The Power Test for Data Dependence

Michael Wolfe
Chau-Wen Tseng

CRPC-TR90108
December, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

The Power Test for Data Dependence

Michael Wolfe* Chau-Wen Tseng†

July 18, 1991

Abstract

This paper introduces a data dependence decision algorithm called the Power Test; the Power Test is a combination of the extended GCD algorithm and the Fourier-Motzkin method to eliminate variables in a system of inequalities. This is the first test that can generate the information needed for some advanced transformations, and that can handle complex simultaneous loop limits. This paper briefly reviews previous work in data dependence decision algorithms, and describes the Power Test. Some examples which motivated the development of this test are examined, including those which demonstrate the additional power of the Power Test. Although it may be too expensive for use as a general purpose dependence test in a compiler, the Power Test has proven useful in an interactive program restructuring environment.

1 Introduction

Vectorizing and parallelizing compilers are common in the commercial supercomputer and mini-supercomputer 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 is 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 [2, 4, 7, 8, 9, 15, 24, 35, 36].

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

*Oregon Graduate Institute

†Rice University

```
L1:   for I1 = l1 to u1 do
L2:       for I2 = l2 to u2 do
              ⋮
Ld:           for Id = ld to ud do
S1:               A(f1(I1,...,Id),...,fs(I1,...,Id))
S2:               A(g1(I1,...,Id),...,gs(I1,...,Id))
              endfor
              ⋮
          endfor
      endfor
```

The dependence test should determine whether there are values of the loop indices that lie within their limits such that the subscript expressions are simultaneously equal; *i.e.*, determine whether there exist integer values

$$i_1, i_2, \ldots, i_d \text{ and } j_1, j_2, \ldots, j_d$$

such that

$$l_k \leq i_k \leq u_k,\ l_k \leq j_k \leq u_k,\ \forall k, \text{ and } f_m(\bar{i}) = g_m(\bar{j}),\ \forall m$$

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 $1 \leq k \leq d$, if that is constant), or a dependence direction in each dimension (the sign of the dependence distance). In the general case, there may be d loops in all, but only c of them enclose both statements; there may be loops L_{c+1} through L_b that enclose only statement S_1 and loops L_{b+1} through L_d that enclose only statement S_2. In that case, the first c loops are called the *common loops*, and the dependence distance or direction information is valid only for 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 (including the Power Test) require the subscript expressions to be linear combinations of the loop index variables with known constant coefficients. For most common purposes, a simple test will suffice, applied on one dimension at a time, such as Banerjee's Inequalities [8]. For more advanced restructuring transformations, however, more precision is necessary; the Power Test addresses this need.

The Power Test is a combination of Fourier-Motzkin variable elimination with an extension of Euclid's GCD algorithm. Its name is derived from the power and precision of the method, and from the fact that in the worst case it takes exponential time (in the number of loop index variables). The Power Test finds only integer solutions, accommodating loop limits that represent triangular, trapezoidal and other convex loop limits; as we shall see, it is imprecise in some cases when the loop limits or other conditions cannot be handled exactly. The extended GCD algorithm finds whether a set of linear equations with integer coefficients has any integer solutions, and gives a way to enumerate all those solutions [20, 7]. Since it is derived from the extended GCD algorithm, the Power Test solves the dependence equations for all dimensions simultaneously. It suffers from an worst-case exponential time, so it may not be practical for use as a general dependence test, but it has advantages in particular cases that arise for advanced loop restructuring transformations, as explained in the following section.

Any data dependence problem may be formulated as an integer programming problem; this is discussed

briefly in the comparison section. Other uses of Fourier-Motzkin elimination for dependence testing are described by Kuhn [22] and Triolet et al [29]. The Power Test shares some characteristics of those methods, and a comparison is also given later.

2 Motivation

The main motivation for the Power Test came during the construction of an interactive program restructuring research tool, called TINY. One of the main loop restructuring transformations was loop interchanging, which is usually defined only for tightly-nested loops (where the outer loop encloses the inner loop, but no additional statements) [33, 3]. We planned to add the ability to directly interchange non-tightly-nested loops [35]. In particular, we wanted to be able to generate all 6 versions of the LU decomposition program (factorization of a general real matrix into the product of lower and upper triangular matrices) through loop restructuring; the basic KIJ form of LU decomposition is:

```
        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
```

To generate the JKI form requires interchanging the I and J loops, then interchanging the non-tightly-nested K and J loops to get:

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

Notice where S_1 must be placed in relation to the inner loop. As explained in earlier work [34, 35], the data dependence test for interchanging non-tightly-nested loops (unlike simple loop interchanging) is not a direction vector test. What is needed is a data dependence test which would (1) tell when a data dependence relation would be violated by interchanging non-tightly-nested loops and (2) tell what the direction vectors (or distance vectors) would be after interchanging. Commonly used decision algorithms can generate distance or direction vector information, which determine relative values of the same index variable involved in a dependence relation. The Power Test can also determine the relative values of different index variables, and

so allows our tool to successfully obtain this form of the program.

An additional 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 do
    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, a dependence test must take advantage of the extra knowledge of the simultaneous constraints of the multiple lower and upper loop limits. Current dependence tests handle only a single lower and upper limit expression for each loop index.

For these examples, existing data dependence decision algorithms fall short. While many algorithms have been extended to produce dependence distance or direction vector information, none will generate the information necessary for interchanging non-tightly-nested loops, and all consider only a single lower and upper limit on each loop.

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 next 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 second subsequent iteration of the loop. The first case is called a *def-use* ordering, or a *flow-dependence*; for shorthand, we use the normal terminology $S_1 \ \delta \ S_2$. The second case is a *use-def* ordering, called *anti-dependence*, written $S_3 \ \bar{\delta} \ S_2$. There can also be *def-def* orderings, called *output-dependence*, which uses the symbol δ^o .

In this paper, all loops have an increment (step) of one, to simplify the presentation. Other increments can be handled in several ways, either by ***normalizing*** the loop [4, 10] or by including the increment in the dependence system. In the same way, all linear induction variables for a loop can be expressed as linear functions of a single loop index variable. As with most work on dependence testing, we assume this substitution has been done.

Saying statement 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 [4]. 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 is two for $S_3\ \overline{\delta}\ S_2$. 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 there is 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. These distances are usually written as a *distance vector*; here the distance vector would be $(1,0)$. Often the dependence relation is subscripted with the distance vector, as in $S_1\ \delta_{(1,0)}\ S_2$.

Frequently the dependence distance is not constant; rather than finding all possible dependence distances, it may be sufficient to find 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. The sign of the distance is also saved as a vector; in this case, the dependence relation would be written $S_1\ \delta_{(<)}\ S_2$, meaning that there is a dependence relation from some $S_1[i_1]$ to some $S_2[j_1, j_2]$ where $i_1 < j_1$. When the direction vector consists of all $=$ directions, the dependence relation is called a ***loop-independent*** dependence [4]; otherwise, the dependence relation is said to be *carried* by the outermost loop with a positive distance.

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. We assume that:

1. There are d loops, with loop index variables numbered from I_1 through I_d.

2. The array has s dimensions, and the subscript expressions are linear combinations of the loop index variables with constant integer coefficients, so that the subscript expressions are as below, for $1 \leq m \leq s$:

$$\begin{aligned} f_m(I_1, \ldots, I_d) &= f_{m,0} + f_{m,1} I_1 + f_{m,2} I_2 + \cdots + f_{m,d} I_d \\ g_m(I_1, \ldots, I_d) &= g_{m,0} + g_{m,1} I_1 + g_{m,2} I_2 + \cdots + g_{m,d} I_d \end{aligned}$$

For simplicity, when any $f_{m,x}$ or $g_{m,x}$ is not defined by the subscript expressions, we treat that coefficient as equal to zero (as when $c \neq d$, where c is the number of common loops). A subsequent section

will discuss how to deal with unknown variables in the subscript expressions. Thus, each dimension will generate one dependence equation:

$$f_{m,1}i_1 - g_{m,1}j_1 + f_{m,2}i_2 - g_{m,2}j_2 + \cdots + f_{m,d}i_d - g_{m,d}j_d = g_{m,0} - f_{m,0}$$

or, after renaming (and eliminating terms with zero coefficients):

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

where $n \leq 2d$ is the total number of index variables involved, and where $\mathbf{h}$ is a renaming of the index variables, from outer loop to inner loop.

3. The lower and upper loop limits are also linear combinations of outer loop index variables, with constant coefficients, so that:

$$\begin{aligned} h_k &\geq l_{k,0} + l_{k,1}h_1 + l_{k,2}h_2 + \cdots + l_{k,k-1}h_{k-1} \\ h_k &\leq u_{k,0} + u_{k,1}h_1 + u_{k,2}h_2 + \cdots + u_{k,k-1}h_{k-1} \end{aligned}$$

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

4 Review of the Extended GCD Algorithm

In Knuth [20], the ideas behind Euclid's GCD algorithm are extended to find a general integer solution to a set of linear equations with integer coefficients; Banerjee [7] describes a matrix formulation of this algorithm to attack the data dependence problem. Since the Power Test begins with the extended GCD algorithm, it is briefly reviewed here. The extended GCD algorithm starts by filling an $n \times s$ coefficient matrix $\mathbf{A}$ with the coefficients of the subscript expressions. If there are linearly dependent dependence equations, the GCD algorithm will find and eliminate the redundant equations.

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots \\ a_{1,2} & a_{2,2} & \cdots \\ a_{1,3} & a_{2,3} & \cdots \\ a_{1,4} & a_{2,4} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

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

$$\mathbf{D} = \begin{pmatrix} d_{1,1} & d_{1,2} & d_{1,3} & \cdots \\ 0 & d_{2,2} & d_{2,3} & \cdots \\ 0 & 0 & d_{3,3} & \cdots \\ 0 & 0 & 0 & \ddots \end{pmatrix}$$

During this phase, if a diagonal element is found to be identically zero, then that column (equation) must be a linear combination of previous columns (equations), and is eliminated from further consideration (effectively reducing s by one). Applying the same elementary integer row operations to the identity matrix produces a matrix $\mathbf{U}$ which is unimodular ($\det(\mathbf{U}) = \pm 1$) and that satisfies the equation $\mathbf{UA} = \mathbf{D}$.

If there is an integer solution $\mathbf{t}$ such that $\mathbf{tD} = \mathbf{c}$, then $\mathbf{h} = \mathbf{tU}$ is a solution to the dependence equations $\mathbf{hA} = \mathbf{c}$ (as shown in [7]). After finding $\mathbf{D}$ and $\mathbf{U}$, the test finds values for t_1 through t_s by solving $\mathbf{tD} = \mathbf{c}$ using a simple back-substitution algorithm.

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 to be solved are:

$$\begin{array}{rcl@{\quad\rightsquigarrow\quad}rcl} i_1 + i_2 + 1 & = & j_1 + j_2 & i_1 - j_1 + i_2 - j_2 & = & -1 \\ i_2 + 1 & = & j_2 & i_2 - j_2 & = & -1 \end{array}$$

The dependence matrix $\mathbf{hA} = \mathbf{c}$ is:

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

The extended GCD algorithm augments $\mathbf{A}$ with the identity matrix to get $(\mathbf{U} \mid \mathbf{D})$:

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

($\mathbf{U}$ comprises the first four columns, $\mathbf{D}$ the final two.) The algorithm performs elementary row operations to reduce $\mathbf{D}$ to an upper triangular matrix:

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

The reader can verify that $\mathbf{UA} = \mathbf{D}$. Now solve $\mathbf{tD} = \mathbf{c}$,

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

Because $\mathbf{D}$ is an upper triangular matrix, t_3 and t_4 do not figure into the computation. The first equation is $-t_1 = -1$ or $t_1 = 1$. The second equation is $-t_1 + t_2 = -1$ or $t_2 = 0$. Since there is a feasible solution, the extended GCD algorithm stops here and assumes dependence.

5 Uses of the Extended GCD Solution

The extended GCD algorithm, like the single-dimension GCD test, tells whether there are any integer solutions to the dependence equations, ignoring loop limits. 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 matrix product $\mathbf{h} = \mathbf{tU}$. The first use that we make of the extended GCD solution is to find constant dependence distances by subtracting corresponding equations. That is, the dependence distance for loop level k ($1 \leq k \leq c$) is found by subtracting the equations for i_k and j_k. Suppose that $i_k \equiv h_{2k-1}$, and $j_k \equiv h_{2k}$; the two equations are subtracted by looking at:

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

(where $\mathbf{U}_{\star,x}$ is column x of the matrix $\mathbf{U}$). 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 is found from $\mathbf{t}_{1:s}(\mathbf{U}_{1:s,2k-1} - \mathbf{U}_{1:s,2k})$.

This sometimes allows a dependence distance vector to be constructed from the $\mathbf{U}$ matrix and the solutions to $t_1, ..., t_s$. The dependence direction vector can 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 [4]. Using the extended GCD algorithm is more precise than linearizing the array references and using a single-equation GCD test [8, 9].

Example. We continue the example from the previous section. After solving for t_1 and t_2, the original dependence matrix equation was $\mathbf{hA} = \mathbf{c}$; the extended GCD algorithm found that there are integer solutions $\mathbf{t}$ such that $\mathbf{tD} = \mathbf{c}$. Since $\mathbf{UA} = \mathbf{D}$, we have $\mathbf{tUA} = \mathbf{c}$, which means that $\mathbf{tU} = \mathbf{h}$. We use this to find equations for the index variables that solve the dependence matrix equation in terms of the free variables by multiplying $\mathbf{tU}$:

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

This gives the equations:

$$\begin{aligned} i_1 &= t_3 \\ j_1 &= t_2 + t_3 \\ i_2 &= t_4 \\ j_2 &= t_1 - t_2 + t_4 \end{aligned}$$

The dependence distance vector is computed as $(j_1 - i_1, j_2 - i_2)$, which in this case is: $(t_2, t_1 - t_2)$. Since these are already solved quantities, the dependence distance can be computed to be $(0, 1)$.

6 The Power Test

The method from the previous section will find constant dependence distances, but does not consider all the constraints on the dependence system, such as loop limits. The Power Test continues from this point. Loop limits, direction vector information or non-direction vector relations comprise a set of linear inequalities or constraints on the set of free variables. Fourier-Motzkin variable elimination [11, 12], modified to find integer solutions, is used to test for an empty solution space.

The Power Test constructs 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:

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

with $lb_k > 0$. Since we want only the integer solutions, we can take the ceiling of lower bounds (or the floor of upper bounds), as suggested by Kuhn [22]:

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

These bounds are derived from the constraints on the index variables, such as the loop limits. Each index variable h_m is defined by the extended GCD algorithm as some linear combination of the free variables. In addition, the upper and lower limits of each index variable are themselves linear combinations of outer loop index variables. Thus, the constraint $h_m \geq l_m$ can be algebraically reduced to an inequality constraint on one of the free variables, of the form of (1). 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: $\mathbf{tU}_{\star,m} \geq 1$. 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 are 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 is simplified 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 $\lceil lb_0 / lb_k \rceil$ are computed by the compiler. This gives a precise integer bound. If the divisions are not exact, then the ceiling or floor operators will be a source of imprecision, since the Power Test will 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 points near the boundaries. In particular, this method may include some 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:

$$\begin{aligned} lb'_0 &= \lceil lb_0 / g \rceil \\ lb'_r &= lb_r / g, \quad \forall s+1 \leq r \leq k-1 \\ lb'_k &= lb_k / g \end{aligned}$$

Example. Take the program:

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

The dependence matrix equation $\mathbf{hA} = \mathbf{c}$ is:

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

The extended GCD algorithm finishes with the matrix:

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

Solving $\mathbf{tD} = \mathbf{c}$ we get $t_1 = 0, t_2 = 0$. From these solutions and $\mathbf{h} = \mathbf{tU}$ we get:

$$\begin{aligned} i_1 &= t_3 \\ j_1 &= t_4 \\ i_2 &= t_4 \\ j_2 &= t_3 \end{aligned}$$

The dependence distance is not constant, since $(j_1 - i_1, j_2 - i_2)$ is a function of the remaining free variables.

From the lower limit for I_1, we have the inequality $i_1 \geq 1$; since $i_1 = t_3$, 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:

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

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 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 false solution. 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: $t_3 + 1 \leq t_3 - 1$, or $1 \leq -1$, which is clearly inconsistent; thus, this dependence system has no solution, and the two references are independent.

6.1 Direction Vectors

The Power Test will also 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 $h_{2k} - h_{2k-1} \geq 1$. We then replace the index variables by their formula in terms of the free variables:

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

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

Example. In the following program:

```
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 equation $\mathbf{hA} = \mathbf{c}$ is:

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

The extended GCD algorithm produces:

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

Solving $\mathbf{tD} = \mathbf{c}$ gives $t_1 = 0$, so that multiplying out $\mathbf{h} = \mathbf{tU}$ gives:

$$\begin{aligned} i_1 &= t_2 \\ j_1 &= t_3 \\ i_2 &= t_4 \\ j_2 &= t_2 \end{aligned}$$

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 $t_4 \geq t_2 + 1$, 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:

$$\begin{aligned} 1 \leq\ & t_2 \leq N \\ 1 \leq\ & t_3 \leq \begin{cases} t_2 - 1 \\ N \end{cases} \\ t_2 + 1 \leq\ & t_4 \leq N \end{aligned}$$

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 loop. From $i_1 < j_1$ we derive the additional bound:

$$t_2 + 1 \leq t_3$$

This is inconsistent with one of the previous bounds, so there is no dependence with a $(<)$ direction in the first loop. Likewise, if we test for a $(<)$ direction in the second loop, 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 negated dependence equation, which corresponds to an anti-dependence [35]. Thus, the Power Test correctly identifies the anti-dependence with the precise direction vector.

7 The Power of the Power Test

This section shows by example how the Power Test handles several interesting cases of dependence. More examples can be found in the technical report [37].

7.1 Non-Direction-Vector Constraints

One goal of the TINY project was to be able to generate all six versions of LU decomposition by simple loop restructuring. In the motivation section, we started with the normal KIJ form, then interchanged the tightly nested I and J loops, using classical loop interchanging techniques. The dependence test for loop interchanging is that there must be no dependence relations with $(<, >)$ direction vectors in the loops being interchanged [33, 3]; this condition is satisfied here, so interchanging is legal. 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. A new transformation was designed to interchange non-tightly-nested loops directly [34].

Let us first 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 (i_1, i_3, i_4) of S_2 to iteration (j_1, j_2) of S_1 such that $i_1 < j_1$ and $i_3 > j_1$ [35]. Let us inspect the dependence from $S_2 : A(I_4, I_3)$ to $S_1 : A(I_2, I_1)$. The dependence matrix equation $\mathbf{hA} = \mathbf{c}$ is:

$$(i_1, j_1, j_2, i_3, i_4) \begin{pmatrix} 0 & 0 \\ 0 & -1 \\ -1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = (0, 0)$$

The extended GCD algorithm ends with:

$$(\mathbf{U} \mid \mathbf{D}) = \left(\begin{array}{ccccc|cc} 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}\right)$$

The solution of $\mathbf{tD} = \mathbf{c}$ finds $t_1 = 0$ and $t_2 = 0$, so the index variables are defined as:

$$\begin{aligned} i_1 &= t_3 \\ j_1 &= t_4 \\ j_2 &= t_5 \\ i_3 &= t_4 \\ i_4 &= t_5 \end{aligned}$$

The loop limits bound the free variables as follows:

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

It is easy to see that testing for $i_1 \geq j_1$ would add the constraint $t_4 \leq t_3$, which is inconsistent. Thus, i_1 must be less than j_1; this corresponds to the dependence $S_2 \ \delta_{(<)} \ S_1$. Also, adding the constraint $i_3 > j_1$ gives

the inconsistency $t_4 > t_4$, so this dependence does not prevent interchanging. Moreover, after interchanging, the loops around S_2 will be reordered to (i_3, i_1, i_4); thus after interchanging, the direction vector should be the sign of the difference $j_1 - i_3$. It is easy to realize that this is $t_4 - t_4 = 0$. Thus, the Power Test easily performs non-direction-vector tests, which other current dependence tests cannot handle.

7.2 Multiple Loop Limits

As mentioned earlier, some program transformations generate multiple lower or upper limits. As a case in point, the Gaussian Elimination above can be reindexed automatically 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 $S_2 : A(I_2, I_3)$, the algorithm has to deal with the upper limit of the I_3 loop, which is the minimum of two simple expressions. The Power Test handles this case effectively by enforcing multiple limits on the free variables, one for each loop limit. The extended GCD algorithm for this dependence system returns the definitions:

$$\begin{aligned} i_1 &= t_3 \\ j_1 &= t_4 \\ i_4 &= t_5 \\ j_2 &= t_5 \\ j_3 &= t_3 \end{aligned}$$

Enforcing the upper limit for j_3 gives $j_3 \leq j_1 - 1$ or $t_3 \leq t_4 - 1$, and $j_3 \leq j_2 - 1$ or $t_3 \leq t_5 - 1$; adding these to the other limits gives the solution space:

$$\begin{aligned} 1 &\leq t_3 \leq n \\ \left.\begin{matrix} t_3+1 \\ 1 \end{matrix}\right\} &\leq t_4 \leq n \\ \left.\begin{matrix} t_3+1 \\ 2 \end{matrix}\right\} &\leq t_5 \leq n \end{aligned}$$

When testing for a (>) direction in the I_1 loop, for instance, the Power Test adds the inequality $i_1 \geq j_1 + 1$ which derives $t_4 \leq t_3 - 1$ which generates the inconsistency $t_3 + 1 \leq t_4 \leq t_3 - 1$.

7.3 Handling Unknown Variables

Unknown variables frequently occur in subscript expressions or in loop limits, and thus in dependence systems. The Power Test handles many of these cases naturally by treating the unknown variables as additional index variables which have no limits. To compute dependence in the program:

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

the variable N used in the loop limit and the second subscript equation must be handled. Rather than assuming dependence or treating this as a special case, the Power Test treats the unknown variable N as an additional index variable and builds the dependence system with three index variables. The dependence equation in matrix form is:

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

The extended GCD algorithm ends up with the definitions:

$$\begin{aligned} N &= t_2 \\ i_1 &= t_2 + t_3 \\ j_1 &= t_3 \end{aligned}$$

Since the value of N is unknown, there is no bound on t_2; the loop limits for i_1 produce $1 - t_2 \leq t_3 \leq 0$, while the limits for j_1 produce $1 \leq t_3 \leq t_2$, which are clearly inconsistent. Thus the two references are independent, as we already know; 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 expression:

```
for I = 1 to N do
    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, adding the assertion as another constraint on the dependence system.

Not all unknown variables are 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 expression appear nonlinear, as in the case of manually linearized arrays:

```
for I = 2 to N do
    for J = 1 to M do
        A(I+J*N) = ...
        ...= A(I+J*N-1)
    endfor
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. Here we compare the Power Test with existing single and multi-dimension dependence tests.

8.1 Integer and Linear Programming

Since testing linear subscript expressions for dependence is equivalent to finding simultaneous integer solutions within loop limits, one approach is to employ integer programming methods based either on the simplex method [13] or Fourier-Motzkin elimination [31, 32]. Linear programming techniques such as Shostak's loop residue method [27] are also applicable, though integer solutions are not guaranteed. Unfortunately, while integer programming techniques are suitable for solving large systems of equations, their high initialization costs and implementation complexity make them less desirable for dependence testing. In addition, though integer programming can be extended for complex loop limits and direction vector constraints, they cannot be used to calculate distance vectors.

8.2 Single Dimension Tests

There also exists several single dimension dependence tests. The GCD and Banerjee tests [6] will determine the existence of *unconstrained* integer and *constrained* real solutions, respectively. Banerjee's inequalities have also been extended for direction vectors [33], dependence levels [4], and triangular loop limits [7, 18]. Researchers have also shown that Banerjee's inequalities are exact in many common cases [5, 19, 23]. The I-test [21] combines the GCD and Banerjee tests, and can usually prove integer solutions. If only one index variable appears in each subscript reference, the Single-Index-Exact test [6, 33] may be applied. In comparison, the Power Test subsumes the Single-Index-Exact test, and will always be at least as precise as any combination of the GCD or Banerjee tests.

8.3 Multi-Dimension Tests

Testing dependences for multi-dimensional arrays is difficult because simultaneous solutions must be discovered for all array dimensions. Early approaches include intersecting direction vectors from each dimension [33] and linearization [9], but are not precise in many cases. In the following sections we will compare the Power Test with several precise multi-dimensional dependence tests.

8.3.1 Fourier-Motzkin Elimination

Previous researchers have also utilized Fourier-Motzkin elimination. Kuhn [22] represents array accesses in a canonical convex hull using *dependence arc sets*. Dependences are calculated by intersecting these convex sets using Fourier-Motzkin elimination. Nonempty sets can be checked for integer solutions by *enumerating* possible solutions; this is impractical for loops with unknown symbolic limits. Kuhn suggests compile-time array bounds checking, dataflow anomaly detection, and calculating array kills as other possible applications. Triolet [29] uses convex sets called *regions* to summarize array accesses in compound statements (*e.g.*, procedure calls). Dependences are calculated by intersecting regions using Fourier-Motzkin elimination. This determines whether calls may be executed in parallel. Both techniques support complex loop limits and direction vectors.

The main differences between these methods and the Power Test is that they apply Fourier-Motzkin elimination on the constraints of the original system (*e.g.*, loop limits, dependence directions, subscript expressions). The number of variables and constraints in the system of inequalities in these two methods is larger than in the Power test, due to the way the systems are built. Equality of the subscript equations is handled by adding the constraints $f_s(\vec{i}) \leq g_s(\vec{j})$ and $g_s(\vec{j}) \leq f_s(\vec{i})$, for each subscript s, which is not always precise. In comparison, the Power Test applies Fourier-Motzkin elimination on the *dense* linear system derived from the extended GCD algorithm. As a result, the Power Test is more precise in detecting the lack of simultaneous constrained integer solutions.

8.3.2 Constraint-Matrix Test

The Constraint-Matrix test [30] is a simplex algorithm modified for integer programming. It introduces slack variables for each constraint in the system, then iteratively applies the 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 halts after an arbitrary number of iterations and conservatively assumes dependence. Gaussian elimination must be performed first to eliminate linearly dependent constraints. Although not stated in [30], the Constraint-Matrix test may be extended to compute full direction vectors and accommodate complex loop limits, though it does not compute distance vectors.

The Constraint-Matrix test can detect the lack of simultaneous real-valued solutions. However, comparison with the Power Test is difficult since it stops after an arbitrary number of iterations to avoid cycling. It is especially hard to analyze the ability of the Constraint-Matrix test to detect the lack of simultaneous integer solutions. Since the Constraint-Matrix is based on the simplex algorithm, it has worst-case exponential complexity. For most linear programming problems, simplex algorithms take only linear time and cycling is rare. However, Schrijver [26] states that for combinatorial problems where coefficients tend to be 1, 0, or -1, the simplex algorithm is slow and will cycle for certain pivot rules.

8.3.3 λ-Test

The λ-test [23, 24] is equivalent to a multi-dimensional version of Banerjee's inequalities, since it checks for simultaneous constrained real-valued solutions. As with the Constraint-Matrix test, the λ-test requires Gaussian elimination be first performed to eliminate redundant subscript expressions. The λ-test can also test direction vectors and handle triangular loops, but not multiple loop limits or distance vectors.

The λ-test is applied to *coupled dimensions*, groups of subscript positions sharing one or more index variables. The test forms linear combinations of subscript expressions that eliminate one or more instances of index variables, then tests the result. Simultaneous real-valued solutions exist if and only if Banerjee's inequalities finds solutions in all the linear combinations generated.

We can enhance the precision of the λ-test by also applying the GCD or Single-Index-Exact tests to the linear combinations generated. Unfortunately, there is no obvious method to extend the λ-test to prove the existence of simultaneous integer solutions. Li [23] proves that for two coupled dimensions, the λ-test is exact if unconstrained integer solutions exist and the coefficients of index variables are all 1, 0 or -1. However, even with these restrictions the λ-test is not exact for three or more coupled dimensions. The precision of the λ-test for complex loop limits or direction vectors has not been discussed in the literature.

We now show that the Power Test is more precise than the λ-test. First of all, the Power Test is just as precise in detecting simultaneous real-valued solutions, since it solves the linear programming problem exactly. The following example will show that Power Test can detect the lack of simultaneous constrained integer solutions where the λ-test cannot:

```
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:

$$\begin{aligned} \text{EQ1:} \quad & 3i_1 - j_1 + 2i_2 + j_2 = 6 \\ \text{EQ2:} \quad & -j_1 + 2i_2 - j_2 = 0 \end{aligned}$$

The linear combinations (set of canonical solutions) generated by the λ-test are as follows:

Combination	equation	eliminates	dependence equation	GCD	Banerjee's inequalities
C_1	EQ2	i_1	$-j_1 + 2i_2 - j_2 = 0$	1	$-198 \leq 0 \leq 198$
C_2	EQ1 - EQ2	j_1,i_2	$3i_1 + 2j_2 = 6$	1	$5 \leq 6 \leq 500$
C_3	EQ1 + EQ2	j_2	$3i_1 - 2j_1 + 4i_2 = 6$	1	$-193 \leq 6 \leq 698$

In all cases the GCD test fails to detect independence since the GCD of the coefficients is one. In addition, Banerjee's inequalities show that constrained real-valued solutions exist for all three combinations. Since the GCD test and Banerjee's inequalities fail for all three combinations, the λ-test would assume dependence exists. In comparison, when we apply the Power Test, we get the dependence matrix equation $\mathbf{hA} = \mathbf{c}$:

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

The extended GCD algorithm returns the matrices:

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

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

$$\begin{aligned} i_1 &= 2t_3 + 6 \\ j_1 &= 3t_3 + 2t_4 + 6 \\ i_2 &= t_4 \\ j_2 &= 3t_3 - 6 \end{aligned}$$

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

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

Combining the upper and lower limits 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 or symbolic expressions, since they were not used in the test.

8.3.4 Delta Test

The Delta test [16] is a new multi-dimension dependence test designed for a restricted class of subscript expressions. It works by converting single-index-variable subscript expressions into *constraints* that can be intersected or propagated into other array dimensions. The Delta test is very efficient at calculating exact distance and direction vectors for common subscript expressions, but the Power Test is much more general.

9 Conclusions

The Power Test is useful in advanced program restructuring techniques. It is based on the extended GCD algorithm, and is close to the holy grail of solving simultaneous subscript equations only for integer solutions within the loop limits. It loses precision when it ignores 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 a solution *near* the limits of the loop, or near the bounds imposed by other constraints such as direction vector relations. Experimental evidence is necessary to see whether this is a serious limitation. 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. This becomes important when performing advanced restructuring transformations, as in the research tool TINY.

An obvious consideration when implementing the Power Test is its execution cost. The worst case cost of the search procedure of the Fourier-Motzkin elimination can be exponential in the number of free variables. Triolet [29] found that using Fourier-Motzkin elimination for dependence testing takes from 22 to 28 times longer than conventional dependence tests. By comparison, although the λ-test is an worst-case exponential

cost algorithm, it proves to be quite efficient in practice. Empirical studies [24] show that the λ-test usually increases the cost of dependence testing by a factor of two or less. At this point, more studies are required to characterize the behavior of the dependence tests we have examined. Since the actual number of both loops and dimensions are likely to be small, only experimental results will indicate which tests are efficient enough for practical use. The TINY tool uses the Power Test as the default dependence test, but since TINY was designed for only small programs, it is not a reasonable test vehicle. Even if the cost of the Power Test is too high for inclusion in a critical component such as a compiler, it has been shown to be appropriate in a special environments like TINY. Other dependence tests do not handle the complex dependence systems that have arisen in our research.

10 Acknowledgements

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

References

[1] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. In Houstis et al. [17], pages 194–211.

[2] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *J. Parallel and Distributed Computing*, 5(5):617–640, October 1988. (update of [1]).

[3] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. SIGPLAN '84 Symp. on Compiler Construction*, pages 233–246, Montreal, Canada, June 1984.

[4] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[5] Utpal Banerjee. Data dependence in ordinary programs. M.S. thesis UIUCDCS-R-76-837, Univ. Illinois, Dept. Computer Science, November 1976.

[6] Utpal Banerjee. Speedup of ordinary programs. PhD Dissertation UIUCDCS-R-79-989, Univ. Illinois, Dept. Computer Science, October 1979. (available from Univ. Microfilms Inc., document 80-08967).

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

[8] Utpal Banerjee, Shyh-Ching Chen, David J. Kuck, and Ross A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. on Computers*, C-28(9):660–670, September 1979.

[9] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proc. SIGPLAN '86 Symp. on Compiler Construction* [28], pages 162–175.

[10] William L. Cohagan. Vector optimization for the ASC. In *Proc. Seventh Annual Princeton Conf. on Information Sciences and Systems*, pages 169–174, March 1973.

[11] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *J. Combinatorial Theory (A)*, 14:288–297, 1973.

[12] R. J. Duffin. On Fourier's analysis of linear inequality systems. In *Mathematical Programming Study 1*, pages 71–95. North-Holland, 1974.

[13] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Operationnelle*, 22(3):243–268, September 1988.

[14] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. In Houstis et al. [17], pages 229–254.

[15] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. *J. Parallel and Distributed Computing*, 5(5):587–616, October 1988. (update of [14]).

[16] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. Technical Report TR90-142, Rice Univ., November 1990. To appear in the *ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[17] Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos, editors. *Supercomputing: 1st International Conf.*, volume 297 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[18] Ken Kennedy. Triangular Banerjee inequality. Technical Report Supercomputer Software Newsletter #8, Rice Univ., Dept. Computer Science, October 1986.

[19] David Klappholz, Xiangyun Kong, and Kleanthis Psarris. On the perfect accuracy of an approximate subscript analysis test. In *Proc. 1990 International Conf. on Supercomputing*, pages 201–212, Amsterdam, June 1990.

[20] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[21] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I test: A new test for subscript data dependence. In Padua [25], pages 204–211.

[22] Robert H. Kuhn. Optimization and interconnection complexity for: Parallel processors, single stage networks, and decision trees. PhD Dissertation UIUCDCS-R-80-1009, Univ. Illinois, Dept. Computer Science, February 1980.

[23] Zhiyuan Li. Intraprocedural and interprocedural data dependence analysis for parallel computing. PhD Dissertation CSRD Rpt. No. 910, Univ. Illinois, Dept. Computer Science, August 1989.

[24] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[25] David Padua, editor. *Proc. 1990 International Conf. on Parallel Processing*, volume II, St. Charles, IL, August 1990. Penn State Press.

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

[27] Robert Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, October 1981.

[28] *Proc. SIGPLAN '86 Symp. on Compiler Construction*, Palo Alto, CA, June 1986.

[29] Rémi Triolet, François Irigoin, and Paul Feautrier. Direct parallelization of Call statements. In *Proc. SIGPLAN '86 Symp. on Compiler Construction* [28], pages 176–185.

[30] David R. Wallace. Dependence of multi-dimensional array references. In *Proc. 1988 International Conf. on Supercomputing*, pages 418–428, St. Malo, June 1990.

[31] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *J. Combinatorial Theory (A)*, 21:118–123, 1976.

[32] H. P. Williams. A characterisation of all feasible solutions to an integer program. *Discrete Applied Mathematics*, 5:147–155, 1983.

[33] Michael Wolfe. Optimizing supercompilers for supercomputers. PhD Dissertation UIUCDCS-R-82-1105, Univ. Illinois, Dept. Computer Science, October 1982. (available from Univ. Microfilms Inc., document 83-03027).

[34] Michael Wolfe. Advanced loop interchanging. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proc. 1986 International Conf. on Parallel Processing*, pages 536–543, St. Charles, IL, August 1986.

[35] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and

Distributed Computing. Pitman Publishing, London, 1989. (also available from MIT Press).

[36] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International J. Parallel Programming*, 16(2):137–178, April 1987.

[37] Michael Wolfe and Chau-Wen Tseng. The Power test for data dependence. Technical Report CSE 90-015, Oregon Graduate Institute, August 1990.