Automatic Translation of FORTRAN Programs to Vector Form

RANDY ALLEN and KEN KENNEDY
Rice University

The recent success of vector computers such as the Cray-1 and array processors such as those manufactured by Floating Point Systems has increased interest in making vector operations available to the FORTRAN programmer. The FORTRAN standards committee is currently considering a successor to FORTRAN 77, usually called FORTRAN 8x, that will permit the programmer to explicitly specify vector and array operations.

Although FORTRAN 8x will make it convenient to specify explicit vector operations in new programs, it does little for existing code. In order to benefit from the power of vector hardware, existing programs will need to be rewritten in some language (presumably FORTRAN 8x) that permits the explicit specification of vector operations. One way to avoid a massive manual recoding effort is to provide a translator that discovers the parallelism implicit in a FORTRAN program and automatically rewrites that program in FORTRAN 8x.

Such a translation from FORTRAN to FORTRAN 8x is not straightforward because FORTRAN DO loops are not always semantically equivalent to the corresponding FORTRAN 8x parallel operation. The semantic difference between these two constructs is precisely captured by the concept of *dependence*. A translation from FORTRAN to FORTRAN 8x preserves the semantics of the original program if it preserves the dependences in that program.

The theoretical background is developed here for employing data dependence to convert FORTRAN programs to parallel form. Dependence is defined and characterized in terms of the conditions that give rise to it; accurate tests to determine dependence are presented; and transformations that use dependence to uncover additional parallelism are discussed.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: *Automatic Programming;* D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Processors**]: Optimization

General Terms: Languages

Additional Key Words and Phrases: FORTRAN, detection of parallelism, language translators, vector computing

1. INTRODUCTION

With the advent of successful vector computers such as the Cray-1 [10, 30] and the popularity of array processors such as the Floating Point Systems AP-120 [13, 35], there has been increased interest in making vector operations available to the FORTRAN programmer. One common method is to supply a "vectorizing"

This work was supported by the IBM Corporation.

Authors' address: Department of Computer Science, Brown School of Engineering, Rice University, P.O. Box 1892, Houston, TX 77251-1892.

ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, October 1987, Pages 491–542.

FORTRAN compiler [11] as depicted in Figure 1. Here standard FORTRAN is accepted as input, and, as part of the optimization phase of the compiler, a vectorizing stage attempts to convert the innermost loops to vector operations. The code generator can then produce vector machine code for these operations.

This scheme has two advantages. First, programmers need not learn a new language since the FORTRAN compiler itself takes on the task of discovering where vector operations may be useful. Second, this scheme does not require a major conversion effort to bring old code across.

In practice, however, this system has drawbacks. Uncovering implicitly parallel operations in a program is a subtle intellectual activity—so subtle that most compilers to date have not been able to do a truly thorough job. As a result, the programmer often has to assist the compiler by recoding loops into a form that the compiler can handle. The Cray FORTRAN manual [11], for example, has several pages devoted to such recoding methods. With this system, the programmer is still obligated to rewrite his programs for a new machine, not because the compiler will not accept the old program, but because the compiler is unable to generate suitably efficient code. During a number of visits to Los Alamos Scientific Laboratory, which has several Crays, we have observed the widespread sentiment that every FORTRAN program will need to be rewritten to be acceptably efficient on the Cray.

This presents the question: If we are forced to rewrite FORTRAN programs into vector form anyway, why not write them in a language that permits explicit specification of vector operations, while still maintaining the flavor of FORTRAN? Many such languages have been proposed. VECTRAN [27, 28] is one of the earliest and most influential of such proposals, although there have been numerous others [7, 12, 34]. In fact, it seems clear that the next ANSI standard for FORTRAN, which we shall refer to as FORTRAN 8x, will contain explicit vector operations like those in VECTRAN [5, 26].

Suppose that, instead of a vectorizing FORTRAN compiler, we were to provide FORTRAN 8x compilers for use with vector machines. This would allow programmers to bypass the implicitly sequential semantics of FORTRAN and explicitly code vector algorithms in a language designed for that purpose. However, the basic problem will still be unresolved: What do we do about old code?

One answer is to provide a translator that will take FORTRAN 66 or FORTRAN 77 as input and produce FORTRAN 8x as output. This leads to the system depicted in Figure 2. An advantage of this system is that the translator need not be as efficient as a vectorizing stage embedded in a compiler must be, since the translation from FORTRAN to FORTRAN 8x is usually done only once. Therefore, the translator can attempt more ambitious program transformations, using techniques from program verification and artificial intelligence. Such a translator should uncover significantly more parallelism than a conventional vectorizing compiler.

There is another advantage to this method. If the translator should fail to discover a potential vector operation in a critical program region, the programmer need not try to trick the translator into recognizing it. Instead, he can correct the problem directly in the FORTRAN 8x version. This advantage is very significant, because some loops can be correctly run in vector form even when

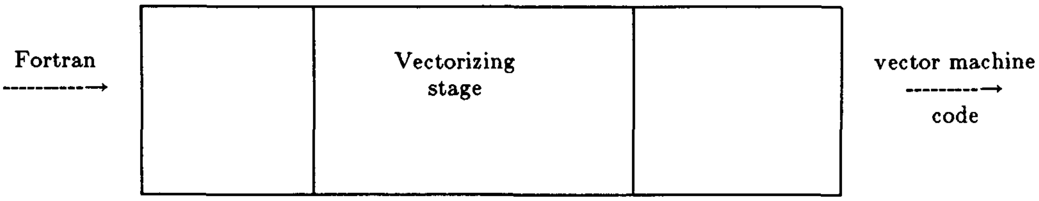


Fig. 1. Vectorizing FORTRAN compiler.

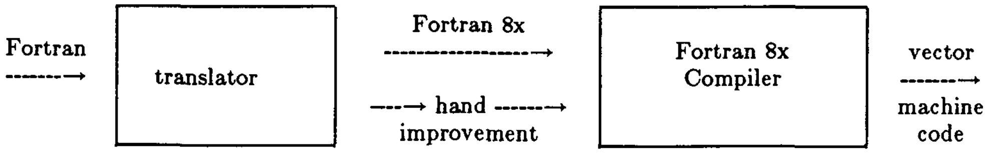


Fig. 2. Vectorizing FORTRAN translator.

a transformation to such a form appears to violate their semantics. Such loops can usually be recoded by a programmer into explicit vector statements in FORTRAN 8x.

This paper discusses the theoretical concepts underlying a project at Rice University to develop an automatic translator, called PFC (for Parallel FORTRAN Converter), from FORTRAN to FORTRAN 8x. The Rice project, based initially upon the research of Kuck and others at the University of Illinois [6, 17–21, 24, 32, 36], is a continuation of work begun while on leave at IBM Research in Yorktown Heights, N.Y. Our first implementation was based on the Illinois PARAFRASE compiler [20, 36], but the current version is a completely new program (although it performs many of the same transformations as PARAFRASE). Other projects that have influenced our work are the Texas Instruments ASC compiler [9, 33], the Cray-1 FORTRAN compiler [15], and the Massachusetts Computer Associates Vectorizer [22, 25].

The paper is organized into seven sections. Section 2 introduces FORTRAN 8x and gives examples of its use. Section 3 presents an overview of the translation process along with an extended translation example. Section 4 develops the concept of interstatement dependence and shows how it can be applied to the problem of vectorization. *Loop carried dependence* and *loop independent dependence* are introduced in this section to extend dependence to multiple statements and multiple loops. Section 5 develops dependence-based algorithms for code generation and transformations for enhancing the parallelism of a statement. Section 6 describes a method for extending the power of data dependence to control statements by the process of *IF conversion.* Finally, Section 7 details the current state of PFC and our plans for its continued development.

2. FUNDAMENTALS OF FORTRAN 8x

It is difficult to describe any language whose definition is still evolving, much less write a language translator for it, but we need some language as the basis for our discussion. In this section, we describe a potential version of FORTRAN 8x, one that is similar to the version presently under consideration by the ANSI X3J3 committee. Our version extends 1977 ANSI FORTRAN to include the proposed features for support of array processing and most of the proposed control structures.

2.1 Array Assignment

Vectors and arrays may be treated as aggregates in the assignment statement. Suppose X and Y are two arrays of the same dimension, then

$$X = Y$$

copies Y into X, element by element. In other words, this assignment is equivalent to

$$\begin{aligned} X(1) &= Y(1) \\ X(2) &= Y(2) \\ &\vdots \\ X(N) &= Y(N). \end{aligned}$$

Scalar quantities may be mixed with vector quantities using the convention that a scalar is expanded to a vector of the appropriate dimensions before operations are performed. Thus

$$X = X + 5.0$$

adds the constant 5.0 to every element of array X.

Array assignments in FORTRAN 8x are viewed as being executed *simultaneously*; that is, the assignment must be treated so that all input operands are fetched before any output values are stored. For instance, consider

$$X = X/X(2).$$

Even though the value of X(2) is changed by this statement, the original value of X(2) is used throughout, so that the result is the same as

$$\begin{aligned} T &= X(2) \\ X(1) &= X(1)/T \\ X(2) &= X(2)/T \\ &\vdots \\ X(N) &= X(N)/T. \end{aligned}$$

This is an important semantic distinction that has a significant impact on the translation process.

2.2 Array Sections

Sections of arrays, including individual rows and columns, may be assigned using *triplet* notation. Suppose A and B are two-dimensional arrays whose subscripts range from 1 to 100 in each dimension; then

```
A(1:100, I) = B(J, 1:100)
```

assigns the Jth row of B to the Ith column of A.

One may also define a range of iteration for vector assignment that is smaller than a whole row or column. Suppose you wish to assign the first M elements of the Jth row of B to the first M elements of the Ith column of A. In FORTRAN 8x, the following assignment could be used:

```
A(1:M, I) = B(J, 1:M).
```

This statement has the effect of the assignments:

```
A(1, I) = B(J, 1)
A(2, I) = B(J, 2)
   .
   .
   .
A(M, I) = B(J, M)
```

even though M might contain a value much smaller than the actual upper bound of these arrays.

The term "triplet" seems to imply that the iteration range specifications such as the one above should have three components. Indeed, the third component, when it appears, specifies a "stride" for the index vector in that subscript position. For example, if we had wished to assign the first M elements of the Jth row of B to the first M elements of the Ith column of A *in odd subscript positions*, the following assignment could have been used.

```
A(1:M*2-1:2, I) = B(J, 1:M).
```

The triplet notation is also useful in dealing with operations involving shifted sections. The assignment

```
A(I, 1:M) = B(1:M, J) + C(I, 3:M + 2)
```

has the effect

```
A(I, 1) = B(1, J) + C(I, 3)
A(I, 2) = B(2, J) + C(I, 4)
   .
   .
   .
A(I, M) = B(M, J) + C(I, M + 2).
```

2.3 Array Identification

Useful as it is, the triplet notation provides no way to skip through elements of a rotated array section, like the diagonal. To do that, one must use the IDENTIFY statement, which allows an array name to be mapped onto an existing array. For example,

```
IDENTIFY /1:M/ D(I) = C(I, I + 1)
```

defines the name D, dimensioned from 1 to M, to be the superdiagonal of C. Thus

$$D = A(1{:}M, J)$$

has the effect

$$\begin{aligned} C(1, 2) &= A(1, J)\\ C(2, 3) &= A(2, J)\\ &\vdots\\ C(M, M + 1) &= A(M, J). \end{aligned}$$

It is important to note that D has no storage of its own; it is merely a pseudonym for a subset of the storage assigned to C.

2.4 Conditional Assignment

The FORTRAN 8x WHERE statement will permit an array assignment to be controlled by a conditional masking array. For example,

```
WHERE(A .GT. 0.0) A = A + B
```

specifies that the vector sum of A and B be formed, but that stores back to A take place only in positions where A was originally greater than zero. The semantics of this statement require that it behave as if only components corresponding to the locations where the controlling condition is true are involved in the computation.

In the special case of statements like

```
WHERE(A .NE. 0.0) B = B/A
```

the semantics require that divide checks arising as a result of evaluating the right-hand side not affect the behavior of the program—the code must hide the error from the user. In other words, any error side-effects that might occur as a result of evaluating the right-hand side in positions where the controlling vector is false are ignored.

2.5 Library Functions

Mathematical library functions, such as SQRT and SIN, are extended on an elementwise basis to vectors and arrays. In addition, new intrinsic functions are provided, such as inner matrix product (DOTPRODUCT) and transpose (TRANSPOSE). The special function

```
SEQ(1, N)
```

returns an index vector from 1 to N. Reduction functions, much like those in APL, are also provided. For example, SUM applied to a vector returns the sum of all elements in that vector.

2.6 User-Defined Subprograms

There are several enhancements to the handling of user-defined subroutines and functions. First, arrays, even identified arrays, may be passed as parameters to subroutines. Second, an array may be returned as the value of a function.

3. THE TRANSLATION PROCESS

Now we are ready to describe, in an idealized way, the process of translating a FORTRAN program into FORTRAN 8x. In so doing, we will illustrate some important aspects of the problem.

Suppose the translator is presented with the following FORTRAN fragment:

```
          DO 20 I = 1, 100
S1          KI = I
            DO 10 J = 1, 300, 3
S2            KI = KI + 2
S3            U(J) = U(J) * W(KI)
S4            V(J + 3) = V(J) + W(KI)
      10    CONTINUE
      20  CONTINUE
```

The goal is to convert statements S_3 and S_4 to vector assignments, removing them from the innermost loop. That will be possible if there is no semantic difference between executing them in a sequential loop and executing them as vector statements. Consider a somewhat simpler case:

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

If we are to convert this to the vector assignment

$$X(1{:}100) = X(1{:}100) + Y(1{:}100)$$

we must be sure that no semantic difference arises. Specifically, a vector assignment requires that the right-hand side be fetched before any stores occur on the left. Thus, it can use only old values of its input operands. If the sequential loop computes a value on one iteration and uses it on a later iteration, it is not semantically equivalent to a vector statement. The following fragment

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

cannot be correctly converted to the vector assignment

$$X(2{:}101) = X(1{:}100) + Y(1{:}100)$$

because each iteration after the first uses a value computed on the previous iteration. The vector assignment would use only old values of X. An iterated statement that depends upon itself in the manner shown is called a *recurrence.*

In order to distinguish the two cases above, the translator must perform a precise test to determine whether or not a statement depends upon itself—that is, whether or not it uses a value that it has computed on some previous iteration. Details of this *dependence test* will be provided in the next section; for now, it is enough to know that certain program transformations are required to make the test possible.

The first of these, *DO-loop normalization*, transforms loops so that the loop induction variables iterate from 1 to some upper bound by increments of 1. Sometimes new induction variables must be introduced to accomplish this. Within the loop, every reference to the old loop induction variable is replaced by an expression in the new induction variable. The effect of DO-loop normalization on our example is

```
       DO 20 I = 1, 100
         KI = I
         DO 10 j = 1, 100
           KI = KI + 2
           U(3 * j - 2) = U(3 * j - 2) * W(KI)
           V(3 * j + 1) = V(3 * j - 2) + W(KI)
    10   CONTINUE
S6       J = 301
    20 CONTINUE
```

Note that the new variable j (written as a lowercase letter to signify that it has been introduced by the translator) is now the inner loop induction variable and that an assignment S_6 has been introduced to define the previous induction variable on exit from the loop. In this form, the upper bound of the loop is precisely the number of times the loop will be executed.

A major goal of this sequence of normalizing transformations is to convert all subscripts to linear functions of loop induction variables. To accomplish this conversion, uses of auxiliary induction variables, such as KI in our example, must be replaced. This transformation, called *induction variable substitution* [36], replaces statements that increment auxiliary induction variables with statements that compute them directly using normal loop induction variables and loop constants. The effect in our example is as follows:

```
       DO 20 I = 1, 100
         KI = I
         DO 10 j = 1, 100
           U(3 * j - 2) = U(3 * j - 2) * W(KI + 2 * j)
           V(3 * j + 1) = V(3 * j - 2) + W(KI + 2 * j)
    10   CONTINUE
         KI = KI + 200
         J = 301
    20 CONTINUE
```

Here the computation of KI has been removed from the loop and all references to KI have been replaced by references to the initial value of KI plus the sum total of increments that can occur by the relevant iteration, expressed as a function of j. At the end of the loop, an assignment updates the value of KI by the aggregate total of all increments in the loop. Note that since it attempts to replace simple additions with multiplications, induction variable substitution is, in a sense, an inverse of the classical optimization technique *operator strength reduction* [2, 8].

The final transformation in preparation for dependence testing is *expression folding*, which substitutes integer expressions and constants forward into subscripts, with simplification where possible. The result in our example is

```
          DO 20 I = 1, 100
            DO 10 j = 1, 100
S3            U(3 * j - 2) = U(3 * j - 2) * W(I + 2 * j)
S4            V(3 * j + 1) = V(3 * j - 2) + W(I + 2 * j)
     10     CONTINUE
S5          KI = I + 200
S6          J = 301
     20   CONTINUE
```

In this example, the first assignment to KI in the outer loop has been removed and references to KI replaced by the right-hand side (I) in statements S_3, S_4, and S_5. It should be noted that statements S_5 and S_6 could now be removed from the loop by *forward substitution*; this is, in fact, done in the actual translator.

Once the subscripts have been transformed, a standard data flow analysis phase can be applied to build the data flow graph for the whole program. This graph can be used to propagate constants throughout the program and to recognize *dead statements*, that is, statements whose output will never be used. In the example above, suppose that KI and J are both dead after the code segment shown. Then all assignments of those variables will be deleted, as shown below.

```
          DO 20 I = 1, 100
            DO 10 j = 1, 100
S3            U(3 * j - 2) = U(3 * j - 2) * W(I + 2 * j)
S4            V(3 * j + 1) = V(3 * j - 2) + W(I + 2 * j)
     10     CONTINUE
     20   CONTINUE
```

The point of this complex assortment of transformations is to attempt to convert all subscripts to a canonical form: linear functions of the DO loop induction variables. This form makes it possible to apply a powerful and precise test for interstatement dependence. In the example above, we have succeeded in putting all subscripts into the desired form, so we can use precise tests to determine what dependences exist among the statements in the inner loop.

Once the dependences have been identified, we are ready for *vector code generation*. Using dependence information, the translator determines which of the remaining statements does not depend on itself. As it happens, statement S_3 does not depend upon itself, while statement S_4 does (and hence represents a recurrence). Therefore, statement S_3 is converted to a vector assignment, while statement S_4 is left in a sequential loop by itself.

```
          DO 20 I = 1, 100
S3          U(1:298:3) = U(1:298:3) * W(I - 2:I + 200:2)
            DO 10 j = 1, 100
S4            V(3 * j + 1) = V(3 * j - 2) + W(I + 2 * j)
     10     CONTINUE
     20   CONTINUE
```

Figure 3 gives an overview of the translation process as implemented in PFC. The *scanner-parser* phase converts the input program to an abstract syntax tree that is used as the intermediate form throughout the translation. The *pretty*

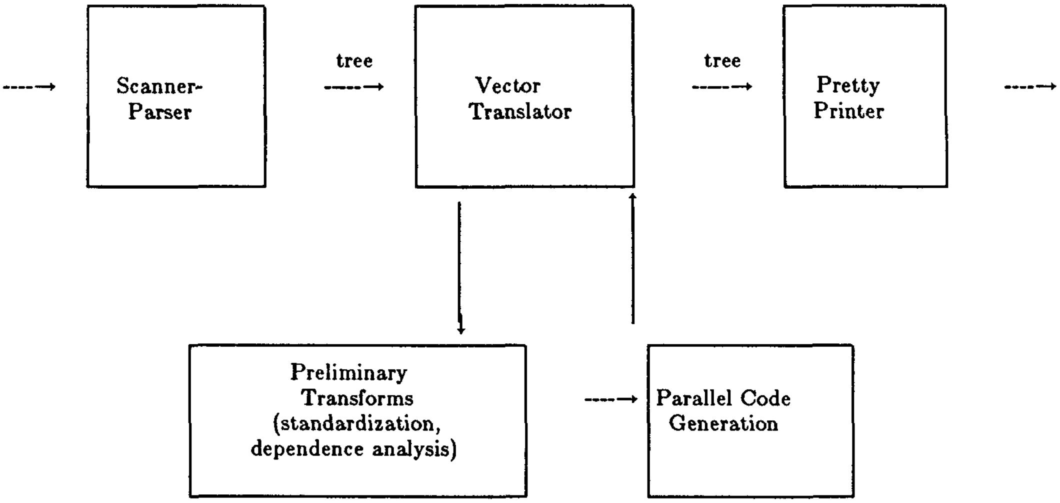


Fig. 3. Overview of PFC.

printer can reconstruct a source program from the abstract syntax tree; it is used throughout the translator. The *vector translation* phase consists of three main subphases:

(1) *subscript standardization*, which encompasses all the transformations that attempt to put subscripts into canonical form;
(2) *dependence analysis*, which builds the interstatement dependence graph;
(3) *parallel code generation*, which generates array assignments where possible.

Each of these will be discussed in more detail. Since the dependence test is fundamental to these phases, it is the subject of the next section.

4. DEPENDENCE ANALYSIS

Since a statement can be directly vectorized only if it does not depend upon itself, the analysis of interstatement dependence is an important part of PFC. In this section we formalize the concept of dependence and introduce a precise test for interstatement dependence in a single loop. We then extend this concept to multiple loops with the concept of *layered* dependence.

4.1 Interstatement Dependence

Informally, a statement S_2 *depends upon* statement S_1 if some execution of S_2 uses as input a value created by some previous execution of S_1. In straight-line code, this condition is easy to determine. Since we are interested in determining whether a statement depends upon itself and since this can only happen if the execution flows from a statement back to itself via a loop, we must be able to determine dependence within loops.

To illustrate the complexity of this problem, consider the following loop:

```
      DO 10 J = 1, N
        X(J) = X(J) + C
10    CONTINUE
```

The statement in this loop does not depend on itself because the input variable X(J) always refers to the old value at that location. By contrast, the similar loop

```
      DO 10 J = 1, N − 1
        X(J + 1) = X(J) + C
   10 CONTINUE
```

forms a recurrence and cannot be directly converted to vector form. The input on any iteration $i + 1$ is always the value of X computed on iteration i. As a result, the direct vector analog will not be equivalent.

In order to understand intrastatement dependence, we need to examine the generalized form of a (possibly dependent) single statement within a loop.

```
      DO 10 i = 1, N
(*)     X(f(i)), = F(X(g(i)))
   10 CONTINUE
```

Here f and g are arbitrary subscript expressions, and **F** is some expression involving its input parameter.

Definition. Statement (*) *depends upon itself* if and only if there exist integers i_1, i_2 such that $1 \leq i_1 < i_2 \leq N$ and $f(i_1) = g(i_2)$.

The integers i_1 and i_2 represent separate iterations of the i loop. On iteration i_1, statement (*) computes a value that is subsequently used on iteration i_2. To put it another way, statement (*) depends upon itself if and only if the *dependence equation*

$$f(x) - g(y) = 0$$

has integer solutions in the region depicted in Figure 4.

If f and g are permitted to be arbitrary functions of the DO loop induction variable, then determining whether statement (*) depends upon itself is an extremely difficult problem. The problem becomes much more tractable when f and g are restricted to be *linear* functions of the induction variable, that is,

$$f(i) = a_0 + a_1 i$$
$$g(i) = b_0 + b_1 i.$$

This is by far the most common case encountered in practice. With this restriction, the dependence equation has solutions if and only if

$$a_1 x - b_1 y = b_0 - a_0.$$

In order for x and y to be viable solutions to the dependence equation, they must be irrtegers. As a result, we are seeking integer solutions to an equation with integer coefficients. Almost any text on number theory (e.g., [14]) will include the following theorem on Diophantine equations.

THEOREM 1. *The linear Diophantine equation $ax + by = n$ has a solution if and only if $\gcd(a, b) \mid n$.*

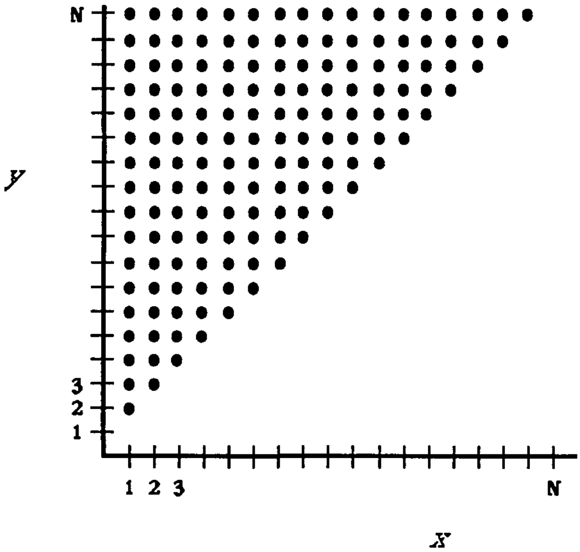


Fig. 4. The region of interest.

Immediately following from the above theorem is a *necessary* requirement for dependence:

COROLLARY 1 (GCD TEST). *Statement* (*) *with* $f(i) = a_0 + a_1 i$ *and* $g(i) = b_0 + b_1 i$ *depends upon itself* only if $gcd(a_1, b_1) \mid b_0 - a_0$.

Note that the gcd test is only necessary for dependence, because an integer solution to the dependence equation is not sufficient to guarantee self-dependence. For that, the solution must exist within the region depicted in Figure 4.

Although the gcd test is interesting, it is of limited usefulness, because the most common case by far is that in which the gcd of a_1 and b_1 is 1. A more effective test can be developed by examining the effects of region constraints on the existence of solutions. The mathematics of determining integer solutions to a Diophantine equation within a restricted region can lead to extremely expensive tests for dependence. As a result, it is more useful to investigate the *real* solutions to the dependence equation in the region of interest.

Consider the real solutions of

$$h(x, y) = f(x) - g(y) = 0$$

in the region R:

$$1 \le x \le N - 1 \qquad 2 \le y \le N \qquad x \le y - 1.$$

A real solution to the dependence equation exists in R if and only if the level curve at 0 for h passes through R, as depicted in Figure 5. If h meets fairly general continuity conditions, the intermediate value theorem guarantees that h has

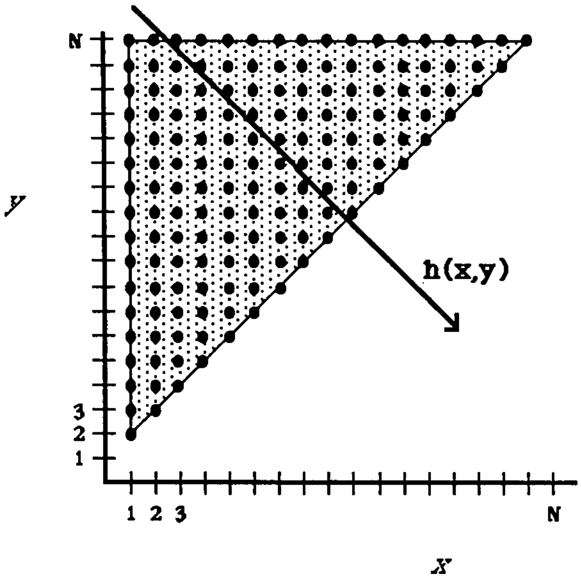


Fig. 5. Real solutions in R.

zeros in R if and only if there exist points (x_1, y_1) and (x_2, y_2) in R such that

$$h(x_1, y_1) \leq 0 \leq h(x_2, y_2).$$

The following theorem summarizes this observation.

THEOREM 2. *If $h(x, y)$ is continuous in R, then there exists a solution to $h(x, y) = 0$ in R if and only if*

$$\min_R h(x, y) \leq 0 \leq \max_R h(x, y).$$

COROLLARY 2. *If $f(x)$ and $g(y)$ are continuous, then statement (*) depends upon itself only if*

$$\min_R (f(x) - g(y)) \leq 0 \leq \max_R (f(x) - g(y)).$$

Once again, this condition is necessary, but not sufficient; the existence of real solutions in R does not imply the existence of integer solutions. As a result, the requirements of Corollary 2 may be satisfied by a statement that is not self-dependent.

Corollary 2 is useful only if there is a fast way to find the maximum and minimum on a region. Such a way is provided by the following theorem, adapted from a result due to Banerjee [6].

THEOREM 3. *If $f(x) = a_0 + a_1x$ and $g(y) = b_0 + b_1y$, then*

$$\max_R (f(x) - g(y)) = a_0 + a_1 - b_0 - 2b_1 + (a_1^+ - b_1)^+(N - 2)$$

$$\min_R (f(x) - g(y)) = a_0 + a_1 - b_0 - 2b_1 - (a_1^- - b_1)^+(N - 2)$$

where the superscript notation is defined by the following:

Definition. If t denotes a real number, then the *positive part* t^+ and the *negative part* t^- of t are defined as

$$t^+ = \begin{cases} t, & \text{if } t \geq 0 \\ 0, & \text{if } t < 0 \end{cases}$$

$$t^- = \begin{cases} -t, & \text{if } t \leq 0 \\ 0, & \text{if } t > 0. \end{cases}$$

Thus $t^+ \geq 0$, $t^- \geq 0$, and $t = t^+ - t^-$. The proof of a multidimensional variant of Theorem 3 is given in the Appendix.

Theorem 2 and Theorem 3 establish the following result.

COROLLARY 3 (BANERJEE INEQUALITY). *If* $f(x) = a_0 + a_1x$ *and* $g(y) = b_0 + b_1y$ *then statement* (*) *depends on itself only if*

$$-b_1 - (a_1^- + b_1)^+(N - 2) \leq b_0 + b_1 - a_0 - a_1 \leq -b_1 + (a_1^+ - b_1)^+(N - 2).$$

PROOF. Immediate from Corollary 2 and Theorem 3 with subtraction of

$$a_0 + a_1 - b_0 - b_1$$

from each side of the inequalities in Corollary 2. □

Corollaries 1 and 3 comprise a necessary test for self-dependence. This test may be expressed algorithmically as follows:

(1) Determine whether f and g are linear. If they are, then compute a_0, b_0, a_1, and b_1.

(2) If either (a) $\gcd(a_1, b_1)$ does not divide $b_0 - a_0$ or (b) Banerjee's inequality does not hold, then the statement does *not* depend upon itself. Otherwise, assume it does (even though it may not).

Testing for self-dependence in the presence of multiple loops is more complicated. Before developing that test, let us examine some applications of dependence.

4.2 Dependence Graphs and Their Application

While determining whether a statement depends upon itself or not is useful, it is clearly a simplified case of a more general phenomenon. In general, a statement may depend upon itself indirectly through a chain of zero (the direct case) or more statements, as the following example illustrates:

```
               DO 10 I = 1, 100
S1               T(I) = A(I) * B(I)
S2               S(I) = S(I) + T(I)
S3               A(I + 1) = S(I) + C(I)
          10   CONTINUE
```

Although statements S_1, S_2, and S_3 all depend upon themselves indirectly, no statement depends directly upon itself. In order to uncover the recurrence, it is necessary to first uncover the individual statement-to-statement dependences.

Kuck and others at the University of Illinois [18, 32] have defined three types of dependence that can hold between statements.

Definition. If control flow within a program can reach statement S_2 after passing through S_1, then S_2 *depends on* S_1, written $S_1 \; \Delta \; S_2$, if

(1) S_2 uses the output of S_1. This type of dependence is known as *true dependence* (denoted δ), and is illustrated by the following

$$\begin{tikzcd} S_1\colon X = \arrow[d, bend left=60, "\delta"] \\ S_2\colon \quad = X. \end{tikzcd}$$

(2) S_1 might wrongly use the output of S_2 if they were reversed in order. This type of dependence is called *antidependence* (denoted $\bar{\delta}$) and is illustrated by the following:

$$\begin{tikzcd} S_1\colon \quad = X \arrow[d, bend left=60, "\bar{\delta}"] \\ S_2\colon X = \end{tikzcd}$$

(3) S_2 recomputes the output of S_1; thus, if they were reversed, later statements might wrongly use the output of S_1. This type of dependence is termed *output dependence* (denoted δ^0) and is illustrated by

$$\begin{tikzcd} S_1\colon X = \arrow[d, bend left=60, "\delta^0"] \\ S_2\colon X = \end{tikzcd}$$

Thus $\Delta = \delta + \bar{\delta} + \delta^0$, where addition means set union.

All three types of dependence must be considered when detecting recurrences that inhibit vectorization. Note that dependence, in this sense, denotes a relation between two statements that captures the order in which they must be executed. This concept of dependence differs from that normally encountered in data flow analysis, where dependence implies that one statement must be present for another to receive the correct values. Antidependence and output dependence are meaningless in such a setting, since they only fix the order of statements. In particular, we would not wish to use either of these *pseudodependences* (as we will henceforth call them) in the dead statement eliminator; it would be ridiculous to refuse to eliminate a particular statement because some useful statement recomputes its output and hence "depends" on it.

In any case, the common element among these types of dependence is the use of the same memory location in two statements (or in two different executions of the same statement). The actual type of dependence created by a common use is determined by which statement (or statements) defines the location and which statement uses the location. As a result, all three types of dependence can be decided by the same test. The only change necessary is to switch the locations from which the subscript functions f and g are taken. We will therefore discuss only the test for true dependence between two statements in a loop, with the understanding that the same methods are easily extended to all types of dependence.

In contrast to the case of self-dependence, there are two completely separate ways in which dependence can arise between different statements. One statement may store into a location on one iteration of the loop; the other statement may fetch from that location on a later iteration of the loop. The dependence of statement S_1 on statement S_3 in the previous example illustrates this type of dependence, known as *loop carried dependence*. The other possibility is that one statement may store into a location on an iteration of the loop; on the same iteration *another* statement may fetch from that location. The dependence of statement S_2 on statement S_1 illustrates this type of dependence, known as *loop independent dependence*. In order for one statement to have a true dependence upon another, it is necessary that the statement defining the common memory location precede (in terms of execution) the statement using that location. Since these two types of dependence completely describe all possible ways that a definition can precede a use, these two types of dependence completely encapsulate all possible data dependences.

Before providing a more formal definition of these types of dependence, it is convenient to introduce some notation.

Definition. Let S_1 and S_2 be two statements that appear in the same DO-loop. We say that S_2 *follows* S_1, or $S_2 > S_1$ if S_1 appears first in the loop and $S_1 \neq S_2$.

Consider two statements S_1 and S_2, both contained in one loop with loop induction variable i. Suppose S_1 is of the form

$$S_1\text{: } \mathrm{X}(f(i)) = \mathbf{F}(\ldots)$$

where f is a subscript expression and $\mathbf{F}$ is an expression, and suppose S_2 is of the form

$$S_2\text{: } \mathbf{A} = \ \mathbf{G}(\mathrm{X}(g(i)))$$

where $\mathbf{A}$ is an arbitrary variable (possibly subscripted), $\mathbf{G}$ is an expression involving $\mathrm{X}(g(i))$, and g is a subscript expression. Then the following definitions are obvious from the above discussion.

Definition. S_2 has a *loop carried dependence* on S_1 (denoted $S_1\ \delta\ S_2$) if there exist i_1 and i_2 such that $1 \le i_1 < i_2 \le N$ and $f(i_1) = g(i_2)$.

Definition. S_2 has a *loop independent dependence* on S_1 (denoted $S_1\ \delta_\infty\ S_2$) if there exists some iteration i, $1 \le i \le N$, such that $S_2 > S_1$ and $f(i) = g(i)$.

Note that self-dependence is merely a special case of loop carried dependence. It is true in the case of self-dependence (as in the case of all loop carried dependences) that the dependence arises because of the iteration of a loop. In particular, there is no way in which a single statement can first define and later use a value unless it is contained within a loop. Loop independent dependences, on the other hand, arise not because of loop iterations, but because of the relative

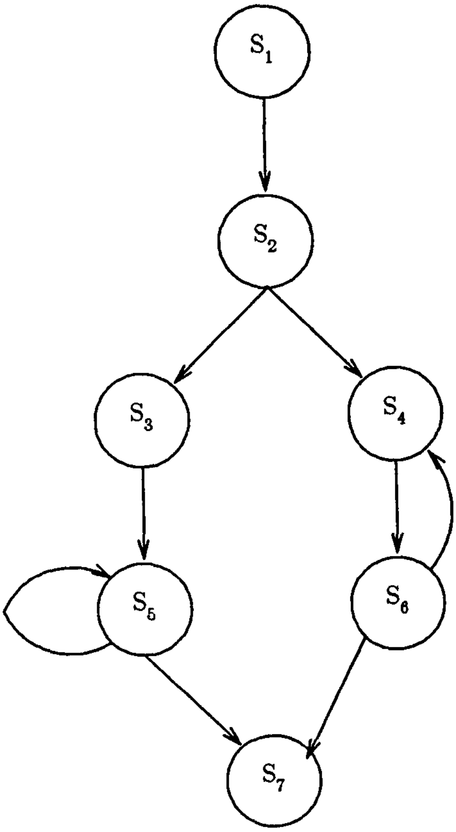


Fig. 6. A sample dependence graph D.

position of two statements within the loop. These dependences do not "cross" loop iterations. A loop carried dependence cannot be limited to a single iteration by its very nature.

Since the primary function of the translator is to detect recurrences, it is useful to see how the concept of dependence aids in that function. In order to apply dependence analysis, it is necessary to

(1) test each pair of statements for dependence (true, anti, or output), building a dependence relation D;
(2) compute the transitive closure D^+ of the dependence relation;
(3) execute each statement that does not depend upon itself in D^+ in parallel; all others are part of a recurrence.

There is a small wrinkle, however. The parallel statements must be executed in an order that is consistent with the dependence relation D^+. To view it in the manner suggested by Kuck [18], consider D as a graph in which individual statements are nodes and in which pairs in the relation are represented by directed edges. Figure 6 contains an example of such a graph. Cycles in this graph

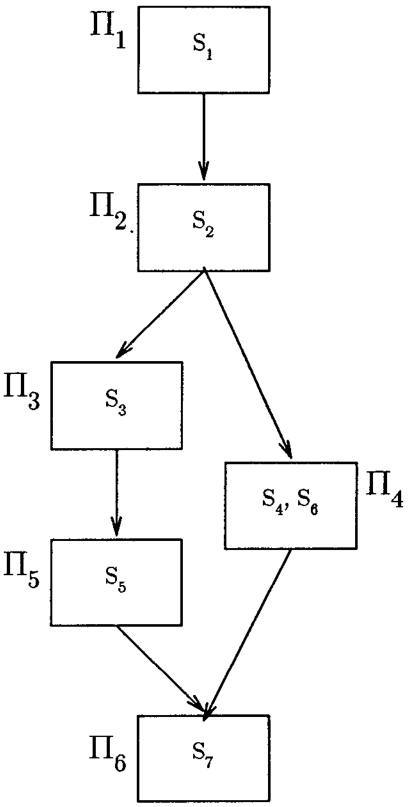


Fig. 7. The derived dependence graph for π-blocks.

represent recurrences. If each cycle and each single statement not part of a cycle are reduced to a single node (called a *π-block*), then the dependence graph D' derived from this transformation on D is acyclic (see Figure 7). Using a topological sort [16], we can then generate code for each π-block in an order that preserves the dependence relation D'.

As an example, consider the program below.

```
          DO 10 I = 1, 99
S1          X(I) = I
S2          B(I) = 100-I
     10   CONTINUE
          DO 20 I = 1, 99
S3          A(I) = F(X(I))
S4          X(I + 1) = G(B(I))
     20   CONTINUE
```

Figure 8 depicts the dependences among the numbered statements in this program, ignoring dependences on the DO statements. Since there are no cycles, all the statements may be executed in vector, but we must be careful to choose an order that preserves dependences. In particular, S_4 must come before S_3 in the final code. Choosing the order (S_1, S_2, S_4, S_3), the result is

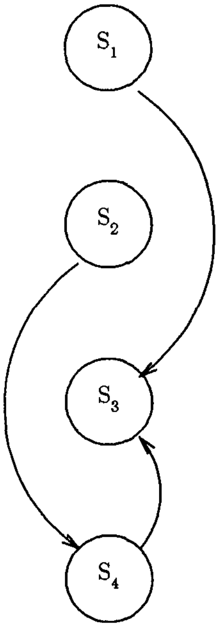


Fig. 8. Dependences in the example program.

the FORTRAN 8x program

```
X(1:99) = SEQ(1, 99, 1)
B(1:99) = SEQ(99, 1, −1)
X(2:100) = G(B(1:99))
A(1:99) = F(X(1:99))
```

which is fully consistent with the original sequential semantics.

Currently, the translator leaves a recurrence coded as a sequential DO-loop.

4.3 Dependence in Multiple Loops

When extending the definition of dependence to multiple loops, it is convenient to precisely pinpoint the loop that creates a loop carried dependence. An example will illustrate this concept.

```
              DO 100 i = 1, 100
                DO 90 j = 1, 100
                  DO 30 k = 1, 100
S1                  X(i, j + 1, k) = A(i, j, k) + 10
      30          CONTINUE
                  DO 80 l = 1, 50
S2                  A(i + 1, j, l) = X(i, j, l) + 5
      80          CONTINUE
      90        CONTINUE
     100      CONTINUE
```

First, statements S_1 and S_2 depend upon each other. On every iteration of the j loop other than the first, S_2 uses a value that was computed on the previous iteration of the j loop by S_1. Similarly, on every iteration of the i loop other than the first, S_1 uses a value computed on the previous iteration by S_2. Neither the k loop nor the l loop can carry a dependence between the statements, because two statements must be nested within a loop in order for it to carry a dependence between them.

It is important to recognize which loop carries a particular dependence if we are to do a good job of translation. This is aptly illustrated by the example above, because S_1 and S_2 may be executed in parallel in two dimensions even though they form a global recurrence. If the outermost loop is left sequential we get

```
          DO 100 i = 1,100
            X(i, 2:101, 1:100) = A(i, 1:100, 1:100) + 10
            A(i + 1, 1:100, 1:50) = X(i, 1:100, 1:50) + 5
      100 CONTINUE
```

Clearly, this partial vectorization is desirable.

The test for loop carried dependence presented earlier can be generalized to detect which loop carries a dependence by the following:

Let f and g be subscript mappings

$$f : Z^{n_1} \rightarrow Z^m$$
$$g : Z^{n_2} \rightarrow Z^m$$

where Z is the set of all integers, n_1 is the number of loops containing statement S_1

$$S_1\text{: } X(f(x_1, x_2, \ldots, x_{n_1})) = \mathbf{F}(\ )$$

n_2 is the number of loops containing statement S_2

$$S_2\text{: } A = \mathbf{G}(X(g(x_1, \ldots, x_{n_2}))$$

and m is the number of subscripts for array X. The symbol $\mathbf{F}(\ )$ denotes an arbitrary left-hand side. We use $x_1, x_2, \ldots$ to denote the induction variables for the loops, with x_1 being the induction variable for the outermost loop. In general, we will number the loops from the outermost to the innermost. The upper bound of the ith loop surrounding S_1 is assumed to be M_i; the upper bound of the ith loop surrounding S_2 is assumed to be N_i; hence $M_i = N_i$ for $1 \le \mathrm{i} \le n$, where n is the number of common loops surrounding the two statements.

Definition. Statement S_2 depends on S_1 with respect to *carrier* k ($k \le n$), written $S_1\ \delta_k\ S_2$, if there exist $(i_1, i_2, \ldots, i_{k-1})$, $(j_{k+1}, j_{k+2}, \ldots, j_{n_1})$, $(l_{k+1}, l_{k+2}, \ldots, l_{n_2})$, and integers ζ_1, ζ_2 in the following regions:

$$\begin{array}{ll} 1 \le i_q \le N_q & \forall_q \text{ s.t. } 1 \le q \text{ and } q < k \\ 1 \le j_q \le M_q & \forall_q \text{ s.t. } k < q \text{ and } q \le n_1 \\ 1 \le l_q \le N_q & \forall_q \text{ s.t. } k < q \text{ and } q \le n_2 \\ 1 \le \zeta_1 < \zeta_2 \le N_k & \end{array}$$

such that the following equation holds:

$$f(i_1, i_2, \ldots, i_{k-1}, \zeta_1, j_{k+1}, \ldots, j_{n_1}) = g(i_1, i_2, \ldots, i_{k-1}, \zeta_2, l_{k+1}, \ldots, l_{n_2}).$$

Intuitively, we test for dependence with respect to carrier loop k by holding the outer loop indices constant and letting the inner loop indices run free. Note that the same definition can be used for antidependence and output dependence as well.

Interstatement dependence can now be defined in terms of dependence with respect to a particular carrier.

Definition. S_2 depends directly on S_1, $S_1 \, \Delta \, S_2$, if and only if there exists some $k \geq 1$ such that $S_1 \, \delta_k \, S_2$.

If we view dependence as a relation, then

$$\Delta = \sum_{k=1}^{\infty} \delta_k,$$

where addition is interpreted as set union.

Now we are ready for the main result of this section—testing for dependence on a particular carrier.

THEOREM 4. *If* $f(x_1, \ldots, x_{n_1}) = a_0 + \sum_{i=1}^{n_1} a_i x_i$, $g(x_1, \ldots, x_{n_2}) = b_0 + \sum_{i=1}^{n_2} b_i x_i$, *and* S_1 *and* S_2 *are of the form*

$$\begin{aligned} &S_1\colon X(f(x_1, \ldots, x_{n_1})) = \mathbf{F}(\ldots) \\ &S_2\colon A = \mathbf{G}(X(g(x_1, \ldots, x_{n_2}))) \end{aligned}$$

and are contained in n common loops (assumed normalized), $n \geq k$, *and the upper bounds of the loops surrounding* S_1 *are* M_i *and the upper bounds of the loops surrounding* S_2 *are* N_i ($M_i = N_i$ *for* $i \leq n$), *then* $S_1 \, \delta_k \, S_2$ *only if*

(*a*) *gcd test*:

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

(*b*) *Banerjee inequality*:

$$\begin{aligned} -b_k - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) - (a_k^- + b_k)^+(N_k - 2) \\ - \sum_{i=k+1}^{n_1} a_i^-(M_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1) \\ \leq \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \\ \leq -b_k + \sum_{i=1}^{ku-1} (a_i - b_i)^+(N_i - 1) + (a_k^+ - b_k)^+(N_k - 2) \\ + \sum_{i=k+1}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1). \end{aligned}$$

The long but straightforward proof is given in the Appendix.

This theorem is an adaptation of a result by Banerjee. The gcd test has been slightly sharpened over Banerjee's and the test has been formulated as a test for dependence with respect to a specific carrier *k*.

When the theory of loop carried dependence is extended to account for multiple loops, it is convenient to determine which loop "creates" the dependence. The previous theorem does exactly that. Extending loop independent dependence to multiple loops is much simpler, since such dependences do not arise from the iteration of loops, but from the relative statement position. The gcd test and Banerjee's inequality can be modified to test for loop independent dependences as follows.

THEOREM 5. *If* $f(x_1, \ldots, x_{n_1}) = a_0 + \sum_{i=1}^{n_1} a_i x_i$, $g(x_1, \ldots, x_{n_2}) = b_0 + \sum_{i=1}^{n_2} b_i x_i$, *and* S_1 *and* S_2 *are of the form*

$$S_1\colon X(f(x_1, \ldots, x_{n_1})) = \mathbf{F}(\ldots)$$
$$S_2\colon A = \mathbf{G}(X(g(x_1, \ldots, x_{n_2})))$$

and are contained in n common loops (assumed normalized), and the upper bounds of the loops surrounding S_1 *are* M_i *and the upper bounds of the loops surrounding* S_2 *are* N_i ($M_i = N_i$ *for* $i \le n$), *then* $S_1\ \delta_\infty\ S_2$ (S_2 *has a loop independent dependence on* S_1) *only if* S_2 *follows* S_1 *and*

(*a*) *gcd test*:

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

(*b*) *Banerjee inequality*:

$$-\sum_{i=1}^{n} (a_i - b_i)^-(N_i - 1) - \sum_{i=n+1}^{n_1} a_i^-(M_i - 1) - \sum_{i=n+1}^{n_2} b_i^+(N_i - 1)$$
$$\le \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i$$
$$\le \sum_{i=1}^{n} (a_i - b_i)^+(N_i - 1) + \sum_{i=n+1}^{n_1} a_i^+(M_i - 1) + \sum_{i=n+1}^{n_2} b_i^-(N_i - 1).$$

4.4 The Depth of a Dependence

The test for dependence given in the previous section leads in a natural way to the concept of dependence depth. Recall that S_2 depends directly on S_1 ($S_1\ \Delta\ S_2$) if and only if there exists $k > 0$ such that $S_1\ \delta_k\ S_2$. Clearly, if we disregard some of the outer loops, holding them constant, the dependence may not exist. Therefore, let us introduce the concept of *depth* into our theory of dependence.

Definition. We say that S_2 *depends on* S_1 *at depth d* (*denoted* $S_1\ \Delta_d\ S_2$), if there exists a $k \ge d$ such that $S_1\ \delta_k\ S_2$. In other words,

$$\Delta_d = \sum_{k=d}^{\infty} \delta_k.$$

Note that in this scheme, a loop independent dependence is a dependence of infinite depth. The reason for this will become clear shortly.

Definition. For statements S_1 and S_2, $\eta^0(S_1, S_2)$, the *nesting level* of the direct dependence of S_2 on S_1, is the maximum depth at which the dependence exists, that is,

$$\eta^0(S_1, S_2) = \begin{cases} \max\{k \geq 1 \mid S_1 \; \delta_k \; S_2\}, & \text{if} \quad S_1 \; \Delta \; S_2 \\ 0, & \text{otherwise.} \end{cases}$$

LEMMA 2

(*a*) *If* $d_1 \geq d_2$ *then* $S_1 \; \Delta_{d_1} \; S_2 \Rightarrow S_1 \; \Delta_{d_2} \; S_2$.
(*b*) *If* $S_1 \; \Delta \; S_2$ *and* $\tau = \eta^0(S_1, S_2)$, *then* $S_1 \; \not\Delta_r \; S_2$ *and* $S_1 \; \Delta_{\tau+1} \; S_2$.

PROOF. Obvious. Clearly, $\eta^0(S_1, S_2)$ is easy to compute for any pair of statements. □

It is customary to view dependence as a transitive relation. That is, if S_2 depends on S_1, and S_3 depends on S_2, then S_3 depends on S_1, albeit indirectly. Henceforth, we will say that S_2 depends on S_1 if $S_1 \; \Delta^+ \; S_2$ where Δ^+ is the transitive closure of Δ, that is,

$$\Delta^+ = \Delta + \Delta^2 + \Delta^3 + \cdots.$$

In other words, $S_1 \; \Delta^+ \; S_2$ if there exist statements $T_0, T_1, \ldots, T_n$ ($n \geq 1$) such that

$$T_0 = S_1$$
$$T_n = S_2$$

and

$$S_1 = T_0 \; \Delta \; T_1 \; \Delta \; T_2 \; \Delta \cdots \Delta \; T_n = S_2.$$

We shall refer to the sequence $(T_0, T_1, \ldots, T_n)$ as a *path* in the dependence graph.

It is also possible to extend the notion of loop carried dependence by taking the transitive closure. That is, $S_1 \; \Delta_d^+ \; S_2$ if there exists a path $T_0, T_1, \ldots, T_n$ ($n \geq 1$) such that

$$S_1 = T_0 \; \Delta_d \; T_1 \; \Delta_d \cdots \Delta_d \; T_n = S_2.$$

Next we extend η^0 to dependence paths.

Definition. Let $P = (T_0, T_1, \ldots, T_n)$ be a path in the dependence graph; in other words, $T_0 \; \Delta \; T_1 \; \Delta \cdots \Delta \; T_n$. The *nesting level* of P, $\eta^0(P)$, is the maximum depth at which all the dependences in the path still exist.

$$\eta^0(P) = \max\{d \geq 1 \mid T_i \; \Delta_d \; T_{i+1} \; \forall i, \; 0 \leq i \leq n-1\}.$$

LEMMA 3. *If* $P = (T_0, T_1, \ldots, T_n)$, *then* $\eta^0(P) = min\{\eta^0(T_i, T_{i+1}), 0 \leq i \leq n-1\}$.

PROOF. Let $\tau = \min\{\eta^0(T_i, T_{i+1}), 0 \leq i \leq n-1\}$. By Lemma 2, all of the dependences $T_i \; \Delta_\tau \; T_{i+1}$ exist, while at least one such dependence, the minimum, does not exist at level $\tau + 1$. □

Finally, we extend the concept of nesting level to arbitrary pairs of statements.

Definition. For arbitrary statements S_1 and S_2, $\eta(S_1, S_2)$, the *nesting level* of the dependence, is the maximum depth d at which there exists a path $(T_0, T_1, \ldots, T_n)$ such that $S_1 = T_0 \Delta_d T_1 \Delta_d \cdots \Delta_d T_n = S_2$, that is,

$$\eta(S_1, S_2) = \begin{cases} \max\{d \geqslant 1 \mid S_1 \Delta_d^+ S_2\}, & \text{if } S_1 \Delta^+ S_2 \\ 0, & \text{otherwise.} \end{cases}$$

Note that we must distinguish $\eta^0(S_1, S_2)$, the depth of a direct dependence, and $\eta(S_1, S_2)$, the depth of a dependence. This is because it is possible that there exists a dependence path from S_1 to S_2 at a depth greater than that of the direct dependence. In other words, $\eta(S_1, S_2) \geq \eta^0(S_1, S_2)$ and the inequality may be strict.

THEOREM 6. *If $S_1 \Delta^+ S_2$, $\eta(S_1, S_2) = max\{\eta^0(P) \mid P$ a dependence path from S_1 to $S_2\}$.*

PROOF. Let $P_0 = (U_0, U_1, \ldots, U_m)$ be the path from S_1 to S_2 with maximum nesting level, and let $\tau = \eta^0(P_0)$. Clearly $S_1 = U_0 \Delta_\tau U_1 \Delta_\tau \cdots \Delta_\tau U_m = S_2$, so

$$\eta(S_1, S_2) \geqslant \tau.$$

Suppose $\eta(S_1, S_2) > \tau$. Then there exists a path $P = (T_0, T_1, \ldots, T_n)$ such that

$$S_1 = T_0 \Delta_d T_1 \Delta_d \cdots \Delta_d T_n = S_2$$

where $d > \tau$. But then $\eta^0(P) = d > \tau$, contradicting the maximality of $\eta(P_0)$. □

Lemma 3 and Theorem 6 establish that the computation of $\eta(S_1, S_2)$ for each pair of statements in the program is just a *shortest path* problem with min replacing + as the operation used to compose costs along a path (Lemma 3) and max replacing min as the operation to compute the resulting cost at a vertex where two paths join (Theorem 6). Hence, Kleene's algorithm can be used to compute $\eta(S_1, S_2)$ for each pair of statements in time proportional to the cube of the number of statements [1].

The concept of depth of a dependence is useful because it permits partial vectorization. 重要

Definition. Consider a statement S that depends upon itself ($S \Delta^+ S$). The *parallelism index* of S, $\rho(S)$ is defined

$$\rho(S) = m - \eta(S, S)$$

where m is the number of loops containing S.

Observe that if $\rho(S) > 0$, then S may be executed in parallel in the innermost $\rho(S)$ loops surrounding it.

As an example, consider the multiple loop from Section 4.3.

```
          DO 100 i = 1, 100
            DO 90 j = 1, 100
              DO 30 k = 1, 100
S1              X(i, j + 1, k) = A(i, j, k) + 10
     30       CONTINUE
              DO 80 l = 1, 50
S2              A(i + 1, j, l) = X(i, j, l) + 5
     80       CONTINUE
     90     CONTINUE
    100   CONTINUE
```

In this loop $S_1 \; \Delta \; S_2$ and $S_2 \; \Delta \; S_1$; however, $\eta(S_1, S_2) = 2$, while $\eta(S_2, S_1) = 1$. From the definitions of η and ρ, we have $\eta(S_1, S_1) = \eta(S_2, S_2) = 1$ and $\rho(S_1) = \rho(S_2) = 2$. Thus both inner loops surrounding each statement may be run in vector. The translated program would be

```
      DO 100 i = 1, 100
        X(i, 2:101, 1:100) = A(i, 1:100, 1:100) + 10
        A(i + 1, 1:100, 1:50) = X(i, 1:100, 1:50) + 5
100   CONTINUE
```

This is the same result that was obtained in Section 4.3.

The depth of a dependence represents the number of loops that, if iterated sequentially, will guarantee that the dependence is satisfied. That is, a level one dependence will be preserved so long as the outer loop is iterated sequentially, regardless of what is done to inner loops or to statement order within the loop. In this context, the depth of a loop independent dependence is correct as infinity, since it is impossible to guarantee that a loop independent dependence is satisfied by any iteration of loops. Rather, relative statement order preserves those dependences, regardless of the iteration of the surrounding loops. Although there exist infinite level dependences, $\eta(S, S)$ can never be greater than the number of loops surrounding statement S. The reason is very simple; any path that has S as both start and end must contain a loop carried dependence, because loop independent dependences are always directed forward. Therefore, $\rho(S)$ is always nonnegative.

The next section presents a general procedure to find $\rho(S)$ for each S in a program and to generate FORTRAN 8x code that runs the innermost $\rho(S)$ loops in parallel.

5. GENERATION OF VECTOR CODE

In this section, we demonstrate how the test for dependence can be used to generate vector code. This material was briefly introduced in Section 4.4. This section generalizes the ideas presented there and discusses several techniques for improving the quality of the generated code.

5.1 The Augmented Dependence Graph

Earlier in the paper, we discussed the concept of a dependence graph, in which each statement was represented by a vertex and each dependence by a directed edge from the statement depended upon to the dependent statement (the edges indicate the direction in which control must flow). In the augmented dependence graph, we shall attach auxiliary information to each edge in the form of a label.

Definition. The *augmented dependence graph D* is an ordered pair (V, E) where V, the set of *vertices*, represents the statements in a program and E, the set of *edges*, represents interstatement dependences. Each edge $e \in E$ may be viewed as a quadruple $\langle S_1, S_2, t, k\rangle$, where S_1 and S_2 are two statements such that $S_1 \; \delta_k \; S_2$ and where t is the type of the dependence (true, anti-, output). The pair $\langle t, k\rangle$ is the *label* of the dependence edge.

Note that in this graph it is possible to have multiple edges linking a pair of statements.

Consider, for example,

```
          DO 100 j = 1, 100
            DO 90 i = 1, 100
S1            X(i, j) = F(A(i, j))
S2            B(i, j) = G(X(i − 1, j), X(i, j − 1))
     90     CONTINUE
    100   CONTINUE
```

Here $S_1\ \delta_1\ S_2$ because of input X(i − 1, j), and $S_1\ \delta_2\ S_2$ because of input X(i, j − 1). Although we will say that $\eta(S_1, S_2) = 2$, it will be useful to distinguish dependences such as these. Therefore multiple edges will be used in the augmented dependence graph.

5.2 Code Generation

Once the augmented dependence graph is constructed, we proceed to generate code for the program as follows:

(1) Find all the strongly connected regions in the dependence graph.
(2) Reduce the dependence graph to an acyclic graph by treating each strongly connected region as a single node, or π-block.
(3) Generate code for each π-block in an order consistent with the dependences. That is, by using a method similar to topological sort, first generate code for blocks that depend on no others, then for blocks that depend only on blocks for which code has already been generated, etc.

This is exactly the method we described in Section 4; however, this time we would like to take advantage of dependence depth to get more parallelism. In the method above, if a statement does not depend upon itself, we will run all the loops that contain it in parallel. But what do we do with a set of statements $S_1, S_2, \ldots, S_n$ that form a recurrence? Observe that, even though these statements all depend upon themselves, some may not depend upon themselves when we consider only inner loops. We would like to see what happens if we iterate one or more loops sequentially.

Let us introduce some terminology.

Definition. Given a dependence graph $D = (V, E)$, the corresponding *level-k dependence graph* D_k is defined as (V, E_k) where

$$E_k = \{\langle S_1, S_2, t, j\rangle \in E \mid j \geq k\}.$$

In other words, in D_k we consider only those edges at nesting level k or greater. In a sense, D represents the relation Δ, while D_k represents Δ_k.

Definition. Let $D = (V, E)$ be any dependence graph, and let S be a subset of V. The *restriction* of D to S, written $D \mid S$, is defined as the dependence graph $(S, E \mid S)$ where

$$E \mid S = \{\langle S_1, S_2, t, k\rangle \in E \mid S_1 \in S \text{ and } S_2 \in S\}.$$

We also need some terminology to define the π-block construction.

Definition. If $D_k = (V_k, E_k)$ is a level-k dependence graph, the corresponding level-k π-graph π_k is defined as

$$\pi_k = (V_k^\pi, E_k^\pi)$$

where $V_k^\pi = \{S \mid S \in V_k \text{ and } S \not\Delta_k^+ S\} \cup \{SC_1, SC_2, \ldots, SC_p\}$ where each SC_j, $1 \le j \le p$, is a maximal strongly connected region in D_k, and

$$E_k^\pi = \{\langle v_1, v_2\rangle \in V_k^\pi \times V_k^\pi \mid S_1 \in v_1, S_2 \in v_2, v_1 \neq v_2 \text{ and } S_1 \Delta_k S_2\}.$$

In other words, π_k is a directed acyclic graph derived from D_k by collapsing each strongly connected region to a single node. Edges in π_k are inherited from D_k in the natural way. Since π_k is acyclic, we can construct an ordering on the elements of V_k^π that is consistent with the partial ordering E_k^π; this is done using topological sort.

The code generation procedure can generate parallel code for all statements that are not part of a recurrence in D_k. However, it need not give up on a recurrence R in D_k; instead, it can call itself recursively with $D_{k+1} \mid R$ as input. That is, it attempts to break the recurrence by increasing the depth of dependence considered.

The procedure *codegen* (Figure 9), written in a Pascal-like specification language, is an encoding of this method. At the outermost level, this procedure would be called as follows:

$$codegen(V, 1, D)$$

where V is the entire set of statements in the original program and D is the dependence graph for the entire program.

We shall illustrate the operation of the code generation procedure by considering a contrived example.

```
          DO 30 I = 1, 100
S1          X(I) = Y(I) + 10
            DO 20 J = 1, 100
S2            B(J) = A(J, N)
              DO 10 K = 1, 50
S3              A(J + 1, K) = B(J) + C(J, K)
     10       CONTINUE
S4            Y(I + J) = A(J + 1, N)
     20     CONTINUE
     30   CONTINUE
```

Let us trace the actions of *codegen* on this example. The dependences are shown in Figure 10. At the top level, statements S_2, S_3, and S_4 are involved in a recurrence, so that code generation will be called recursively for these statements

```
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 $R_\pi$ from $R$ by reducing each $S_i$ to a single node and compute $D_\pi$, the dependence graph
  naturally induced on $R_\pi$ by $D$;
let $\{\pi_1, \pi_2, \ldots, \pi_m\}$ be the nodes of $R_\pi$ numbered in an order consistent with $D_\pi$ (use topological sort to
  do the numbering);

for $i \leftarrow 1$ to $m$ do
  if $\pi_i$ is strongly-connected
  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 $\pi_i$;
    codegen($\pi_i$, $k + 1$, $D_i$);
    generate the level-$k$ CONTINUE statement
  else
    generate a parallel statement for $\pi_i$ in $\rho(\pi_i) - k + 1$ dimensions, where $\rho(\pi_i)$ is the number of loops
      containing $\pi_i$
  fi
  od
end
```

Fig. 9. Parallel code generation routine.

one level down. Vector code will be generated for statement S_1 after the loop for the first three.

```
        DO 30 I = 1, 100
          code for S2, S3, S4
          generated at lower levels
     30 CONTINUE
S1      X(1:100) = Y(1:100) + 10
```

At the next level down, the output dependences of S_2, S_3, and S_4 on themselves, which occur because the array being assigned to does not have enough subscripts for all the surrounding loops, disappear. Also, the antidependence of S_3 on S_4, due to the possibility that A(J + 1, K) is the same as A(J + 1, N) on a successive iteration of the *outer* loop, is broken. This leaves the dependence graph shown in Figure 11. Statements S_2 and S_3 still form a recurrence, but code can now be generated for statement S_4.

```
        DO 30 I = 1, 100
          DO 20 J = 1, 100
            code for S2, S3
            generated at lower levels
     20   CONTINUE
S4        Y(I + 1:I + 100) = A(2:101, N)
     30 CONTINUE
S1      X(1:100) = Y(1:100) + 10
```

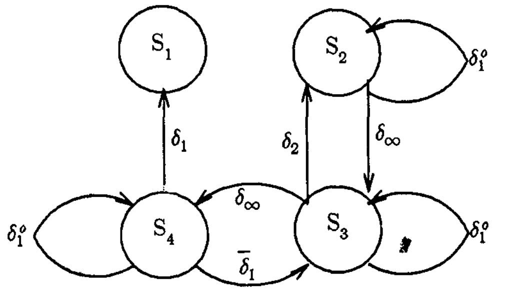
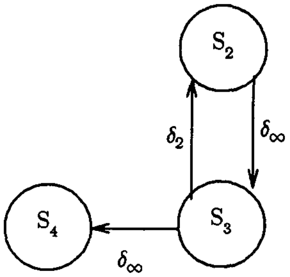


Fig. 10. Dependences for *codegen* example.

Fig. 11. Dependence graph at level 2.



The recurrence involving statements S_2 and S_3 will be broken at the next level down and the final code below will result.

```
         DO 30 I = 1, 100
           DO 20 J = 1, 100
S2           B(J) = A(J, N)
S3           A(J + 1, 1:100) = B(J) + C(J, 1:100)
    20     CONTINUE
S4         Y(I + 1:I + 100) = A(2:101, N)
    30   CONTINUE
S1       X(1:100) = Y(1:100) + 10
```

This example is pleasing because it generates vector statements at three different levels and because it illustrates, in the case of statement S_2, how parallel code generation can also serve as scalar code generation. This happens when we generate code that is parallel in 0 dimensions.

It should be obvious that the code generated by *codegen* for a statement S is vector in the maximal possible number of inner loops, since vector code is generated at the first level at which S does not depend upon itself. Thus S runs in vector in $\rho(S)$ loops. The time required to generate this code is given by the following theorem.

THEOREM 7. *If m is the number of edges in the dependence graph, n the number of vertices, and N the maximum nesting level of any loop in the program, then procedure codegen runs in time $O(mN + nN)$.*

PROOF. If we assume that actually generating a parallel statement takes constant time and ignore, for the moment, the time required in a recursive call to *codegen*, the time required for the first invocation of *codegen* is $O(m + n)$. To see this, observe that the computation consists of three components:

(1) identification of maximal strongly connected regions—this can be done by using Tarjan's depth-first search method in $O(m + n)$ time [31],
(2) topological sorting of the π-blocks to determine the order of code generation—this will require $O(m + n)$ time [16],
(3) the code generation **for**-loop, which takes at most $O(n)$ time, since we are ignoring the time spent in the recursive calls to *codegen*.

Note that both (2) and (3) make use of the fact that the number of π-blocks p is less than or equal to the number of statements n. Thus the total time spent in *codegen* at level 1 is $O(m + n)$. But we have actually shown that the time spent in any single invocation is linear in the number of vertices and edges in the subgraph presented to codegen.

Thus, if strongly connected region S_i has n_i vertices and m_i edges, the total time spent in calls to *codegen* at level 2 is

$$\sum_{i=1}^{p} O(m_i + n_i) = O\left(\sum_{i=1}^{p} m_i + \sum_{i=1}^{p} n_i\right) = O(m + n),$$

where p is the number of strongly-connected regions found at level 1.

Continuing in this manner, it is clear that the time spent in all the invocations of *codegen* at any level is $O(m + n)$. Since there are at most N levels, the total time is no worse than

$$O((m + n)N). \qquad \square$$

As a final observation, note that $\rho(S)$ for each statement in the program could be computed directly from the augmented dependence graph using Kleene's algorithm.

5.3 Increasing the Level of Parallelism

5.3.1 *Loop Interchange.* Although the procedure *codegen* generates parallel code for the maximal number of inner loops, it may be possible to generate vector code for an outer loop even when ρ(S) = 0. Consider the following example:

```
          DO 100 j = 1, 100
            DO 90 i = 1, 100
S:            X(i + 1, j) = F(X(i, j))
      90    CONTINUE
     100  CONTINUE
```

S depends upon itself in the innermost loop; hence $\rho(S) = 0$. However, if we *interchange* the two loops so that the j loop is innermost, $\rho(S)$ is increased to 1 and the j loop can be run in vector.

This approach is not without its pitfalls, however, as the next example illustrates.

```
          DO 100 j = 1, 100
            DO 90 i = 1, 100
              X(i + 1, j + 1) = F(X(i, j + 1), X(i + 2, j))
      90    CONTINUE
     100  CONTINUE
```

Once again, if the loops could be interchanged, the new innermost loop (on j) could be run in vector. Unfortunately, merely interchanging the loops would introduce a semantic difference. In the original version above, the values of X(i + 2, j) used on the right-hand side would be those computed on the previous iteration of the outermost loop. However, if we interchange loops, the values of X(i + 2, j) will be those that existed before entry to the outermost loop, since only the first i + 1 rows will have been computed by the time X(i + 2, j) is used. Therefore, we cannot interchange loops and preserve the semantics.

How can we decide when loop interchange is permissible and when it is not? In general, we can interchange loops if doing so preserves all dependences. To understand this, consider the diagram in Figure 12, which shows a two-dimensional array of nodes where each node represents a single parameterized instance of statement S in the loop structure below.

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

Thus S_{11} represents the execution of S when both i and j are 1, S_{12} when i = 1 and j = 2, S_{21} when i = 2 and j = 1, and so on. To preserve dependence, we must ensure that if $S_{i_1j_1}\ \Delta\ S_{i_2j_2}$ then $S_{i_1j_1}$ is executed before $S_{i_2j_2}$ in any modified loop structure. The diagram illustrates that if we wish to interchange loops, we must ensure that no dependence

$$S_{i_1j_1}\ \Delta\ S_{i_2j_2}$$

is such that $i_1 < i_2$ and $j_1 > j_2$. In Figure 12, if S_{22} depends on S_{13}, then interchanging loops (which can be visualized in the diagram either by transposing the matrix of nodes or by moving down the matrix first and then across) will destroy that dependence by causing S_{22} to be executed before S_{13}. The observation depicted in Figure 12 is due to Wolfe [36].

We now turn to the question of determining when an interchange-preventing dependence exists.

Definition. Suppose $S_1\ \Delta\ S_2$ in the loop below.

```
        DO 100 i = 1, N1
          DO 90 j = 1, N2
S1:         X(f(i, j)) = F(...)
S2:         A = G(X(g(i, j)))
   90     CONTINUE
  100   CONTINUE
```

The dependence is said to be *interchange preventing* if there exist i_1, i_2, j_1, j_2 such that $1 \leq i_1 < i_2 \leq N_1$(outer loop), $1 \leq j_2 < j_1 \leq N_2$(inner loop), and $f(i_1, j_1) = g(i_2, j_2)$.

This is simply a straightforward encoding of the condition expressed graphically in Figure 12. This can be generalized to multiple loops following the model of Section 4.3.

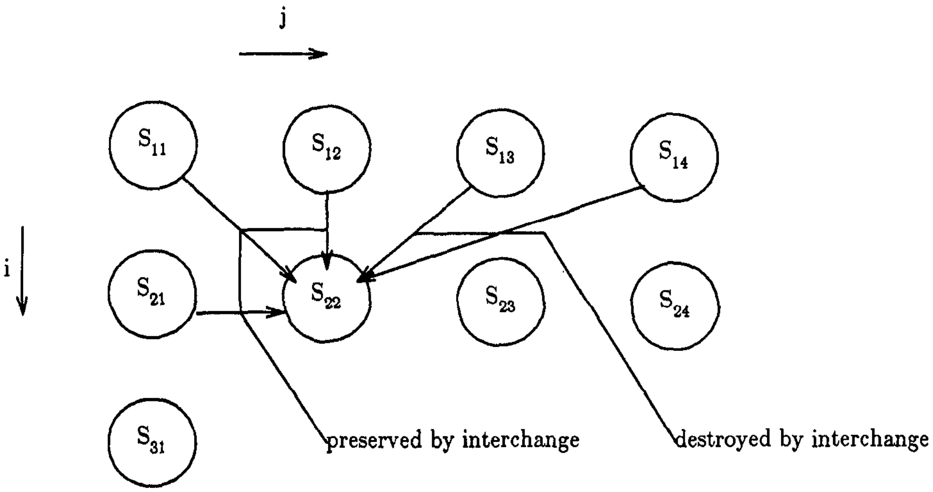


Fig. 12. Dependence patterns in a loop.

Definition. Suppose S_1 is contained in n_1 loops with loop induction variables $(x_1, x_2, \ldots, x_{n_1})$ and S_2 is contained in n_2 loops with induction variables $(x_1, x_2, \ldots, x_{n_2})$, where S_1 and S_2 are of the form

$$\begin{aligned} S_1&: \; \mathrm{X}(f(x_1, \ldots, x_{n_1})) = \mathbf{F}(\ldots) \\ S_2&: \; \mathrm{A} = \mathbf{G}(\mathrm{X}(g(x_1, \ldots, x_{n_2}))). \end{aligned}$$

There exists an *interchange preventing dependence* of S_2 on S_1 with respect to carrier loop k and source loop $k+1$, written $S_1 \, \gamma^k_{k+1} \, S_2$, if there exist

$$\begin{array}{lll} (i_1, i_2, \ldots, i_{k-1}), & 1 \le i_q \le N_q, & \forall q \text{ s.t. } 1 \le q \text{ and } q < k \\ (j_{k+2}, j_{k+3}, \ldots, j_{n_1}), & 1 \le j_q \le N_q, & \forall q \text{ s.t. } k+2 \le q \text{ and } q \le n_1 \\ (l_{k+2}, l_{k+3}, \ldots, l_{n_2}), & 1 \le l_q \le N_q, & \forall q \text{ s.t. } k+2 \le q \text{ and } q \le n_2 \end{array}$$

and integers $\alpha_1, \alpha_2, \beta_1, \beta_2$ where

$$\begin{aligned} 1 \le \alpha_1 < \alpha_2 \le N_k \\ 1 \le \beta_2 < \beta_1 \le N_{k+1} \end{aligned}$$

such that

$$f(i_1, i_2, \ldots, i_{k-1}, \alpha_1, \beta_1, j_{k+2}, \ldots, j_{n_1}) = g(i_1, i_2, \ldots, i_{k-1}, \alpha_2, \beta_2, l_{k+2}, \ldots, l_{n_2}).$$

A useful observation about γ^k_{k+1} is given by the following lemma.

LEMMA 4. $S_1 \gamma^k_{k+1} \, S_2 \Rightarrow S_1 \, \delta_k \, S_2$.

PROOF. Immediate from the definition of δ_k (Section 4.3) with

$$\zeta_1 = \alpha_1, \quad \zeta_2 = \alpha_2, \quad j_{k+1} = \beta_1, \quad l_{k+1} = \beta_2. \qquad \square$$

Lemma 4 tells us that we need not test for $S_1 \, \gamma^k_{k+1} S_2$ unless $S_1 \, \delta_k \, S_2$, a useful prescreen since the test for γ^k_{k+1}, which we shall introduce next, represents a significant additional computation.

THEOREM 8. *Suppose S_1 and S_2 are of the form given above and are contained in at least $k + 1$ common loops. If*

$$f(x_1, \ldots, x_{n_1}) = a_0 + \sum_{i=1}^{n_1} a_i x_i$$

and

$$g(x_1, \ldots, x_{n_2}) = b_0 + \sum_{i=1}^{n_2} b_i x_i$$

then $S_1 \gamma_{k+1}^{k} S_2$ only if

(*a*) *gcd test*:

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

(*b*) *γ-inequality*:

$$\begin{aligned}
(a_{k+1} - b_k) &- \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) \\
&- (a_k^- + b_k)^+(N_k - 2) - (a_{k+1} - b_{k+1}^+)^-(N_{k+1} - 2) \\
&- \sum_{i=k+2}^{n_2} a_i^-(M_i - 1) - \sum_{i=k+2}^{n_2} b_i^+(N_i - 1) \\
&\le \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i \\
&\le (a_{k+1} - b_k) + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1) \\
&+ (a_k^+ - b_k)^+(N_k - 2) + (a_{k+1} + b_{k+1}^-)^+(N_{k+1} - 2) \\
&+ \sum_{i=k+1}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1).
\end{aligned}$$

The proof of this theorem (which resembles the proof of Theorem 4) is also contained in the Appendix.

Since this test and Banerjee's inequality are similar, testing an edge to determine whether it is interchange preventing can easily accomplished at the time of normal dependence testing. Furthermore, this method can be generalized to allow successive interchanges betweeen loops, or to test for interchanges between arbitrary (not necessarily consecutive) loops [4]. Note that the test need not be applied until a dependence with respect to carrier k has been discovered.

Although the property of interchange prevention determines when two loops may be *safely* interchanged, it does not determine when they may be *profitably* interchanged. From the definition of the parallelism index of a statement, it should be obvious that loop interchange is profitable whenever it moves a recurrence *outward*, thereby decreasing $\eta(S, S)$ and increasing the parallelism index. In order to move a recurrence outward, it is important that dependence edges carried by the outer loop not move *inward* with the interchange. Edges that do move inward with an interchange are known as *interchange sensitive* edges.

A glance at Figure 12 should help clarify this idea. The dependence of S_{22} on S_{11} is carried by the level 1 loop. If the two loops are interchanged, this dependence is *still* carried by the level 1 loop; hence it is insensitive to the interchange. On the other hand, the level 1 dependence of S_{22} on S_{12} becomes a level 2 dependence after loop interchange. This dependence is interchange sensitive.

Detecting interchange sensitive edges is easy when dependence testing is performed. Since loop interchange corresponds to a switch of two terms in the dependence tests, Banerjee's inequality and the gcd test can be easily modified to detect interchange sensitive edges. Loop interchange will then be profitable whenever a recurrence at an inner level moves out with the interchange, while the interchange sensitive edges do not form a new recurrence.

A special case when interchange sensitive edges are guaranteed not to exist is given by the following lemma:

LEMMA 5. *If $S_1 \not\delta_k S_2$, and loops k and $k + 1$ are interchanged, then $S_1 \not\delta_{k+1} S_2$; in other words, S_2 cannot depend on S_1 with respect to the new carrier $k + 1$.*

PROOF. Interchanging two loops leaves the subscript functions f, g unchanged; however, we must remember that the interchange makes the kth parameter position correspond to the new position for carrier loop $k + 1$. Suppose that $S_1 \delta_{k+1} S_2$ after interchange. Then there exist

$$(x_1, x_2, \ldots, x_k, \mu_1, y_{k+2}, \ldots, y_{n_1})$$
$$(x_1, x_2, \ldots, x_k, \mu_2, z_{k+2}, \ldots, z_{n_2})$$

such that

$$\begin{array}{ll}
1 \le x_i \le N_i, & 1 \le i \le k \\
1 \le \mu_1 < \mu_2 \le N_{k+1} & \\
1 \le y_i \le N_i, & k + 2 \le i \le n_1 \\
1 \le z_i \le N_i, & k + 2 \le i \le n_2
\end{array}$$

and

$$f(x_1, \ldots, x_{k-1}, \mu_1, x_k, y_{k+2}, \ldots, y_{n_1}) = g(x_1, \ldots, x_{k-1}, \mu_2, x_k, z_{k+2}, \ldots, z_{n_2}).$$

But this trivially implies $S_1 \delta_k S_2$ before loop interchange. □

Lemma 5 says that a loop must carry a dependence in order to carry an interchange sensitive dependence. Thus, loop interchange may be profitable when an inner level carries a recurrence and an outer level carries no dependences. These ideas are more fully developed elsewhere [4].

5.3.2 *Recurrence Breaking.* Some recurrences that one finds have an essential dependence link that represents an output dependence or antidependence. These "pseudo-dependences," as we called them earlier, may often be removed on technical grounds by array expansion [19] or by clever renaming. The simplest case is the single-statement recurrence.

```
      DO 100 i = 1, 100
        X(i − 1) = F(X(i))
100   CONTINUE
```

This statement has an antidependence upon itself. In particular, the input on the first iteration is the same as the output on the second iteration. If vector execution meant running the various iterations concurrently with loads and stores intermixed, we might accidentally store X(2) before loading X(1), particularly if iteration 2 were executed before iteration 1. However the semantics of FORTRAN 8x prevent this problem from arising since they specify that the right-hand side is loaded before any stores occur. Thus we have the following result, which is stated without proof.

THEOREM 9. *Any single-statement recurrence based on antidependence may be ignored.*

The situation with output dependences at first appears more complicated. Consider the following nested loops.

```
      DO 100 i = 1, 100
        DO 90 j = 1, 100
          X(i + j) = F(A(i, j))
 90     CONTINUE
100   CONTINUE
```

In this example different values will be stored into a particular location of X at different times. For example, stores into X(4) occur when

$$\begin{aligned} i &= 1 \text{ and } j = 3\\ i &= 2 \text{ and } j = 2\\ i &= 3 \text{ and } j = 1 \end{aligned}$$

in that order. Here we are saved by the store semantics of FORTRAN 8x. Normally we would attempt to translate the above example as follows:

```
      IDENTIFY/1:100, 1:100/XX(j, i) = X(i + j)
      XX(*, *) = F(A(*, *))
```

The store semantics of this statement specify that the stores will actually occur into XX in column-major order. Thus

$$\begin{aligned} XX(3, 1) &= F(A(3, 1))\\ XX(2, 2) &= F(A(2, 2))\\ XX(1, 3) &= F(A(1, 3)) \end{aligned}$$

will occur in the correct order, and after exit from the outer loop X will contain the right values. In general, all single statement output recurrences can be broken if the parallel code generator is careful in the way in which it uses IDENTIFY statements.

Breaking output dependences in this manner is not so intuitively a correct transformation as breaking antidependences. For instance, assume that there is a statement within the loops of the above example that uses X(i + j):

```
      DO 100 i = 1, 100
        DO 90 j = 1, 100
          X(i + j) = F(A(j, i))
          Y(j, i) = G(X(i + j))
 90     CONTINUE
100   CONTINUE
```

It would be incorrect to generate the following parallel code for this example

```
      IDENTIFY/1:100, 1:100/XX(j, i) = X(i + j)
      XX(*, *) = F(A(*, *))
      Y(*, *) = G(XX(*, *))
```

because Y(3, 1), Y(2, 2), and Y(1, 3) all receive the same value, G(F(A(1, 3))), whereas in the original example they receive intermediate values in the computation. It thus appears that any single statement output recurrence that has a true dependence upon it cannot be ignored.

But appearances in this case are deceiving. Note that when a later statement uses an intermediate value of the computation, which is stored over on another iteration, the recurrence cannot possibly be single statement. The reason is that the store on top of the value used creates an antidependence between the first statement and the last statement. This dependence and the true dependence guarantee that at least two statements will be tied up in the recurrence. Thus, if the code generator is careful in its generation of IDENTIFYs, the following theorem, which is stated without proof, is true.

THEOREM 10. *Any single statement recurrence based on output dependence may be ignored.*

The previous example is instructive for other transformations, also. Note that as written, each statement can be correctly vectorized in the inner loops; that is,

```
      DO 100 i = 1, 100
        X(i + 1:i + 100) = F(A(1:100, i))
        Y(1:100, i) = G(X(i + 1:i + 100))
100   CONTINUE
```

However, we can enhance the parallelism present in this example. The main inhibition to vectorization of the i loop is the antidependence of the first statement on the second; that is, the fact that the first statement stores on top of values that are used earlier by the second statement. If we store the results of the first statement in a temporary, as in

```
      DO 100 i = 1, 100
        DO 90 j = 1, 100
          T(j, i) = F(A(j, i))
          Y(j, i) = G(T(j, i))
          X(i + j) = T(j, i)
 90     CONTINUE
100   CONTINUE
```

then we can break the antidependence and the recurrence. This loop may be translated directly to the vector form

```
IDENTIFY/1:100, 1:100/XX(j, i) = X(i + j)
T(*, *) = F(A(*, *))
Y(*, *) = G(T(*, *))
XX(*, *) = T(*, *)
```

This procedure, also known as the "node splitting" method [19], should be considered cautiously in light of the increased storage required for temporary arrays.

In general, this technique can be used to break any recurrence that contains an antidependence as an essential link. Consider the example below.

```
              DO 10 I = 1, N
S1              A(I) = G(X(I + 1), X(I))
   δ̄                                          δ.
S2              X(I + 1) = F(B(I))
           10 CONTINUE
```

Here S_1 depends on S_2 in the true sense because it uses the output of S_2 on the next iteration. However, we cannot merely interchange S_1 with S_2 and vectorize, because S_1 would then get wrong values for its first input parameter X(I + 1). Thus we have a recurrence involving antidependence.

The obvious remedy is to introduce a new name for the first input parameter to S_1 as below.

```
           DO 10 I = 1, N
             T(I) = X(I + 1)
   δ̄                                  δ
S1           A(I) = G(T(I), X(I))
                                      δ
S2           X(I + 1) = F(B(I))
        10 CONTINUE
```

Now S_2 can be slipped in between S_0 and S_1 and all three statements can be vectorized.

```
T(*) = X(* + 1)
X(* + 1) = F(B(*))
A(*) = G(T(*), X(*))
```

In general, any recurrence involving an essential antidependence link can be broken by introducing new array temporaries. The efficiency gained by this transformation must be weighed against the additional storage costs.

5.3.3 *Thresholds.* On some vector machines, short vector operations are efficient enough to be cost effective, even with vector lengths as small as 4 or 5. On the Cray-1, for example, the crossover point for choosing vector over scalar processing is between 2 and 4 elements [30]. For such machines, we may wish to vectorize recurrences that have a long period or "lag." Consider an example.

```
      DO 100 i = 1, 100
        X(i + 5) = F(X(i))
  100 CONTINUE
```

If this loop is broken up into the following double loop,

```
      DO 100 i = 1, 20
        DO 90 j = 1, 5
          X(i * 5 + j) = F(X(i * 5 + j - 5))
 90   CONTINUE
100   CONTINUE
```

the inner loop now contains no recurrence and can be directly translated to vector form.

```
      DO 100 i = 1, 20
        X(5 * i: 5 * i + 4) = F(X(5 * 1 - 5: 5 * i - 1))
100   CONTINUE
```

Let us develop this idea more formally.

Definition. Suppose $S_1 \, \delta_k \, S_2$. The *direct threshold* of this dependence, written $\tau_k^0(S_1, S_2)$, is one less than the minimum value of N_k (the loop upper bound for the kth loop) at which Banerjee's inequality still holds. Alternatively, $\tau_k^0(S_1, S_2)$ is the maximum value of N_k for which Banerjee's inequality does not hold.

$$\tau_k^0(S_1, S_2) = \max\{n, 0 \le n \le N_k \mid \text{if } N_k \text{ is set to } n,\ S_1 \not\delta_k S_2\}.$$

The threshold, as defined above, indicates the number of times the loop may be executed without creating the dependence. This information can be useful in two ways:

(1) It may be the case that *every* dependence has the same threshold as the initial dependence. In the previous example, every iteration of the statement (other than the first four) uses values that were computed five iterations back. When every dependence between two statements has the same, exact threshold, the threshold is known as a *constant threshold,* and an inner loop that runs up to the threshold will carry no dependences.

(2) It may also be the case that every dependence will be satisfied by breaking the loop across a threshold. For instance, the threshold in

```
      DO 100 I = 1, 100
        A(I) = F(A(101 - I))
100   CONTINUE
```

is 50, since no dependence arises until the 51st iteration. Note that in this case, *every* dependence crosses the 51st iteration. Thus, the statement can be safely executed as

```
100   DO 100 I = 1, 2
        A(50 * I - 49:50 * I) = F(150 - 50 * I:101 - 50 * I)
100   CONTINUE
```

This type of threshold is known as a *crossing threshold.*

Determining whether a threshold is a constant threshold, a crossing threshold, or neither can be easily accomplished using Banerjee's inequality. The actual computation, which is too tedious to be detailed here, is precise; that is, it exactly determines which thresholds are constant and which are crossing.

Crossing thresholds impose very strict requirements on any vector statements that attempt to satisfy them. Since they specify an exact point that *must* be crossed by a vector analog, only certain vector sizes can be used. For instance, the crossing example above can be run in vector sizes of 5 or 10, since these exactly divide 50, but it cannot be correctly run with a vector size of 8. As a result, crossing thresholds are not easily extended to multiple statements.

Constant thresholds, on the other hand, represent the *minimum* threshold of a dependence—any vector size smaller than a constant threshold will satisfy the dependence (note that the first example can be correctly run with a vector size of 2, 3, 4, or 5). As a result, constant thresholds can be extended to multiple statements and to indirect dependences.

To extend the concept of threshold to indirect dependences, we need some way to compute a *composite* threshold. To put things in perspective, the threshold may be thought of as the minimum number of iterations between the definition and the use of the variable involved in the dependence. Since an indirect dependence ceases to exist whenever one of the direct dependences in the path ceases to exist, the threshold of an indirect dependence is simply the *maximum* of the direct thresholds in the path. (Note that the first dependence to disappear is the one with the largest threshold.) There may be multiple paths between two statements, however. Since the dependence will disappear only when *all* the indirect dependences disappear, the threshold of a *composite* dependence is the *minimum* of the thresholds of the individual paths. These observations are summarized in the following definitions.

Definition. If $P = (T_0, T_1, \ldots, T_n)$ is a sequence of statements such that

$$T_0 \; \Delta_d \; T_1 \; \Delta_d \; \cdots \; \Delta_d \; T_n$$

then the *direct threshold* of the dependence path P at depth d, written $\tau_d^0(P)$, is given by

$$\tau_d^0(P) = \max_{0 \le i \le n} \{\tau_d^0(T_i, T_{i+1})\}.$$

Definition. If $S_1 \; \Delta_d^+ \; S_2$, then the *depth d threshold* of the dependence, written $\tau_d(S_1, S_2)$ is given by

$$\tau_d(S_1, S_2) = \min\{\tau_d^0(P) \mid P \text{ is a dependence path from } S_1 \text{ to } S_2\}.$$

Note that we must distinguish a direct threshold τ^0 and the more general indirect threshold τ, because there may be a dependence path from S_1 to S_2 that has a smaller cumulative threshold than the direct threshold of the dependence of S_2 on S_1. In other words,

$$\tau_k(S_1, S_2) \le \tau_k^0(S_1, S_2)$$

and it is possible for the inequality to be strict.

Now suppose the translator has an input parameter θ, which specifies the minimum number of elements for which vector operations should be assumed profitable. If statement S depends upon itself, that is, $S \; \Delta^+ \; S$, and $\rho(S) = m$, the level of parallelism can be profitably increased through the use of thresholding if

$$\tau_{n-m+1}(S, S) \geq \theta$$

where n is the number of loops containing S. In general, thresholding may be desirable only if $\rho(S) = 0$.

In order to properly implement thresholding, the augmented dependence graph will need to associate a threshold with each directed edge. Since each such edge represents a single dependence $S_1\ \delta_k\ S_2$, the associated threshold will be $\tau_k^0(S_1, S_2)$, the direct threshold.

5.4 Unknown Constants

The dependence test described in Section 4 is fine so long as all the coefficients $(a_0, a_1, \ldots, a_{n_1}, b_0, \ldots, b_{n_2})$ are known constants. However, in real programs we often find unknown variables, possibly subroutine parameters, inside subscripts. For example, consider

```
          DO 100 i = 1, 100
S1          X(i, K) = F(A(i, K))
            DO 50 J = 1, 100
S2            A(i, j) = G(X(i, j + K))
       50 CONTINUE
      100 CONTINUE
```

The test described in Section 4 has no provision for handling unknown constants like K, so the translator must assume $S_1\ \Delta\ S_2$, even though no dependence exists. Can we revise the test to discover that S_1 does not depend on S_2 and thus avoid the presumption that S_1 and S_2 form a recurrence?

One obvious technique would be to evaluate Banerjee's inequality symbolically as far as possible. In the example above, this approach results in the elimination of K from the inequality since

$$b_0 - a_0 = K - K = 0.$$

Symbolic evaluation was used in the vectorizing compiler for the Texas Instruments Advanced Scientific Computer [9].

The method fails, however, if the two subscripts contain terms aK and bK, where a and b are constants and $a \neq b$. In many cases we may be able to establish rough upper and lower bounds on the value of K by examining the program logic. For example, $K \leq 0$ might be excluded by a specific text and branch on entry to the subroutine. We can make use of such bounds on K by treating K as an induction variable for an outer loop containing both S_1 and S_2. Then the dependence test of Section 4 could be routinely applied.

Using this technique, we can develop more precise dependence information than would be possible if we blindly presume dependence whenever unknown variables appear.

6. CONDITIONAL STATEMENTS

The theory of vector translation presented so far is based on an idea of data dependence. However, other types of dependence can occur within a program. In particular, conditional statements and branches introduce *control dependences*. If loops containing such statements are to be vectorized using the previous theory, then control dependences must be explicitly represented in the dependence graph.

One method for dealing with control dependences is to convert all statements under the control of a branch into *conditional assignments* whose controlling conditions are stated explicitly in the statements themselves. Once this is done, the condition controlling the statement can be viewed as another input expression. The following loop is an example:

```
      DO 100 I = 1, N
        IF (A(I) .LE. 0) GOTO 100
          A(I + 1) = B(I) + 3
100   CONTINUE
```

It is difficult to determine whether the code can be vectorized as is. However, if it is converted to an equivalent form:

```
      DO 100 I = 1, N
        BR1 = A(I) .LE. 0
        IF (.NOT. BR1) A(I + 1) = B(I) + 3
100   CONTINUE
```

the data dependences form an obvious recurrence. This example cannot be vectorized because of the recurrence; however, the following slightly different example does not contain a recurrence.

```
      DO 100 I = 1, N
        IF (A(I) .LE. 0) GOTO 100
        A(I) = B(I) + 3
100   CONTINUE
```

Converting all the control dependences into data dependences yields

```
      DO 100 I = 1, N
        BR1 = A(I) .LE. 0
        IF (.NOT. BR1) A(I) = B(I) + 3
100   CONTINUE
```

If the scalar variable BR1 in the above example is transformed into an array:

```
      DO 100 I = 1, N
        BR1(I) = A(I) .LE. 0
        IF (.NOT. BR1(I)) A(I) = B(I) + 3
100   CONTINUE
```

the loop can be transformed in a straightforward way to two vector statements.

```
      BR1(1:N) = A(1:N) .LE. 0
      WHERE (.NOT. BR1(1:N)) A(1:N) = B(1:N) + 3
```

The process of expanding scalars into arrays for vectorization is known as *scalar expansion* and is described in detail by Wolfe [36]. The process of converting control dependences into data dependences is known as *IF conversion* [3].

IF conversion is accomplished by converting all statements under the control of an IF or a branch into conditional statements. These conditional statements can then be translated, where possible, into conditional vector statements by viewing the condition as just another input to the statement. As such, all of the methods of dependence analysis described earlier apply to these statements, and *conditional dependence* as defined by Kuck et al. [19] becomes a special case of true dependence.

IF conversion is not a trivial task, as the following example illustrates:

```
             DO 100 I = 1, 100
               IF (A(I).GT.10) GO TO 60
S1               A(I) = A(I) + 10
                 IF (B(I).GT.10) GO TO 80
S2                 B(I) = B(I) + 10
S3      60       A(I) = B(I) + A(I)
S4      80     B(I) = A(I) + 5
       100   CONTINUE
```

Here there are two conditions that we shall call c_1 and c_2, where

$$c_1 = \text{A(I).GT. 10}$$
$$c_2 = \text{B(I).GT. 10}$$

Statements in the loop are controlled by conditions as follows.

Statement	*Controlling Condition*
S_1	$\neg c_1$
S_2	$\neg c_1 \wedge \neg c_2$
S_3	$c_1 \vee (\neg c_1 \wedge \neg c_2)$
S_4	$c_1 \vee (\neg c_1 \wedge c_2) \vee (\neg c_1 \wedge \neg c_2)$

The translator must be able to recognize identities and simplify logical expressions if it is to prevent the proliferation of long expressions involving conditions like c_1 and c_2. For example, it should surely recognize that the condition controlling S_4 above is always true, and hence S_4 is not under the control of any IF statement. The Rice translator incorporates a version of the Quine-McCluskey prime implicant simplifier algorithm [23, 29] to simplify such expressions. As a result, the IF conversion module in the translator converts the example loop above to 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)    B(I) = B(I) + 10
S3             IF (BR1 .OR. .NOT. BR2)           A(I) = B(I) + A(I)
S4             B(I) = A(I) + 5
       100   CONTINUE
```

Note that the condition guarding statement S_3 has also been slightly simplified. When we first ran this example, we briefly thought there was an error in the simplifier; then we realized that the new version of S_3 is indeed correct.

If dependence analysis, properly extended to handle conditional assignments, is applied to this loop, it will report that all statements in the loop can be converted to vector assignments or conditional vector assignments. This will not always be the case, however, since an IF statement can be part of a recurrence, as the very first example in this section demonstrates.

IF conversion, as implemented in the Rice translator, is an extremely powerful process that can convert any combination of branches into the equivalent sequence of conditional assignments. This transformation is treated in more detail elsewhere [3].

IF conversion differs significantly from the way that IF statements are handled in PARAFRASE. In that system, IF statements are classified using dependence analysis, and special techniques are selected based upon that classification [20, 32]. In particular, "mode vectors" are used to control parallel execution where it is determined that parallel execution is possible; these vectors are sometimes computed by fast Boolean recurrence solvers. In the Rice system, these techniques would be handled as special cases of recurrence-breaking transformations.

7. STATUS OF THE IMPLEMENTATION

As we said in the introduction, we began with the PARAFRASE compiler, developed by Kuck's group at the University of Illinois. This provided us with an excellent starting point and a convenient vehicle for familiarizing ourselves with Kuck's techniques. However, we soon discovered that the early version of PARAFRASE with which we were working was simply too inefficient and unreliable to support the new techniques we wanted to try. So we abandoned this effort and began work on an entirely new translator, which we called PFC. We were able to bring up the initial version of PFC in roughly three months. Since that time it has been extensively enhanced to include most of the transformations discussed in this paper, including loop interchange, thresholds, recurrence breaking, and IF conversion.

The current version of PFC runs in a three-megabyte machine under VM/CMS and consists of roughly 25,000 lines of PL/I. However, this figure is somewhat misleading because we make extensive use of the preprocessor to implement data abstraction. In particular, a high-level language for manipulating linked lists (known as *Polylist*) is implemented entirely in macros.

The performance of the translator has been a pleasant surprise. Not only does it do an excellent job of uncovering parallelism, but it also is extremely fast in doing so. For example, a composite case of kernels from Los Alamos that we use to benchmark the translator requires roughly a minute of CPU time to vectorize, even with string and array bounds checking enabled. By comparison, our last version of PARAFRASE required roughly 10 minutes of CPU time to vectorize the same program.

APPENDIX. PROOF OF THEOREMS 4 AND 8

An important property of the positive and negative parts of a real number is given by the following lemma.

LEMMA 1. *Let t, s, and z denote real numbers. If* $0 \le z \le s$ *then*

$$-t^-s \le tz \le t^+s.$$

Furthermore, there exist values z_1, z_2, $0 \le z_1, z_2 \le s$ such that

$$tz_1 = -t^- s$$
$$tz_2 = t^+ s.$$

PROOF. (a) If $t \ge 0$, then $0 \le tz \le ts$; but $t = t^+$ and $t^- = 0$, so $-t^- s \le tz \le ts$, $tz_1 = -t^- s$ when $z_1 = 0$ and $tz_2 = t^+ s$ when $z_2 = s$.

(b) If $t < 0$, then $ts \le tz \le 0$; but $t = -t^-$, $t^+ = 0$, so $-t^- s \le tz \le t^+ s$, $tz_1 = -t^- s$ when $z_1 = s$ and $tz_2 = t^+ s$ when $z_2 = 0$. □

THEOREM 4. *If $f(x_1, \ldots, x_{n_1}) = a_0 + \sum_{i=1}^{n_1} a_i x_i$ and $g(x_1, \ldots, x_{n_2}) = b_0 + \sum_{i=1}^{n_2} b_i x_i$ and S_1 and S_2 are of the form*

$$S_1\colon \mathbf{X}(f(x_1, \ldots, x_{n_1})) = \mathbf{F}(\ldots)$$
$$S_2\colon \mathbf{A} = \mathbf{G}(\mathbf{X}(g(x_1, \ldots, x_{n_2})))$$

and are contained in n common loops, $n \ge k$, and the upper bounds of the loops surrounding S_1 are M_i and the upper bounds of the loops surrounding S_2 are N_i ($M_i = N_i$ for $i \le n$), then $S_1\ \delta_k\ S_2$ only if

(*a*) *gcd test*:

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

(*b*) *Banerjee inequality*:

$$\begin{aligned}
&-b_k - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) - (a_k^- + b_k)^+(N_k - 2)\\
&- \sum_{i=k+1}^{n_1} a_i^-(M_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1)\\
&\le \sum_{i=0}^{n_2} b_i - \sum_{i=0}^{n_1} a_i\\
&\le -b_k + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1) + (a_k^+ - b_k)^+(N_k - 2)\\
&+ \sum_{i=k+1}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1).
\end{aligned}$$

PROOF. Consider the equation

$$h(x_1, \ldots, x_{n_1}, y_1, \ldots, y_{n_2}) = f(x_1, \ldots, x_{n_1}) - g(y_1, \ldots, y_{n_2}) = 0.$$

This is equivalent to

$$\sum_{i=1}^{n_1} a_i x_i - \sum_{i=1}^{n_2} b_i y_i = b_0 - a_0.$$

By the definition of δ_k, $S_1\ \delta_k\ S_2$ if and only if there exist

$$(r_i, \ldots, r_{k-1})\ (s_k, \ldots, s_{n_1})\ (t_k, \ldots, t_{n_2})$$

such that

$$\sum_{i=1}^{k-1} (a_i - b_i)r_i + \sum_{i=k}^{n_1} a_i s_i + \sum_{i=k}^{n_2} b_i t_i = b_0 - a_0.$$

A standard result from the theory of Diophantine equations [14] tells us this has a solution if and only if

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

Thus $S_1 \; \delta_k \; S_2 \Rightarrow (a)$.

To show (b), assume there exists a real solution of

$$h(x_1, \ldots, x_{n_1}, y_1, \ldots, y_{n_2}) = 0$$

in the region R given by

$$\begin{array}{cc}
\multicolumn{2}{c}{1 \le x_1 = y_1 \le N_1} \\
\multicolumn{2}{c}{1 \le x_2 = y_2 \le N_2} \\
\multicolumn{2}{c}{\vdots} \\
\multicolumn{2}{c}{1 \le x_{k-1} = y_{k-1} \le N_{k-1}} \\
\multicolumn{2}{c}{1 \le x_k < y_k \le N_k} \\
1 \le x_{k+1} \le M_{k+1}, & 1 \le y_{k+1} \le N_{k+1} \\
1 \le x_{k+2} \le M_{k+2} & 1 \le y_{k+2} \le N_{k+2} \\
\multicolumn{2}{c}{\vdots} \\
1 \le x_{n_1} \le M_{n_1} & 1 \le y_{n_2} \le N_{n_2}.
\end{array}$$

From the intermediate value theorem of calculus, we have

$$\min_R h \le 0 \le \max_R h.$$

Let us now derive formulas for both the $\min_R h$ and $\max_R h$:

$$\begin{aligned} h &= a_0 - b_0 + \sum_{i=1}^{n_1} a_i x_i - \sum_{i=1}^{n_2} b_i y_i \\ &= a_0 - b_0 + \sum_{i=1}^{k-1} (a_i - b_i) y_i + a_k x_k - b_k y_k + \sum_{i=k+1}^{n_1} a_i x_i - \sum_{i=k+1}^{n_2} b_i y_i. \end{aligned}$$

We will maximize and minimize each of the variable terms separately. Let

$$t_i = x_i - 1 \quad \text{and} \quad s_i = y_i - 1.$$

(1) Since $x_i = y_i$ for $1 \le i < k$, then

$$\sum_{i=1}^{k-1} (a_i x_i - b_i y_i) = \sum_{i=1}^{k-1} (a_i - b_i) x_i = \sum_{i=1}^{k-1} (a_i - b_i) + \sum_{i=1}^{k-1} (a_i - b_i) t_i$$

Furthermore, $0 \le t_i \le N_i - 1$, which means that

$$-(a_i - b_i)^-(N_i - 1) \le (a_i - b_i)t_i \le (a_i - b_i)^+(N_i - 1)$$

and there exist values of t_i that give equality. Hence

$$\begin{aligned}
&\sum_{i=1}^{k-1} (a_i - b_i) - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) \\
&\le \sum_{i=1}^{k-1} (a_i x_i - b_i y_i) \\
&\le \sum_{i=1}^{k-1} (a_i - b_i) + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1).
\end{aligned}$$

(2) Since $1 \le x_k \le (y_k - 1) \le (N_k - 1)$,

$$0 \le t_k \le s_k - 1 \le N_k - 2$$

so $a_k x_k = a_k + a_k t_k$ and $-a_k^-(s_k - 1) \le a_k t_k \le a_k^+(s_k - 1)$ with equality for legal values of t_k. Hence

$$\begin{aligned}
&a_k - b_k - a_k^-(s_k - 1) - b_k s_k \\
&\quad \le a_k x_k - b_k y_k \\
&\quad \le a_k - b_k + a_k^+(s_k - 1) - b_k s_k.
\end{aligned}$$

But since $0 \le (s_k - 1) \le N_i - 2$, we get (using Lemma 1 twice)

$$(-a_k^- - b_k)^-(N_k - 2) = -(a_k^- + b_k)^+(N_k - 2) \le (-a_k^- - b_k)(s_k - 1)$$

and

$$(a_k^+ - b_k)(s_k - 1) \le (a_k^+ - b_k)^+(N_k - 2).$$

Putting these inequalities together we get

$$\begin{aligned}
&a_k - 2b_k - (a_k^- + b_k)^+(N_k - 2) \\
&\quad \le a_k x_k - b_k y_k \\
&\quad \le a_k - 2b_k + (a_k^+ - b_k)^+(N_k - 2)
\end{aligned}$$

with equality for legal values of x_k and y_k.

(3) For any i, $k < i \le n_1$,

$$a_i x_i = a_i + a_i s_i.$$

Since $0 \le s_i \le M_i - 1$,

$$-a_i^-(M_i - 1) \le a_i s_i \le a_i^+(M_i - 1)$$

so

$$\begin{aligned}
&\sum_{i=k+1}^{n_1} a_i - \sum_{i=k+1}^{n_1} a_i^-(M_i - 1) \\
&\qquad \le \sum_{i=k+1}^{n_1} a_i x_i \\
&\qquad \le \sum_{i=k+1}^{n_i} a_i + \sum_{i=k+1}^{n_i} a_i^+(M_i - 1).
\end{aligned}$$

(4) Similarly $-b_i y_i = -b_i - b_i t_i$.
Since $0 \le t_i \le N_i - 1$,

$$-(-b_i)^-(N_i - 1) \le -b_i t_i \le (-b_i)^+(N_i - 1)$$

or

$$-b_i^+(N_i - 1) \le -b_i t_i \le b_i^-(N_i - 1).$$

Hence

$$\begin{aligned} & - \sum_{i=k+1}^{n_2} b_i - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1) \\ \le & - \sum_{i=k+1}^{n_2} b_i y_i \\ \le & - \sum_{i=k+1}^{n_2} b_i + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1). \end{aligned}$$

Gathering these inequalities we get

$$\begin{aligned} \max_R h = & \sum_{i=0}^{n_1} a_i - \sum_{i=0}^{n_2} b_i + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1) \\ & - b_k + (a_k^+ - b_k)^+(N_k - 2) + \sum_{i=k+1}^{n_1} a_i^+(N_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1) \end{aligned}$$

and

$$\begin{aligned} \min_R h = & \sum_{i=0}^{n_1} a_i - \sum_{i=0}^{n_2} b_i - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1) \\ & - b_k - (a_k^+ - b_k)^+(N_k - 2) - \sum_{i=k+1}^{n_1} a_i^-(N_i - 1) - \sum_{i=k+1}^{n_2} b_i^+(N_i - 1). \end{aligned}$$

By subtracting $\sum_{i=0}^{n_1} a_i - \sum_{i=0}^{n_2} b_i$ from both sides of each inequality in min $h \le 0 \le$ max h, we see that there is a real solution of $h = 0$ in R if and only if Banerjee's inequality holds. Since there are integer solutions only if there are real solutions, the theorem follows. □

THEOREM 8. *Suppose S_1 and S_2 are of the form given above, and are contained in at least $k + 1$ common loops. If*

$$f(x_1, \ldots, x_{n_1}) = a_0 + \sum_{i=1}^{n_1} a_i x_i$$

and

$$g(x_1, \ldots, x_{n_2}) = b_0 + \sum_{i=1}^{n_2} b_i x_i$$

then $S_1 \, \gamma_{k+1}^k \, S_2$ only if

(*a*) *gcd test:*

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

(*b*) γ − *inequality*:

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

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

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

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

$$+ (a_k^+ - b_k)^+(N_k - 2) + (a_{k+1} + b_{k+1}^-)^+(N_{k+1} - 2)$$

$$+ \sum_{i=k+1}^{n_1} a_i^+ + (M_i - 1) + \sum_{i=k+1}^{n_2} b_i^-(N_i - 1).$$

PROOF. $S_1\gamma_{k+1}^k S_2 \Rightarrow$ (a) since $S_1\gamma_{k+1}^k S_2 \Rightarrow S_1\delta_k S_2$ by Lemma 4 and $S_1\delta_k S_2 \Rightarrow$ (a) by Theorem 4. We must show that $S_1\gamma_{k+1}^k S_2 \Rightarrow$ (b). Consider the real solutions of

$$h = f(x_1, \ldots, x_n) - g(y_1, \ldots, y_{n_2}) = 0$$

or

$$\sum_{i=1}^{n_1} a_i x_i - \sum_{i=1}^{n_2} b_i y_i + (a_0 - b_0) = 0$$

in the region S:

$$\begin{gathered} 1 \leq x_1 = y_1 \leq N_1 \\ \cdots \\ 1 \leq x_k < y_k \leq N_k \\ 1 \leq y_{k+1} < x_{k+1} \leq N_{k+1} \\ 1 \leq x_{k+2} \leq M_{k+2} \qquad 1 \leq y_{k+2} \leq N_{k+2} \\ \cdots \\ 1 \leq x_{n_1} \leq M_{n_1} \qquad 1 \leq y_{n_2} \leq N_{n_2} \end{gathered}$$

A real solution exists if and only if

$$\min_S h \leq 0 \leq \max_S h$$

By rearrangement we have

$$\begin{aligned} h = a_0 - b_0 &+ \sum_{i=1}^{k-1} (a_i x_i - b_i y_i) \\ &+ (a_k x_k - b_k y_k) + (a_{k+1} x_{k+1} - b_{k+1} y_{k+1}) \\ &+ \sum_{i=k+2}^{n_1} a_i x_i - \sum_{i=k+2}^{n_2} b_i y_i \end{aligned}$$

If we maximize and minimize these terms separately we have, from the proof of Theorem 4,

$$\begin{aligned}
&\sum_{i=1}^{k-1} (a_i - b_i) - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1)\\
&\le \sum_{i=1}^{k-1} (a_i x_i - b_i y_i)\\
&\le \sum_{i=1}^{k-1} (a_i - b_i) + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1)
\end{aligned}$$

and

$$\begin{aligned}
&\sum_{i=k+2}^{n_1} a_i - \sum_{i=k+2}^{n_2} b_i - \sum_{i=k+2}^{n_1} a_i^-(M_i - 1) - \sum_{i=k+2}^{n_2} b_i^+(N_i - 1)\\
&\le \sum_{k=k+2}^{n_1} a_i x_i - \sum_{i=k+2}^{n_2} b_i y_i\\
&\le \sum_{i=k+2}^{n_1} a_i - \sum_{i=k+2}^{n_2} b_i + \sum_{i=k+2}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+2}^{n_2} b_i^-(N_i - 1)
\end{aligned}$$

It only remains to maximize and minimize the terms for k and $k + 1$

$$\begin{aligned}
a_k x_k - b_k y_k &+ a_{k+1} x_{k+1} - b_{k+1} y_{k+1}\\
&= a_k + a_k(x_k - 1) - 2b_k - b_k(y_k - 2)\\
&+ 2a_{k+1} + a_{k+1}(x_{k+1} - 2) - b_{k+1} - b_{k+1}(y_{k+1} - 1)
\end{aligned}$$

Since $1 \le x_k < y_k \le N_k$, it follows that

$$0 \le x_k - 1 \le y_k - 2 \le N_k - 2 \qquad \text{(A1)}$$

and

$$-a_k^-(y_k - 2) \le a_k(x_k - 1) \le a_k^+(y_k - 2) \qquad \text{(A2)}$$

by Lemma 1. Also $1 \le y_{k+1} < x_{k+1} \le N_{k+1}$ implies

$$0 \le y_{k+1} - 1 \le x_{k+1} - 2 \le N_{k+1} - 2$$

so

$$-b_{k+1}^-(x_{k+1} - 2) \le b_{k+1}(y_{k+1} - 1) \le b_{k+1}^+(x_{k+1} - 2) \qquad \text{(A3)}$$

again by Lemma 1.

Combining (A1)–(A3) gives

$$\begin{aligned}
&a_k - 2b_k - (a_k^- + b_k)(y_k - 2) + 2a_{k+1} - b_{k+1} + (a_{k+1} - b_{k+1}^+)(x_{k+1} - 2)\\
&\le (a_k x_k - b_k y_k) + (a_{k+1} x_{k+2} - b_{k+1} y_{k+1})\\
&\le a_k - 2b_k + (a_k^+ - b_k)(y_k - 2) + 2a_{k+1} - b_{k+1} + (a_{k+1} + b_{k+1}^-)(x_{k+1} - 2).
\end{aligned}$$

Applying Lemma 1 twice more on each side and rearranging, we get

$$
\begin{aligned}
&(a_k - b_k) + (a_{k+1} - b_{k+1}) + (a_{k+1} - b_k)\\
&\quad - (a_k^- + b_k)^+(N_k - 2) - (a_{k+1} - b_{k+1}^+)^-(N_{k+1} - 2)\\
&\quad \le (a_k x_k - b_k y_k) + (a_{k+1} x_{k+1} - b_{k+1} y_{k+1})\\
&\quad \le (a_k - b_k) + (a_{k+1} - b_{k+1}) + (a_{k+1} - b_k)\\
&\quad + (a_k^+ - b_k)^+(N_k - 2) + (a_{k+1} + b_{k+1}^-)^+(N_{k+1} - 2).
\end{aligned}
$$

Since Lemma 1 implies that there exist values of x_i and y_i such that equality exists in each case where it was applied,

$$
\begin{aligned}
\min_S h = a_0 - b_0 &+ \sum_{i=1}^{k-1} (a_i - b_i) - \sum_{i=1}^{k-1} (a_i - b_i)^-(N_i - 1)\\
&+ (a_k - b_k) + (a_{k+1} - b_k) - (a_k + b_k)^+(N_k - 2)\\
&+ (a_{k+1} - b_{k+1}) - (a_{k+1} - b_{k+1})^-(N_{k+1} - 2)\\
&+ \sum_{i=k+2}^{n_1} a_i - \sum_{i=k+2}^{n_2} b_i - \sum_{i=k+2}^{n_1} a_i^-(M_i - 1) - \sum_{i=k+2}^{n_2} b_i^+(N_i - 1)
\end{aligned}
$$

$$
\begin{aligned}
\max_S h = a_0 - b_0 &+ \sum_{i=1}^{k-1} (a_i - b_i) + \sum_{i=1}^{k-1} (a_i - b_i)^+(N_i - 1)\\
&+ (a_k - b_k) + (a_{k+1} - b_k) + (a_k^+ - b_k)^+(N_k - 2)\\
&+ (a_{k+1} - b_{k+1}) + (a_{k+1} + b_{k+1}^-)^+(N_{k+1} - 2)\\
&= \sum_{i=k+2}^{n_1} a_i - \sum_{i=k+2}^{n_2} b_i + \sum_{i=k+2}^{n_1} a_i^+(M_i - 1) + \sum_{i=k+2}^{n_2} b_i^-(N_i - 1).
\end{aligned}
$$

Subtracting $\sum_{i=0}^{n_1} a_i - \sum_{i=0}^{n_2} b_i$ from both sides of each inequality in the expression

$$\min_S h \le 0 \le \max_S h$$

yields the desired result. □

ACKNOWLEDGMENTS

This project was started on a sabbatical at IBM Research in Yorktown Heights, New York. George Paul was responsible for many of the connections with IBM and for the early direction towards language systems for vector machines. Much of the early development was based upon the ideas of David Kuck and his group, illuminated in discussions with Utpal Banerjee and Michael Wolfe. Although some of the programming was our own, a great deal more is due to Ron Cytron, Carrie Porterfield, Keith Cooper, Kenny Zadeck, David Chase, Scott Comer, Joe Warren, Alan Weingarten, Robert Warfield, and David Callahan. Finally, Horace Flatt has provided all the steady, even-handed support that one might hope for in a project monitor. To all these people go our heartfelt thanks.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass, 1974.

2. ALLEN, F. E., COCKE, J., AND KENNEDY, K. Reduction of operator strength. In *Program Flow Analysis: Theory and Applications*, N. D. Jones and S. S. Muchnick, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1981.
3. ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Tx., Jan. 1983). ACM, New York, 1983.
4. ALLEN, J. R. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. dissertation, Dept. of Mathematical Sciences, Rice University, Houston, Tx., April 1983.
5. American National Standards Institute, Inc. Proposals approved for Fortran 8x. X3J3/S6.80 (preliminary document). ANSI, New York, Nov. 30, 1981.
6. BANERJEE, U. Data dependence in ordinary programs. Report 76-837, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill., Nov. 1976.
7. Burroughs Corporation. Implementation of FORTRAN. Burroughs Scientific Processor Brochure, 1977.
8. COCKE, J., AND KENNEDY, K. An algorithm for reduction of operator strength. *Commun. ACM 20*, 11 (Nov. 1977), 850–856.
9. COHAGEN, W. L. Vector optimization for the ASC. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems* (Dept. of Electrical Engineering, Princeton, N.J., 1973), pp. 169–174.
10. Cray Research, Inc. *Cray-1 Computer System Reference Manual.* Publication 2240004, Cray Research, Inc., Bloomington, Minn., 1976.
11. Cray Research, Inc. *Cray-1 Computer System FORTRAN (CFT) Reference Manual.* Publication 2240009, Cray Research, Inc., Mendota Heights, Minn., 1980.
12. Department of Energy, Advanced Computing Committee Language Working Group. FORTRAN language requirements, fourth report. Draft report, Department of Energy, Aug. 1979.
13. Floating Point Systems, Inc. *AP-120 Programmers Reference Manual.* Publication 860-7319-000, Floating Point Systems, Inc., Beaverton, Ore., 1978.
14. GRIFFIN, H. *Elementary Theory of Numbers.* McGraw-Hill, New York, 1954.
15. HIGBEE, L. Vectorization and conversion of FORTRAN programs for the Cray-1 CFT compiler. Publication 2240207, Cray Research Inc., Mendota Heights, Minn., June 1979.
16. KNUTH, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1973.
17. KUCK, D. J. A survey of parallel machine organization and programming. *Comput. Surv. 9*, 1 (March 1977), 29–59.
18. KUCK, D. J. *The Structure of Computers and Computations*, Vol. 1, Wiley, New York, 1978.
19. KUCK, D. J., KUHN, R. H., LEASURE, B., PADUA, D. A., AND WOLFE, M. Compiler transformation of dependence graphs. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 1981). ACM, New York, 1981.
20. KUCK, D. J., KUHN, R. H., LEASURE, B., AND WOLFE, M. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the IEEE Computer Society Fourth International Computer Software and Applications Conference* (Chicago, Oct. 1980). IEEE, New York, 1980.
21. LEASURE, B. R. Compiling serial languages for parallel machines. Report-76-805, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill., Nov. 1976.
22. LEVESQUE, J. M. Application of the Vectorizer for effective use of high-speed computers. In *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Eds. Academic Press, New York, 1977, pp. 447–449.
23. MCCLUSKEY, E. J. Minimization of Boolean functions. *Bell System Tech. J. 35*, 5 (Nov. 1956), 1417–1444.
24. MURAOKA, Y. Parallelism exposure and exploitation in programs. Report 71-414, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana Ill., Feb. 1971.
25. MYSZEWSKI, M. The Vectorizer System: Current and proposed capabilities. Report CA-17809-1511 Massachusetts Computer Associates, Inc., Wakefield, Mass., Sept. 1978.
26. PAUL, G., private communication, 1980.

27. PAUL, G., AND WILSON, M. W. An Introduction to VECTRAN and its use in scientific applications programming. In *Proceedings of the LASL Workshop on Vector and Parallel Processors.* (Los Alamos, N.M., Sept. 1978). University of California, Los Alamos Scientific Laboratory, Los Alamos, N.M., 1978, pp. 176–204.
28. PAUL, G., AND WILSON, M. W. The VECTRAN language: An experimental language for vector/matrix array processing. IBM Palo Alto Scientific Center Report G320-3334, Palo Alto, Calif., Aug. 1975.
29. QUINE, W. V. The problem of simplifying truth functions. *Am. Math. Monthly 59*, 8 (Oct. 1952), 521–531.
30. RUSSELL, R. M. The CRAY-1 computer system. *Commun. ACM 21*, 1 (Jan. 1978), 63–72.
31. TARJAN, R. E. Depth first search and linear graph algorithms. *SIAM J. Comput. 1*, 2 (1972), 146–160.
32. TOWLE, R. A. Control and data dependence for program transformations. Ph.D. dissertation, Report. 76-788, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill., March 1976.
33. WEDEL, D. Fortran for the Texas Instruments ASC system. *ACM SIGPLAN Not. 10*, 3 (March 1975), 119–132.
34. WETHERELL, C. Array processing for FORTRAN. Report CID-30175, Lawrence Livermore Laboratory, Livermore, Calif., April 1979.
35. WITTMAYER, W. R. Array processor provides high throughput rates. *Comput. Design* (March 1978).
36. WOLFE, M. J. Techniques for improving the inherent parallelism in programs. Report 78-929, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill., July 1978.

Received May 1981; revised January 1985, September 1985, February 1986; accepted February 1986