Practical Dependence Testing

Gina Goff
Ken Kennedy
Chau-Wen Tseng

CRPC-TR 90103
November 1990

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

In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, CA, June 1991.

Practical Dependence Testing *

Gina Goff Ken Kennedy Chau-Wen Tseng

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Precise and efficient dependence tests are essential to the effectiveness of a parallelizing compiler. This paper proposes a dependence testing scheme based on classifying pairs of subscripted variable references. Exact yet fast dependence tests are presented for certain classes of array references, as well as empirical results showing that these references dominate scientific Fortran codes. These dependence tests are being implemented at Rice University in both PFC, a parallelizing compiler, and ParaScope, a parallel programming environment.

1 Introduction

In the past decade, high performance computing has become vital for scientists and engineers alike. Much progress has been made in developing large-scale parallel architectures composed of powerful commodity microprocessors. To exploit parallelism and the memory hierarchy effectively for these machines, compilers must be able to analyze data dependences precisely for array references in loop nests. Even for a single microprocessor, optimizations utilizing dependence information can result in integer factor speedups for scientific codes [11]. However, because of its expense, few if any scalar compilers perform dependence analysis.

Parallelizing compilers have traditionally relied on two dependence tests to detect data dependences between pairs of array references: Banerjee's inequalities and the GCD test [8, 55]. However, these tests are usually more general than necessary. This paper presents empirical results showing that most array references in scientific Fortran programs are fairly simple. For these simple references, we demonstrate a suite of highly exact yet efficient dependence tests. We feel that these tests will significantly reduce the cost of performing dependence analysis, making it more practical for all compilers. We begin with some definitions.

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, by IBM Corporation, and by the Cray Research Foundation.

1.1 Data Dependence

The theory of *data dependence*, originally developed for automatic vectorizers, has proved applicable to a wide range of optimization problems. We say that a data dependence exists between two statements S_1 and S_2 if there is a path from S_1 to S_2 and both statements access the same location in memory. There are four types of data dependence [32, 33]:

True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads.

Anti dependence occurs when S_1 reads a memory location that S_2 later writes.

Output dependence occurs when S_1 writes a memory location that S_2 later writes.

Input dependence occurs when S_1 reads a memory location that S_2 later reads.

Dependence analysis is the process of computing all such dependences in a program.

1.2 Dependence Testing

Calculating data dependence for arrays is complicated by the fact that two array references may not always access the same memory location. *Dependence testing* is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest. For the purposes of this explication, we will ignore any control flow except for the loops themselves. Suppose that we wish to test whether or not there exists a dependence from statement S_1 to S_2 in the following model loop nest:

```
      DO i1 = L1, U1
        DO i2 = L2, U2
          ...
          DO in = Ln, Un
S1          A(f1(i1,...,in),...,fm(i1,...,in)) = ...
S2          ... = A(g1(i1,...,in),...,gm(i1,...,in))
          ENDDO
          ...
        ENDDO
      ENDDO
```

Let α and β be vectors of n integer indices within the ranges of the upper and lower bounds of the n loops in the example. There is a dependence from S_1 to S_2 if and only if there exist α and β such that α is lexicographically less than or equal to β and the following system of *dependence equations* is satisfied:

$$f_i(\alpha) = g_i(\beta) \ \forall i, \ 1 \leq i \leq m$$

Otherwise the two references are *independent*.

1.3 Distance and Direction Vectors

Data dependences may be characterized by their access pattern between loop iterations using distance and direction vectors. Suppose that there exists a data dependence for $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$. Then the *distance vector* $\mathbf{D} = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The *direction vector* $\mathbf{d} = (d_1, \ldots, d_n)$ of the dependence is defined by the equation:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

The elements are always displayed in order left to right, from the outermost to the innermost loop in the nest. For example, consider the following loop nest:

```
      DO 10 i
        DO 10 j
          DO 10 k
10          A(i+1, j, k-1) = A(i, j, k) + C
```

The distance and direction vectors for the dependence between the definition and use of array A are $(1, 0, -1)$ and $(<, =, >)$, respectively. Since several different values of α and β may satisfy the dependence equations, a set of distance and direction vectors may be needed to completely describe the dependence.

Direction vectors, first introduced by Wolfe [53], are useful for calculating the level of *loop-carried* dependences [1, 4, 25]. A dependence is *carried* by the outermost loop for which the direction in the direction vector is not '='. For instance, the direction vector $(<, =, >)$ for the dependence above shows the dependence is carried on the i loop.

Carried dependences are important because they determine which loops cannot be executed in parallel without synchronization. Direction vectors are also useful in determining whether loop interchange is legal and profitable [4, 25, 53].

Distance vectors, first used by Kuck and Muraoka [34, 42], are more precise versions of direction vectors that specify the actual distance in loop iterations between two accesses to the same memory location. They may be used to guide optimizations to exploit parallelism [23, 27, 36, 51, 54] or the memory hierarchy [11, 19, 43].

Dependence testing thus has two goals. First, it tries to disprove dependence between pairs of subscripted references to the same array variable. If dependences may exist, it tries to characterize them in some manner, usually as a minimal complete set of distance and direction vectors. Dependence testing must also be *conservative* and assume the existence of any dependence it cannot disprove. Otherwise the validity of any optimizations based on dependence information is not guaranteed.

1.4 Exact Tests

When array subscripts are linear expressions of the loop index variables, dependence testing is equivalent to the problem of finding integer solutions to systems of linear Diophantine equations, an NP-complete problem [15, 17]. In practice most dependence tests, such as Banerjee's inequalities [8], seek efficient approximate solutions. *Exact* tests, on the other hand, are dependence tests that will detect dependences if and only if they exist.

1.5 Indices and Subscripts

In this paper we will use the term *index* to mean the index variable for some loop surrounding both of the references. We assume that all *auxiliary induction variables* have been detected and replaced by linear functions of the loop indices [2, 3, 5, 52].

In addition, we will use the term *subscript* to refer to one of the subscripted positions in a pair of array references; *i.e.*, the pair of subscripts in some dimension of the two array references. Dependence tests always consider a pair of array references, but for brevity we refer to a subscript pair simply as a subscript. For example, in the pair of references to array A in the following loop nest,

```
      DO 10 i
        DO 10 j
          DO 10 k
10          A(i, j) = A(i, k) + C
```

we say that index i occurs in the first subscript and indices j and k occur in the second subscript.

2 Classification

In this section we present two orthogonal criteria for classifying subscripts in a pair of array references. The first criterion, *complexity*, refers to the number of indices appearing within the subscript. The second criterion, *separability*, describes whether a given subscript interacts with other subscripts for the purpose of dependence testing.

2.1 Complexity

When testing for dependence, we classify subscript positions by the total number of distinct loop indices they contain. A subscript is said to be ZIV (zero index variable) if the subscript position contains no index in either reference. A subscript is said to be SIV (single index variable) if only one index occurs in that position. Any subscript with more than one index is said to be MIV (multiple index variable). For instance, consider the following loop:

```
      DO 10 i
        DO 10 j
          DO 10 k
10          A(5, i+1, j) = A(N, i, k) + C
```

When testing for a true dependence between the two references to A in the code below, the first subscript is

ZIV, the second is SIV, and the third is MIV. For the sake of simplicity, we will ignore output dependences in this and all future examples.

2.2 Separability

When testing multidimensional arrays, we say that a subscript position is *separable* if its indices do not occur in the other subscripts [1, 10]. If two different subscripts contain the same index, we say they are *coupled* [38]. For example, in the loop below,

```
      DO 10 i
        DO 10 j
          DO 10 k
10          A(i, j, j) = A(i, j, k) + C
```

the first subscript is separable, but the second and third are coupled because they both contain the index j. ZIV subscripts are vacuously separable because they contain no indices.

Separability is important because multidimensional array references can cause imprecision in dependence testing. One suggested approach, called *subscript-by-subscript testing*, is to test each subscript separately and intersect the resulting sets of direction vectors [53]. However, this method provides a conservative approximation to the set of directions within a coupled group—it may yield direction vectors that do not exist. For instance, consider the following loop:

```
      DO 10 i
10      A(i+1, i+2) = A(i, i) + C
```

A subscript-by-subscript test would yield the single direction vector $(<)$. But a careful examination of the statement reveals that this direction vector is invalid since no dependence exists!

On the other hand, if all subscripts are separable, we may compute the direction vector for each subscript independently, and merge the direction vectors on a positional basis with full precision. For example, in the loop nest below,

```
      DO 10 i
        DO 10 j
          DO 10 k
10          A(i+1, j, k-1) = A(i, j, k) + C
```

the leftmost direction in the direction vector is determined by testing the first subscript, the middle direction by testing the second subscript and the rightmost direction by testing the third subscript. The resulting direction vector, $(<,=,>)$, is precise. The same approach applied to distances allow us to calculate the exact distance vector $(1, 0, -1)$.

We know from linear algebra that systems of equations with distinct variables may be solved independently, and their solutions merged to form an exact solution set. Previous tests have used this property for array references consisting of only separable SIV subscripts [1, 23, 34, 36, 42]. More recently, Li *et al.* formalized and applied this method in the λ-test to array references also containing MIV or coupled subscripts [38]. Our treatment of constraint propagation in Section 5.3 was inspired by their work.

3 Dependence Testing

The goal of dependence testing in this paper is to construct the complete set of distance and direction vectors representing potential dependences between an arbitrary pair of subscripted references to the same array variable. Since distance vectors may be treated as precise direction vectors, we will simply refer to direction vectors for the rest of the paper. For the sake of simplicity we will also assume that all loops have a step of 1. Non-unit step values may be normalized on the fly as needed.

3.1 Partition-Based Algorithm

The classifications presented in the previous section may be used naturally in a *partition-based* dependence testing algorithm as follows:

1. Partition the subscripts into separable and minimal coupled groups.
2. Label each subscript as ZIV, SIV or MIV.
3. For each separable subscript, apply the appropriate single subscript test (ZIV, SIV, MIV) based on the complexity of the subscript. This will produce independence or direction vectors for the indices occurring in that subscript.
4. For each coupled group, apply a multiple subscript test to produce a set of direction vectors for the indices occurring within that group.
5. If any test yields independence, no dependences exist.
6. Otherwise merge all the direction vectors computed in the previous steps into a single set of direction vectors for the two references.

This algorithm is implemented in both PFC, an automatic vectorizing and parallelizing compiler [3, 4], and ParaScope, a parallel programming environment [12, 27, 28].

Our dependence testing algorithm takes advantage of separability by classifying all subscripts in a pair of array references as separable or part of some minimal coupled group. A coupled group is *minimal* if it cannot be partitioned into two non-empty subgroups with distinct sets of indices. Once a partition is achieved, each separable subscript and each coupled group have completely disjoint sets of indices. Each partition may then be tested in isolation and the resulting distance or direction vectors merged without any loss of precision.

Subscripts may be partitioned using the algorithm in Figure 1. An alternative algorithm based on UNION/FIND is implemented in PFC. The dependence testing algorithm may halt and return independence as soon as the test for any separable subscript or coupled group yields independence, since no simultaneous solutions are possible once we prove no solutions exist for some subset of the entire system.

INPUT: A pair of m-dimensional array references
containing subscripts $S_1 \ldots S_m$
enclosed in n loops with indices $I_1 \ldots I_n$

OUTPUT: A set of partitions $P_1 \ldots P_{n'}$, $n' \leq n$, each
containing a separable or minimal coupled group

```
for each i, 1 ≤ i ≤ n do
    P_i ← {S_i}
endfor
for each index I_i, 1 ≤ i ≤ n do
    k ← ⟨none⟩
    for each remaining partition P_j do
        if ∃ S_l ∈ P_j such that S_l contains I_i then
            if k = ⟨none⟩ then
                k ← j
            else
                P_k ← P_k ∪ P_j
                discard P_j
            endif
        endif
    endfor
endfor
```

Figure 1: Subscript Partition Algorithm

3.1.1 Merge

The *merge* operation described in the test algorithm merits more explanation. Since each separable and coupled subscript group contains a unique subset of indices, merge may be thought of as Cartesian product. In this loop nest,

```
      DO 10 i
        DO 10 j
10        A(i+1, j) = A(i, j) + C
```

the first position yields the direction vector $(<)$ for the i loop. The second position yields the direction vector $(=)$ for the j loop. The resulting Cartesian product is the single vector $(<, =)$. A more complex example is shown below:

```
      DO 10 i
        DO 10 j
10        A(i+1, 5) = A(i, N) + C
```

The first subscript yields the direction vector $(<)$ for the i loop. Since j does not appear in any subscript, we must assume the full set of direction vectors

$$\{\ (<), (=), (>)\ \}$$

for the j loop. The merge thus yields the following set of direction vectors:

$$\{\ (<,<), (<,=), (<,>)\ \}$$

Dependence test results for ZIV subscripts are treated specially. If a ZIV subscript proves independence, the dependence test algorithm halts immediately. If independence is not proved, the ZIV test does not produce direction vectors, so no merge is necessary.

4 Single Subscript Tests

We first consider dependence tests for single separable subscripts. All tests presented in this paper assume that the subscript being tested contains expressions that are *linear* in the loop index variables. A subscript expression is linear if it has the form:

$$a_1 i_1 + a_2 i_2 + \ldots + a_n i_n + e$$

where i_k is the index for the loop at nesting level k; all a_k, $1 \leq k \leq n$, are integer constants; and e is an expression possibly containing loop-invariant symbolic expressions. We assume in PFC that all direction vectors are possible for nonlinear subscripts.

4.1 ZIV Test

The ZIV test is a dependence test that takes two loop-invariant expressions. If the system determines that the two expressions cannot be equal, it has proved independence. Otherwise the subscript does not contribute any direction vectors and may be ignored. The ZIV test can be easily extended for symbolic expressions. Simply form the expression representing the difference between the two subscript expressions. If the difference simplifies to a non-zero constant, we have proved independence.

4.2 SIV Tests

A number of authors, notably Banerjee, Cohagan, and Wolfe [8, 14, 55], have published a Single-Index exact test for linear SIV subscripts based on finding all solutions to a simple Diophantine equation in two variables. Here we present a new exact test based on the idea of treating the most commonly occurring SIV subscripts as special cases. It provides greater efficiency and is easily extended to handle symbolics and coupled subscripts. We begin by separating SIV subscripts into two categories: *strong SIV* and *weak SIV* subscripts.

4.2.1 Strong SIV Subscripts

An SIV subscript for index i is said to be *strong* if it has the form $\langle ai + c_1, ai' + c_2 \rangle$; *i.e.*, if it is linear and the coefficients of the two occurrences of the index i are constant and equal [1, 10]. For strong SIV subscripts, we define the *dependence distance* as:

$$d \ = \ i' - i \ = \ \frac{c_1 - c_2}{a}$$

Then a dependence exists if and only if d is an integer and $|\, d\,| \leq U - L$, where U and L are the loop upper and lower bounds. For dependences that do exist, the *dependence direction* is given by:

$$\textit{direction} \ = \ \begin{cases} < & \text{if } d > 0 \\ = & \text{if } d = 0 \\ > & \text{if } d < 0 \end{cases}$$

The strong SIV test is thus an exact test that can be implemented very efficiently in a few operations. Since we calculate distance vectors in any case, we get the test for almost no additional cost.

Another advantage of the strong SIV test is that it can be easily extended to handle loop-invariant sym-

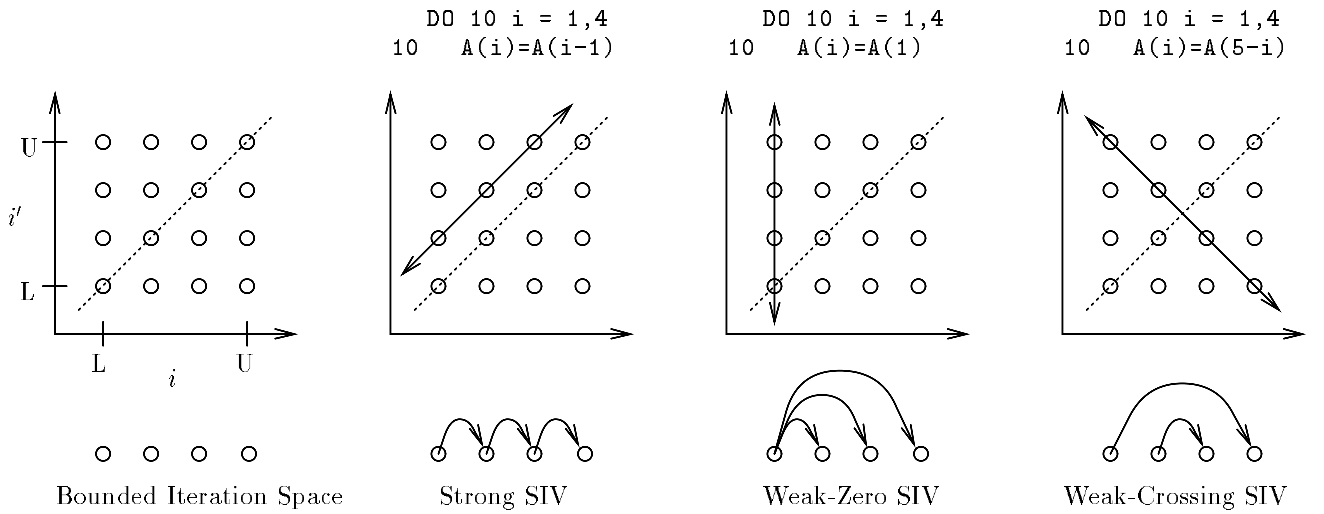

Figure 2: Geometric View of SIV Tests

bolic expressions. The trick is to first evaluate the dependence distance, d, symbolically. If the result is a constant, then the test may be performed as above. Otherwise calculate the difference between the loop bounds and compare the result with d symbolically. For instance, consider the following loop:

```
      DO 10 i = 1, N
  10    A(i+2N) = A(i+N) + C
```

The strong SIV test can evaluate the dependence distance, d, as $2N - N$, which simplifies to N. This is compared with the loop bounds symbolically, proving independence since $N > N - 1$.

4.2.2 Weak SIV Subscripts

A *weak* SIV subscript has the form $\langle a_1 i + c_1, a_2 i' + c_2 \rangle$, where the coefficients of the two occurrences of index i have different constant values. As stated previously, weak SIV subscripts may be solved using the Single-Index exact test. However, we also find it helpful to view the problem geometrically, where the dependence equation:

$$a_1 i + c_1 = a_2 i' + c_2$$

describes a line in the two dimensional plane with i and i' as the axes [10]. The weak SIV test can then be formulated as determining whether the line derived from the dependence equation intersects with any integer points in the space bounded by the loop upper and lower bounds, as shown in Figure 2. In particular, we find it advantageous to identify the following two special cases.

Weak-zero SIV Subscripts. We call the case where $a_1 = 0$ or $a_2 = 0$ a *weak-zero SIV* subscript. If a_2 is equal to zero, the dependence equation reduces to:

$$i = \frac{c_2 - c_1}{a_1}$$

We simply need to check that the resulting value for i is an integer and within the loop bounds. A similar check applies when a_1 is zero.

The weak-zero SIV test finds dependences caused by a particular iteration i. In scientific codes, i is usually the first or last iteration of the loop, eliminating one possible direction vector for the dependence. More importantly, weak-zero dependences caused by the first or last loop iteration may be eliminated by applying the *loop peeling* transformation [28]. For instance, consider the following simplified loop in the program *tomcatv* from the SPEC benchmark suite [49]:

```
      DO 10 i = 1, N
  10    Y(i, N) = Y(1, N) + Y(N, N)
```

The weak-zero SIV test can determine that the use of $Y(1, N)$ causes a loop-carried true dependence from the first iteration to all other iterations. Similarly, with aid from symbolic analysis the weak-zero SIV test can discover that the use of $Y(N, N)$ causes a loop-carried anti dependence from all iterations to the last iteration. By identifying the first and last iterations as the only cause of dependences, the weak-zero SIV test advises the user or compiler to peel the first and last iterations of the loop, resulting in the following parallel loop:

```
      Y(1, N) = Y(1, N) + Y(N, N)
      DO 10 i = 2, N-1
  10    Y(i, N) = Y(1, N) + Y(N, N)
      Y(N, N) = Y(1, N) + Y(N, N)
```

Weak-crossing SIV Subscripts. We label as *weak-crossing SIV* all subscripts where $a_2 = -a_1$; these subscripts typically occur as part of Cholesky decomposition. In these cases we set $i = i'$ and derive the dependence equation:

$$i = \frac{c_2 - c_1}{2a_1}$$

This corresponds to the intersection of the dependence equation with the line $i = i'$. To determine whether dependences exist, we simply need to check that the resulting value i is within the loop bounds, and is either an integer or has a non-integer part equal to $1/2$.

Weak-crossing SIV subscripts cause *crossing* dependences, loop-carried dependences whose endpoints all cross iteration i [1, 4]. These dependences may be eliminated using the *loop splitting* transformation [28]. For instance, consider the following loop from the Callahan-Dongarra-Levine vector test suite [13]:

```
    DO 10 i = 1, N
10    A(i) = A(N-i+1) + C
```

The weak-crossing SIV test determines that dependences exist between the definition and use of A, and that they all cross iteration $(N+1)/2$. Splitting the loop at that iteration results in two parallel loops:

```
    DO 10 i = 1,(N+1)/2
10    A(i) = A(N-i+1) + C
    DO 20 i = (N+1)/2 + 1, N
20    A(i) = A(N-i+1) + C
```

Both forms of weak SIV tests are also useful for testing coupled subscripts, described in Section 5. We rely on the Single-Index exact test to handle the general case.

4.3 Complex Iteration Spaces

SIV tests can be extended to handle complex iteration spaces, where loop bounds may be functions of other loop indices; for example, triangular or trapezoidal loops. We need to compute the minimum and maximum loop bounds for each loop index. Starting at the outermost loop nest and working inwards, we replace each index in a loop upper bound with its maximum value (or minimal value if it is a negative term). We do the opposite in the lower bound, replacing each index with its minimal value (or maximal if it is a negative term). We evaluate the resulting expressions to calculate the minimal and maximal values for the loop index, then repeat for the next inner loop. This algorithm returns the maximal range for each index, all that is needed for SIV tests.

4.4 MIV Tests

The Banerjee-GCD test [4, 8, 25, 55] may be employed to construct all legal direction vectors for linear subscripts containing multiple indices. In most cases the test can also determine the minimal dependence distance for the carrier loop. Since the literature in this area is extensive, we will not discuss it further here. PFC employs a special version of the Banerjee-GCD test enhanced for triangular loop nests [8, 26].

We note a special case of MIV subscripts called RDIV (Restricted Double Index Variable) subscripts that have form $\langle a_1 i + c_1, a_2 j + c_2 \rangle$. They are similar to SIV subscripts, except that i and j are distinct indices. By observing different loop bounds for i and j, SIV tests may also be extended to exactly test RDIV subscripts [55].

4.5 Symbolic Tests

As we have pointed out in the text, we can perform dependence testing in a natural way for subscripts with loop-invariant symbolic additive constants. The basic idea is that $c_2 - c_1$, the difference between the constant terms of each subscript expression, may be formed symbolically and simplified. The result may then be used like a constant.

In this section we describe a special test for independence between references to a subscripted variable that are contained in two different loops at the nesting level of the SIV index. In the pair of loops below,

```
    DO 10 i = 1, N₁
10    A(a₁i + c₁) = ...
    DO 20 j = 1, N₂
20    ... = A(a₂j + c₂)
```

we can use the following general test. Assume for the sake of simplicity that a_1 is greater or equal to zero. A dependence exists if the following dependence equation is satisfied:

$$a_1 i - a_2 j = c_2 - c_1$$

for some value of $i,\ 1 \le i \le N_1$ and $j,\ 1 \le j \le N_2$.

There are two cases to consider. First, a_1 and a_2 may have the same sign. In this case, $a_1 i - a_2 j$ assumes its maximum value for $i = N_1$ and $j = 1$ and its minimum value for $i = 1$ and $j = N_2$ (remember, a_1 and a_2 are non-negative). Hence, there is a dependence only if:

$$a_1 - a_2 N_2 \le c_2 - c_1 \le a_1 N_1 - a_2$$

If either inequality is violated, the dependence cannot exist.

In the second case, a_1 and a_2 have different signs. In this case, $a_1 i - a_2 j$ assumes its maximum for $i = N_1$ and $j = N_2$, so there is a dependence only if:

$$a_1 - a_2 \le c_2 - c_1 \le a_1 N_1 - a_2 N_2$$

If either inequality is violated, the dependence cannot exist.

It should be noted that these inequalities are just special cases of the Banerjee inequality. However, when they are stated in this form, it is obvious that they can be formulated for symbolic values of $c_1,\ c_2,\ N_1$ and N_2. Furthermore, this test may also be used to test for dependence in the same loop, with $N_1 = N_2$.

Our empirical study in Section 6 shows that symbolic testing techniques significantly enhance the effectiveness of dependence tests in PFC. Any symbolic expressions that remain at the end of dependence testing may also be used as a *user query* in an interactive system, or as a condition to *break* the dependence at run-time.

5 Delta Test

The tests used for separable subscripts can also be used on each subscript of a coupled group—if any test proves independence, then no dependence exists. However, we have already seen that subscript-by-subscript testing in

INPUT: coupled SIV and/or MIV subscripts

OUTPUT: hybrid distance/direction vector,
  constrained MIV subscripts

initialize elements of constraint vector $\vec{C}$ to $\langle none \rangle$
while $\exists$ untested SIV subscripts **do**
  apply SIV test to all untested SIV subscripts,
    return independence or
    derive new constraint vector $\vec{C'}$
  $\vec{C'} \leftarrow \vec{C} \cap \vec{C'}$
  if $\vec{C'} = \emptyset$ **then**
    return independence
  else if $\vec{C} \neq \vec{C'}$ **then**
    $\vec{C} \leftarrow \vec{C'}$
    propagate constraint $\vec{C}$ into MIV subscripts,
      possibly creating new ZIV or SIV subscripts
    apply ZIV test to untested ZIV subscripts,
      return independence or continue
  endif
endwhile
while $\exists$ untested RDIV subscripts **do**
  test and propagate RDIV constraints
endwhile
test remaining MIV subscripts, then
  intersect resulting direction vectors with $\vec{C}$
return distance/direction vectors from $\vec{C}$

Figure 3: Delta Test Algorithm

a coupled group may yield false dependences. Some recent research has focused on overcoming this deficiency [38, 50, 56]. In this section we present the Delta test, a multiple subscript test designed to be exact yet efficient for common coupled subscripts. Figure 3 presents an overview of the Delta test algorithm.

The main insight behind the Delta test is that *constraints* derived from SIV subscripts may be propagated into other subscripts in the same coupled group efficiently, usually without any loss of precision. Since most coupled subscripts in scientific Fortran codes are simple, in practice the Delta test is an exact yet fast multiple subscript test.

The Delta test can detect independence if any of its component ZIV or SIV tests determine independence. Otherwise it converts all SIV subscripts into constraints, propagating them into MIV subscripts where possible. It repeats until no new constraints are found, then propagates constraints for coupled RDIV subscripts. Remaining MIV subscripts are tested; the results are intersected with existing constraints. We describe the Delta test algorithm in greater detail in the following sections.

5.1 Constraints

Constraints are assertions on indices derived from subscripts. For instance, the subscript $\langle a_1 i + c_1, a_2 i' + c_2 \rangle$ generates the constraint $a_1 i - a_2 i' = c_2 - c_1$ for index i. A dependence distance is an example of a simple constraint. The *constraint vector* $\vec{C} = (\delta_1, \delta_2, \ldots, \delta_n)$ is a vector with one constraint for each of the n indices in the coupled subscript group. It is used in the Delta test to store constraints generated from SIV tests, and can be easily converted to distance or direction vectors. A constraint δ may have the following form:

- *dependence line* — a line $\langle ax + by = c \rangle$ representing the dependence equation
- *dependence distance* — the value $\langle d \rangle$ of the dependence distance; it is equivalent to the dependence line $\langle x - y = -d \rangle$
- *dependence point* — a point $\langle x, y \rangle$ representing dependence from iteration x to y

Dependence distances and lines derive directly from the strong and weak SIV tests. Dependence points result from intersecting constraints, as described in the next section.

5.2 Intersecting Constraints

Since dependence equations from all subscripts must be solved simultaneously for dependences to exist, *intersecting* constraints from each subscript results in greater precision. If the result of the intersection is the empty set, no dependence is possible. Constraint intersection has been employed for both direction vectors [53] and coupled SIV subscripts [1, 10]. The version employed by the Delta test is equivalent to an exact multiple subscript SIV test.

Dependence distances are the easiest to intersect; a simple comparison suffices. If all distances are not equal, then no dependences exist. For example, reconsider the following loop nest from Section 2.2:

```
   DO 10 i
10   A(i+1, i+2) = A(i, i) + C
```

Applying the strong SIV test to the first subscript derives a dependence distance of 1. Doing the same for the second subscript derives a distance of 2. To intersect the two constraints we perform a comparison. This results in the empty set, proving independence.

It turns out that even complex constraints from SIV subscripts may be intersected exactly. Recall that each dependence equation from a SIV subscript may be viewed as a line in a two-dimensional plane. Intersecting constraints from multiple SIV subscripts then corresponds to calculating the point(s) of intersection for lines in a plane. No dependence exists if the lines do not intersect at a common point within the loop bounds, or if the coordinates of this point do not have integer values. If all dependence equations intersect at a single *dependence point*, its coordinates are the only two iterations that actually cause dependence.

```
   DO 10 i
10   A(i, i) = A(1, i-1) + C
```

INPUT: constraints δ_1, δ_2 and loop bounds U, L

OUTPUT: new constraint δ or $\emptyset$

```
{∗ either δ1 or δ2 = ⟨none⟩ ∗}
if δ1 = ⟨none⟩ then
   return δ2
else if δ2 = ⟨none⟩ then
   return δ1
{∗ both δ1 and δ2 are dependence distances ∗}
else if δ1 = ⟨d1⟩ and δ2 = ⟨d2⟩ then
   if d1 = d2 then
      return ⟨d1⟩
   else
      return ∅
   endif
{∗ both δ1 and δ2 are dependence points ∗}
else if δ1 = ⟨x1, y1⟩ and δ2 = ⟨x2, y2⟩ then
   if x1 = x2 and y1 = y2 then
      return ⟨x1, y1⟩
   else
      return ∅
   endif
{∗ both δ1 and δ2 are dependence lines/distances ∗}
else if δ1 = ⟨a1x + b1y = c1⟩ and
        δ2 = ⟨a2x + b2y = c2⟩ then
   {∗ lines are parallel if slopes are equal ∗}
   if a1/b1 = a2/b2 then
      if c1/b1 = c2/b2 then
         return ⟨a1x + b1y = c1⟩
      else
         return ∅
      endif
   {∗ lines must intersect if not parallel ∗}
   else
      ⟨x1, y1⟩ ← intersection of δ1 and δ2
      if L ≤ x1 ≤ U and x1 is an integer and
            L ≤ y1 ≤ U and y1 is an integer then
         return ⟨x1, y1⟩
      else
         return ∅
      endif
   endif
{∗ either δ1 is a dependence line/distance        ∗}
{∗  and δ2 is a dependence point, or vice versa  ∗}
{∗ without loss of generality, assume the former ∗}
else δ1 = ⟨a1x + b1y = c1⟩ and δ2 = ⟨x1, y1⟩
   if a1x1 + b1y1 = c1 then
      return ⟨x1, y1⟩
   else
      return ∅
   endif
endif
```

Figure 4: Constraint Intersection

For instance, in this example loop testing the first and second subscripts in the pair of references to A derives the dependence lines $\langle i = 1\rangle$ and $\langle i = i' - 1\rangle$, respectively. These dependence lines intersect at the dependence point $\langle 1, 2\rangle$, indicating that the only dependence is from the first to the second iteration. Since calculating the intersection of lines in a plane can be performed precisely, constraint intersection is exact.

The full constraint intersection algorithm is shown in Figure 4. Note that for simplicity dependence distances are also treated as lines at places in the algorithm.

5.3 Propagating Constraints

5.3.1 SIV Constraints

A major contribution of the Delta test is its ability to *propagate* constraints derived from SIV subscripts into coupled MIV subscripts, usually without loss of precision. The resulting *constrained* subscript can then be tested with greater efficiency and precision. Figure 5 shows the constraint propagation algorithm. Its goal is to utilize SIV constraints for each index to eliminate instances of that index in the target MIV subscript. We demonstrate the algorithm in the following example:

```
      DO 10 i
        DO 10 j
10        A(i+1, i+j) = A(i, i+j)
```

Applying the strong SIV test to the first subscript of array A derives a dependence distance of $\langle 1\rangle$ for index i. We can propagate this constraint into the second subscript to eliminate both occurrences of i, resulting in the constrained SIV subscript $\langle j - 1, j\rangle$. We then apply the strong SIV test to derive a distance of -1 on loop j. All subscripts have been tested, so the Delta test is finished. We merge the elements of the constraint vector to determine that a dependence exists with distance vector $(1, -1)$.

Constraint propagation in this example is *exact* because we were able to eliminate both instances of index i in the constrained subscript. Our empirical study in Section 6 shows that this is frequently the case for scientific codes. In general the algorithm may only eliminate one occurrence of an index. This results in improved precision when testing coupled groups, but is not exact. If desired, additional precision may be gained by utilizing the constraint to reduce the range of the remaining index, as in Fourier-Motzkin Elimination [44].

The constraint propagation algorithm is an incremental adaptation of the λ-test heuristic for selecting linear combinations of subscript expressions. It has also been extended to efficiently handle constraints from SIV tests and linearly dependent subscripts [38]. Below we present some more examples of the Delta test.

Multiple Passes The Delta test algorithm iterates if MIV subscripts are reduced to SIV subscripts, since they may produce new constraints. The following loop nest demonstrates this:

INPUT: MIV subscript with form
$\langle a_1 i_1 + \ldots + a_n i_n + e,\ a'_1 i'_1 + \ldots + a'_n i'_n + e' \rangle$,
and constraint vector $\vec{C} = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$

OUTPUT: *constrained* ZIV, SIV, or MIV subscript

for each index i_k with nonzero a_k or a'_k **do**
  if δ_k = dependence distance $\langle d \rangle$ **then**
    $e \leftarrow e - a_k d$; $a_k \leftarrow 0$; $a'_k \leftarrow a'_k - a_k$
  else if δ_k = dependence line $\langle \alpha x + \beta y = c \rangle$ **then**
    if $\alpha = 0$ **then**
      $e \leftarrow e - a'_k c / \beta$; $a'_k \leftarrow 0$
    else if $\beta = 0$ **then**
      $e \leftarrow e + a_k c / \alpha$; $a_k \leftarrow 0$
    else if $\alpha = \beta$ **then**
      $e \leftarrow e + a_k c / \alpha$; $a_k \leftarrow 0$; $a'_k \leftarrow a'_k + a_k$
    else
      {* multiply terms of subscript by α to *}
      {* retain integer coefficients in result *}
      for each $\tau \in \{a_1, \ldots, a_n, a'_1, \ldots, a'_n, e, e'\}$ **do**
        $\tau \leftarrow \alpha \tau$
      endfor
      $e \leftarrow e + a_k c$; $a_k \leftarrow 0$; $a'_k \leftarrow a'_k + a_k \beta$
    endif
  else if δ_k = dependence point $\langle x, y \rangle$ **then**
    $e \leftarrow e + a_k x - a'_k y$
    $a_k \leftarrow 0$; $a'_k \leftarrow 0$
  endif
endfor

Figure 5: Constraint Propagation

```
    DO 10 i
      DO 10 j
        DO 10 k
10        A(j-i, i+1, j+k) = A(j-i, i, j+k)
```

In the first pass of the Delta test, the second subscript is tested, producing a dependence distance of $\langle 1 \rangle$ on the i loop. This constraint can be propagated into the first subscript, resulting in the subscript $\langle j+1, j \rangle$.

Since a new SIV subscript has been created, the algorithm repeats. On the second pass, the new subscript is tested to produce a distance of $\langle 1 \rangle$ on the j loop. This constraint is then propagated into the third subscript to derive the subscript $\langle k-1, k \rangle$. The new SIV subscript causes another pass that discovers a distance of -1 on the k loop. Since all SIV subscripts have been tested, the Delta test halts at this point, returning the distance vector $(1, 1, -1)$.

Improved Precision The Delta test may also improve the precision of other dependence tests on any remaining constrained MIV subscripts.

```
    DO 10 i = 1,100
      DO 10 j = 1,100
10      A(i-1, 2i) = A(i, i+j+110)
```

When applied to each subscript in this example loop, Banerjee's inequalities show possible dependence for both subscripts. The Delta test can improve this by converting the first subscript into a dependence distance of $\langle -1 \rangle$ and propagating it into the second subscript to produce the constrained MIV subscript $\langle 2, j - i + 110 \rangle$. Banerjee's inequalities can now detect independence for the constrained subscript.

```
    DO 10 i
      DO 10 j
10      A(i, 2j+i) = A(i, 2j-i+5)
```

Similarly, in this example loop the GCD test shows integer solutions for both subscripts. However, propagating the distance constraint $\langle 0 \rangle$ for i from the first subscript into the second subscript yields the constrained MIV subscript $\langle 2j, 2j - 2i + 5 \rangle$. The GCD test can now detect independence since the GCD of the coefficients of all the indices is 2, which does not divide evenly into the constant term 5.

Distance Vectors The Delta test is particularly useful for analyzing dependences in *skewed* loops [27, 36, 54], including upper triangular loops skewed by *loop normalization* [3, 53]. Consider the following simplified kernel from the *Livermore Loops* [41]:

```
    DO 10 i = 1, N
      DO 10 j = 1, N
10      A(i,j) = A(i-1,j) + A(i,j-1)
               + A(i+1,j) + A(i,j+1)
```

Since all subscripts are separable, the strong SIV test can be applied to calculate distance vectors of $(1, 0)$ and $(0, 1)$ for the dependences in the loop nest. This dependence information can be used to skew the inner loop to expose parallelism, resulting in the following loop nest:

```
    DO 10 i = 1, N
      DO 10 j = 1+i, N+i
10      A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
                 + A(i+1,j-i) + A(i,j-i+1)
```

At this point, most dependence tests are unable to calculate distance vectors due to the presence of MIV subscripts. However, the Delta test can easily propagate distance constraints for i from the first subscript into the second subscript to derive the distance vectors $(1, 1)$ and $(0, 1)$. This dependence information may then be used to guide further optimizations such as loop interchange, loop blocking, or scalar replacement [11, 51, 55].

5.3.2 Restricted DIV Constraints

In the previous section we showed how SIV constraints may be propagated. Propagating MIV constraints is expensive in the general case. However, we present a method to handle an important special case consisting of coupled RDIV subscripts (discussed in Section 4.4). For simplicity, we consider array references with the following form:

```
    DO 10 i
      DO 10 j
10      A(i1 + c1, i2 + c2) = A(i3 + c3, i4 + c4)
```

program	type	lines	subrs	array pairs tested 1D	2D	3D	4D	total	subscript pairs sep	coup	total	nonlin subs
	RiCEPS											
baro	Shallow Water Atmosphere	1002	7	34	360	0	0	394	754	0	754	0
euler	1D Unsteady Euler	1200	14	106	294	0	0	400	694	0	694	21
heat2d	Heat Conduction System	336	2	352	251	0	0	603	822	32	854	149
linpackd	Linear Algebra Benchmark	797	11	52	36	0	0	88	74	50	124	29
mhd2d	2D MHD Equations	927	14	280	196	21	0	497	711	24	735	232
onedim	Eigenfunction/Eigenenergies	1016	16	297	209	0	0	506	567	148	715	47
shear	3D Turbulence	915	15	787	0	828	0	1615	2415	856	3271	368
simple	2D Hydrodynamics	1892	18	174	163	0	0	337	500	0	500	1
sphot	Particle Transport	1144	7	282	77	0	0	359	436	0	436	21
vortex	Vortex Simulation	709	20	174	42	0	0	216	258	0	258	0
	Perfect											
adm (APS)	Pseudospectral Air Pollution	6105	97	647	160	521	0	1328	2506	24	2530	19
arc2d (SRS)	2D Fluid Flow Solver	3965	39	168	405	4396	0	4969	13172	994	14166	9
bdna (NAS)	Molecular Dynamics of DNA	3980	43	1787	269	0	0	2056	2085	240	2325	562
dyfesm (SDS)	Structural Dynamics	7608	78	268	866	7	7	1148	1897	152	2049	0
flo52 (TFS)	Transonic Inviscid Flow	1986	28	54	256	1624	0	1934	5284	154	5438	0
mdg (LWS)	Molecular Dynamics of Water	1238	16	928	5	0	0	933	934	4	938	296
mg3d (SMS)	Depth Migration	2812	28	749	52	133	40	974	1412	0	1412	494
ocean (OCS)	2D Ocean Simulation	4343	36	439	29	0	0	468	497	0	497	322
qcd (LGS)	Quantum Chromodynamics	2102	34	7774	171	3	0	7948	8115	10	8125	0
spec77 (WSS)	Weather Simulation	3885	65	1187	477	54	0	1718	2255	48	2303	122
spice (CSS)	Circuit Simulation	18521	128	470	51	0	0	521	546	26	572	50
track (MTS)	Missile Tracking	3735	34	242	115	0	0	357	464	8	472	7
trfd (TIS)	2 Electron Integral Transform	485	7	39	64	0	0	103	135	32	167	6
	SPEC											
doduc	Thermohydraulical Modelization	5334	41	656	895	0	0	1551	2446	0	2446	11
fpppp	2 Electron Integral Derivative	2718	38	997	108	0	0	1105	1213	0	1213	3
matrix300	Matrix Multiplications	439	6	0	8	0	0	8	14	2	16	0
nasa7	NASA Ames Fortran Kernels	1105	17	155	106	329	347	937	2267	475	2742	24
tomcatv	Mesh Generation	195	1	0	207	0	0	207	272	142	414	0
	Math Libraries											
eispack	Eigensystems Library	11519	75	2995	10295	0	0	13290	10823	12762	23585	2052
linpack	Linear Algebra Library	7427	51	3224	869	0	0	4093	4228	734	4962	290

Table 1: Program Characteristics

	ZIV		SIV												MIV			Delta				symbolics used	
			strong			weak-zero			weak-cross			other											
program	A	S/I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	P	S	I
baro	386	78	294	294	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	17	17
euler	267	162	240	236	3	13	0	0	4	0	0	0	0	0	6	0	0	0	0	0	0	114	2
heat2d	450	127	127	127	0	43	32	0	0	0	0	0	0	0	20	16	0	11	6	0	5	1	0
linpackd	13	0	57	57	0	4	0	0	0	0	0	0	0	0	2	2	0	23	16	0	2	10	0
mhd2d	55	28	384	384	1	115	12	0	4	0	0	0	0	0	107	0	0	12	0	0	0	4	1
onedim	39	11	452	448	0	91	91	4	0	0	0	0	0	0	32	31	25	31	14	2	14	158	29
shear	179	72	1327	1327	151	537	386	168	4	0	0	0	0	0	223	32	32	233	40	0	0	44	1
simple	112	44	358	357	0	5	5	0	0	0	0	0	0	0	2	2	0	0	0	0	0	374	44
sphot	280	191	134	134	0	1	1	0	0	0	0	0	0	0	21	0	0	0	0	0	0	12	0
vortex	42	24	216	216	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0
adm	632	210	1355	1327	142	17	17	0	97	22	12	52	52	0	32	12	12	0	0	0	0	142	21
arc2d	9501	3628	2821	2807	0	149	111	12	0	0	0	0	0	0	0	0	0	107	38	38	0	3024	454
bdna	85	21	1393	1249	14	177	69	0	0	0	0	0	0	0	79	33	0	119	28	0	1	454	14
dyfesm	1073	629	514	514	0	55	55	0	0	0	0	0	0	0	26	14	1	69	36	1	6	86	0
flo52	2372	1043	2607	2599	6	101	92	0	16	2	0	8	8	0	0	0	0	77	9	0	0	52	10
mdg	464	415	177	177	0	211	0	0	0	0	0	0	0	0	85	0	0	0	0	0	0	20	0
mg3d	362	20	456	456	54	385	2	0	0	0	0	36	36	0	41	0	0	0	0	0	0	5	1
ocean	20	2	108	107	8	152	1	0	4	0	0	8	4	0	157	34	0	0	0	0	0	24	1
qcd	7945	6725	149	107	0	22	15	0	0	0	0	0	0	0	1	1	0	5	5	5	0	27	1
spec77	385	62	1660	1631	273	94	18	0	12	12	12	0	0	0	91	9	0	24	2	2	0	286	201
spice	258	102	229	210	8	1	1	0	0	0	0	10	0	0	8	8	1	9	9	1	3	46	8
track	230	95	181	181	5	3	3	0	0	0	0	0	0	0	7	0	0	4	0	0	0	15	0
trfd	6	0	71	71	0	24	24	0	0	0	0	0	0	0	50	28	0	16	0	0	0	18	0
doduc	795	424	1638	1627	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	58	0
fpppp	902	11	232	232	110	0	0	0	1	0	0	0	0	0	15	15	0	0	0	0	0	4	0
matrix300	3	0	12	12	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
nasa7	832	410	948	947	0	196	154	0	5	1	0	0	0	0	28	26	13	101	37	1	3	515	12
tomcatv	32	13	204	204	0	78	73	1	26	26	0	0	0	0	0	0	0	71	4	0	0	0	0
eispack	4215	471	6680	6227	2	2707	2548	92	329	225	0	0	0	0	2772	915	404	5328	2641	892	314	7937	871
linpack	2719	227	1281	1271	88	59	50	0	111	6	0	0	0	0	118	118	2	329	102	3	28	572	95

Table 2: Dependence Test Application/Success/Independence Frequencies

When i_1, i_2 are instances of index i, and i_3, i_4 are instances of index j, a constraint between i and j is derived from the first subscript that may be propagated into the second subscript employing the algorithms for SIV subscripts discussed previously. The only additional consideration is that bounds for i and j may differ.

More commonly, i_1, i_4 are instances of index i, and i_2, i_3 are instances of index j. This yields the following set of dependence equations:

$$\begin{aligned} i + c_1 &= j' + c_3 \\ j + c_2 &= i' + c_4 \end{aligned}$$

Each dependence equation may be tested separately without loss of precision when checking for dependence. However, both equations must be considered simultaneously when determining which distance or direction vectors are possible.

We can propagate constraints for these coupled RDIV subscripts by considering instances of index i in the second reference as $i + \Delta_i$, where Δ_i is the dependence distance between the two occurrences of i. We do the same for index j to produce the following set of dependence equations:

$$\begin{aligned} i + c_1 &= j + \Delta_j + c_3 \\ j + c_2 &= i + \Delta_i + c_4 \end{aligned}$$

It is clear that these the first two equations may be combined to result in the equation:

$$\Delta_i + \Delta_j = c_1 + c_2 - c_3 - c_4$$

We can then check this dependence equation when testing for a specific distance or direction vector.

Array Transpose We show how RDIV constraints may be used in this array transpose example:

```
      DO 10 i
        DO 10 j
10        A(i, j) = A(j, i) + C
```

Propagating RDIV constraints results in the dependence equation $\Delta_i + \Delta_j = 0$. As a result, distance vectors must have the form $(d, -d)$, and the only valid direction vectors are $(<, >)$ and $(=, =)$. The direction vector $(>, <)$ may be ignored since it is equivalent to a reversed dependence with direction vector $(<, >)$ [9]. All dependences are thus carried on the outer loop; the inner loop may be executed in parallel.

5.4 Precision and Complexity

The precision of the Delta test depends on the nature of the coupled subscripts being tested. The SIV tests applied in the first phase are exact. The constraint intersection algorithm is also exact, since we can calculate the intersection of any number of lines in a plane precisely. The Delta test is thus exact for any number of coupled SIV subscripts.

In the constraint propagation phase, weak-zero SIV constraints and dependence points may always be applied exactly, since they assign values to occurrences of an index in a subscript. Dependence distances (from strong SIV subscripts) may also be propagated into MIV subscripts without loss of precision when the coefficients of the corresponding index are equal. Fortunately, this is frequently the case in scientific codes.

When constraints can be propagated exactly and all subscripts *uncoupled* by eliminating shared indices, the Delta test prevents loss of precision due to multiple subscripts. At its conclusion, if the Delta test has tested all subscripts using ZIV and SIV tests, the answer is exact. If only separable MIV subscripts remain, the Delta test is limited by the precision of the single subscript tests applied to each subscript. Research has shown that the Banerjee-GCD test is usually exact for single subscripts [6, 30, 37], so the Delta test is also likely to be exact for these cases.

There are three sources of imprecision for the Delta test. First, constraint propagation of dependence lines and distances may be imprecise if an index cannot be completely eliminated from both references in the target subscripts. Second, complex iteration spaces such as triangular loops may impose constraints between subscripts not utilized by the Delta test.

Finally, the Delta test does not propagate constraints from general MIV subscripts. As a result, coupled MIV subscripts may remain at the end of the Delta test. More general but expensive multiple subscript dependence tests such as the λ or Power tests may be used in these cases [38, 56].

Since each subscript in the coupled group is tested at most once, the complexity of the Delta test is linear in the number of subscripts. However, constraints may be propagated into subscripts multiple times.

6 Empirical Results

In this section we present empirical results to demonstrate that our dependence tests are applicable for scientific Fortran codes. PFC currently performs the following dependence tests:

- subscript classification and partitioning
- ZIV test (symbolic)
- strong SIV test (symbolic)
- weak SIV test (including special cases)
- MIV tests (GCD, triangular Banerjee)
- Delta test (constraint intersection, propagation of distance constraints only)

For this study we measured the number times each dependence test was applied by PFC when processing four groups of Fortran programs: RiCEPS (Rice Compiler Evaluation Program Suite), the Perfect and SPEC benchmark suites [16, 49], and two math libraries, *eispack* and *linpack*.

Explanation Table 1 provides the number of lines and subroutines for each program, a histogram of the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscripts pairs found.

category	ZIV	SIV				MIV	Delta	symbolics used
		strong	weak-zero	weak-cross	other			
summed over all programs								
% of all tests applied	44.76	33.98	6.77	0.79	0.15	5.07	8.49	
% of all successful tests	30.97	51.88	7.64	0.60	0.20	2.63	6.07	28.52
% of all proven independences	85.43	4.85	1.55	0.13	0	2.75	5.30	9.99
% of applications that were successful	43.99	97.08	71.77	47.96	87.72	33.04	45.46	
% of applications that proved independence	43.99	3.29	5.28	3.92	0	12.49	14.38	
averaged over all programs								
% of all tests applied	36.38	45.76	7.72	0.61	0.30	5.02	4.22	
% of all successful tests	21.85	67.33	4.79	0.41	0.44	2.98	2.20	15.88
% of all proven independences	73.97	17.82	2.28	0.25	0	3.58	2.10	14.64
% of applications that were successful	34.74	97.88	65.42	27.41	70.00	49.47	32.03	
% of applications that proved independence	34.74	4.03	1.86	9.36	0	9.09	9.08	

Table 3: Comparison of Dependence Tests

Table 2 describes the usage and success frequencies of the dependence tests for each program. For each test, the table shows the number of times the test was (A) *applied*, (S) *succeeded* in eliminating at least one direction vector, and (I) proved *independence*. Note that the S and I columns are combined for the ZIV test because they are always identical.

The A, S, and I columns for the Delta test reflect frequencies measured for constraint intersection only. A separate column (P) indicates the number of times distance constraints were *propagated* into MIV subscripts. Results for dependence tests applied on the constrained subscripts are credited to the test invoked. The last two columns in Table 2 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

Table 3 summarizes the effectiveness of each dependence test relative to other tests by presenting the percentage contribution of each test to the total number of applications, successes, and independences. Also displayed is the absolute effectiveness of each test; *i.e.*, the percentage of applications of each test that proved independence or was successful in eliminating one or more direction vectors.

In order to limit bias toward either large or small programs, two groups of results are presented. In the first group, percentages are calculated after summing results over all programs. In the second group, percentages are calculating for each program and then averaged.

Analysis PFC applied dependence tests 74889 times (88% of all subscript pairs). Subscript pairs were not tested if they were nonlinear (6%), or if tests on other subscripts in the same multidimensional array have already proven independence. Over all array reference pairs tested, most subscript pairs were ZIV (49%) or strong SIV (37%). Few of the subscripts tested were MIV (5.5%). The ZIV and strong SIV tests combined for most of the successful tests (82%). The ZIV test accounted for almost all reference pairs proven independent (85%).

Most subscripts were separable. Coupled subscripts (20% overall) were concentrated in a few programs, notably *eispack* (75% of all coupled subscripts). Most of the 8449 coupled groups found were of size two; some coupled groups of size three were also encountered.

The Delta test constraint intersection algorithm tested 6570 coupled groups exactly (78%). Propagation of distance constraints was applied in 376 cases (4.4%), converting MIV subscripts into SIV form in all but 28 cases. The Delta test thus managed to test 6918 coupled groups exactly (82%), using only constraint intersection and propagation of dependence distances. We expect this percentage to improve once we implement full constraint propagation, including propagation of RDIV constraints.

Our results show that the SIV and Delta tests presented in this paper tested most subscripts exactly. MIV tests such as the Banerjee-GCD test are only needed for a small fraction of all subscripts (5%), though they are important for certain programs. Many of the successful tests required PFC's ability to manipulate symbolic additive constants (28.5%). This indicates the importance of symbolic analysis and dependence testing.

7 Related Work

In this section, we discuss the large body of work in the field of dependence testing. The suite of tests presented in this paper are distinguished by the fact that they combine high precision and efficiency by targeting a simple yet common subset of all possible subscripts.

7.1 Integer and Linear Programming

Since testing linear subscript functions for dependence is equivalent to finding simultaneous integer solutions within loop limits, one approach is to employ integer programming methods [18, 44]. Linear programming techniques such as Shostak's loop residue [46] or Karmarkar's method [24] are also applicable, though integer solutions are not guaranteed. Unfortunately, while integer and linear programming techniques are suitable for solving large systems of equations, their high initialization costs and implementation complexity make them less desirable for dependence testing.

7.2 Single Subscript Tests

The earliest work on dependence tests concentrated on deriving distance vectors from strong SIV subscripts

[34, 36, 42]. Cohagan [14] described a test that analyzes general SIV subscripts symbolically. Banerjee and Wolfe [7, 53] developed the current form of the Single-Index exact test.

For MIV subscripts, the GCD test may be used to check *unconstrained* integer solutions [6, 25]. Banerjee's inequalities provide a useful general-purpose single subscript test for *constrained* real solutions [7]. It has also been adapted to provide many different types of dependence information [4, 8, 9, 25, 26, 53]. Research has shown that Banerjee's inequalities are exact in many common cases [6, 30, 37], though results have not yet been extended for direction vectors or complex iteration spaces.

The I-test developed by Kong *et al.* integrates the GCD and Banerjee tests and can usually prove integer solutions [31]. Gross and Steenkiste propose an efficient interval analysis method for calculating dependences for arrays [21]. Unfortunately their method does not handle coupled subscripts, and is unsuitable for most loop transformations since distance and direction vectors are not calculated. Lichnewsky and Thomasset describe symbolic dependence testing in the VATIL vectorizer [39]. Haghighat and Polychronopoulos propose a flow analysis framework to aid symbolic tests [22].

Execution conditions may also be used to refine dependence tests. Wolfe's All-Equals test checks for loop-independent dependences invalidated by control flow within the loop [53]. Lu and Chen's *subdomain test* incorporates information about indices from conditionals within the loop body [40]. Klappholz and Kong have extended Banerjee's inequalities to do the same [29].

7.3 Multiple Subscript Tests

Early approaches to impose simultaneity in testing multidimensional arrays include intersecting direction vectors from each dimension [53] and linearization [9, 20]; they proved inaccurate in many cases. True multiple subscript tests provide precision at the expense of efficiency by considering all subscripts simultaneously. In comparison, the Delta test propagates constraints incrementally as needed.

Fourier-Motzkin Elimination Many of the earliest multiple subscript tests utilized Fourier-Motzkin elimination, a linear programming method based on pairwise comparison of linear inequalities. Kuhn [35] and Triolet *et al.* [48] represent array accesses in convex *regions* that may be intersected using Fourier-Motzkin elimination. Regions may also be used to summarize memory accesses for entire segments of the program. These techniques are flexible but expensive. Triolet found that using Fourier-Motzkin elimination for dependence testing takes from 22 to 28 times longer than conventional dependence tests [47].

Constraint-Matrix The Constraint-Matrix test developed by Wallace is a simplex algorithm modified for integer programming [50]. Its precision and expense are difficult to ascertain since it halts after an arbitrary number of iterations to avoid cycling. The simplex algorithm has worst case exponential complexity, but takes only linear time for most linear programming problems. However, Schrijver states that in combinatorial problems where coefficients tend to be 1, 0, or -1, the simplex algorithm is slow and will cycle for certain pivot rules [44].

λ-test Li *et al.* present the λ-test, a multidimensional version of Banerjee's inequalities that checks for simultaneous constrained real-valued solutions [38]. The λ-test forms linear combinations of subscripts that eliminate one or more instances of indices, then tests the result using Banerjee's inequalities. Simultaneous real-valued solutions exist if and only if Banerjee's inequalities finds solutions in all the linear combinations generated.

The λ-test can test direction vectors and triangular loops. Its precision may be enhanced by also applying the GCD or Single-Index exact tests to the pseudosubscripts generated. However, there is no obvious method to extend the λ-test to prove the existence of simultaneous integer solutions. The λ-test is exact for two dimensions if unconstrained integer solutions exist and the coefficients of index variables are all 1, 0 or -1 [37]. However, even with these restrictions it is not exact for three or more coupled dimensions.

The Delta test may be viewed as a restricted form of the λ-test that trades generality for greater efficiency and precision.

Multidimensional GCD Banerjee's multidimensional GCD test checks for simultaneous unconstrained integer solutions in multidimensional arrays [8]. It applies Gaussian elimination modified for integers to create a compact system where all integer points provide integer solutions to the original dependence system. It can also be extended to provide an exact test for distance vectors [56].

Power Test Wolfe and Tseng's Power test gains great precision by applying loop bounds using Fourier-Motzkin elimination to the dense system resulting from the multidimensional GCD test [56]. The Power test is expensive, but is also flexible and well-suited for providing precise dependence information such as direction vectors in imperfectly nested loops, loops with complex bounds, and non-direction vector constraints.

Both the Constraint-Matrix and λ-tests require that a pretest be used to eliminate linearly dependent subscripts. In comparison, the Power and Delta tests can detect and discard linearly dependent subscripts as part of their basic algorithm.

7.4 Empirical Studies

Li *et al.* showed that for coupled subscripts, multiple subscript tests may detect independence in up to 36% more cases than subscript-by-subscript tests in libraries such as *eispack* [38]. Our results for *eispack* demonstrate that the Delta test is as effective in testing coupled subscripts. A comprehensive empirical study of array subscripts and conventional dependence tests was performed by Shen *et al.* [45]. Our study focuses on the complexity of subscripted references and the effectiveness of our partition-based dependence tests. We also provide some data on the efficacy of symbolic dependence tests.

8 Conclusions

This paper presents a strategy for dependence testing based on the thesis that array references in real codes have simple subscripts. Our empirical results show that in practice the dependence tests described in this paper are extremely precise, fast, and applicable to the vast majority of all subscripts in scientific codes.

In the few cases where our tests are inapplicable, we can afford applying more expensive tests since their cost may be effectively amortized. Experience has shown that dependence analysis can be highly useful for both scalar and parallel compilers [2, 11, 33]. We feel that the dependence tests described in this paper make dependence analysis more efficient and hence practical for every compiler.

9 Acknowledgements

As with most research, the suite of dependence tests we have described in this paper owes much to the contributions of others. The first version of PFC [3, 25] employed subscript-by-subscript testing using the Banerjee-GCD test extended to calculate the level, minimum distance, and interchange information for each dependence. This strategy proved very useful for PFC's layered vectorization algorithm.

In the mid-eighties, Randy Allen and others implemented the strong SIV test [1] that was applied to each subscript if all the subscripts were SIV and separable—this change dramatically improved the efficiency of dependence testing. Both the strong SIV and Banerjee tests were enhanced to deal with symbolic additive constants. Some simultaneity was also added for multidimensional arrays by comparing dependence distances for each index.

David Callahan later extended PFC to handle weak SIV subscripts and general constraint intersection [10]. In addition, he proposed the RDIV constraint propagation algorithm and conducted an initial study on the complexity of array subscripts. Paul Havlak extended PFC's ability to test subscripts containing symbolic expressions, and assisted in developing the symbolic test described in Section 4.5.

Unfortunately, dependence testing in PFC was hindered due to an oversight in the original algorithm that employed SIV tests only if *all* subscripts were SIV subscripts. The presence of a single MIV subscript in a multidimensional array would cause PFC to fall back on subscript-by-subscript testing with the Banerjee-GCD tests. The introduction of the λ-test [38] motivated us to reexamine PFC's test strategy, exposing the obvious mistake. Some additional work led to the current form of the Delta test.

We are grateful to all the people named above for their contributions in the development of the dependence tests described in this paper, and to the PFC and ParaScope research groups, especially Paul Havlak, for their help in conducting our empirical study. We also wish to thank Kathryn McKinley, Doug Moore, and Vicky Dean for their assistance on this paper.

References

[1] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[2] J. R. Allen. Unifying vectorization, parallelization, and optimization: The Ardent compiler. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Third International Conference on Supercomputing*, 1988.

[3] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

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

[5] Z. Ammarguellat and W. Harrison. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[6] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976. Report No. 76-837.

[7] U. Banerjee. *Speedup of ordinary programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

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

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[10] D. Callahan. Dependence testing in PFC: Weak separability. Supercomputer Software Newsletter 2, Dept. of Computer Science, Rice University, August 1986.

[11] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[12] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[13] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[14] W. Cohagan. Vector optimization for the ASC. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems*, Princeton, NJ, March 1973.

[15] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of Third Annual ACM Symposium on Theory of Computing*, New York, NY, 1971.

[16] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[17] G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[18] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[19] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[20] M. Girkar and C. Polychronopoulos. Compiling issues for supercomputers. In *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

[21] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[22] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high performance parallelizing compilers. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[23] R. Heuft and W. Little. Improved time and parallel processor bounds for Fortran-like loops. *IEEE Transactions on Computers*, C-31(1):78–81, January 1982.

[24] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing*, 1984.

[25] K. Kennedy. Automatic translation of Fortran programs to vector form. Technical Report 476-029-4, Dept. of Mathematical Sciences, Rice University, October 1980.

[26] K. Kennedy. Triangular Banerjee inequality. Supercomputer Software Newsletter 8, Dept. of Computer Science, Rice University, October 1986.

[27] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[28] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[29] D. Klappholz and X. Kong. Extending the Banerjee-Wolfe test to handle execution conditions. Technical Report 9101, Dept. of EE/CS, Stevens Institute of Technology, 1991.

[30] D. Klappholz, K. Psarris, and X. Kong. On the perfect accuracy of an approximate subscript analysis test. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[31] 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*, St. Charles, IL, August 1990.

[32] D. Kuck. *The Structure of Computers and Computations, Volume 1.* John Wiley and Sons, New York, NY, 1978.

[33] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[34] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[35] R. Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1980.

[36] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[37] Z. Li and P. Yew. Some results on exact data dependence analysis. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing.* The MIT Press, 1990.

[38] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[39] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[40] L. Lu and M. Chen. Subdomain dependence test for massive parallelism. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[41] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.

[42] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971. Report No. 71-424.

[43] A. Porterfield. *Software Methods for Improvement of Cache Performance.* PhD thesis, Rice University, May 1989.

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

[45] Z. Shen, Z. Li, and P. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[46] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.

[47] R. Triolet. Interprocedural analysis for program restructuring with Parafrase. CSRD Rpt. No. 538, Dept. of Computer Science, University of Illinois at Urbana-Champaign, December 1985.

[48] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[49] J. Uniejewski. SPEC Benchmark Suite: designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[50] D. Wallace. Dependence of multi-dimensional array references. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[51] M. E. Wolf and M. Lam. Maximizing parallelism via loop transformations. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[52] M. J. Wolfe. Techniques for improving the inherent parallelism in programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, July 1978.

[53] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[54] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986.

[55] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[56] M. J. Wolfe and C. Tseng. The Power test for data dependence. Technical Report CS/E 90-015, Dept. of Computer Science and Engineering, Oregon Graduate Institute, August 1990. To appear in *IEEE Transactions on Parallel and Distributed Systems.*