The Power Test for Data Dependence

Michael Wolfe
Chau-Wen Tseng

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-015

August, 1990

The Power Test for Data Dependence

Michael Wolfe
Chau-Wen Tseng

Oregon Graduate Institute of Science and Technology
Department of Computer Science and Engineering
19600 NW von Neumann Drive
Beaverton, OR 97006
(503)-690-1153
mwolfe@cse.ogi.edu

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251
(713)-527-8101
tseng@rice.edu

The Power Test for Data Dependence

Abstract

This paper introduces a data dependence decision algorithm, called the Power Test; the Power Test is a combination of Banerjee's Generalized GCD dependence algorithm and the Fourier-Motzkin method to eliminate variables in a system of inequalities. In addition to having certain advantages over previous dependence algorithms (such as increased precision, the ability to handle multiple subscripts simultaneously, the ability to handle complex multiple loop limits, and others), it can also easily generate dependence direction vector information. This paper briefly reviews previous work in data dependence decision algorithms, and describes the Power Test. Several examples which motivated the development of this test are examined, including those which demonstrate additional power of the Power Test.

1. Introduction

Vectorizing and parallelizing compilers are common in the supercomputer and mini-supercomputer commercial market; these compilers inspect the patterns of data usage in programs, especially array usage in loops, often representing these patterns as a data dependence graph. With this information, compilers can often automatically detect parallelism in loops, or report to the user specific reasons why a particular loop cannot be executed in parallel. Additional performance improvement can be attained by using certain program transformations to take advantage of architectural features, such as improving memory locality to take advantage of cache memories. In order to determine what restructuring transformations are legal, data dependence tests are devised to detect those programs or loops whose semantics will be violated by the transformation. General literature on this subject is widely available [AlK87, ABC87, BCK79, Ban88, BuC86, GJG87, LY90, WoB87, Wol89].

In order to allow the most freedom in applying restructuring transformations, a compiler needs a precise data dependence test. Much of the theory behind data dependence testing for array references in loops can be reduced to solving simultaneous diophantine equations. The data dependence problem for array references can be stated as:

> Given a set of nested loops surrounding two statements (not necessarily distinct) where each statement contains a reference to an array:

```
L1:  for I1 = ℓ1 to u1 do
L2:   for I2 = ℓ2 to u2 do
      ...
Lc:    for Ic = ℓc to uc do
Lc+1:   for Ic+1 = ℓc+1 to uc+1 do
        ...
Lc+d1:    for Ic+d1 = ℓc+d1 to uc+d1 do
S1:       A(f1(I1,...,Ic,Ic+1,...,Ic+d1),...,fs(I1,...,Ic+d1))
        endfor
        ...
      endfor
Lc+d1+1: for Ic+d1+1 = ℓc+d1+1 to uc+d1+1 do
        ...
Lc+d1+d2:  for Ic+d1+d2 = ℓc+d1+d2 to uc+d1+d2 do
S2:        A(g1(I1,...,Ic,Ic+d1+1,...,Ic+d1+d2),...,gs(I1,...,Ic+d1+d2))
         endfor
         ...
       endfor
      endfor
      ...
    endfor
  endfor
```

We want to determine whether there are values of the loop indices that lie within their limits such that the subscript functions are simultaneously equal; i.e., we want to determine whether there exist integer values:

$i_1, i_2, \ldots, i_c, i_{c+1}, \ldots, i_{c+d1}$ and $j_1, j_2, \ldots, j_c, j_{c+d1+1}, \ldots, j_{c+d1+d2}$

such that:

$$\ell_k \leq i_k \leq u_k, \quad \forall k \text{ and } \ell_k \leq j_k \leq u_k, \quad \forall k$$

and

$$f_m(\bar{i}) = g_m(\bar{j}) \quad \forall m$$

Additional information may be desired if there is a solution, such as the dependence distance in each dimension (the value of $j_k - i_k$ for $1 \leq k \leq c$), or a dependence direction in each dimension (the sign of the dependence distance). Loops L_1 through L_c are called the *common loops.*

A great deal of research has gone into the development of various data dependence decision algorithms, which vary in generality, precision and complexity. Most decision algorithms require the subscript functions to be linear combinations of the loop index variables with known constant coefficients. A test in common use is Banerjee's Inequalities [BCK79], which is efficient, but tests each subscript independently (reducing precision). The array references can be "linearized" to solve for simultaneous solutions, but this does not always improve precision [BCK79,BuC86]. The Lambda test was developed as another method to add simultaneity to Banerjee's Inequalities [Gru90,LY90]. Banerjee's Inequalities have also been extended to provide direction vector information [Wol82,Wol89]. Banerjee's Inequalities test for real (not integer) solutions of the dependence equation; they have recently been extended to handle triangular as well as rectangular loop limits when the coefficients are known constants [Ban88].

Another simple dependence test is the single subscript GCD test. Banerjee has also developed a Generalized GCD algorithm which tests multiple subscripts simultaneously. These decision algorithms test for integer solutions of the dependence equation, but ignore loop limits. The Generalized GCD test can also be trivially extended to provide dependence distance information, as shown later in this paper.

Other methods have been attempted for use as decision algorithms, but are generally more expensive, such as Shostak's loop residue method [Sho81]. For many purposes, a simple single-index-variable test will suffice, applied on one subscript at a time. For more advanced restructuring transformations, however, more precision is necessary.

The Power Test is a combination of Banerjee's Generalized GCD test with the Fourier-Motzkin variable elimination method. Its name is derived from the power and precision of the method, and from the fact that in the worst case it can take exponential time (in the number of loop index variables). The Power Test finds only integer solutions and considers the loop limits,

and can handle triangular, trapezoidal and complicated convex loop limits; as we shall see, it is imprecise in some cases when the loop limits or other conditions cannot be handled exactly and an integer solution occurs "near" the solution space boundary. Since it is derived from the Generalized GCD test, it solves for all subscripts simultaneously.

2. Motivation

The first motivation behind the development of the Power Test was a challenge from a colleague. We have recently been constructing a program restructuring research tool, called TINY, which parses a tiny language, builds a data dependence graph, then applies (under interactive user control) a series of loop restructuring transformations. Among the transformations implemented are loop interchanging [AlK84], loop skewing [Wol86a], loop reversal [Wed75] and loop distribution [BCK79]. While demonstrating the tool to a colleague, he asked if it could build a precise dependence graph for the following program:

```
      for I = 1 to N do
        for J = I+1 to N do
S1:     A(I) = A(J)
        endfor
      endfor
```

In particular, this colleague wanted a precise *direction vector* in addition to the correct data dependence relations (terminology used here is defined in the following section). The first attempt failed to find any dependence relations whatever. After fixing several serious bugs in the tool, a second attempt found the dependence relations, but with a rather imprecise direction vector. Looking at the iteration space of this loop:

	J=2	J=3	J=4	J=5
I=1	A(1)=A(2)	A(1)=A(3)	A(1)=A(4)	A(1)=A(5)
I=2		A(2)=A(3)	A(2)=A(4)	A(2)=A(5)
I=3			A(3)=A(4)	A(3)=A(5)
I=4				A(4)=A(5)

we find two data dependence relations: an obvious data output-dependence relation due to the reassignment of A(I) on each iteration of the J loop (for the dependence relation $S_1\ \delta^{o}_{(=,<)}\ S_1$), and a data anti-dependence relation due to the use and subsequent assignment of element A(K) from each iteration S_1[1:K-1,K] to each iteration S_1[K,K+1:N]. The dependence relation with precise direction vector is $S_1\ \bar{\delta}_{(<,<)}\ S_1$. Note that there is no fixed *dependence distance* here, though the direction vector (corresponding to the sign of the dependence distance) is precise. Note also that "normalizing" the inner loop would change the shape

of the iteration space, and would affect the direction vector; we do not normalize loops.

The second motivation also came during the construction of TINY. One of the restructuring transformations planned was interchanging of non-tightly-nested loops [Wol89]. In particular, we wanted to be able to generate all 6 versions of the Cholesky decomposition program (LL^T factorization of a symmetric matrix) through loop restructuring; the basic KIJ form of Cholesky decomposition is:

```
      for K = 1 to N do
S1:     A(K,K) = sqrt(A(K,K))
        for I = K+1 to N do
S2:       A(I,K) = A(I,K) / A(K,K)
          for J = K+1 to I do
S3:         A(I,J) = A(I,J) - A(I,K)*A(J,K)
          endfor
        endfor
      endfor
```

The data dependence relations just involving S_1 for this loop are:

$S_3\ \delta_{(<)}\ S_1$

$S_1\ \delta_{(=)}\ S_2$

$S_3\ \delta^{\circ}_{(<)}\ S_1$

$S_3\ \overline{\delta}_{(<)}\ S_1$

Note the loop-independent dependence relation from S_1 to S_2, requiring S_1 to lexically precede S_2; also note that S_1 is bound in a dependence cycle with the inner loop, preventing distribution of the K loop. To generate the IKJ form requires interchanging the imperfectly-nested K and I loops to get:

```
      for I = 1 to N do
        for K = 1 to I-1 do
S2:       A(I,K) = A(I,K) / A(K,K)
          for J = K+1 to I do
S3:         A(I,J) = A(I,J) - A(I,K)*A(J,K)
          endfor
        endfor
S1:     A(I,I) = sqrt(A(I,I))
      endfor
```

Notice where S_1 must be placed in relation to the inner loop. The dependence relations involving S_1 in the restructured loop are now:

$S_3\ \delta_{(=)}\ S_1$

$S_1\ \delta_{(<)}\ S_2$

$S_3\ \delta^{\circ}_{(=)}\ S_1$

$S_3\ \overline{\delta}_{(=)}\ S_1$

Note the loop independent dependence relations coming into S_1, requiring its placement below

the inner loop. In both versions of the loop, there is no option in the placement of S_1.

As explained in [Wol86b, Wol89], the data dependence test for interchanging imperfectly nested loops (unlike simple loop interchanging) is not a direction vector test. What we needed was a data dependence test which would (1) tell when a data dependence relation would be violated by interchanging imperfectly nested loops and (2) what the direction vectors would be after interchanging.

An additional motivation was to be able to compute dependence for all subscript equations simultaneously, which we found to be critical in some instances. Yet another motivation was the result of the ability to interchange loops with trapezoidal limits, as in:

```
for I = 2 to N-1 do
  for J = I+2 to I+N-1 do
    ...
```

the interchanged limits of the inner loop involve maxima and minima:

```
for J = 4 to 2*N-2
  for I = max(2,J-N+1) to min(N-1,J-2) do
    ...
```

In order to compute dependence relations in the modified loop as precisely as possible, we want to take advantage of the extra knowledge of the simultaneous constraints of the multiple lower and upper loop limits.

In these examples, existing data dependence decision algorithms fall short. As it turns out, Banerjee's Inequalities extended with triangular loop limits [Ban88] will correctly handle the first example, but they are harder (or impossible) to extend to the second case. Also, Banerjee's Inequalities handle only a single subscript equation at a time, and only a single lower and upper loop limit. The method for extending Banerjee's multiple-index-variable dependence test [Ban79], boldly (and improperly) called the "Exact Algorithm for Multiple Indices" in [Wol89], has been shown not only to be imprecise, but is incorrect; all attempts to correct it made it even more expensive (in time and space) and reduced its claims for precision to the point where we gave up. So we embarked on the search for a unified dependence decision algorithm that would properly handle all these potentially important cases.

3. Definitions and Terminology

For the purposes of this paper, we are concerned about data dependence between array references in loops; we assume imperative language loop semantics (as in Fortran, C or Pascal). For instance, in the loop:

```
      for I = 2 to N-2 do
S1:     A(I) = B(I)
S2:     C(I) = A(I-1)
S3:     D(I) = C(I+2)
      endfor
```

the array element assigned to A(i) in iteration I=i of statement S_1 is fetched by statement S_2 in the second subsequent iteration of the loop. On the other hand, array element C(i+2) fetched by statement S_3 is reassigned by statement S_2 in the next iteration of the loop. The first case is called a *def-use* ordering, or a *flow-dependence;* for shorthand, we say $S_1 \delta S_2$. The second case is a *use-def* ordering, called *anti-dependence;* we write $S_3 \bar{\delta} S_2$. There can also be *def-def* orderings, called *output-dependence.*

Saying S_2 depends on both statements S_1 and S_3 means that some iteration of S_2 depends on some iteration of S_1 and some iteration of S_3. For loop vectorization, this may be enough information [AlK87]. For other program transformations, more precise information is useful. We use the notation $S_2[2]$ to mean the instance of S_2 when the loop variable I=2. If instance $S_2[j]$ depends on instance $S_1[i]$, then the *dependence distance* is defined to be j-i. In our example above, the dependence distance for $S_1 \delta S_2$ is one, while the dependence distance for $S_3 \bar{\delta} S_2$ is two. These definitions would have to be modified if the loop increment were something other than +1.

In multiple loops, there is an independent distance in each loop. Take the program:

```
      for I1 = 1 to N do
        for I2 = 2 to N-1 do
S1:       A(I1,I2) = B(I1,I2)
S2:       C(I1,I2) = A(I1-1,I2)
        endfor
      endfor
```

Here we have one data dependence relation, $S_1 \delta S_2$. The distance for the I_1 loop is one, while the distance for the I_2 loop is zero. We usually write these distances as a *distance vector*; here the distance vector would be (1,0). We sometimes subscript the dependence relation with the distance vector, as in $S_1 \delta_{(1,0)} S_2$.

Sometimes the dependence distance is not constant; rather than finding all possible dependence distances, we simplify the problem by finding the signs of all possible distances. In the program:

```
      for I1 = 1 to N do
S1:     X(I1) = A(I1)
        for I2 = 1 to I1-1 do
S2:       C(I1,I2) = X(I2)
        endfor
      endfor
```

the value assigned to $X(i_1)$ in $S_1[i_1]$ is used in $S_2[j_1, j_2]$ for every j_1 such that $i_1 < j_1$.

Note that there is no dependence from $S_1[i_1]$ to $S_2[j_1, j_2]$ when $i_1=j_1$ or when $i_1>j_1$. Thus, though the dependence distance varies in magnitude, it is always strictly greater than zero. We can therefore just save the sign of the distance as a vector; in this case, we would save (+) or $S_1\ \delta_{(+)}\ S_2$. Current notational conventions use <, = and > as direction vector elements (instead of +, 0 and -, respectively). We would then write $S_1\ \delta_{(<)}\ S_2$, meaning that we have dependence from some $S_1[i_1]$ to some $S_2[j_1, j_2]$ where $i_1<j_1$. as the possible relations between i_1 and j_1.

Terminology. In the following sections, bold lower case letters refer to row vectors and bold upper case letters refer to matrices; all vector and matrix entries are integers. Let us assume the following:

1) the loop index variables are numbered from I_1 through $I_{c+d1+d2}$

2) there are s subscripts, and the subscript functions are linear combinations of the loop index variables with constant coefficients, so that the subscript functions are as below, for $1 \leq m \leq s$:

$$f_m(I_1, \ldots, I_c, I_{c+1}, \ldots, I_{c+d1}) = \begin{cases} f_{m,0}+f_{m,1}I_1+f_{m,2}I_2+\ldots+f_{m,c}I_c+ \\ f_{m,c+1}I_{c+1}+\ldots+f_{m,c+d1}I_{c+d1} \end{cases}$$

$$g_m(I_1, \ldots, I_c, I_{c+d1+1}, \ldots, I_{c+d1+d2}) = \begin{cases} g_{m,0}+g_{m,1}I_1+g_{m,2}I_2+\ldots+g_{m,c}I_c+ \\ g_{m,c+d1+1}I_{c+d1+1}+\ldots+g_{m,c+d1+d2}I_{c+d1+d2} \end{cases}$$

For simplicity, we assume that any $f_{m,x}$ or $g_{m,x}$ not defined by the subscript functions are equal to zero (such as $f_{m,c+d1+1}$ or $g_{m,c+1}$). A subsequent section will argue how to deal with unknown variables in the subscript functions. Thus, each subscript will generate one dependence equation:

$$f_{m,1}i_1-g_{m,1}j_1+f_{m,2}i_2-g_{m,2}j_2+\ldots+f_{m,c}i_c-g_{m,c}j_c+f_{m,c+1}i_{c+1}+\ldots$$
$$+f_{m,c+d1}i_{c+d1}-g_{m,c+d1+1}j_{c+d1+1}-\ldots-g_{m,c+d1+d2}j_{c+d1+d2}=g_{m,0}-f_{m,0}$$

or, after renaming:

$$a_{m,1}h_1+a_{m,2}h_2+\ldots+a_{m,n}h_n=c_m$$

where $n=c+d1+c+d2$ is the total number of index variables involved, and where **h** is a renaming of these index variables:

$$\mathbf{h} = (h_1, h_2, \ldots, h_n) = (i_1, j_1, \ldots, i_c, j_c, i_{c+1}, i_{c+2}, \ldots, i_{c+d1}, j_{c+d1+1}, \ldots, j_{c+d1+d2})$$

3) the lower and upper limits of the loop index variables are also linear combinations of outer loop index variables, with constant coefficients:

$$h_k \geq \ell_{k,0}+\ell_{k,1}h_1+\ell_{k,2}h_2+\ldots+\ell_{k,k-1}h_{k-1}$$
$$h_k \leq u_{k,0}+u_{k,1}h_1+u_{k,2}h_2+\ldots+u_{k,k-1}h_{k-1}$$

The set of dependence equations with the inequalities induced by the lower and upper limits together comprise the *dependence system.*

4. Review of the Generalized GCD Algorithm

The Power Test begins with the Generalized GCD algorithm given in [Ban88]; we briefly review that algorithm here. It will be important that the subscript functions be linearly independent, but since the Generalized GCD algorithm naturally finds linearly-dependent subscript functions, we need not worry about that. The Generalized GCD Algorithm starts by filling an $n \times s$ coefficient matrix **A** with the coefficients of the subscript functions.

$$\begin{matrix} a_{1,1} & a_{2,1} & \cdots \\ a_{1,2} & a_{2,2} & \cdots \\ a_{1,3} & a_{2,3} & \cdots \\ a_{1,4} & a_{2,4} & \cdots \\ \cdots & \cdots & \end{matrix}$$

The goal is to discover whether there is an integer vector **h** that solves all dependence equations simultaneously, **hA=c**, (where **c** has s elements, one for each subscript). The algorithm initializes an $n \times s$ matrix **D** with **A**, and an $n \times n$ unimodular matrix **U** with the identity matrix. These two matrices are stored in one combined $n \times (n+s)$ matrix **UD**. By a series of elementary integer row operations (essentially equivalent to Gaussian Elimination adapted for integers) the **D** matrix is reduced to upper triangular form. This means that column k of **D** will have zero elements in rows $k+1$ through n (for $1 \leq k < n$):

$$\begin{matrix} d_{1,1} & d_{1,2} & d_{1,3} & \cdots \\ 0 & d_{2,2} & d_{2,3} & \cdots \\ 0 & 0 & d_{3,3} & \cdots \\ 0 & 0 & 0 & \cdots \end{matrix}$$

During this phase, if a diagonal element is found to be identically zero, then that column (subscript equation) must be a linear combination of previous columns (subscript equations), and can be eliminated from further consideration (effectively reducing s by one). Applying the same elementary integer row operations to the **U** matrix produces a unimodular matrix with the property that **UA=D**.

If we find an integer solution **t** such that **tD=c**, then **h=tU** is a solution to the dependence equations **hA=c** (as shown in [Ban88]). After finding **U**, we can solve for t_1 through t_s by

solving **tD=c** using a simple back-substitution algorithm.

Example. Take the program:

```
        for I1 = 1 to 100
          for I2 = 1 to 100
S1:         A(4*I1-2*I2-1, -I1-2*I2) = ...
S2:         ... = A(I1+2*I2+1, -2*I1+1)
          endfor
        endfor
```

The dependence equations to be solved are:

$$4i_1-2i_2-1=j_1+2j_2+1$$
$$-i_1-2i_2=-2j_1+1$$

or, rewritten to:

$$4i_1-j_1-2i_2-2j_2=2$$
$$-i_1+2j_1-2i_2=1$$

The dependence matrix **hA=c** is:

$$(i_1, j_1, i_2, j_2)\begin{pmatrix} 4 & -1 \\ -1 & 2 \\ -2 & -2 \\ -2 & 0 \end{pmatrix} = (2, 1)$$

We augment **A** with the identity matrix to get **UD**:

$$\left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 4 & -1 \\ 0 & 1 & 0 & 0 & -1 & 2 \\ 0 & 0 & 1 & 0 & -2 & -2 \\ 0 & 0 & 0 & 1 & -2 & 0 \end{array}\right)$$

(**U** comprises the first four columns, **D** the final two.) After performing the elementary row operations to reduce **D** to an upper triangular matrix, we have:

$$\mathbf{UD} = \left(\begin{array}{cccc|cc} 0 & 1 & 0 & 0 & -1 & 2 \\ 1 & 4 & 3 & -3 & 0 & 1 \\ -2 & -8 & -7 & 7 & 0 & 0 \\ 0 & -2 & -2 & 3 & 0 & 0 \end{array}\right)$$

We can verify that **UA=D**. Now we solve **tD=c**,

$$(t_1, t_2, t_3, t_4)\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = (2, 1)$$

Because **D** is an upper triangular matrix, t_3 and t_4 do not figure into the computation. The first equation is $-t_1=2$ or $t_1=-2$. The second equation is $2t_1+t_2=1$ or $t_2=5$.

Since there is a feasible solution, Banerjee's Generalized GCD algorithm stops here and assumes dependence.

5. Extension of the Generalized GCD Algorithm

The Generalized GCD algorithm, like the single-subscript GCD test, tells whether there can be any integer solution to the dependence equation, ignoring loop limits. However, it also gives formulae that can be used to specify the index variables h_1, h_2, ..., h_n in terms of the "free" variables t_{s+1}, t_{s+2}, ..., t_n, derived from the equation **h=tU**. The first extension that we make to the Generalized GCD algorithm is to find all constant dependence distances by subtracting corresponding equations. That is, the dependence distance for loop level k ($1 \le k \le c$) can be found by subtracting the equations for i_k and j_k. Suppose that the $i_k \equiv h_{2k-1}$, and $j_k \equiv h_{2k}$; we subtract the equations by looking at:

$$\mathbf{tU}_{.,2k-1} - \mathbf{tU}_{.,2k}$$

(where $\mathbf{U}_{.,x}$ is column x of the matrix **U**). If the dependence distance is fixed, this will have non-zero coefficients only for t_1 through t_s, which were previously solved. If there are non-zero coefficients for any other t_v, where $v>s$, the dependence distance is not constant; equivalently, if $\mathbf{U}_{s+1:n,2k-1} - \mathbf{U}_{s+1:n,2k}$ is the zero vector, the dependence distance is constant, and can be found from $\mathbf{t}_{1:s}(\mathbf{U}_{1:s,2k-1} - \mathbf{U}_{1:s,2k})$.

This allows a dependence distance vector to sometimes be constructed from the **U** matrix and the solutions to $t_1, \ldots, t_s$. The dependence direction vector can also be constructed from the signs of the distance vector. Of course, if the dependence distance for any loop exceeds the maximum trip count (number of iterations) of that loop, the references are independent [AlK87]. The Generalized GCD algorithm is more precise than linearizing the array references and using a single-equation GCD test [BCK79,BuC86].

Example. Take the program:

```
for I1 = 1 to 3 do
  for I2 = 1 to 3 do
    X(I1+I2+1,I2+1) = . . .
     . . .  = X(I1+I2,I2)
  endfor
endfor
```

The dependence equations are:

$$i_1 - j_1 + i_2 - j_2 = -1$$
$$i_2 - j_2 = -1$$

The dependence matrix **hA=c** is:

$$(i_1, j_1, i_2, j_2) \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 1 \\ -1 & -1 \end{pmatrix} = (-1, -1)$$

After augmenting this with the identity matrix to get **UD**, and reducing **D** to an upper triangular matrix via elementary row operations, we have

$$\mathbf{UD} = \left(\begin{array}{cccc|cc} 0 & 0 & 0 & 1 & -1 & -1 \\ 0 & 1 & 0 & -1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right)$$

From this we solve **tD=c**:

$$(t_1, t_2, t_3, t_4) \begin{pmatrix} -1 & -1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = (-1, -1)$$

This produces the equations:

$$-t_1 = -1$$

$$-t_1 + t_2 = -1$$

which we solve to get

$$t_1 = 1, t_2 = 0$$

Using the equations to derive **h** from **tU** we get:

$$(h_1, h_2, h_3, h_4) = (i_1, j_1, i_2, j_2) = (t_1, t_2, t_3, t_4) \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

This generates the equations:

$$i_1 = t_3$$

$$j_1 = t_2 + t_3$$

$$i_2 = t_4$$

$$j_2 = t_1 - t_2 + t_4$$

The dependence distance vector is computed as $(j_1 - i_1, j_2 - i_2)$, which is: $(t_2, t_1 - t_2)$. But these are already solved quantities, so the dependence distance is $(0, 1)$.

This could also be computed by subtracting column $\mathbf{U}_{.,1}$ (which corresponds to i_1) from column $\mathbf{U}_{.,2}$ (which corresponds to j_1), and $\mathbf{U}_{.,3}$ from $\mathbf{U}_{.,4}$, to get:

$$(d_1, d_2) = (t_1, t_2, t_3, t_4) \begin{pmatrix} 0 & 1 \\ 1 & -1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = (1, 0, t_3, t_4) \begin{pmatrix} 0 & 1 \\ 1 & -1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

6. The Power Test

The method from the previous section does nothing for more precision in cases where the dependence distance is not fixed in one or more index dimensions. Using Fourier-Motzkin variable elimination [DaE73,Duf74], the Power Test continues from this point.

We construct a list of upper and lower bounds on each free variable t_{s+1} through t_n. For a free variable t_k, each lower and upper bound will be a linear combination of t_{s+1}, t_{s+2}, ..., t_{k-1}. These give the boundaries to the solution space of the dependence equation; if the solution space is non-empty, then the dependence equation has solutions that satisfy all the conditions. For instance, each lower bound for t_k will be of the form:

$$lb_k t_k \geq lb_0 + lb_{s+1} t_{s+1} + \cdots + lb_{k-1} t_{k-1}$$

with $lb_k > 0$. Since we are dealing with integers, we can take the ceiling of lower bounds (or the floor of upper bounds):

$$t_k \geq \left\lceil (lb_0 + lb_{s+1} t_{s+1} + \cdots + lb_{k-1} t_{k-1}) / lb_k \right\rceil$$

and similarly for upper bounds. These bounds are derived from the constraints on the index variables, such as the loop limits. For instance, if the lower limit on index variable h_m is 1, we have the inequality $h_m \geq 1$. From this we replace h_m by its equivalent in terms of the free variables: $\mathbf{tU}_{.,m} \geq 1$. This inequality generates a bound for the highest numbered free variable which has a non-zero coefficient. It will correspond to a lower bound if that non-zero coefficient is positive, or a upper bound if the coefficient is negative.

The ceiling or floor operators are a source of precision if they can be used to advantage; in the lower bound above, if lb_k divides all of lb_{s+1} through lb_{k-1} exactly, then the bound can be reduced to:

$$t_k \geq \left\lceil \frac{lb_0}{lb_k} \right\rceil + \frac{lb_{s+1}}{lb_k} t_{s+1} + \cdots + \frac{lb_{k-1}}{lb_k} t_{k-1})$$

where all the divisions, including $\lceil lb_0 / lb_k \rceil$ can be computed by the compiler. This gives a more precise bound:

$$t_k \geq \left\lceil (lb'_0 + lb'_{s+1} t_{s+1} + \cdots + lb'_{k-1} t_{k-1}) / lb'_k \right\rceil$$

where lb'_k is equal to one. If the divisions are not exact, then the ceiling or floor operators will be a source of imprecision, since the Power Test will essentially ignore them to remain in the realm of integer computation. The method used to handle inexact division, equivalent to LP-relaxation, essentially solves the linear programming problem rather than the integer programming problem by enlarging the solution space to potentially include some integer points near the boundaries. In particular, this method may include some integer points in an otherwise empty solution space. A slightly more general method to find a more precise bound would be to find the GCD of lb_{s+1}, lb_{s+2}, ..., lb_{k-1} and lb_k, (let g be the name of this GCD); then make the following substitutions:

$$lb'_0 = \left\lceil lb_0/g \right\rceil$$

$$lb'_r = lb_r/g, \quad \forall s+1 \leq r \leq k-1$$

$$lb'_k = lb_k/g$$

Example. Take the program:

```
for I1 = 1 to 100
  for I2 = I1+1 to 100
    X(I1,I2) =  . . .
     . . .   = X(I2,I1)
  endfor
endfor
```

The dependence equations are:

$$i_1 - j_2 = 0$$

$$i_2 - j_1 = 0$$

The dependence matrix **hA=c** is:

$$(i_1, j_1, i_2, j_2) \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ 0 & 1 \\ -1 & 0 \end{pmatrix} = (0,0)$$

The Generalized GCD algorithm gives the matrix:

$$\mathbf{UD} = \left(\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

Solving **tD = c** we get $t_1=0, t_2=0$ From these solutions and **h = tU** we get:

$$i_1 = t_3$$

$$j_1 = t_4$$

$$i_2 = t_4$$

$$j_2 = t_3$$

The dependence distance is not constant.

From the lower limit for I_1, we have the inequality $i_1 \geq 1$ from which we derive

$$t_3 \geq 1$$

Likewise, from $j_1 \geq 1$ we derive

$$t_4 \geq 1$$

From the lower limit for I_2 we have $i_2 \geq i_1 + 1$ and $j_2 \geq j_1 + 1$ from which we derive

$$t_4 \geq t_3 + 1$$

and

$$t_3 \geq t_4 + 1$$

Note that we will adjust this last inequality to create an upper bound for t_4:

$$t_4 \leq t_3 - 1$$

After adding the inequalities for the upper limit expressions, we have the following lower and upper bounds for each free variable:

$$1 \leq t_3 \leq 100$$

$$\left.\begin{matrix} 1 \\ t_3+1 \end{matrix}\right\} \leq t_4 \leq \begin{cases} 100 \\ t_3 - 1 \end{cases}$$

Given a list of lower and upper bounds for each free variable, the Power Test visits each free variable (from t_n down to t_{s+1}) comparing each lower bound to each upper bound. Each comparison will be of the form:

$$(lb_0 + lb_1 t_1 + \cdots + lb_{k-1} t_{k-1}) / lb_k \leq t_k \leq (ub_0 + ub_1 t_1 + \cdots + ub_{k-1} t_{k-1}) / ub_k$$

from which we can derive:

$$(lb_k ub_0 - ub_k lb_0) + (lb_k ub_1 - ub_k lb_1) t_1 + \cdots + (lb_k ub_{k-1} - ub_k lb_{k-1}) t_{k-1} \geq 0$$

We assume that the floor and ceiling operators have already been handled as discussed above, if possible reducing the magnitude of lb_k and ub_k to one. If lb_k and ub_k are both one, there is no loss of precision here; if either is greater than one, there is a potential loss of precision, which

may result in finding a solution when in fact there is none. If any of the coefficients are non-zero, this will derive another lower or upper limit on another lower-numbered free variable. If all the coefficients are zero, then we have the simple inequality:

$$lb_k ub_0 - ub_k lb_0 \geq 0$$

If this inequality is not satisfied (if the left hand side is in fact negative), then there is no solution to the dependence system, and thus no dependence.

Example. In the previous example, when we compare each lower bound to each upper bound for t_4, we eventually compare

$$t_3+1 \leq t_4 \leq t_3-1$$

from which we derive the inequality:

$$t_3+1 \leq t_3-1, \quad \text{or} \quad 1 \leq -1$$

which is clearly inconsistent; thus, this system of equations and limits has no solution, and the two references are independent.

Direction Vectors. The Power Test can also be extended to test for particular direction vectors. Each direction vector element being tested corresponds simply to another inequality which derives a lower or upper bound on one of the free variables. Suppose for instance we are testing for a (<) in position k of the direction vector, and that $i_k \equiv h_{2k-1}$ and $j_k \equiv h_{2k}$. The (<) direction means we want to test for dependence when $i_k < j_k$, or $h_{2k-1} < h_{2k}$, or $h_{2k} - h_{2k-1} > 0$, or $h_{2k} - h_{2k-1} \geq 1$. We then replace the index variables by their formula in terms of the free variables:

$$\mathbf{t}U_{.,2k} - \mathbf{t}U_{.,2k-1} \geq 1$$

This will again derive either a lower or upper bound on one of the free variables, or will produce a simple constant inequality which can be tested for consistency.

Example. Study again one of the examples in the motivation:

```
for I1 = 1 to N do
  for I2 = I1+1 to N do
    A(I1) = A(I2)
  endfor
endfor
```

The dependence equation is:

$$i_1 - j_2 = 0$$

so the dependence matrix **hA=c** is:

$$(i_1, j_1, i_2, j_2) \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} = (0)$$

The Generalized GCD test produces:

$$\mathbf{UD} = \left(\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right)$$

Solving **tD=c** gives $t_1=0$, so that multiplying out **h=tU** gives:

$$i_1 = t_2$$

$$j_1 = t_3$$

$$i_2 = t_4$$

$$j_2 = t_2$$

From the loop limits we derive the lower and upper bounds on the free variables. For instance, from $i_2 \geq i_1+1$ we get $t_4 \geq t_2$, while $j_2 \geq j_1+1$ derives $t_2 \geq t_3+1$, or equivalently, $t_3 \leq t_2-1$. In this way we find the following bounds on the free variables:

$$1 \leq t_2 \leq N$$

$$1 \leq t_3 \leq \begin{cases} t_2-1 \\ N \end{cases}$$

$$t_2+1 \leq t_4 \leq N$$

With just the constraints implied by the loop limits, the system is still consistent. Now suppose we want to test for a particular dependence direction, such as the (<) direction in the first dimension. From $i_1<j_1$ we derive the additional bound:

$$t_3 \geq t_2+1$$

This is inconsistent with one of the previous bounds, so there can be no dependence with a (<) direction in the first dimension. Likewise, if we test for a (<) direction in the second dimension, meaning $i_2<j_2$, we derive the additional bound:

$$t_4 \leq t_2-1$$

which again is inconsistent with the lower bounds for t_4. Continuing in this way, we find that the only consistent direction vector is (>,>), which means that $i_1>j_1$ and $i_2>j_2$. This corresponds to a (<,<) direction for the negative of the dependence equation, which

corresponds to an anti-dependence [Wol89]. Thus, the Power Test correctly identifies the anti-dependence with the precise direction vector.

7. The Power of the Power Test

The previous section showed how the Power Test handled the first example from the motivation section. Let us see how the other examples are handled.

Multiple Loop Limits. As mentioned earlier, some program transformations generate multiple lower or upper limits. As a case in point, take the Gaussian Elimination program:

```
      for I1 = 1 to N do
        for I2 = I1+1 to N do
S1:       A(I2,I1) = A(I2,I1) / A(I1,I1)
          for I3 = I1+1 to N do
S2:         A(I2,I3) = A(I2,I3) - A(I1,I3)*A(I2,I1)
          endfor
        endfor
      endfor
```

Using advanced program restructuring techniques, we can reindex this to the following loop structure:

```
      for I1 = 1 to N do
        for I2 = 2 to N do
          for I3 = 1 to min(I1-1,I2-1) do
S2:         A(I2,I1) = A(I2,I1)-A(I3,I1)*A(I2,I3)
          endfor
        endfor
        for I4 = I1+1 to N do
S1:       A(I4,I1) = A(I4,I1)/A(I1,I1)
        endfor
      endfor
```

In this case, to test for dependence between $S_1: A(I_4, I_1)$ and $S_2: A(I_2, I_3)$, we have to somehow deal with the upper limit of the i_3 loop, which is the minimum of two simple expressions. We shall see that the Power Test handles this case effectively.

The dependence equations are:

$$i_4 - j_2 = 0$$

$$i_1 - j_3 = 0$$

The dependence matrix **hA=c** is:

$$(i_1, j_1, i_4, j_2, j_3) \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} = (0,0)$$

The Generalized GCD test returns the matrices:

$$\mathbf{UD} = \left(\begin{array}{ccccc|cc} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

From the equation $\mathbf{tD=c}$, we solve for $t_1=t_2=0$; the index variables are then defined as:

$$i_1 = t_3$$

$$j_1 = t_4$$

$$i_4 = t_5$$

$$j_2 = t_5$$

$$j_3 = t_3$$

Enforcing the lower and upper limits for each of the five index variables derives the following limits on the free variables:

$$1 \leq t_3 \leq n$$

$$\left.\begin{matrix} t_3+1 \\ 1 \end{matrix}\right\} \leq t_4 \leq n$$

$$\left.\begin{matrix} t_3+1 \\ 2 \end{matrix}\right\} \leq t_5 \leq n$$

In particular, the upper limit of i_3 derives two bounds:

$$j_3 \leq j_2-1 \text{ derives } t_3+1 \leq t_5$$

$$j_3 \leq j_1-1 \text{ derives } t_3+1 \leq t_4$$

When testing for a $(\geq)$ direction in the i_1 loop, we add the inequality $i_1 \geq j_1$ which derives

$$t_4 \leq t_3$$

which generates the inconsistency

$$t_3+1 \leq t_4 \leq t_3$$

The Power Test correctly decides that this dependence has only the (<) direction vector.

Non-Direction Vector Constraints. In the example above we skipped over the restructuring necessary to change one form of the Gaussian Elimination into the other. The series of steps is as follows: start with the normal KIJ form:

```
        for K = 1 to N do
          for I = K+1 to N do
S1:         A(I,K) = A(I,K) / A(K,K)
            for J = K+1 to N do
S2:           A(I,J) = A(I,J) - A(K,J)*A(I,K)
            endfor
          endfor
        endfor
```

We start by *distributing* the I loop. Loop distribution is legal if there are no dependence cycles; the only dependence cycle here is carried by the outer loop, so the I loop can be distributed:

```
        for K = 1 to N do
          for I = K+1 to N do
S1:         A(I,K) = A(I,K)/A(K,K)
          endfor
          for I = K+1 to N do
            for J = K+1 to N do
S2:           A(I,J) = A(I,J)-A(K,J)*A(I,K)
            endfor
          endfor
        endfor
```

Next we interchange the tightly nested I and J loops. The dependence test for loop interchanging is that there must be no dependence relations with (<,>) direction vectors in the loops being interchanged; this condition is satisfied here, so interchanging is legal:

```
        for K = 1 to N do
          for I = K+1 to N do
S1:         A(I,K) = A(I,K)/A(K,K)
          endfor
          for J = K+1 to N do
            for I = K+1 to N do
S2:           A(I,J) = A(I,J)-A(K,J)*A(I,K)
            endfor
          endfor
        endfor
```

The next step involves interchanging the J and K loops. Note that these loops are not tightly nested, and distribution of the K loop is not legal. We want to interchange these loops directly. Let us reindex the loops as:

```
      for I1 = 1 to N do
        for I2 = I1+1 to N do
S1:       A(I2,I1) = A(I2,I1)/A(I1,I1)
        endfor
        for I3 = I1+1 to N do
          for I4 = I1+1 to N do
S2:         A(I4,I3) = A(I4,I3)-A(I1,I3)*A(I4,I1)
          endfor
        endfor
      endfor
```

One condition that must be satisfied for legal interchanging is that there must be no dependence relation from iteration (j_1, j_3, j_4) of S_2 to iteration (i_1, i_2) of S_1 such that $j_1 < i_1$ and $j_3 > i_1$ [Wol89]. Let us inspect the dependence between $S_2: A(I_4, I_3)$ and $S_1: A(I_2, I_1)$. The dependence matrix **hA=c** is:

$$(i_1, j_1, i_2, j_3, j_4) \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \\ 0 & -1 \\ -1 & 0 \end{pmatrix} = (0,0)$$

The Generalized GCD test ends with:

$$\mathbf{UD} = \left(\begin{array}{ccccc|cc} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}\right)$$

The equation **tD=c** results in $t_1=0$ and $t_2=0$, so we have

$$i_1 = t_3$$

$$j_1 = t_4$$

$$i_2 = t_5$$

$$j_3 = t_3$$

$$j_4 = t_5$$

The loop limits bounds the free variables as follows:

$$1 \le t_3 \le N$$

$$\left.\begin{matrix} t_3+1 \\ t_4+1 \end{matrix}\right\} \le t_5 \le N$$

$$1 \le t_4 \le \begin{cases} N \\ t_3-1 \end{cases}$$

It is easy to see that testing for $j_1 \ge i_1$ would add the constraint $t_4 \ge t_3$, which is inconsistent. Thus, j_1 must be less than i_1; this corresponds to the dependence $S_2 \ \overline{\delta}_{(<)} \ S_1$. Also, if we add the constraint $j_3 > i_1$ we get the inconsistency $t_3 > t_3$, so this dependence does not prevent interchanging. Moreover, after interchanging, the loops around S_2 will be reordered to (j_3, j_1, j_4); thus after interchanging, the direction vector should be the sign of the difference $i_1 - j_3$. It is easy to realize that this is $t_3 - t_3 = 0$. Thus, the Power Test is easily extended to perform non-direction-vector tests.

Handling Equal Directions. Special handling of the (=) direction is needed for the Power Test to be precise. Take the example at the end of section 4. That example ended with the equation **tD=c**:

$$(t_1, t_2, t_3, t_4) \begin{pmatrix} -1 & 2 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} = (2, 1)$$

from which we computed $t_1 = -2$ and $t_2 = 5$. From the equation **j=tU** we get:

$$(i_1, j_1, i_2, j_2) = (-2, 5, t_3, t_4) \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 4 & 3 & -3 \\ -2 & -8 & -7 & 7 \\ 0 & -2 & -2 & 3 \end{pmatrix}$$

from which we compute

$$i_1 = -2t_3 + 5$$

$$j_1 = -8t_3 - 2t_4 + 18$$

$$i_2 = -7t_3 - 2t_4 + 15$$

$$j_2 = 7t_3 + 3t_4 - 15$$

The loop limits give the bounds:

$$-47 \leq t_3 \leq 2$$

$$\left.\begin{matrix}\lceil(-8t_3-82)/2\rceil \\ \lceil(-7t_3-85)/2\rceil \\ \lceil(-7t_3+16)/3\rceil\end{matrix}\right\} \leq t_4 \leq \left\{\begin{matrix}\lfloor(-8t_3+17)/2\rfloor \\ \lfloor(-7t_3+14)/2\rfloor \\ \lfloor(-7t_3+115)/3\rfloor\end{matrix}\right.$$

The limits for t_4 can be simplified to:

$$\left.\begin{matrix}-4t_3-41 \\ \lceil(-7t_3-85)/2\rceil \\ \lceil(-7t_3+16)/3\rceil\end{matrix}\right\} \leq t_4 \leq \left\{\begin{matrix}-4t_3+8 \\ \lfloor(-7t_3+14)/2\rfloor \\ \lfloor(-7t_3+115)/3\rfloor\end{matrix}\right.$$

Suppose we now want to test for the (=) direction in the first dimension, so we want to test for dependence under the condition $i_1=j_1$. Since the Power Test deals with inequalities, one way to test for an (=) direction is with two inequalities $i_1 \leq j_1$ and $j_1 \leq i_1$. These two inequalities give rise to the bounds:

$$\lceil(-6t_3+13)/2\rceil \leq t_4 \leq \lfloor(-6t_3+13)/2\rfloor$$

which can be simplified to:

$$-3t_3+7 \leq t_4 \leq -3t_3+6$$

It is immediately obvious that these bounds are inconsistent, so there is no dependence with an (=) direction for that loop. Using this approach for the second loop, we would have the inequalities:

$$\lceil(-14t_3+30)/5\rceil \leq t_4 \leq \lfloor(-14t_3+30)/5\rfloor$$

from which the floor and ceiling operators cannot be trivially eliminated. The Power Test would then ignore them, giving rise to potential imprecision.

Another way to test for the (=) direction is to set $i=j$, and solve for one of the free variables; in the first case, $i_1=j_1$, we get:

$$-6t_3+13 = 2t_4$$

By the GCD test, the GCD(6,2) must divide 13 for an integer solution; since it does not, there can be no dependence with an (=) direction here. In the second loop, $i_2=j_2$, we get:

$$-14t_3+30 = 5t_4$$

Here the GCD test gives no information. Instead, we scale the bounds of t_4 by a factor of 5

and replace $5t_4$ by its equivalent expression:

$$\left.\begin{matrix} -20t_3-205 \\ \lceil(-35t_3-425)/2\rceil \\ \lceil(-35t_3+80)/3\rceil \end{matrix}\right\} \leq 5t_4 = -14t_3+30 \leq \left\{\begin{matrix} -20t_3+40 \\ \lfloor(-35t_3+70)/2\rfloor \\ \lfloor(-35t_3+575)/3\rfloor \end{matrix}\right.$$

From this system of inequalities we can derive other bounds on t_3, such as $t_3 \geq -39$ from $-20t_3-205 \leq -14t_3+30$.

A third (expensive) method to handle an (=) direction is to build a reduced set of dependence equations using the equality $i_2=j_2$:

$$4i_1-j_1-4i_2=2$$
$$-i_1+2j_1-2i_2=1$$

with the dependence matrix **hA=c**:

$$(i_1, j_1, i_2)\begin{pmatrix} 4 & -1 \\ -1 & 2 \\ -4 & -2 \end{pmatrix} = (2, 1)$$

The Generalized GCD algorithm would give the result **UD**:

$$\left(\begin{array}{ccc|cc} 0 & 1 & 0 & -1 & 2 \\ 3 & 4 & 2 & 0 & 1 \\ 10 & 12 & 7 & 0 & 0 \end{array}\right)$$

Solving **tD=c** results in the assignments:

$$i_1=10t_3+15$$
$$j_1=12t_3+18$$
$$i_2=j_2=7t_3+10$$

The bounds on t_3 are:

$$\left.\begin{matrix} \lceil -14/10\rceil \\ \lceil -9/7\rceil \\ \lceil -11/12\rceil \end{matrix}\right\} \leq t_3 \leq \left\{\begin{matrix} \lfloor 85/10\rfloor \\ \lfloor 90/7\rfloor \\ \lfloor 81/12\rfloor \end{matrix}\right.$$

which simplifies to $-1 \leq t_3 \leq 6$. Indeed, when $t_3=-1$ we have $S_1[5,3]\ \delta\ S_2[6,3]$ for array element $A(13,-11)$, and when $t_3=6$ we have $S_1[75,52]\ \delta\ S_2[90,52]$ for array element $A(195,-179)$. This is related to the technique of taking advantage of the equal direction for the standard GCD test, as shown in [AlK87].

Handling Unknown Variables. Unknown variables occur in some subscript functions or in loop limits. The Power Test can handle many of these cases naturally by treating the unknown variables as additional index variables which have no limits. For instance, suppose we are

computing dependence in the program:

```
        for I1 = 1 to 100
          for I2 = 1 to N
S1:         A(I1,I2) = . . .
          endfor
          for I3 = N+1 to 100
S2:         . . .  = A(I1,I3)
          endfor
        endfor
```

Rather than ignoring the information in the loop limits where unknown variables occur, or treating this as a special case, we can treat the unknown variable N as an additional index variable and build the dependence system. The dependence equations, in matrix form, are:

$$(N, i_1, j_1, i_2, j_3) \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} = (0,0)$$

The Generalized GCD returns with **UD**:

$$\left(\begin{array}{ccccc|cc} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right)$$

and ends up with the equations:

$N=t_3$
$i_1=j_1=t_4$
$i_2=j_3=t_5$

Since the value of N is unknown, there is no bound on t_3; the bounds of the other free variables are:

$$1 \le t_4 \le 100$$

$$\left.\begin{matrix} 1 \\ t_3+1 \end{matrix}\right\} \le t_5 \le \left\{\begin{matrix} 100 \\ t_3 \end{matrix}\right.$$

The bounds for t_5 are inconsistent, so the Power Test, without any special handling of this case, correctly detects independence.

Another example of where this applies would be a program such as:

```
for I = 1 to N
  A(I) =  · · ·
   · · ·  = A(I+N)
endfor
```

which generates the dependence matrix:

$$(N, i_1, j_1)\begin{pmatrix}-1\\1\\-1\end{pmatrix} = 0$$

The Generalized GCD produces **UD**:

$$\left(\begin{array}{ccc|c}1&0&0&-1\\1&1&0&0\\0&1&1&0\end{array}\right)$$

and the equations:

$N=t_2$
$i_1=t_2+t_3$
$j_1=t_3$

The loop limits for i_1 are $1 \leq i_1 \leq n$, which produce $1 \leq t_2+t_3 \leq t_2$, or

$$1-t_2 \leq t_3 \leq 0$$

while the loop limits for j_1 produce

$$1 \leq t_3 \leq t_2$$

which are clearly inconsistent; thus the two references are independent. Again, the Power Test handles this without resorting to special case analysis.

Another important case arises when some unknown variable other than a loop limit appears in a subscript function:

```
for I = 1 to N
  A(I) =  · · ·
   · · ·  = A(I+INC)
endfor
```

Depending on the sign of INC, this could correspond to a flow or an anti-dependence. Some parallelizing compilers now accept *assertions* about the sign of INC, such as

```
assert relation ( INC > 0 )
```

The Power Test, by treating INC as another index variable and adding the assertion to the system of inequalities, will naturally handle this case.

Not all unknown variables can be handled by this method; when an index variable is multiplied by some unknown value, treating the unknown value as another index variable will make the subscript function appear nonlinear:

```
for I = 1 to N
  A(I) =  . . .
   . . .  = A(INC*I)
endfor
```

This is also true when the loop increment value is unknown; in fact, if the loop limits and increment are all unknown values, then the compiler can't even tell if the loop counts up or down (for some languages), so any dependence test (including the Power Test) is severely limited.

8. Comparison with Other Methods

There exists a large body of work in the field of dependence tests. These tests may be categorized based on their precision and on whether they test multiple subscripts simultaneously. Here we compare the Power Test with several single subscript exact tests, as well as two multi-dimensional tests, the Lambda test and the Constraint Matrix test.

8.1. Single Subscript Exact Tests

Since testing linear subscripts for dependence is equivalent to finding simultaneous integer solutions within loop limits, one approach is to employ integer programming techniques such as [DaE73,Fea88]. There are even integer programming algorithms based on Fourier-Motzkin elimination [Wil76,Wil83]. Unfortunately, these tests are expensive since integer programming is NP-complete [Coo71].

In the realm of exact dependence tests, there is a Single Index Variable exact test for simple subscripts with the same index variable in each reference [Ban79,Wol82]. Work has also been done towards proving exactness conditions for Banerjee's Inequalities. Banerjee showed that his inequalities are exact if the coefficients of the index variables are all 1, 0, or -1 [Ban76]. Li et al recently showed that the Banerjee's Inequalities are exact if the coefficient of one index variable is +1 or -1, and the magnitudes of all other coefficients are less than the range (determined by the loop limits) for that index variable [LY90]. Klappholz et al proved that the Banerjee's Inequalities are exact if and only if the coefficient of one index variable is +1 or -1, and there exists a permutation of the remaining index variables such that (loosely stated) the coefficient of each index variable is less than the sum of the products of the coefficients and ranges for all the previous index variables [KKP90a].

The I Test is a new single subscript test for dependence testing based on refining a combination of the GCD and Banerjee tests [KKP90b]. It is better able to detect the lack of integer solutions in more cases than simply applying both the GCD and Banerjee tests, and can usually prove the existence of integer solutions.

When comparing the Power Test with these tests, we note that the Power Test totally subsumes the Single Index Variable exact test. In addition, we believe that it will always be as precise as any combination of GCD or Banerjee tests. The Power Test handles complex loop limits, as well as multiple dimensions simultaneously. Since exactness conditions for Banerjee's Inequalities have not been proven for complex loop limits, and in general cannot be extended to multiple subscripts, the Power Test can be considered to be more precise than these tests. However, for the sake of efficiency it is desirable to employ simpler tests where they are known to be exact.

8.2. Lambda Test

The Lambda test is introduced in [LY90]. Its precision is equivalent to a multi-dimensional version of Banerjee's Inequalities, since it checks for simultaneous real-valued solutions for all subscripts within the loop limits. Like Banerjee's Inequalities, it can also be used to test for direction vectors. The Lambda test is applied to "coupled subscripts", which are groups of subscript sharing identical index variables. The test proceeds by selectively forming linear combinations of these subscripts and testing the result. The linear combinations selected are exactly those which eliminate one or more instances of index variables. The authors prove that when all such combinations have been generated and tested with Banerjee's Inequalities, real-valued solutions exist if and only if they exist in in all the linear combinations tested.

We can enhance the precision of the Lambda test for detecting independence quite simply as follows. When testing each linear combination, the Lambda test uses Banerjee's Inequalities. However, where the exactness condition for Banerjee's Inequalities do not hold, we may apply the GCD or Single Index Variable tests as appropriate. This will allow us to improve the ability of the Lambda test to detect some cases where no integer solutions exist within the loop limits, even though real valued solutions do.

Unfortunately, there is no obvious method to enhance the Lambda test to prove the existence of simultaneous integer solutions. The Lambda test is not exact even when exact single subscript tests may be applied to all linear combinations generated, since the presence of constrained integer solutions in all linear combinations does not guarantee simultaneous constrained integer solutions. For two coupled subscripts, Li et al were able to prove that the Lambda test is exact if unconstrained integer solutions exist and the coefficients of index variables are all +1, 0 or -1. Even with these restrictions, they showed that the Lambda test is not always exact for three or more coupled subscripts [LY90].

Precision Comparison. When compared with the Power Test, we show that the Power Test will detect independence whenever the Lambda test does, even with our suggested

enhancements. First of all, the Power Test is just as precise in detecting the presence of simultaneous real-valued solutions, since it solves the linear programming problem exactly. As the Power Test subsumes the Generalized GCD test, it will also capture all the cases where applying the GCD test detects the lack of unconstrained integer solutions. Finally, the ability of the Power Test to create a dense solution space guarantees that it succeeds in all the cases where applying the Single Index Variable test would show independence in the Lambda test.

On the other hand, we can show that the the Power Test will be able to detect the lack of simultaneous integer solutions where the Lambda test cannot, even when the Power Test cannot apply exact floor and ceiling operators. We show an example where the Power Test is more precise than the Lambda test:

```
for I1 = 1 to 100 do
  for I2 = 1 to 100 do
    A(3*I1 + 2*I2, 2*I2) = A(I1 - I2 + 6, I1 + I2)
  endfor
endfor
```

The dependence equations are:

$$EQ1: 3i_1 - j_1 + 2i_2 + j_2 = 6$$
$$EQ2: - j_1 + 2i_2 - j_2 = 0$$

The linear combinations (set of canonical solutions) that the Lambda test would generate are as follows:

Combination	equation	eliminates	dependence equation
C_1	EQ2	i_1	$-j_1+2i_2-j_2=0$
C_2	EQ1-EQ2	j_1,i_2	$3i_1+2j_2=6$
C_3	EQ1+EQ2	j_2	$3i_1-2j_1+4i_2=6$

In all cases the GCD test fails to detect independence since the GCD of the coefficients is one. Banerjee's inequalities applied to the three combinations give rise to the comparisons:

$$C_1 \quad -198 \leq 0 \leq 198$$
$$C_2 \quad 5 \leq 6 \leq 500$$
$$C_3 \quad -193 \leq 6 \leq 698$$

For all three combinations, Banerjee's show that real-valued solutions exist with the loop limits. Since the GCD test and Banerjee's inequalities fail for all three combinations, the Lambda test would assume dependence.

In comparison, when we apply the Power Test, we get the dependence matrix $\mathbf{hA} = \mathbf{c}$:

$$(i_1, j_1, i_2, j_2) \begin{pmatrix} 3 & 0 \\ -1 & -1 \\ 2 & 2 \\ 1 & -1 \end{pmatrix} = (6, 0)$$

The Generalized GCD test returns the matrices:

$$\mathbf{UD} = \left(\begin{array}{cccc|cc} 0 & 0 & 0 & 1 & 1 & -1 \\ 1 & 1 & 0 & -2 & 0 & 1 \\ 2 & 3 & 0 & -3 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 \end{array}\right)$$

From the equation $\mathbf{tD} = \mathbf{c}$ we solve for $t_1 = t_2 = 6$; the index variables are then defined as:

$$i_1 = 2t_3 + 6$$

$$j_1 = 3t_3 + 2t_4 + 6$$

$$i_2 = t_4$$

$$j_2 = 3t_3 - 6$$

Examining just the lower limits for each of the four index variables derives the following limits on the free variables:

$$\begin{array}{lcrcl} i_1 \geq 1 & \rightarrow & 2t_3+6 \geq 1 & \rightarrow & t_3 \geq \lceil -5/2 \rceil \\ j_1 \geq 1 & \rightarrow & 3t_3+2t_4+6 \geq 1 & \rightarrow & t_4 \geq \left\lceil (-3t_3-5)/2 \right\rceil \\ i_2 \geq 1 & \rightarrow & t_4 \geq 1 & \rightarrow & t_4 \geq 1 \\ j_2 \geq 1 & \rightarrow & -3t_3-6 \geq 1 & \rightarrow & t_3 \leq \lfloor -7/3 \rfloor \end{array}$$

Examining the bounds for t_3, we see that

$$\lceil -5/2 \rceil \leq t_3 \leq \lfloor -7/3 \rfloor$$

which derives the inconsistent condition $-2 \leq t_3 \leq -3$, proving that no simultaneous integer solutions exist. In fact, the Power Test would have detected independence even if the loop upper limits were unknown symbolic expressions, since it only needed to use the loop lower limits.

Limitations. The Lambda test assumes that no subscript tested can be formed by a linear combination of other subscripts. This requires first performing Gaussian elimination to detect redundant subscripts. The Power Test doesn't assume independence between subscripts; dependent subscripts are eliminated during the Generalized GCD test.

Both the Lambda test and the Power Test may be used to calculate full direction vectors, though there is no discussion in the literature on how the Lambda test may be used to generate distance vectors.

Since it is based on Banerjee's inequalities, the Lambda test is unable to handle complex loop limits that use min and max functions that may be introduced by advanced loop

interchanging. In addition, the precision of the Lambda test for simple triangular or trapezoidal loops has not been discussed in the literature. We show with an example that the Lambda test is less precise than the Power Test for non-rectangular loops. In the following triangular loop, the Lambda test cannot detect that there is no dependence (because there is only one iteration of the I_2 loop when $I_1 = 100$).

```
for I1 = 1 to 100 do
  for I2 = I1 to 100 do
    A(I1, I2+1) = A(100, I2)
  endfor
endfor
```

First of all, the Lambda test may not even consider the two subscripts to be coupled, since there are no shared index variables. Even if it was applied, there is no way for the Lambda test to propagate the constraint on I_1 from the first subscript into the bounds for I_2 in the second subscript. This example also shows that detecting simultaneous solutions depends on more than just coupled subscript functions for non-rectangular loops.

Complexity Comparison. As with the Power Test, the Lambda test is an exponential cost algorithm for the general case. The expensive part of the test is in the number of linear combinations formed. Given that there are n index variables and s subscripts, there are up to $\binom{n}{s-1}$ possible linear combinations which must be created and tested. This clearly grows exponentially with respect to the number of index variables in general. However, if the number of subscripts is kept low, the Lambda test is quite efficient. For instance, with only two coupled subscripts, the number of linear combinations to be tested grows only linearly with the number of index variables. The cost of the Lambda test grows much faster with respect to the number of subscripts (up to $n/2$ subscripts)

In comparison, the expensive part of the Power Test is in checking the convex hull through Fourier-Motzkin elimination. The cost of this step is exponential with respect to the number of index variables, but actually decreases with respect to the number of subscripts. If efficiency was the only factor to be considered, it seems that the Power Test would be preferable for references with large numbers of subscripts and few index variables.

8.3. Constraint Matrix Test

The Constraint Matrix test is a modified simplex algorithm for solving integer programming problems, presented in [Wal88]. Instead of first parameterizing the system and then checking the consistency of the loop limits as in the Power Test, the algorithm introduces slack variables for each constraint and adds them to the system. The Constraint Matrix test then iteratively reduces rows in the system using a reduction row pivot method, until the test either

converges or detects the lack of solutions. Since cycling may result for degenerate cases, the Constraint Matrix test also halts after a fixed number of iterations and conservatively assumes dependence.

Limitations. Like the Lambda test, the Constraint Matrix algorithm requires that all subscripts be independent. Gaussian elimination must thus be performed as a preliminary step. Although not directly stated in [Wal88], the Constraint Matrix test may compute full direction vectors by introducing new slack variables for each direction. This requires that the test be applied from scratch for each direction vector tested. The Constraint Matrix test does not compute distance vectors. [Wal88] also does not mention complex loop limits, but the same techniques we present for the Power Test may be applied to the Constraint Matrix test as well.

Comparison. The Constraint Matrix test is a multi-dimensional test, and is guaranteed to detect the lack of simultaneous real-valued solutions (when cycling does not occur). However, it is not an exact test, and it is not even clear that it matches the ability of the Generalized GCD test to detect simultaneous unconstrained integer solutions. In addition, the inability of the Constraint Matrix Test to detect cycling forces it to impose an arbitrary limit on the number of iterations allowed. This has an unknown impact on the precision of the test, and makes it difficult to compare the Constraint Matrix test with the Power Test, especially in its ability to detect the lack of simultaneous integer solutions.

Since the Constraint Matrix is based on the simplex algorithm, it also has worst case exponential complexity. For most real linear programming problems, simplex algorithms tend to have near linear time complexity, and cycling is rare. However, [Sch86] states that for combinatorial problems, where coefficients tend to be 1, 0, or -1, the simplex algorithm is slow and tends to cycle for certain pivot rules.

At this point, more studies are required to characterize the behavior of the the dependence tests we have examined. In the end, since the actual number of both index variables and subscripts is likely to be small, only experimental results will indicate which test is more efficient.

9. Proof of the Power Test

This section proves two important theorems about the Power Test. First, we prove that the Power Test is conservative; that is, the Power Test will never claim independence if there are simultaneous integer solutions that satisfy the constraints of the dependence system. Second, we prove that in many well-defined cases, the Power Test is exact; that is, in many cases it will claim a solution to the dependence system only if it can prove that there are

simultaneous integer solutions that satisfy the constraints. Moreover, it has a simple mechanism to distinguish when it is precise and when it is not.

We distinguish two sources of imprecision; in the first case, the dependence system itself may not be a precise characterization of the data dependence problem. If the subscript functions are not linear combinations of the index variables, then the dependence system cannot be built; a compiler using the Power Test may assume dependence in these cases even when the references are independent. Unknown variables, in loop limits or in the subscript functions, can also cause imprecision, as in the case:

```
for I = 1 to N do
  for J = M to 100 do
    A(I,J) = A(J,I)
  endfor
endfor
```

The relative values of M and N, will determine whether references are or are not dependent. Since the dependence system cannot characterise the relative values, the system will be imprecise; the Power Test (or any other solution method for the dependence system) will assume dependence even if they are in fact independent. Some compilers use special case analysis to generate code that detects at run time whether there is or is not a dependence relation [BDH87], and execute different code if there is not. Although the dependence system may be imprecise, it is always conservative; that is, if there is an actual solution to the data dependence problem, that solution will also appear as a solution to the dependence system. The dependence system may be imprecise in that a solution to the dependence system may not correspond to an actual solution to the data dependence problem, as shown above. In this section, we show that the Power Test will always conservatively solve the dependence system, and will sometimes exactly solve the system. We recognize that the dependence system itself may be imprecise, but that is beyond the scope of this work.

In the Power Test, a second source of imprecision occurs when the floor or ceiling operators are ignored to solve the system of inequalities. This imprecision arises from trying to solve an integer system of inequalities with the Fourier-Motzkin method for linear programming. However, it is also easy for the Power Test to detect when a floor or ceiling operator has been ignored; when a tool using the Power Test reports to the user the presence of a parallelism-restricting dependence relation, the tool can also tell the user how confident it is that the dependence actually exists.

Banerjee's Generalized GCD Algorithm starts by filling an $n \times s$ coefficient matrix **A** with the coefficients of the subscript functions. The goal is to find whether there is an integer vector that solves the dependence system, $\mathbf{hA}=\mathbf{c}$. The algorithm finds an $n \times s$ upper triangular matrix **D** and a $n \times n$ unimodular matrix **U** that satisfy $\mathbf{UA}=\mathbf{D}$. If an integer vector **t** can be found

such that **tD=c**, then **h=tU** is a solution to the dependence system. Banerjee proved this algorithm correct. Solving **tD=c** actually solves for t_1 through t_s, leaving only t_{s+1} through t_n as free variables; if there is no integer solution to **tD=c**, then there is no integer solution to the original dependence equations, regardless of the loop limits. If there is a solution, then there is an integer solution somewhere, but it may or may not be within the loop limits. All integer solutions to the dependence equations can be enumerated by letting the free variables $t_{s+1}, \ldots, t_n$ range through the integers (any integer value of the free variables derives a solution to the dependence equations).

Multiplying **h=tU** gives **h** in terms of **t**. These can be substituted into the loop limit and direction vector inequalities to get inequalities relating the free variables. In the Power Test, we rearrange each inequality to be an upper or lower bound on the highest numbered free variable with a non-zero coefficient. This gives us potentially a list of upper and lower bounds for for free variable t_{s+1} through t_n. Each upper and lower bound will be expressed as a linear combination of lower-numbered free variables. For instance, each lower bound for t_k will be of the form:

$$lb_{k,x,k}t_k \geq lb_{k,x,0}+lb_{k,x,s+1}t_{s+1}+\cdots+lb_{k,x,k-1}t_{k-1} \quad (1)$$

and each upper bound of the form:

$$ub_{k,y,k}t_k \leq ub_{k,y,0}+ub_{k,y,s+1}t_{s+1}+\cdots+ub_{k,y,k-1}t_{k-1} \quad (2)$$

The first subscript of each lb or ub coefficient is the free variable for which this is a bound, the last subscript is the free variable for which this is a coefficient, and the middle subscript x ranges over the number of lower bounds (and y over the number of upper bounds) for t_k. Note that $lb_{k,x,k}>0$ and $ub_{k,y,k}>0$, by construction.

These bounds express the boundaries of the solution space of the dependence system in $(n-s)$-space exactly; that is, if there are any integer points in the $(n-s)$-dimensional convex region bounded by these inequalities, those integer points are values of the free variables that will generate (integer) values of the index variables which will solve the original dependence system. If the original dependence system is exact, this solution will be exact.

The first question is whether the Power is always conservative; that is, is there a case in which there is in fact an integer solution but the Power Test will (incorrectly) show independence. Duffin [Duf74] shows in his Lemma 1 that Fourier-Motzkin pairwise elimination works. We reproduce the statement of his Lemma 1 here:

Lemma 1 (Duffin).

> Pairwise elimination of the variable x_1 from a system of linear inequalities gives an eliminant system of linear inequalities. Then $x'_2, \ldots, x'_m$ is a solution to the eliminant

system if and only if there is an x_1' such that $x_1', x_2', \ldots, x_n'$ is a solution of the original system.

Proof.

See [Duf74].

Given that the set of linear inequalities of the form shown in (1-2), if there is an integer solution to the inequalities then there must be a real solution; approximating the integer solution by a real solution is LP-relaxation. By Lemma 1, pairwise elimination to remove the one of the free variables, say t_k, will generate an eliminant system which will have real solutions if and only if the original system had real solutions. Thus, if the original system had an integer solution, the eliminant system will have a real solution. This gives us the first theorem about the Power Test.

Theorem 1.

If the Power Test relaxes all the floor and ceiling operators, then it will be conservative; that is, it will not assert independence when there is in fact an integer solution.

Proof.

Immediately from Lemma 1.

While this is nice, we are really interested only in integer solutions. We use the following two Lemmas.

Lemma 2.

Given an inequality of the form (1) or (2), where all the coefficients are integers. This inequality has integer solutions for the free variables if and only if the corresponding inequality (1') or (2') has integer solutions.

$$t_k \geq \left\lceil (lb_{k,x,0} + lb_{k,x,s+1} t_{s+1} + \cdots + lb_{k,x,k-1} t_{k-1}) / lb_{k,x,k} \right\rceil \qquad (1')$$

$$t_k \leq \left\lfloor (ub_{k,y,0} + ub_{k,y,s+1} t_{s+1} + \cdots + ub_{k,y,k-1} t_{k-1}) / ub_{k,y,k} \right\rfloor \qquad (2')$$

Proof.

Directly from the properties of integers.

Lemma 3.

If $lb_{k,x,m}/lb_{k,x,k}$ is integer for $s+1 \leq m \leq k-1$, then an inequality of the form (1') is equivalent to:

$$t_k \geq \left\lceil \frac{lb_{k,x,0}}{lb_{k,x,k}} \right\rceil + \frac{lb_{k,x,s+1}}{lb_{k,x,k}} t_{s+1} + \cdots + \frac{lb_{k,x,k-1}}{lb_{k,x,k}} t_{k-1} \qquad (1'')$$

Similarly, if $ub_{k,y,m}/ub_{k,y,k}$ is an integer for $s+1 \leq m \leq k-1$, then an inequality of the form (2') is equivalent to:

$$t_k \leq \left\lfloor \frac{ub_{k,y,0}}{ub_{k,y,k}} \right\rfloor + \frac{ub_{k,y,s+1}}{ub_{k,y,k}} t_{s+1} + \cdots + \frac{ub_{k,y,k-1}}{ub_{k,y,k}} t_{k-1} \qquad (2'')$$

Proof.

Directly from the properties of integers.

The following theorem is the claim for the proof of correctness of the Power Test, when some or all of the floor and ceiling operators are computable.

Theorem 2.

When the Power Test exercises floor and ceiling operators as in Lemma 3, it is conservative.

Proof.

The Power Test starts with a linear system of inequalities of the form (1) and (2). Any integer solution to these inequalities generates a solution to the dependence system.

At each elimination step k in the Power Test, where k ranges from n down to $s+1$, the Power Test eliminates one free variable, t_k. Step k starts with a set of inequalities of the form (1) and (2), which we call the primary set for step k, or $\texttt{Primary}_k$. Examine the inequalities bounding t_k. Suppose that the floor and ceiling operators cannot be exercised as described in Lemmas 2 and 3. Elimination of t_k via pairwise elimination will generate an eliminant system, not involving t_k, which will have solutions if and only if the "original" system had solutions; the "original" system in this case is $\texttt{Primary}_k$. Thus, if there is an integer solution to $\texttt{Primary}_k$, the eliminant system will also have that same integer solution. Thus, we need only look for integer solutions in the eliminant system. Let $\texttt{Primary}_{k-1}$ be this eliminant system, and proceed by induction.

Suppose instead that one or more floor and ceiling operators of t_k can be exercised. By Lemma 2, since we only want integer solutions of t_k, we can convert the inequalities for which the floor or ceiling operators can be exercised to the form (1') and (2'), and by Lemma 3 and the premise that the floor and ceiling operators can be exercised, these replacement inequalities can then be converted to the form (1") and (2"), replacing the original inequalities. After performing the replacement, we have the secondary set of inequalities for step k, called $\texttt{Secondary}_k$. $\texttt{Secondary}_k$ has potentially a smaller real solution space than the primary set, but the same integer solution space. Let

$$lb'_{k,x,m} = lb_{k,x,m}/lb_{k,x,k}, \quad s+1 \le m \le k-1, \ \forall x$$

$$lb'_{k,x,0} = \left\lceil lb_{k,x,0}/lb_{k,x,k} \right\rceil \forall x$$

$$ub'_{k,y,m} = ub_{k,y,m}/ub_{k,y,k}, \quad s+1 \le m \le k-1, \ \forall y$$

$$ub'_{k,y,0} = \left\lfloor ub_{k,y,0}/ub_{k,y,k} \right\rfloor \forall y$$

Then (1") and (2") are equivalent to the linear system of inequalities:

$$t_k \ge lb'_{k,x,0} + lb'_{k,x,s+1} t_{s+1} + \cdots + lb'_{k,x,k-1} t_{k-1} \qquad (1")$$

$$t_k \le ub'_{k,y,0} + ub'_{k,y,s+1} t_{s+1} + \cdots + ub'_{k,y,k-1} t_{k-1} \qquad (2")$$

We have taken advantage of the properties of integers to change the coefficient of t_k to one in some (perhaps all) of its lower and upper bounds, and to (possibly) slightly reduce the size of the convex region of the solution space for t_k. By Lemma 1, eliminating t_k by pairwise elimination will generate an eliminant system, not involving t_k, which will have (real, and hence integer) solutions if and only if the "original" set of inequalities had solutions; the "original" set of inequalities in this case is $\texttt{Secondary}_k$. Thus, if there is an integer solution to $\texttt{Secondary}_k$, the eliminant system will also have an integer solution. Thus, we need only look for integer solutions in the eliminant system, which becomes $\texttt{Primary}_{k-1}$.

By induction, we see that if there is an integer solution to $\texttt{Primary}_n$, then there will be an integer solution to $\texttt{Primary}_s$. Thus, the Power Test will not assert independence if the original dependence system in fact has an integer solution, whether or not some or all of the floor or ceiling operators are exercised.

The final theorem shows when the Power Test is exact.

Theorem 3.

When the Power Test exercises all floor and ceiling operators as in Lemma 3, it will find an integer solution if and only if there is an integer solution to the original dependence system.

Proof of "if".

By Theorem 2, exercising floor and ceiling operators is conservative.

Proof of "only if".

The procedure shown above in Theorem 2 generates a set of simple affine lower and upper bounds for each free variable. If the Power Test asserts dependence, then the bounds for the final free variable, t_{s+1}, will be simple integers:

$$t_{s+1} \ge \max\left(lb'_{k,x,0} \mid \forall x\right)$$

$$t_{s+1} \leq \min(ub'_{k,y,o} \mid \forall y)$$

(if some limits are unknown, then t_{s+1} may be unbounded in one or both directions). Thus, t_{s+1} has at least one, and perhaps many, known integer values within its bounds; choose one such value, say $\bar{t}_{s+1}$. Use the value $\bar{t}_{s+1}$ to find lower and upper bounds for t_{s+2}; these will be integers, since the bounds for t_{s+2} are simple linear combinations of t_{s+1} with integer coefficients. By Lemma 1, any value in the solution range of t_{s+1} can be used in the bounds of t_{s+2} and will generate a non-empty solution space. Since the lower and upper bounds must be integer, there must be at least one integer value of t_{s+2} in this range; choose one, say $\bar{t}_{s+2}$. In such a way, we can find integer values for each of the free variables, $\bar{t}_{s+1}$, $\bar{t}_{s+2}$, ..., $\bar{t}_n$. By the Generalized GCD, any integer value of the free variables satisfies the dependence equations. By Lemma 1, only those values that lie within the loop limit and direction vector inequalities will appear within the final set of bounds. Thus, this set of integer values will generate an integer solution to the dependence system that satisfies all the inequalities.

10. Conclusions

The Power Test can be useful in advanced program restructuring techniques. Since it is based on Banerjee's Generalized GCD test, it is close to the holy grail of solving simultaneous subscript equations only for integer solutions within the loop limits. It loses some precision because it might ignore pertinent ceiling and floor operators. This precision loss is equivalent to enlarging the solution space somewhat; in other words, it may return a false positive if there is an integer solution *near* the limits of the loop, or near the bounds imposed by other constraints such as direction vector relations. The Power Test is also extensible beyond most other dependence decision algorithms, allowing non-direction vector tests and simultaneous multiple upper and lower loop limits.

The obvious consideration when implementing the Power Test is the execution cost. The worst case cost of the search procedure can be exponential in the number of free variables. This cost may be too high for inclusion in a critical component such as a compiler, but may be appropriate when applying certain "power transformations" in an interactive environment.

11. Acknowledgements

The authors thanks Jaspal Subhlok, Paul Havlak and Ken Kennedy of Rice University, Utpal Banerjee of Intel Corporation, and David Callahan of Tera Computer Corporation for their comments and discussions during the preparation of this paper.

This work was supported in part by NSF Grants CCR-8906909 and CCR-8809615, by DARPA Grant MDA972-88-J-1004, and by the Cray Research Foundation.

References

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction*, New York, June 1984, 233-246.

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542.

[ABC87] F. Allen, M. Burke, P. Charles, R. Cytron and J. Ferrante, An Overview of the PTRAN Analysis System for Multiprocessing, in *Supercomputing, 1st International Conference*, vol. 297, Springer-Verlag, Berlin, 1987, 194-211.

[Ban76] U. Banerjee, Data Dependence in Ordinary Programs, UIUCDCS-R-76-837, Univ. Illinois, Dept. Computer Science, Urbana, IL, November 1976.

[Ban79] U. Banerjee, *Speedup of Ordinary Programs*, PhD Thesis, Univ. of Illinois, October 1979. (UMI 80-08967).

[BCK79] U. Banerjee, S. Chen, D. J. Kuck and R. A. Towle, Time and Parallel Processor Bounds for Fortran-Like Loops, *IEEE Trans. on Computers C-28*, 9 (September 1979), 660-670.

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[BuC86] M. Burke and R. Cytron, Interprocedural Dependence Analysis and Parallelization, in *Proc. of the SIGPLAN 86 Symp. on Compiler Construction*, Palo Alto, CA, June 25-27, 1986, 162-175.

[BDH87] M. Byler, J. Davies, C. Huson, B. Leasure and M. Wolfe, Multiple Version Loops, in *Proc. of the 1987 International Conf. on Parallel Processing*, S. K. Sahni (ed.), Penn State Press, University Park, PA, 1987, 312-318. August 17-21, 1987.

[Coo71] S. Cook, The Complexity of Theorem-Proving Procedures, in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ACM, New York, 1971, 151-158.

[DaE73] G. B. Dantzig and B. C. Eaves, Fourier-Motzkin Elimination and Its Dual, *Journal of Combinatorial Theory (A) 14*(1973), 288-297.

[Duf74] R. J. Duffin, On Fourier's Analysis of Linear Inequality Systems, in *Mathematical Programming Study 1*, North-Holland, 1974, 71-95.

[Fea88] P. Feautrier, Parametric Integer Programming, *RAIRO Recherche Operationnelle 22*, 3 (September 1988), 243-268.

[GJG87] D. Gannon, W. Jalby and K. Gallivan, Strategies for Cache and Local Memory Management by Global Program Transformation, in *Supercomputing, 1st International Conference*, vol. 297, Springer-Verlag, Berlin, 1987, 229-254.

[Gru90] D. Grunwald, The Lambda Test Revisited, in *Proc. of the 1990 International Conference on Parallel Processing*, 1990.

[KKP90a] D. Klappholz, X. Kong and K. Psarris, On the Perfect Accuracy of an Approximate Subscript Analysis Test, in *Proceedings of the 1990 International Conference on Supercomputing*, ACM, June 1990. Amsterdam, Holland.

[KKP90b] X. Kong, D. Klappholz and K. Psarris, The I Test: A New Test for Subscript Data Dependence, in *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[LY90] Z. Li, P. C. Yew and C. Q. Zhu, An Efficient Data Dependence Analysis for Parallelizing Compilers, *IEEE Trans. on Parallel and Distributed Systems 1*, 1 (January 1990), 26-34.

[Sch86] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, Chichester, Great Britain, 1986.

[Sho81] R. Shostak, Deciding Linear Inequalities by Computing Loop Residues, *J. of the ACM 28*, 4 (October 1981), 769-779.

[Wal88] D. R. Wallace, Dependence of Multi-Dimensional Array References, in *Proc. of the 1988 International Conf. on Supercomputing*, ACM, 1988, 418-428. St. Malo, France, July 4-8, 1988.

[Wed75] D. Wedel, Fortran for the Texas Instruments ASC System, *SIGPLAN Notices 10*, 3 (March 1975), 119-132.

[Wil76] H. P. Williams, Fourier-Motzkin Elimination Extension to Integer Programming Problems, *Journal of Combinatorial Theory (A) 21*(1976), 118-123.

[Wil83] H. P. Williams, A Characterisation of All Feasible Solutions to an Integer Program, *Discrete Applied Mathematics 5*(1983), 147-155.

[Wol82] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, Univ. of Illinois UIUCDCS-82-1105, Urbana, IL, October 1982. (UMI 83-03027).

[Wol86a] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Intl J. Parallel Programming 15*, 4 (August 1986), 279-294.

[Wol86b] M. Wolfe, Advanced Loop Interchanging, in *Proc. of the 1986 Intl Conf. on Parallel Processing*, K. Hwang, S. M. Jacobs and E. E. Swartzlander (ed.), St. Charles, IL, August 19-22, 1986, 536-543.

[WoB87] M. Wolfe and U. Banerjee, Data Dependence and Its Application to Parallel Processing, *Intl Journal of Parallel Programming 16*, 2 (April 1987), 137-178.

[Wol89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, London, 1989.