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

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 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.

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 exposition, 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:

```plaintext
DO $i_1 = L_1, U_1
   DO $i_2 = L_2, U_2
      \ldots
      DO $i_n = L_n, U_n
      \quad S_1 \quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots
      \quad S_2 \quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))
   \quad \ldots
\quad \ldots
\end{DO}
\end{DO}
\end{DO}
```

Let $\alpha$ and $\beta$ 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 $\alpha$ and $\beta$ such that $\alpha$ is lexi-
cographically less than or equal to $\beta$ and the following system of dependence equations is satisfied:

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

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 $D = (D_1, \ldots, D_n)$ is defined as $\beta - \alpha$. The direction vector $d = (d_1, \ldots, d_n)$ 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 $\alpha$ and $\beta$ 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+1, 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,

\[
\begin{align*}
\text{DO } & 10 \text{ } i \\
\text{DO } & 10 \text{ } j \\
\text{DO } & 10 \text{ } k \\
\end{align*}
\]

\(10 \quad A(i+1, j, j) = A(i, j, k) + C\)

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:

\[
\begin{align*}
\text{DO } & 10 \text{ } i \\
\text{DO } & 10 \text{ } j \\
\text{DO } & 10 \text{ } k \\
\end{align*}
\]

\(10 \quad A(i+1, 1+2) = A(i, i) + C\)

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,

\[
\begin{align*}
\text{DO } & 10 \text{ } i \\
\text{DO } & 10 \text{ } j \\
\text{DO } & 10 \text{ } k \\
\end{align*}
\]

\(10 \quad A(i+1, j, k-1) = A(i, j, k) + C\)

the leftmost index 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 allows 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 \(\lambda\)-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 ParaScop, 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 \(\text{UNION/FIND}\) 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 $i$, $1 \leq i \leq n$ do
  $P_i \leftarrow \{S_i\}$
endfor

for each index $I_i$, $1 \leq i \leq n$ do
  $k \leftarrow \langle none \rangle$
  for each remaining partition $P_j$ do
    if $\exists S_l \in P_j$ such that $S_l$ contains $I_i$ then
      if $k = \langle none \rangle$ then
        $k \leftarrow j$
      else
        $P_k \leftarrow P_k \cup P_j$
        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
  $A(i+i, j) = A(i, j) + C$
```

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
  $A(i+i, 5) = A(i, 5) + C$
```

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

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

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

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

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:

$$a_1i_1 + a_2i_2 + \ldots + a_ni_n + \epsilon$$

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 $\epsilon$ 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 symbolic 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 $(ai + c_1, ai + c_2)$; 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:

$$d = |d| = \frac{c_1 - c_2}{a}$$

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:

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

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-
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:

\[
\text{DO } 10 \ i = 1, N \\
10 \ A(i+2N) = A(i+N) + C
\]

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 $(a_1i + c_1, a_2i' + c_2)$, 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:

\[
a_1i + c_1 = a_2i' + c_2
\]

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 tomcat from the SPEC benchmark suite [49]:

\[
\text{DO } 10 \ i = 1, N \\
10 \ Y(i, N) = Y(1, N) + Y(N, N)
\]

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:

\[
\text{DO } 10 \ i = 2, N-1 \\
10 \ Y(i, N) = Y(1, N) + Y(N, N)
\]

\[
Y(N, N) = Y(1, N) + Y(N, N)
\]

#### 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 \( i = j \). 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 suite \([13]\):

\[
\begin{align*}
\text{DO 10 } & i = 1, N \\
10 & A(i) = A(N-i+1) + C
\end{align*}
\]

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:

\[
\begin{align*}
\text{DO 10 } & i = 1, (N+1)/2 \\
10 & A(i) = A(N-i+1) + C \\
\text{DO 20 } & i = (N+1)/2 + 1, N \\
20 & A(i) = A(N-i+1) + C
\end{align*}
\]

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 \( \{a_i i + c_1, a_j j + c_2\} \). 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,

\[
\begin{align*}
\text{DO 10 } & i = 1, N_1 \\
10 & A(a_i i + c_1) = \ldots \\
\text{DO 20 } & j = 1, N_2 \\
20 & \ldots = A(a_j j + c_2)
\end{align*}
\]

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 \( i \), \( 1 \leq i \leq N_1 \) and \( j \), \( 1 \leq j \leq N_2 \).

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:

\[
a_1 - a_2 N_2 \leq c_2 - c_1 \leq a_1 N_1 - a_2
\]

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:

\[
a_1 - a_2 \leq c_2 - c_1 \leq a_1 N_1 - a_2 N_2
\]

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


\textbf{INPUT:} coupled SIV and/or MIV subscripts
\textbf{OUTPUT:} hybrid distance/direction vector, constrained MIV subscripts

initialize elements of constraint vector \( \bar{C} \) to \( \langle \text{none} \rangle \)

\textbf{while} \( \exists \) untested SIV subscripts \textbf{do}

- apply SIV test to all untested SIV subscripts, return independence or
- derive new constraint vector \( \bar{C}' \)
  \( \bar{C}' = \bar{C} \cap \bar{C} \)

  \textbf{if} \( \bar{C}' = \emptyset \) \textbf{then}
  return independence

  \textbf{else if} \( \bar{C} \neq \bar{C}' \) \textbf{then}
  \( \bar{C} = \bar{C}' \)
  propagate constraint \( \bar{C} \) into MIV subscripts, possibly creating new ZIV or SIV subscripts
  apply ZIV test to untested ZIV subscripts, return independence or continue

\textbf{endif}
\textbf{endwhile}

\textbf{while} \( \exists \) untested RDIV subscripts \textbf{do}

- test and propagate RDIV constraints
\textbf{endwhile}

test remaining MIV subscripts, then intersect resulting direction vectors with \( \bar{C} \)
return distance/direction vectors from \( \bar{C} \)

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{delta_algorithm.png}
\caption{Delta Test Algorithm}
\end{figure}

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 \( \bar{C} = (\delta_1, \delta_2, \ldots, \delta_n) \) 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 \( \delta \) 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:

\begin{verbatim}
DO 10 i
  10 A(i+1, i+2) = A(i, i) + C
\end{verbatim}

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.

\begin{verbatim}
DO 10 i
  10 A(i, i) = A(1, i-1) + C
\end{verbatim}
input: constraints $\delta_1, \delta_2$ and loop bounds $U, L$
output: new constraint $\delta$ or $\emptyset$

{\* either $\delta_1$ or $\delta_2 = \langle \text{none} \rangle$ *}
if $\delta_1 = \langle \text{none} \rangle$ then
    return $\delta_2$
else if $\delta_2 = \langle \text{none} \rangle$ then
    return $\delta_1$
{\* both $\delta_1$ and $\delta_2$ are dependence distances *}
else if $\delta_1 = \langle d_1 \rangle$ and $\delta_2 = \langle d_2 \rangle$ then
    if $d_1 = d_2$ then
        return $\langle d_1 \rangle$
    else
        return $\emptyset$
endif
{\* both $\delta_1$ and $\delta_2$ are dependence points *}
else if $\delta_1 = \langle x_1, y_1 \rangle$ and $\delta_2 = \langle x_2, y_2 \rangle$ then
    if $x_1 = x_2$ and $y_1 = y_2$ then
        return $\langle x_1, y_1 \rangle$
    else
        return $\emptyset$
endif
{\* both $\delta_1$ and $\delta_2$ are dependence lines/distances *}
else if $\delta_1 = \langle a_1 x + b_1 y = c_1 \rangle$ and
        $\delta_2 = \langle a_2 x + b_2 y = c_2 \rangle$ then
    {\* lines are parallel if slopes are equal *}
    if $a_1/b_1 = a_2/b_2$ then
        if $c_1/b_1 = c_2/b_2$ then
            return $\langle a_1 x + b_1 y = c_1 \rangle$
        else
            return $\emptyset$
        endif
    else
    {\* lines must intersect if not parallel *}
    endif
    {\* either $\delta_1$ is a dependence line/distance *}
    {\* and $\delta_2$ is a dependence point, or vice versa *}
    {\* without loss of generality, assume the former *}
else $\delta_1 = \langle a_1 x + b_1 y = c_1 \rangle$ and $\delta_2 = \langle x_1, y_1 \rangle$
    if $a_1 x_1 + b_1 y_1 = c_1$ then
        return $\langle x_1, y_1 \rangle$
    else
        return $\emptyset$
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 $(i = 1)$ and $(i = j - 1)$, respectively. These dependence lines intersect at the dependence point $(1, 2)$, 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 $(1)$ for index $i$. We can propagate this constraint into the second subscript to eliminate both occurrences of $i$, resulting in the constrained SIV subscript $(j - 1, j)$. 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 $\lambda$-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:
When applied to each subscript in this example loop, Banerjee’s inequalities show possible dependence for input:

\[ (a_1 i_1 + \ldots + a_n i_n + \epsilon, a'_1 i'_1 + \ldots + a'_n i'_n + \epsilon'), \]

and constraint vector \( C = (\delta_1, \delta_2, \ldots, \delta_n) \)

OUTPUT: constrained ZIV, SIV, or MIV subscript

for each index \( i_k \) with nonzero \( a_k \) or \( a'_k \) do

if \( \delta_k \) is dependence distance \( \langle d \rangle \) then

\[ e = -a_k d ; \quad a_k = 0 ; \quad d'_k = d'_k - a_k \]

else if \( \delta_k \) is dependence line \( \langle ax + \beta y = c \rangle \) then

if \( \alpha = 0 \) then

\[ e = -a'_k c / \beta ; \quad a'_k = 0 \]

else if \( \beta = 0 \) then

\[ e = -a_k c / \alpha ; \quad a_k = 0 \]

else if \( \alpha = \beta \) then

\[ e = -a_k c / \alpha ; \quad a_k = 0 ; \quad d'_k = d'_k + a_k \]

else

\{ multiply terms of subscript by \( \alpha \) to \( \beta \) \}
\{ retain integer coefficients in result \}

for each \( \tau \in \{ a_1, \ldots, a_n, a'_1, \ldots, a'_n, \epsilon, \epsilon' \} \) do

\( \tau \leftarrow \alpha \tau \)

endfor

\[ e = -a_k c / \alpha ; \quad a_k = 0 ; \quad d'_k = d'_k + a_k \beta \]

endif

else if \( \delta_k \) is dependence point \( \langle x, y \rangle \) then

\[ e = -a_k x - a'_k y \]

\[ a_k = 0 ; \quad a'_k = 0 \]

endif

endfor

Figure 5: Constraint Propagation

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

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

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 \( (1) \) on the second loop. This constraint is then propagated into the third subscript to derive the subscript \( (k-1, k) \). The new SIV subscript causes another pass that discovers a distance of \( -1 \) on the second 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
  A(i-1, 2i) = A(i, i+j+110)
END

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 \( (1) \) and propagating it into the second subscript to produce the constrained MIV subscript \( (2j, j-i+110) \). Banerjee’s inequalities can now detect independence for the constrained subscript.

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

Similarly, in this example loop the GCD test shows integer solutions for both subscripts. However, propagating the distance constraint \( (0) \) for \( i \) from the first subscript into the second subscript yields the constrained MIV subscript \( (2j, j-2i+5) \). 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
  A(i, j) = A(i-1, j) + A(i, j-1) + A(i+1, j) + A(i, j+1)
END

Since all subscripts are separable, the strong SIV test can be applied to calculate distance vectors of \( (0, 1) \) and \( (1, 0) \) 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
  A(i, j-i) = A(i-1, j-i) + A(i, j-i-1) + A(i+1, j-i) + A(i, j-i+1)
END

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
  A(i+1, i+j+110) = A(i, i+j+110)
Table 1: Program Characteristics

<table>
<thead>
<tr>
<th>program</th>
<th>A S/I</th>
<th>S I</th>
<th>A S/I</th>
<th>S I</th>
<th>strong</th>
<th>weak-zero</th>
<th>weak-cross</th>
<th>other</th>
<th>MIV</th>
<th>Delta</th>
<th>symbols used</th>
</tr>
</thead>
<tbody>
<tr>
<td>baro</td>
<td>1001</td>
<td>7</td>
<td>1002</td>
<td>14</td>
<td>34</td>
<td>206</td>
<td>206</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>euler</td>
<td>1200</td>
<td>14</td>
<td>1003</td>
<td>14</td>
<td>1004</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>heat2d</td>
<td>336</td>
<td>6</td>
<td>352</td>
<td>15</td>
<td>206</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>limpack</td>
<td>797</td>
<td>11</td>
<td>52</td>
<td>36</td>
<td>206</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mhld2d</td>
<td>427</td>
<td>14</td>
<td>200</td>
<td>196</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>onedim</td>
<td>1016</td>
<td>16</td>
<td>209</td>
<td>209</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>shear</td>
<td>91.5</td>
<td>15</td>
<td>787</td>
<td>828</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>simple</td>
<td>2018</td>
<td>18</td>
<td>743</td>
<td>600</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>splot</td>
<td>1144</td>
<td>7</td>
<td>282</td>
<td>77</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vortex</td>
<td>709</td>
<td>20</td>
<td>174</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

\[
\begin{align*}
i + c_1 &= j + c_3 \\
j + c_2 &= j + c_4
\end{align*}
\]

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 \( \Delta_i \) 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:

\[
\begin{align*}
i + c_1 &= j + \Delta_j + c_3 \\
j + c_2 &= i + \Delta_i + c_4
\end{align*}
\]

It is clear that these the 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
      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 \( \lambda \) 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.
Table 3: Comparison of Dependence Tests

<table>
<thead>
<tr>
<th>category</th>
<th>ZIV</th>
<th>SIV</th>
<th>MIV</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>strong</td>
<td>weak-zero</td>
<td>weak-cross</td>
<td>other</td>
</tr>
<tr>
<td>summed over all programs</td>
<td>44.76</td>
<td>33.98</td>
<td>6.77</td>
<td>0.79</td>
</tr>
<tr>
<td>% of all tests applied</td>
<td>39.97</td>
<td>51.88</td>
<td>7.64</td>
<td>0.60</td>
</tr>
<tr>
<td>% of all successful tests</td>
<td>85.43</td>
<td>4.85</td>
<td>1.55</td>
<td>0.13</td>
</tr>
<tr>
<td>% of all proven independences</td>
<td>43.99</td>
<td>97.08</td>
<td>71.77</td>
<td>47.96</td>
</tr>
<tr>
<td>% of applications that were successful</td>
<td>43.99</td>
<td>3.29</td>
<td>5.28</td>
<td>3.92</td>
</tr>
<tr>
<td>averaged over all programs</td>
<td>36.38</td>
<td>45.76</td>
<td>7.72</td>
<td>0.61</td>
</tr>
<tr>
<td>% of all tests applied</td>
<td>21.85</td>
<td>67.33</td>
<td>4.79</td>
<td>0.41</td>
</tr>
<tr>
<td>% of all successful tests</td>
<td>73.97</td>
<td>17.82</td>
<td>2.28</td>
<td>0.25</td>
</tr>
<tr>
<td>% of all proven independences</td>
<td>34.74</td>
<td>97.88</td>
<td>65.42</td>
<td>27.41</td>
</tr>
<tr>
<td>% of applications that were successful</td>
<td>34.74</td>
<td>4.03</td>
<td>1.86</td>
<td>9.36</td>
</tr>
</tbody>
</table>

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 independent. 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 *aispack* (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 is 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 Karhmark'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

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 VAIL 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 Triplet 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. Triplet 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 pseudosub-scripts 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 cispark [38]. Our results for cispark 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, 35]. 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 the Banerjee-GCD test extended to calculate the level, activated us to reexamine PFC’s test strategy, exposing some data on the efficacy of symbolic 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


