RICE UNIVERSITY

Practical Techniques to Augment Dependence Analysis in the Presence of Symbolic Terms

by

Gina Goff

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Computer Science
Chair

Keith D. Cooper, Associate Professor
Computer Science

Mary Wheeler,
Ernest and Virginia Cockrell Professor
Computational and Applied Mathematics
University of Texas at Austin

Houston, Texas

May, 1997

Practical Techniques to Augment Dependence Analysis in the Presence of Symbolic Terms

Gina Goff

Abstract

Dependence analysis is an indispensable tool in the automatic vectorization and parallelization of sequential programs, but performing symbolic dependence analysis can be costly and may fail to resolve many unknown terms. In this thesis, we explore ways to overcome the problems symbolic terms create for dependence analysis. We investigate three approaches to enhancing code optimization in the presence of symbolic terms: run-time testing, inserting user assertions, and compiling the program together with its input.

Breaking conditions are created by the dependence analyzer when the presence of symbolic terms make it impossible to prove or disprove independence. If a breaking condition is satisfied at run-time, then optimized code can be executed. We show that the use of breaking conditions was responsible for more than one-fifth of all dependences eliminated in our test suite.

Index array assertions are user-inserted directives describing special properties of the index arrays used in a program. We show that such directives can eliminate additional dependences by providing information that cannot be obtained using dependence analysis.

Partial evaluation can be used to inspect a program's input file and disseminate the actual values of symbolic variables before dependence analysis is done. We show that partial evaluation assists dependence analysis by reducing the number of unknown symbolic terms.

Algorithms for these techniques, developed and implemented in a parallelizing compiler, are presented, along with the results of several empirical studies.

Acknowledgments

I would like to thank my thesis advisor, Ken Kennedy, for his steadfast support throughout a somewhat beleaguered graduate career. Thanks go to my other committee members, Keith Cooper and Mary Wheeler, as well as to Ken, for providing encouragement, advice, and broadening perspectives on the nature of my work. Thanks also to Chuck Koelbel and John Mellor-Crummey for their questions and comments over the years.

I owe much to the insights and companionship of fellow graduate students, such as Mark Anderson, Ervan Darnell, Marina Kalem, Nat McIntosh, Dejan Mircevski, Mike Paleczny, Rebecca Parsons, and Jerry Roth. I should particularly like to acknowledge Chau-Wen Tseng, who co-authored the paper which forms the basis of Chapter 2, and Kathryn McKinley and Paul Havlak, for their highly useful input on the subject of index arrays.

I should also express my gratitude to the multitudinous non-technical support staff who accompanied me through this scientific version of *Last Year at Marienbad*. My thanks to Michael Piller, for unwittingly showing me what I wanted, and to Tom Jaber, for unstintingly giving me something I needed. My thanks also to a trio of swashbuckling librarians, Kay Flowers, Jennifer Juday, and Kerry Keck, to a duo (among dozens) of musicians, David Malone and David Sloat, to the jello-shot producing singularity that is Mary Hanks, to Lisa Hazard for being a lizard, and to Tim Lynch for wearing the PJ's. Finally, I would like to thank my family for all their understanding and humor, especially when things were neither comprehensible nor amusing.

This thesis is dedicated to the memory of my father, who believed this day would come even when I doubted.

Contents

Illustrations

Tables

Chapter 1

Introduction

The importance of good software support for scientific computing has increased as the emphasis in hardware has shifted from single processor to multiprocessor architectures. Application programs must execute wholly or partially in parallel to take advantage of multiprocessor architectures, but writing correct parallel programs by hand places the burden of predicting all possible execution traces on the programmer's shoulders. Such a tedious task can distract the programmer from the application problem he is interested in.

Parallelizing compilers free the programmer from these housekeeping details by analyzing sequential programs and providing the necessary parallel constructs. Compilers can also relieve the programmer of the task of continually re-tuning his code: the application program is written only once, in a machine-independent manner, and the back end of the compiler will be rewritten as necessary to capitalize on the idiosyncrasies of new architectures.

Effective and efficient dependence analysis is a vital part of vectorizing and parallelizing compilers because we must know which dependences exist in order to restructure programs safely. The *conservative* property requires us to assume the existence of any dependences we cannot disprove; this ensures that a parallelized program produces the same output as its sequential ancestor.

The potential dependences in a program may be grouped into three categories:

- those we know must exist,
- those we know cannot exist,
- those we must assume in the absence of further information.

As the quality of dependence analysis improves, the number of dependences in the "uncertain" category decreases. Where does this uncertainty arise?

1.1 Dependence Analysis and Symbolic Terms

Dependence analysis concentrates on array references because large reductions in program execution time are likely to be found by parallelizing or vectorizing loops that iterate through arrays. Our tests handle *affine* subscript references, i.e., references which are linear functions of the index variables. Nonlinear or symbolic references can force us to assume dependences which may not exist. If I is a loop index variable, then the reference

$$\mathtt{A[I^2]}$$

is obviously not linear, while the reference

$$\mathtt{A[M * I - 1]}$$

may or may not be linear, depending on the value of the symbolic term M. Without some sort of symbolic analysis, we would be forced to assume that the reference above is nonlinear and therefore beyond the scope of our dependence tests.

Symbolic terms impair the effectiveness of dependence testing, but these terms cannot be ignored without compromising the accuracy of the dependence graph. In one empirical study, over 50% of the array references in some numerical packages contained at least one unknown term [32], while another found that the ability to manipulate symbolic additive constants was required in 25% of the cases where dependence tests were successful [31].

Symbolic analysis, while useful, may not always be able to handle unknown symbolic terms. In one study of dependence testing [31], 6% of all subscript pairs tested were classified as nonlinear and the dependences associated with those pairs were consequently assumed to exist. An informal survey of the test programs from that study reveals some common reasons for references to be considered nonlinear: inadequate constant propagation for variables shared through a common block, no information about the values of variables read from an input file, and a lack of information about index arrays.

Symbolic terms can thus represent a significant obstacle to the optimization of programs for high performance computing. Ideally, a compiler should be able to sidestep at least a portion of this obstacle without becoming prohibitively slow.

Compiling with Varying Amounts of Information

Instead of viewing the compiler as a monolithic entity, let us consider it to be part of a continuum. In this layered compiler scheme, each layer represents the amount of information which may be at our disposal when compilation is done. Such a scheme could appear as follows:

module
program
program with data and/or user info

At the lowest level, a single module is compiled. At the intermediate level, all modules comprising a program are compiled, perhaps with the assistance of interprocedural analysis. At the highest level, the entire program plus additional information gleaned from the input data set or from the programmer is available. The quality of dependence analysis should improve as more and more information about the program is presented to the compiler.

We believe that large programs often follow the pattern of reading in data at the beginning of the program and then processing it, such that the input data read changes very slowly relative to the program calculations. In large calculations, it could be worthwhile to read and preprocess this data.

Alternatively, there may be information about the program which cannot be conveyed to the compiler via ordinary programming language statements. For example, the programmer may know that a certain array is used as perturbation array and its values are therefore unique; this knowledge could be used to rule out spurious dependences which may block optimization.

We hypothesize that programs in both of these categories exist and that the optimization of these programs and others can be improved without incurring a punitive additional cost in compilation time. Our goal is to implement and test alternative methods to improve the run-time performance of parallelized programs. These methods should improve the quality of dependence analysis and also use the results of that analysis as thoroughly as possible. We believe that the effectiveness of dependence analysis can be dramatically enhanced by using information obtained from run-time execution, from the user, and from the input data set. This work pursues three specific methods corresponding to these general areas of attack.

1.2 Research Overview

1.2.1 Breaking and Proving Conditions

In the first method, breaking and proving conditions are generated by the dependence analyzer. Each condition is passed along to the code generator, which creates a chunk of optimized code guarded by that condition. If the condition is satisfied at run-time, then an optimized version of the code can be executed instead of the unoptimized one. Testing these conditions at run-time will allow us to use the results of dependence analysis to their fullest advantage.

1.2.2 User Assertions Pertaining to Index Arrays

The second method uses directives that make assertions about the properties of index arrays. These directives, which are inserted into the source code by the user, can be used to eliminate dependences. For example, the assertion

$$\mathsf{INDEX(I) > INDEX(J)\ for\ I > J}$$

could be used to safely eliminate the loop-carried true dependence on A in the following loop.

```
DO I = 1, N, 1
   A(INDEX(I)) = A(INDEX(I+1))          (S1)
ENDDO
```

User assertions assist in the analysis of array references by providing information that cannot be obtained using dependence analysis.

1.2.3 Partial Evaluation

The third method relies on a partial evaluation of the program. Many scientific programs read in parameter values from an input file. In an ordinary compilation system, parameters of this kind thwart interprocedural analyses like constant propagation, but the user may be open to the idea of recompiling whenever these parameters change. To exploit this, we will explore the use of data file inputs in the global optimization process.

In this scheme, the program's data file is read and input values are associated with the appropriate variables. Constant propagation is used to disseminate the values of symbolic variables before dependence analysis is done for the residual program. Thus, partial evaluation can be used to reduce the number of unknown symbolic terms confronting the dependence analyzer.

1.2.4 Empirical Results

An integral part of this research is a series of studies intended to provide empirical answers to the following questions:

- How often do symbolic terms appear in dependence testing?
- How prevalent are index arrays and what properties do they frequently possess?
- How precise are dependence tests, i.e., how often can we prove that a dependence does or does not exist?
- How much improvement in the precision of dependence tests and in the execution time of the programs is observed when the techniques presented here are employed?

The answers to these questions should indicate whether the efficacy of these techniques justifies their cost.

The studies have been done using PFC, a parallelizing compiler. Test programs include programs from the RiCEPS, Perfect, and SPEC benchmarks, as well as two math libraries, *eispack* and *linpack*. These programs are written in Fortran, but this research should be applicable to programs written in other imperative languages.

1.2.5 Organization

This dissertation describes the implementation of three techniques to improve the effectiveness of dependence analysis in the presence of symbolic terms. Chapter 2 gives an overview of the dependence testing scheme used in PFC, and presents new experimental results for the study done in [31]. These results confirm our original conclusion that symbolic terms play a significant role in the analysis of programs.

Chapter 3 provides an overview of PFC and addresses the implementation issues associated with this thesis, putting the proposed techniques into context by describing the environment they have been integrated into.

Chapter 4 shows how breaking and proving conditions can be used to generate run-time tests that utilize dependence information which previously had been discarded. Chapter 5 discusses how user-inserted assertions about the properties of index arrays can improve dependence analysis. Chapter 6 presents a method for reading input data associated with a certain class of programs and replacing symbolic terms with their constant values before dependence analysis takes place. Each of these chapters will discuss experimental results measuring the effectiveness of their respective techniques.

Chapter 7 will discuss future research directions and summarize the contributions of this work.

Chapter 2

Dependence Analysis

To properly evaluate the effectiveness of the techniques proposed in this thesis, one must first understand the dependence analysis framework the techniques have been integrated into. This chapter describes a dependence testing scheme based on classifying pairs of subscripted variable references and also presents experimental results showing that symbolic terms do play a significant role in the analysis of programs. Some of the programs from this study will be used in later chapters for experiments measuring the effectiveness of these techniques.

Section 2.2 describes our system for classifying subscript references, while Section 2.3 details a dependence testing algorithm based on that classification system. Section 2.4 describes the dependence tests we apply to separable subscripts and Section 2.5 discusses the Delta test, which is applied to subscripts in coupled groups. Section 2.6 presents the results of our empirical study, and we conclude in Section 2.7. First, however, we begin with some basic definitions.

2.1 Definitions

2.1.1 Data Dependence

The theory of *data dependence*, originally developed for automatic vectorizers, has proven 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 [44, 45]:

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.

2.1.2 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 $i_1 = L_1, U_1$
      DO $i_2 = L_2, U_2$
        ...
        DO $i_n = L_n, U_n$
$S_1$        A($f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)$) = ...
$S_2$        ... = A($g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)$)
        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*.

2.1.3 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 [80], are useful for calculating the level of *loop-carried* dependences [1, 4, 38]. 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, 38, 80].

Distance vectors, first used by Kuck and Muraoka [46, 59], 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 [34, 40, 47, 78, 81] or the memory hierarchy [14, 29, 63].

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.

2.1.4 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 [21, 25]. In practice most dependence tests, such as Banerjee's inequalities [10], seek efficient approximate solutions. *Exact* tests, on the other hand, are dependence tests that will detect dependences if and only if they exist.

2.1.5 Indices and Subscripts

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, 6, 79].

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.

2.2 Subscript Classification

There are 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.

2.2.1 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.

2.2.2 Separability

When testing multidimensional arrays, we say that a subscript position is *separable* if its indices do not occur in the other subscripts [1, 13]. If two different subscripts contain the same index, we say they are *coupled* [53]. 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 [80]. 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, 34, 46, 47, 59]. More recently, Li *et al.* formalized and applied this method in the λ-test to array references also containing MIV or coupled subscripts [53]. Our treatment of constraint propagation as described in [31] was inspired by their work.

2.3 Dependence Testing

The goal of our dependence testing 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 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.

2.3.1 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 has been implemented in both PFC, an automatic vectorizing and parallelizing compiler [3, 4], and ParaScope, a parallel programming environment [15, 40, 41].

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 nonempty 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. The partitioning algorithm is given in [31].

2.4 Single Subscript Tests

We first consider dependence tests for single separable subscripts. All tests presented here 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.

2.4.1 ZIV Test

The ZIV test compares two loop-invariant expressions. If the test determines these expressions cannot be equal, we have proven independence. Otherwise, this subscript contributes no direction vectors and may be ignored. For expressions containing symbolic terms, the ZIV test forms an expression representing the difference between the two subscript expressions. If the difference simplifies to a non-zero constant, we have proven independence.

2.4.2 SIV Tests

Many authors have published a single-index exact test for linear SIV subscripts based on finding all solutions to a simple Diophantine equation in two variables [10, 18, 82]. We use an exact test based on the idea of treating commonly occurring SIV subscripts as special cases. This approach 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.

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, 13]. For strong SIV subscripts, we define the *dependence distance* as:

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

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:

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

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 symbolic 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 $N > N - 1$.

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:

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

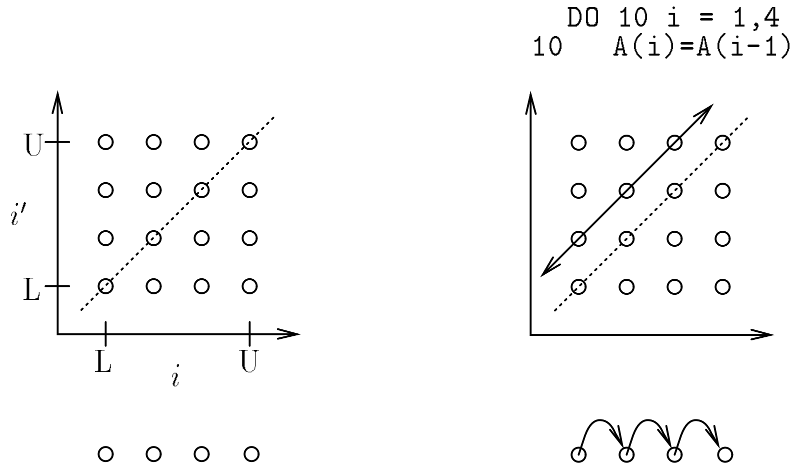
describes a line in the two dimensional plane with i and i' as the axes [13]. The weak SIV test can then be formulated as determining whether the line derived from the dependence equation intersects with any integer points in the space bounded by the loop upper and lower bounds, as shown in Figure 2.1. 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.

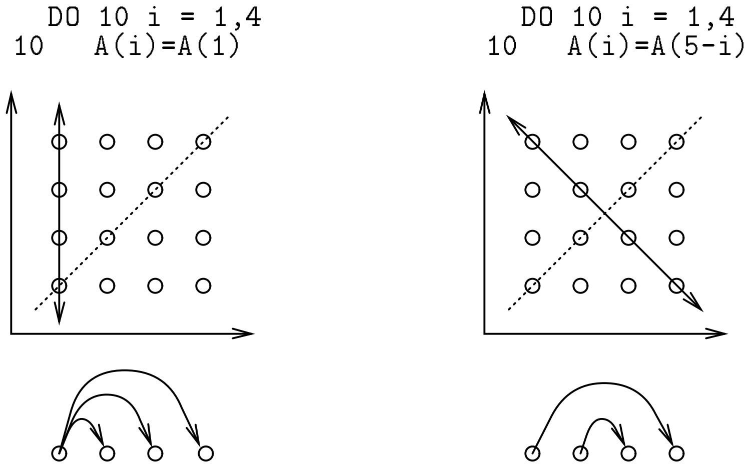
The weak-zero SIV test finds dependences caused by a particular iteration i. In scientific codes, i is usually the first or last iteration of the loop, eliminating one possible direction vector for the dependence. More importantly, weak-zero dependences caused by the first or last loop iteration may be eliminated by applying the *loop peeling* transformation [41]. For instance, consider the following simplified loop in the program *tomcatv* from the SPEC benchmark suite [74]:



```
  DO 10 i = 1,4
10   A(i)=A(i-1)
```

Bounded Iteration Space

Strong SIV



```
  DO 10 i = 1,4
10   A(i)=A(1)
```

```
  DO 10 i = 1,4
10   A(i)=A(5-i)
```

Weak-Zero SIV

Weak-Crossing SIV

Figure 2.1 Geometric View of SIV Tests.

```
    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 [41]. For instance, consider the following loop from the Callahan-Dongarra-Levine vector test suite [16]:

```
    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 2.5. We rely on the Single-Index exact test to handle the general case.

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.

2.4.3 MIV Tests

The Banerjee-GCD test [4, 10, 38, 82] 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 [10, 39].

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 [82].

2.4.4 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, $N_1$
10      A($a_1$i + $c_1$) = ...
      DO 20 j = 1, $N_2$
20      ... = A($a_2$j + $c_2$)
```

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 $N_1 = N_2$.

2.5 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 a coupled group may yield false dependences. Some research has focused on overcoming this deficiency [53, 76, 83]. In this section we present the Delta test, a multiple subscript test designed to be exact yet efficient for common coupled subscripts. Figure 2.2 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.

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.

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 [31].

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: 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 2.2 Delta Test Algorithm.

```
      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 [40, 47, 81], including upper triangular loops skewed by *loop normalization* [3, 80]. Consider the following simplified kernel from the *Livermore Loops* [58]:

```
     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 [14, 78, 82].

2.5.1 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 [8, 42, 52], 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 [53, 83].

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.

2.6 Experimental Results

In this section we present empirical results to demonstrate that our dependence tests are applicable for scientific Fortran codes. PFC 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)
- 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 [24, 74], and two math libraries, *eispack* and *linpack*. These are not the results originally published in [31], although this study recapitulates the experiment performed there. The results of the original experiment appear in the next chapter, together with an explanation of the differences between the original and current results.

Table 2.1 gives some basic information for each of the 30 programs in our test suite: the number of lines and subroutines for each program are listed, together with

program	type	lines	subrs
	RiCEPS		
baro	Shallow Water Atmosphere	1002	7
euler	1D Unsteady Euler	1200	14
heat2d	Heat Conduction System	336	2
linpackd	Linear Algebra Benchmark	797	11
mhd2d	2D MHD Equations	927	14
onedim	Eigenfunction/Eigenenergies	1016	16
shear	3D Turbulence	914	15
simple	2D Hydrodynamics	1892	18
sphot	Particle Transport	1144	7
vortex	Vortex Simulation	709	20
	Perfect		
adm (APS)	Pseudospectral Air Pollution	6105	97
arc2d (SRS)	2D Fluid Flow Solver	3965	39
bdna (NAS)	Molecular Dynamics of DNA	3980	43
dyfesm (SDS)	Structural Dynamics	7610	78
flo52 (TFS)	Transonic Inviscid Flow	1987	28
mdg (LWS)	Molecular Dynamics of Water	1238	16
mg3d (SMS)	Depth Migration	2812	28
ocean (OCS)	2D Ocean Simulation	4343	36
qcd (LGS)	Quantum Chromodynamics	2327	35
spec77 (WSS)	Weather Simulation	3885	65
spice (CSS)	Circuit Simulation	18524	128
track (MTS)	Missile Tracking	3735	34
trfd (TIS)	2 Electron Integral Transform	485	7
	SPEC		
doduc	Thermohydraulical Modelization	5334	41
fpppp	2 Electron Integral Derivative	2718	38
matrix300	Matrix Multiplications	439	6
nasa7	NASA Ames Fortran Kernels	1105	17
tomcatv	Mesh Generation	195	1
	Math Libraries		
eispack	Eigensystems Library	11886	76
linpack	Linear Algebra Library	7427	51
Total		100037	988
Average		3335	33

Table 2.1 Basic Program Characteristics.

a brief description of the program's application. Although many of the programs were only a few thousand lines in length, some programs were larger, such as *spice*, which weighed in at over 18,000 lines. The average number of subroutines was 33, another indication that our test suite is not made up of toy programs.

Table 2.2 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. One-dimensional array references were the most common, accounting for 50.3% of all array references. 35.5% of the references were two-dimensional, 13.5% were three-dimensional, and only 6.7% were four-dimensional. None of the array references had more than four dimensions.

77% of the subscript pairs were separable; the remaining pairs were coupled, with most being in coupled groups of size two. Only two programs, *nasa7* and *shear*, had coupled groups of size three. Only 1.4% of the coupled groups were of size three and no coupled groups were larger than that. Coupled subscripts were not distributed evenly throughout the test suite; several programs had none, while *eispack* accounted for 81% of all coupled subscripts.

PFC applied dependence tests 94861 times, accounting for 89.6% of all subscript pairs. Pairs were not tested if they were nonlinear (6.1%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Table 2.3 gives the number of index variables for each subscript pair. 44% of all subscript pairs had zero indices, 47.1% had only one, 6.8% had two, 1.7% had three, and only 0.27% had four indices; none had more than four. Thus, 91% of all subscript pairs had either zero or one indices, indicating that special-case tests oriented towards such pairs would be applied frequently.

Tables 2.4 and 2.5 detail how often and how successfully each type of dependence test was employed. Table 2.4 is devoted to the various SIV tests while Table 2.5 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*. Note that the S and I columns are combined for the ZIV test because they are always identical.

No results are presented for the triangular MIV test because those numbers were considered unreliable. Triangular MIV cases were factored out by counting the number of times they would have been applied (2.5 %) and using this number only when determining how often each test type is used. Once that percentage was calculated,

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
baro	56	360	0	0	416	776	0	776	5
euler	106	294	0	0	400	694	0	694	8
heat2d	151	159	0	0	310	447	22	469	20
linpackd	52	36	0	0	88	74	50	124	25
mhd2d	280	196	21	0	497	711	24	735	232
onedim	297	209	0	0	506	567	148	715	47
shear	787	0	828	0	1615	2415	856	3271	368
simple	165	163	0	0	328	491	0	491	0
sphot	282	77	0	0	359	436	0	436	21
vortex	179	42	0	0	221	263	0	263	0
adm	642	160	521	0	1323	2501	24	2525	19
arc2d	168	405	4396	0	4969	13172	994	14166	5
bdna	1787	269	0	0	2056	2085	240	2325	409
dyfesm	268	866	7	7	1148	1897	152	2049	0
flo52	54	256	1624	0	1934	5284	154	5438	0
mdg	928	5	0	0	933	934	4	938	92
mg3d	749	52	133	40	974	1412	0	1412	491
ocean	439	29	0	0	468	497	0	497	322
qcd	7804	271	11	0	8086	8369	10	8379	0
spec77	1176	500	54	0	1730	2290	48	2338	50
spice	4529	51	0	0	4580	4605	26	4631	47
track	250	115	0	0	365	472	8	480	2
trfd	39	64	0	0	103	135	32	167	6
doduc	657	885	0	0	1542	2427	0	2427	11
fpppp	1023	108	0	0	1131	1239	0	1239	0
matrix300	0	8	0	0	8	14	2	16	0
nasa7	155	104	329	347	935	2263	475	2738	30
tomcatv	0	207	0	0	207	272	142	414	0
eispack	3001	13846	0	0	16847	12719	17974	30693	3441
linpack	3491	1084	0	0	4575	4897	762	5659	199
Total	29515	20821	7924	394	58654	74358	22147	96505	5850
Average	984	694	264	13	1955	2479	738	3217	195

Table 2.2 Dimensionality, Separability, and Linearity of Pairs Tested.

	subscript pair indices				
program	0	1	2	3	4
baro	380	315	0	0	0
euler	267	259	6	0	0
heat2d	269	131	23	0	0
linpackd	13	106	5	0	0
mhd2d	73	506	45	93	0
onedim	39	607	54	0	0
shear	179	2024	340	93	0
simple	106	361	2	0	0
sphot	280	135	21	0	0
vortex	42	221	0	0	0
adm	632	1517	24	8	0
arc2d	9501	3086	0	0	0
bdna	85	1807	418	0	0
dyfesm	1132	623	41	0	0
flo52	2410	2885	0	0	0
mdg	464	389	84	0	1
mg3d	362	884	104	0	0
ocean	20	276	108	93	0
qcd	8144	231	1	0	0
spec77	418	1788	99	3	0
spice	4302	317	12	0	0
track	242	189	7	0	0
trfd	6	95	64	2	0
doduc	795	1632	0	0	0
fpppp	902	258	18	0	1
matrix300	3	13	0	0	0
nasa7	832	1276	83	0	0
tomcatv	32	379	0	0	0
eispack	5445	19047	4529	1316	244
linpack	3271	2145	196	0	0
Total	40646	43502	6284	1608	246
Average	1355	1450	209	54	8

Table 2.3 Number of Subscript Pair Indices.

	SIV											
	strong			weak-zero			weak-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
baro	310	310	0	0	0	0	0	0	0	0	0	0
euler	251	247	3	2	0	0	4	0	0	0	0	0
heat2d	96	80	0	15	2	0	0	0	0	0	0	0
linpackd	81	65	0	0	0	0	0	0	0	0	0	0
mhd2d	384	384	1	12	0	0	4	4	4	0	0	0
onedim	473	421	0	92	8	4	0	0	0	0	0	0
shear	1327	1327	151	474	261	129	4	0	0	0	0	0
simple	356	356	0	5	5	0	0	0	0	0	0	0
sphot	134	134	0	1	0	0	0	0	0	0	0	0
vortex	221	221	0	0	0	0	0	0	0	0	0	0
adm	1351	1345	154	17	1	0	97	0	0	52	52	0
arc2d	2821	2808	0	260	68	12	0	0	0	0	0	0
bdna	1415	1271	14	86	1	0	0	0	0	0	0	0
dyfesm	531	524	0	92	0	0	0	0	0	0	0	0
flo52	2683	2683	6	178	45	0	16	2	0	8	8	0
mdg	382	353	98	0	0	0	0	0	0	0	0	0
mg3d	459	458	54	2	0	0	0	0	0	36	36	0
ocean	108	107	8	1	1	0	4	0	0	8	4	0
qcd	198	148	0	31	2	0	0	0	0	0	0	0
spec77	1717	1654	272	37	0	0	12	12	12	13	0	0
spice	224	197	8	37	19	18	0	0	0	10	0	0
track	183	183	1	5	0	0	0	0	0	0	0	0
trfd	71	71	0	24	0	0	0	0	0	0	0	0
doduc	1619	1608	0	2	2	0	0	0	0	0	0	0
fpppp	254	254	118	3	2	0	1	0	0	0	0	0
matrix300	12	12	0	1	0	0	0	0	0	0	0	0
nasa7	973	965	0	290	18	0	5	5	5	0	0	0
tomcatv	204	204	0	149	50	1	26	16	16	0	0	0
eispack	13242	9866	0	3459	80	2	1389	0	0	0	0	0
linpack	1523	1200	87	344	0	0	113	3	1	0	0	0
Total	33603	29456	975	5619	565	166	1675	42	38	127	100	0
Average	1120	982	32	187	19	6	56	1	1	4	3	0

Table 2.4 Application/Success/Independence Frequencies for SIV Test.

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
baro	380	86	0	0	0	0	0	0	0	4	4
euler	267	162	0	0	0	0	0	0	0	113	2
heat2d	269	79	19	17	11	1	0	0	2	1	0
linpackd	13	0	0	0	0	23	0	0	2	2	0
mhd2d	73	10	12	0	0	0	0	0	0	0	0
onedim	39	11	1	1	0	58	0	0	12	38	0
shear	179	66	284	18	0	84	0	0	0	4	1
simple	106	45	2	2	0	0	0	0	0	4	0
sphot	280	185	0	0	0	0	0	0	0	11	0
vortex	42	24	0	0	0	0	0	0	0	18	0
adm	632	164	13	13	12	0	0	0	0	139	21
arc2d	9501	3630	0	0	0	107	0	0	0	1456	445
bdna	85	15	2	2	0	119	0	0	1	454	14
dyfesm	1132	558	14	12	0	69	0	0	6	50	0
flo52	2410	996	0	0	0	77	0	0	0	52	10
mdg	464	27	0	0	0	0	0	0	2	38	0
mg3d	362	18	0	0	0	0	0	0	0	7	1
ocean	20	0	34	34	0	0	0	0	0	24	1
qcd	8144	6609	1	1	0	5	0	0	0	187	154
spec77	418	18	61	33	0	16	2	2	0	313	201
spice	4302	1995	0	0	0	4	0	0	9	2005	1966
track	242	77	6	5	0	4	0	0	0	15	0
trfd	6	0	10	0	0	8	2	0	0	12	0
doduc	795	314	0	0	0	0	0	0	0	58	0
fpppp	902	10	1	1	0	0	0	0	0	6	0
matrix300	3	0	0	0	0	1	0	0	0	0	0
nasa7	832	389	49	12	8	86	0	0	9	21	0
tomcatv	32	13	0	0	0	71	0	0	0	0	0
eispack	5445	185	1727	915	182	7366	320	296	958	7521	257
linpack	3271	561	6	6	4	326	1	1	49	882	431
Total	40646	16247	2242	1072	217	8425	325	299	1050	13435	3508
Average	1355	542	75	36	7	281	11	10	35	448	117

Table 2.5 Dependence Test Application/Success/Independence Frequencies.

the triangular MIV statistics were dropped from consideration, i.e., these numbers are not included in the "% of all applications that were successful" (either as applications or as successes) and other similar statistics given in Table 2.6. The rationale for such a decision is as follows: we wanted to know how often the triangular MIV test was being applied, relative to other tests, so we included it when calculating "% of all tests applied". Once that was done, however, we could not include the triangular MIV "successful" or "independence" figures, as we had little confidence in them, yet including the "applied" triangular MIV figures and assuming no successes or independences proven would have skewed the results, so no triangular MIV statistics could be included.

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 2.5 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

The Delta test constraint intersection algorithm tested 8425 coupled groups exactly (76.7%). Propagation of distance constraints was applied in 1050 cases (9.5%), converting MIV subscripts into SIV form in all but 33 cases. The Delta test thus managed to test 9442 coupled groups exactly (86%), using only constraint intersection and propagation of dependence distances.

Table 2.6 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 were 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 calculated for each program and then averaged.

In both groups, ZIV and strong SIV tests dominate the results, accounting for 78% of the applications, 95% of the successes, and 96% of the proven independences. Even some of the least-used tests, such as the general SIV test, enjoy a respectable degree of success. In contrast, the Delta test *appears* ineffective, but that is because

category	ZIV	SIV				MIV	Delta	symbolics used
		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	42.85	35.42	5.92	1.77	0.13	2.36	8.88	
% of all successful tests	33.28	60.33	1.16	0.09	0.20	2.20	0.67	27.52
% of all proven independences	90.55	5.43	0.93	0.21	0.00	1.21	1.67	19.55
% of apps that were successful	39.97	87.66	10.06	2.51	78.74	47.81	3.86	
% of apps that proved independence	39.97	2.90	2.95	2.27	0.00	9.68	3.55	
averaged over all programs								
% of all tests applied	38.37	48.23	4.71	0.70	0.40	1.93	3.36	
% of all successful tests	21.69	73.66	1.47	0.26	0.44	1.65	0.18	13.42
% of all proven independences	68.78	22.14	2.55	3.16	0.00	1.70	1.68	13.27
% of apps that were successful	28.42	94.69	23.16	31.39	58.33	70.38	2.34	
% of apps that proved independence	28.42	4.75	3.29	30.20	0.00	14.34	0.93	

Table 2.6 Comparison of Dependence Tests.

most of the credit has been given to the individual tests invoked by the Delta test (i.e., ZIV, SIV) and not to the Delta test itself.

Symbolic constants were present in 27.5% of all successful tests and in 19.6% of all cases where independence was proven. The ZIV and all SIV tests combined were responsible for 86% of all test applications. 41% of those tests involved symbolic terms. 70% of those symbolic tests were ZIV, 13% were strong SIV, 13% were weak-zero SIV, 5% were weak-crossing SIV, and 0.1% were general SIV.

2.7 Conclusions

Our results show that most array reference pairs can be tested using relatively simple and inexpensive tests such as the ZIV test or the strong SIV test. Perhaps more importantly, the SIV and Delta tests presented in this chapter tested most subscripts exactly. MIV tests such as the Banerjee-GCD test are needed only for a small fraction of all subscripts, though they are important for certain programs. 28% of the successful tests required PFC's ability to manipulate symbolic additive constants, as did 20% of the tests where independence was proven. These figures underscore the importance of symbolic analysis in dependence testing.

Chapter 3

Implementation Considerations

"PFC is a graceful ballerina... with only one leg." – a former PFC coder

This chapter briefly outlines PFC, the Parallel Fortran Converter used as our experimental vehicle. It would be extremely misleading to imply that the techniques presented in this thesis can be implemented in a vacuum. This chapter is intended to give a basic understanding of the environment the research was conducted in, as well as a feel for how much infrastructure the techniques rely on.

In Section 3.1, we sketch the history of PFC's development. Section 3.2 describes the current structure of PFC and Section 3.3 lists some of the improvements which have been made to support this thesis. Section 3.4 compares the results of our original dependence testing study with the new experimental results from Chapter 2, then describes another experiment intended to gauge the effect of constant propagation on the quality of our dependence analysis. General conclusions are presented in Section 3.5.

3.1 Background Information

PFC is the result of a research project begun in the late 1970's. The initial goal of this project was to produce an automatic vectorizing translator that would accept Fortran 66 or Fortran 77 input and produce Fortran 8x output, thus eliminating the need to re-code old programs to take advantage of vector machines. PFC's basic purpose, to alleviate the programmer of the burden of writing correct parallelized/vectorized code, remains, but the tool has altered considerably over the years. A 1984 tech report later published as [4] contains the following passage:

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

PFC still makes extensive use of macros, but the large test programs we parallelize today require 32-128 megabytes of memory and PFC's size has grown to 180,000 lines of PL/I. PFC expanded rapidly during the 1980's as it was continually extended to handle more difficult programs and features such as interprocedural analysis were introduced. With the advent of ParaScope [15], an interactive programming environment, implementation effort in PFC dwindled until only one person was left working on PFC. The advantage of such a solitary effort is that no one makes code changes that are incompatible with your own efforts. The disadvantage is that every problem encountered is yours.

3.2 Structure of PFC

There are several steps in PFC's translation process:

1. **Scanner/parser**. The program is read and converted into an abstract syntax tree. This *AST* is the intermediate representation used throughout PFC.

2. **Inline expansion**. The default is to do no inline expansion, but inline expansion of the main program segment will be done here if we are doing a partial evaluation of the program.

3. **Interprocedural analysis**. This includes the collection of summary information, flow-sensitive analysis, and constant propagation.

4. **Normalization**. Loop bounds and induction variables are shifted if necessary so that all loops will run from 1 to some upper bound N. If inline expansion is being performed at the user's request (and not to facilitate partial evaluation), it will be done here. Backward GOTOs are converted into WHILE loops, and control dependences are calculated and added to the dependence graph.

5. **Dependence analysis**. This entails identifying all possible dependences, testing each potential dependence, and adding edges to the dependence graph representing every dependence which cannot be disproved.

6. **Code generation**. The dependence graph is divided into strongly connected regions, corresponding to loops in the program, and code is generated for each region after loop fusion has been attempted.

3.3 Functional Improvements to PFC

The actual coding of the techniques described in the thesis represents only a small portion of the implementation effort. Here is a brief summary of the state of PFC when we began our research:

> **Scanner/parser**. This was in excellent condition. Its only problem was occasionally encountering bits of nonstandard Fortran.
>
> **Inline expansion**. This was in good condition, but seldom used.
>
> **Interprocedural analysis**. The quality of analysis was excellent, but the reliability of this code was poor, partly because of its inherent complexity and partly because no one had had the time to fix the problems encountered when IP analysis was done on larger, more difficult codes.
>
> **Normalization**. This was in fairly good condition. There were few problems with induction variable substitution, but the handling of control flow occasionally ran into difficulties.
>
> **Dependence analysis**. This was nearing the end of a major revision when we began our research and had not yet been fully tested or debugged.
>
> **Code generation**. This was considered unusable. At this time, programs were often analyzed in PFC and the resulting dependence graph files were shipped to ParaScope for code generation, exploiting the strengths of both tools.

and this is a summary of the changes made to implement or support this research:

> **Scanner/parser**. Underwent a minor change to handle index array assertions.
>
> **Interpreter**. Written long ago and no longer part of PFC, this was dusted off and extended to include additional data types and statement types, as well as handling common blocks. The real challenge here was integrating it into interprocedural analysis.
>
> **Interprocedural analysis**. Many of its problems have been fixed, but some bugs remain. To give an idea of the degree of improvement in interprocedural constant propagation, we need only look at the experiment originally done for [31] and repeated for Chapter 2. When the experiment was originally done, almost all of the programs ran with interprocedural analysis enabled, but none of them ran when interprocedural constant propagation was attempted. Today, all of those programs run with interprocedural constant propagation enabled.

Dependence analysis. The rewrite of the dependence testing scheme was finished and debugged, and a series of minor revisions have taken place. New code was also added to handle index array references, which had previously been categorized as nonlinear, to generate breaking conditions, and to gather new statistics.

The list above only indicates major areas of effort, as most parts of PFC have required some attention during the course of this work.

3.4 PFC – Then vs. Now

In this section we present the empirical results from [31] and compare them to the results obtained when that same experiment was repeated for Chapter 2. As will be seen, the fundamental conclusions drawn from both iterations of the experiment are the same, but some of the numbers are quite different, due to the refinements PFC has experienced over the years. We will describe the nature of the tables containing the original numbers, then go on to discuss additional tables comparing the old results with the new ones.

3.4.1 Results From Original Experiment

As in Chapter 2, the test suite consists of four groups of Fortran programs: RiCEPS (Rice Compiler Evaluation Program Suite), the Perfect and SPEC benchmark suites [24, 74], and two math libraries, *eispack* and *linpack*. Table 3.1 gives some basic information for each of these: the number of lines and subroutines for each program, as well as a brief description of the program's application. Table 3.1 is quite similar to Table 2.1, but they are not completely identical, as a shorter version of *eispack* was used in the initial study.

Table 3.2 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. One-dimensional array references were the most common, accounting for 50% of all array references. 33.6% of the references were two-dimensional, 15.6% were three-dimensional, and only 0.8% were four-dimensional. None of the array references had more than four dimensions.

80% of the subscript pairs were separable; the remaining pairs were coupled, with most being in coupled groups of size two. Only two programs, *nasa7* and *shear*, had coupled groups of size three. Only 1.4% of the 8232 coupled groups were of size three

program	type	lines	subrs
	RiCEPS		
baro	Shallow Water Atmosphere	1002	7
euler	1D Unsteady Euler	1200	14
heat2d	Heat Conduction System	336	2
linpackd	Linear Algebra Benchmark	797	11
mhd2d	2D MHD Equations	927	14
onedim	Eigenfunction/Eigenenergies	1016	16
shear	3D Turbulence	915	15
simple	2D Hydrodynamics	1892	18
sphot	Particle Transport	1143	7
vortex	Vortex Simulation	709	20
	Perfect		
adm (APS)	Pseudospectral Air Pollution	6105	97
arc2d (SRS)	2D Fluid Flow Solver	3965	39
bdna (NAS)	Molecular Dynamics of DNA	3980	43
dyfesm (SDS)	Structural Dynamics	7608	78
flo52 (TFS)	Transonic Inviscid Flow	1986	28
mdg (LWS)	Molecular Dynamics of Water	1238	16
mg3d (SMS)	Depth Migration	2812	28
ocean (OCS)	2D Ocean Simulation	4343	36
qcd (LGS)	Quantum Chromodynamics	2327	34
spec77 (WSS)	Weather Simulation	3885	65
spice (CSS)	Circuit Simulation	18521	128
track (MTS)	Missile Tracking	3735	34
trfd (TIS)	2 Electron Integral Transform	485	7
	SPEC		
doduc	Thermohydraulical Modelization	5334	41
fpppp	2 Electron Integral Derivative	2718	38
matrix300	Matrix Multiplications	439	6
nasa7	NASA Ames Fortran Kernels	1105	17
tomcatv	Mesh Generation	195	1
	Math Libraries		
eispack	Eigensystems Library	11519	75
linpack	Linear Algebra Library	7427	51
Total		99664	986
Average		3322	33

Table 3.1 Basic Program Characteristics.

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
baro	34	360	0	0	394	754	0	754	0
euler	106	294	0	0	400	694	0	694	21
heat2d	352	251	0	0	603	822	32	854	149
linpackd	52	36	0	0	88	74	50	124	29
mhd2d	280	196	21	0	497	711	24	735	232
onedim	297	209	0	0	506	567	148	715	47
shear	787	0	828	0	1615	2415	856	3271	368
simple	174	163	0	0	337	500	0	500	1
sphot	282	77	0	0	359	436	0	436	21
vortex	174	42	0	0	216	258	0	258	0
adm	647	160	521	0	1328	2506	24	2530	19
arc2d	168	405	4396	0	4969	13172	994	14166	9
bdna	1787	269	0	0	2056	2085	240	2325	562
dyfesm	268	866	7	7	1148	1897	152	2049	0
flo52	54	256	1624	0	1934	5284	154	5438	0
mdg	928	5	0	0	933	934	4	938	296
mg3d	749	52	133	40	974	1412	0	1412	494
ocean	439	29	0	0	468	497	0	497	303
qcd	7774	171	3	0	7948	8115	10	8125	0
spec77	1187	477	54	0	1718	2255	48	2303	122
spice	470	51	0	0	521	546	26	572	50
track	242	115	0	0	357	464	8	472	7
trfd	39	64	0	0	103	135	32	167	6
doduc	656	895	0	0	1551	2446	0	2446	11
fpppp	997	108	0	0	1105	1213	0	1213	3
matrix300	0	8	0	0	8	14	2	16	0
nasa7	155	106	329	347	937	2267	475	2742	24
tomcatv	0	207	0	0	207	272	142	414	0
eispack	2995	10295	0	0	13290	10823	12762	23585	2052
linpack	3224	869	0	0	4093	4228	734	4962	290
Total	25317	17036	7916	394	50663	67796	16917	84713	5116
Average	844	568	264	13	1689	2260	564	2824	171

Table 3.2 Dimensionality, Separability, and Linearity of Pairs Tested.

and no coupled groups were larger than that. Coupled subscripts were not distributed evenly throughout the test suite; several programs had none, while *eispack* accounted for 75.4% of all coupled subscripts.

PFC applied dependence tests 74889 times, accounting for 88% of all subscript pairs. Pairs were not tested if they were nonlinear (6%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Tables 3.3 and 3.4 detail how often and how successfully each type of dependence test was employed. Table 3.3 is devoted to the various SIV tests while Table 3.4 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *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 3.4 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

Over all array reference pairs tested, most subscript pairs were ZIV (47%) or strong SIV (35%). Few of the subscripts tested were MIV (3%). (The MIV numbers presented in Table 3.4 do include numbers for the triangular MIV test, unlike those in Table 2.5.) The ZIV and strong SIV tests combined for most of the successful tests (86%). The ZIV test accounted for most of the reference pairs proven independent (84%).

The Delta test constraint intersection algorithm tested 6950 coupled groups exactly (84%). Propagation of distance constraints was applied in 376 cases (4.6%), converting MIV subscripts into SIV form in all but 33 cases. The Delta test thus managed to test 7326 coupled groups exactly (89%), using only constraint intersection and propagation of dependence distances.

Table 3.5 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 were successful in eliminating one or more direction vectors. In order to limit bias

	SIV											
	strong			weak-zero			weak-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
baro	294	294	0	0	0	0	0	0	0	0	0	0
euler	240	236	3	0	0	0	4	0	0	0	0	0
heat2d	127	127	0	43	43	0	0	0	0	0	0	0
linpackd	57	57	0	0	0	0	0	0	0	0	0	0
mhd2d	384	384	1	12	12	0	4	0	0	0	0	0
onedim	448	448	0	29	29	4	4	4	0	62	0	0
shear	1327	1327	151	346	346	168	4	0	0	40	0	0
simple	358	357	0	5	5	0	0	0	0	0	0	0
sphot	134	134	0	0	0	0	0	0	0	1	0	0
vortex	216	216	0	0	0	0	0	0	0	0	0	0
adm	1341	1327	142	17	17	0	111	22	12	52	52	0
arc2d	2821	2807	0	111	111	12	0	0	0	38	0	0
bdna	1393	1249	14	69	69	0	0	0	0	0	0	0
dyfesm	514	514	0	19	19	0	0	0	0	36	0	0
flo52	2599	2599	6	87	87	0	2	2	0	36	8	0
mdg	177	177	0	0	0	0	0	0	0	0	0	0
mg3d	456	456	54	2	2	0	0	0	0	36	36	0
ocean	108	107	8	1	1	0	4	0	0	8	4	0
qcd	149	107	0	15	15	0	0	0	0	7	0	0
spec77	1650	1631	273	24	24	0	12	12	12	27	0	0
spice	229	210	8	1	1	0	0	0	0	10	0	0
track	181	181	5	3	3	0	0	0	0	0	0	0
trfd	71	71	0	18	24	0	0	0	0	6	0	0
doduc	1638	1627	0	2	2	0	0	0	0	0	0	0
fpppp	232	232	110	0	0	0	1	0	0	0	0	0
matrix300	12	12	0	0	0	0	0	0	0	0	0	0
nasa7	947	947	0	184	184	0	6	1	0	12	1	0
tomcatv	204	204	0	78	78	1	26	26	0	0	0	0
eispack	6732	6228	2	726	726	92	293	225	0	1842	0	0
linpack	1348	1271	88	15	15	0	45	6	0	55	0	0
Total	26387	25537	865	1807	1807	277	516	298	24	2268	101	0
Average	880	851	29	60	60	9	17	10	1	76	3	0

Table 3.3 Application/Success/Independence Frequencies for SIV Test.

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
baro	386	78	0	0	0	0	0	0	0	17	17
euler	267	162	0	0	0	0	0	0	0	114	2
heat2d	450	127	20	16	0	11	3	0	5	1	0
linpackd	13	0	2	2	0	23	16	0	2	10	0
mhd2d	55	28	0	0	0	12	0	0	0	4	1
onedim	39	11	32	31	25	31	5	2	14	97	34
shear	179	72	100	32	32	233	76	40	0	4	0
simple	112	44	2	2	0	0	0	0	0	373	44
sphot	280	191	0	0	0	0	0	0	0	11	0
vortex	42	24	0	0	0	0	0	0	0	18	0
adm	632	210	13	12	12	0	0	0	0	142	21
arc2d	9501	3628	0	0	0	107	38	38	0	3024	454
bdna	85	21	76	33	0	119	28	12	1	454	14
dyfesm	1073	629	26	14	1	69	32	27	6	50	0
flo52	2372	1043	0	0	0	77	14	14	0	47	10
mdg	464	415	0	0	0	2	2	0	0	20	0
mg3d	362	20	0	0	0	0	0	0	0	5	1
ocean	20	2	34	34	0	0	0	0	0	24	1
qcd	7945	6725	1	1	0	5	5	5	0	27	1
spec77	385	62	27	9	0	24	7	7	0	286	201
spice	258	102	8	8	1	9	9	1	3	46	8
track	230	95	1	0	0	4	2	2	0	15	0
trfd	6	0	44	28	0	16	0	0	0	12	0
doduc	795	424	0	0	0	0	0	0	0	58	0
fpppp	902	11	15	15	0	0	0	0	0	4	0
matrix300	3	0	0	0	0	1	1	1	0	0	0
nasa7	832	410	28	26	13	101	11	1	3	503	12
tomcatv	32	13	0	0	0	71	0	0	0	0	0
eispack	4215	471	1671	915	407	5698	2645	1179	314	6116	1191
linpack	2754	227	118	118	2	337	99	8	28	537	95
Total	34689	15245	2218	1296	493	6950	2993	1337	376	12021	2107
Average	1156	508	74	43	16	232	100	45	13	401	70

Table 3.4 Dependence Test Application/Success/Independence Frequencies.

	ZIV	SIV				MIV	Delta	symbolics
category		strong	wk-zero	wk-cross	other			used
summed over all programs								
% of all tests applied	46.32	35.29	2.41	0.69	3.03	2.98	9.28	
% of all successful tests	32.24	54.00	3.82	1.18	0.21	2.77	6.33	25.42
% of all proven independences	83.58	4.74	1.52	0.13	0	2.70	7.33	11.55
% of apps that were successful	43.95	96.64	100.00	57.75	4.45	58.67	43.06	
% of apps that proved independence	43.95	3.27	15.33	4.65	0	22.05	19.24	
averaged over all programs								
% of all tests applied	38.24	49.40	2.81	0.64	1.43	3.03	4.45	
% of all successful tests	21.91	67.54	3.87	0.84	0.43	3.31	2.50	14.94
% of all proven independences	70.34	16.65	2.06	0.24	0	3.30	7.42	14.96
% of apps that were successful	34.74	97.70	100.00	40.51	17.53	74.61	41.75	
% of apps that proved independence	34.74	4.02	3.96	8.52	0	15.49	22.04	

Table 3.5 Comparison of Dependence Tests.

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 calculated for each program and then averaged.

In both groups, ZIV and strong SIV tests dominate the results, accounting for 82% of all applied tests, 86% of all successful tests, and 88% of all proven independences. Symbolic additive constants were found in 25% of all successful tests and 15% of all cases where independence was proven.

3.4.2 Difference Between Original and New Results

Tables 2.1 through 2.6 describe the results obtained from a recent re-execution of this dependence testing study. Table 3.6 gives the difference between the numbers in Table 2.3 and Table 3.2, which describe the nature of the pairs tested. In some cases, most notably for *heat2d*, *spice*, and *eispack*, there were significant differences in the number of array pairs tested. While the current *eispack* is one subroutine longer than the original, most of the differences should be attributed to changes in PFC and not the test programs. The marked difference in the numbers for *heat2d* is probably due to backward-goto loops which PFC did not process correctly. In a similar vein, when the original study was done, many of the programs could be analyzed only when while-conversion (converting while loops into inductive ones) was turned off. PFC recognizes only inductive loops, so an unconverted while loop would not have been analyzed. Today, all of the programs except one (*ocean*) run with while-conversion enabled. Other options, such as interprocedural flow analysis and interprocedural constant propagation, were also rarely enabled in the original study but are now quite commonplace.

Tables 3.7 and 3.8 show the increase in test frequencies between the old results and the new. In general, more dependence tests were applied, which is a function of PFC's improved ability to recognize programs. There were fewer successes for many of the test types, but the raw figures are somewhat misleading. Table 3.9, which compares Tables 2.6 and 3.5, shows that the percentage of successful tests increased for all test types except the general SIV test. The decrease in the number of successful tests for the general SIV case is the result of a bug fix; the original numbers were incorrect.

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
baro	22	0	0	0	22	22	0	22	5
euler	0	0	0	0	0	0	0	0	-13
heat2d	-201	-92	0	0	-293	-375	-10	-385	-129
linpackd	0	0	0	0	0	0	0	0	-4
mhd2d	0	0	0	0	0	0	0	0	0
onedim	0	0	0	0	0	0	0	0	0
shear	0	0	0	0	0	0	0	0	0
simple	-9	0	0	0	-9	-9	0	-9	-1
sphot	0	0	0	0	0	0	0	0	0
vortex	5	0	0	0	5	5	0	5	0
adm	-5	0	0	0	-5	-5	0	-5	0
arc2d	0	0	0	0	0	0	0	0	-4
bdna	0	0	0	0	0	0	0	0	-153
dyfesm	0	0	0	0	0	0	0	0	0
flo52	0	0	0	0	0	0	0	0	0
mdg	0	0	0	0	0	0	0	0	-204
mg3d	0	0	0	0	0	0	0	0	-3
ocean	0	0	0	0	0	0	0	0	19
qcd	30	100	8	0	138	254	0	254	0
spec77	-11	23	0	0	12	35	0	35	-72
spice	4059	0	0	0	4059	4059	0	4059	-3
track	8	0	0	0	8	8	0	8	-5
trfd	0	0	0	0	0	0	0	0	0
doduc	1	-10	0	0	-9	-19	0	-19	0
fpppp	26	0	0	0	26	26	0	26	-3
matrix300	0	0	0	0	0	0	0	0	0
nasa7	0	-2	0	0	-2	-4	0	-4	6
tomcatv	0	0	0	0	0	0	0	0	0
eispack	6	3551	0	0	3557	1896	5212	7108	1389
linpack	267	215	0	0	482	669	28	697	-91
Total	4198	3785	8	0	7991	6562	5230	11792	734
Average	140	126	0	0	266	219	174	393	24

Table 3.6 Difference in Dimensionality, Separability, and Linearity of Pairs Tested Between Current and Original Results.

	SIV											
	strong			weak-zero			weak-cross			other		
	A	S	I	A	S	I	A	S	I	A	S	I
Total	7216	3919	110	3812	-1242	-111	1159	-256	14	-2141	-1	0
Average	241	131	4	127	-41	-4	39	-9	0	-71	0	0

Table 3.7 SIV Test Frequencies – Difference Between Current and Original Results.

	ZIV		MIV			Delta				symbolics used	
	A	S/I	A	S	I	A	S	I	P	S	I
Total	5957	1002	24	-224	-276	1475	-2668	-679	-1038	1414	1401
Average	199	33	1	-7	-9	49	-89	-23	-35	47	47

Table 3.8 Test Frequencies – Difference Between Current and Original Results.

	ZIV	SIV				MIV	Delta	sym
category		strong	wk-zero	wk-cross	other			used
summed over all programs								
% of all tests applied	3.47	-0.13	-3.51	-1.08	2.90	0.62	0.40	
% of all successful tests	-1.04	-6.33	2.66	1.09	0.01	0.57	5.66	-2.10
% of all proven independences	-6.97	-0.69	0.59	-0.08	0.00	1.49	5.66	-8.00
% of apps that were successful	3.98	8.98	89.94	55.24	-74.29	10.86	39.21	
% of apps that proved independence	3.98	0.37	12.38	2.38	0.00	12.37	15.69	
averaged over all programs								
% of all tests applied	-0.13	1.17	-1.90	-0.06	1.03	1.10	1.09	
% of all successful tests	0.22	-6.12	2.40	0.58	-0.01	1.66	2.32	1.52
% of all proven independences	1.56	-5.49	-0.49	-2.92	0.00	1.60	5.74	1.69
% of apps that were successful	6.32	3.01	76.84	9.12	-40.80	4.23	39.41	
% of apps that proved independence	6.32	-0.73	0.67	-21.68	0.00	1.15	21.11	

Table 3.9 Comparison of Dependence Tests – Current Vs. Original Results.

3.4.3 PFC and Interprocedural Constant Propagation

Interprocedural constant propagation was not enabled when our original dependence testing study was done, but is used in the recent study. How much of the improvement in the successfulness of dependence tests can be ascribed to better constant propagation?

To determine this, we ran our test suite from Chapter 2 with constant propagation disabled, to see how much our test performance deteriorated. Table 3.10 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. The distribution of array dimensions and the separability of subscript pairs is the same as given in Table 2.2, but the number of nonlinear subscripts is larger. An increase in nonlinear subscripts is to be expected, as constant propagation can reduce the number of unknown symbolic terms which might otherwise force us to assume an expression is nonlinear.

PFC applied slightly fewer dependence tests (94626 vs. 94861) when no constant propagation was used, testing 89.3% of all subscript pairs. Pairs were not tested if they were nonlinear (6.4%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Table 3.11 gives the number of index variables for each subscript pair. These numbers differ so slightly from those given in Table 2.3 that we will not present a table comparing the two. *baro* is the only program whose subscript pair indices varied when no constant propagation was used; there were 32 fewer subscripts with zero indices and 18 fewer with one.

Tables 3.12 and 3.13 detail how often and how successfully each type of dependence test was employed. Table 3.12 is devoted to the various SIV tests while Table 3.13 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*.

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 3.13 show the number of times symbolic additive constants were

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
baro	56	360	0	0	416	776	0	776	5
euler	106	294	0	0	400	694	0	694	19
heat2d	151	159	0	0	310	447	22	469	20
linpackd	52	36	0	0	88	74	50	124	29
mhd2d	280	196	21	0	497	711	24	735	232
onedim	297	209	0	0	506	567	148	715	47
shear	787	0	828	0	1615	2415	856	3271	368
simple	165	163	0	0	328	491	0	491	0
sphot	282	77	0	0	359	436	0	436	21
vortex	179	42	0	0	221	263	0	263	0
adm	642	160	521	0	1323	2501	24	2525	19
arc2d	168	405	4396	0	4969	13172	994	14166	5
bdna	1787	269	0	0	2056	2085	240	2325	409
dyfesm	268	866	7	7	1148	1897	152	2049	0
flo52	54	256	1624	0	1934	5284	154	5438	0
mdg	928	5	0	0	933	934	4	938	296
mg3d	749	52	133	40	974	1412	0	1412	494
ocean	439	29	0	0	468	497	0	497	322
qcd	7804	271	11	0	8086	8369	10	8379	0
spec77	1176	500	54	0	1730	2290	48	2338	108
spice	4529	51	0	0	4580	4605	26	4631	47
track	250	115	0	0	365	472	8	480	7
trfd	39	64	0	0	103	135	32	167	6
doduc	657	885	0	0	1542	2427	0	2427	11
fpppp	1023	108	0	0	1131	1239	0	1239	0
matrix300	0	8	0	0	8	14	2	16	0
nasa7	155	104	329	347	935	2263	475	2738	30
tomcatv	0	207	0	0	207	272	142	414	0
eispack	3001	13846	0	0	16847	12719	17974	30693	3441
linpack	3491	1084	0	0	4575	4897	762	5659	199
Total	29515	20821	7924	394	58654	74358	22147	96505	6135
Average	984	694	264	13	1955	2479	738	3217	204

Table 3.10 Dimensionality, Separability, and Linearity of Pairs Tested (Without Constant Propagation).

	subscript pair indices				
program	0	1	2	3	4
baro	412	333	0	0	0
euler	267	259	6	0	0
heat2d	269	131	23	0	0
linpackd	13	106	5	0	0
mhd2d	73	506	45	93	0
onedim	39	607	54	0	0
shear	179	2024	340	93	0
simple	106	361	2	0	0
sphot	280	135	21	0	0
vortex	42	221	0	0	0
adm	632	1517	24	8	0
arc2d	9501	3086	0	0	0
bdna	85	1807	418	0	0
dyfesm	1132	623	41	0	0
flo52	2410	2885	0	0	0
mdg	464	389	84	0	1
mg3d	362	884	104	0	0
ocean	20	276	108	93	0
qcd	8144	231	1	0	0
spec77	418	1788	99	3	0
spice	4302	317	12	0	0
track	242	189	7	0	0
trfd	6	95	64	2	0
doduc	795	1632	0	0	0
fpppp	902	258	18	0	1
matrix300	3	13	0	0	0
nasa7	832	1276	83	0	0
tomcatv	32	379	0	0	0
eispack	5445	19047	4529	1316	244
linpack	3271	2145	196	0	0
Total	40678	43520	6284	1608	246
Average	1356	1451	209	54	8

Table 3.11 Number of Subscript Pair Indices (Without Constant Propagation).

	SIV											
	strong			weak-zero			weak-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
baro	328	328	0	0	0	0	0	0	0	0	0	0
euler	240	236	3	2	0	0	4	0	0	0	0	0
heat2d	96	80	0	15	2	0	0	0	0	0	0	0
linpackd	77	61	0	0	0	0	0	0	0	0	0	0
mhd2d	384	384	1	12	0	0	4	0	0	0	0	0
onedim	473	421	0	92	8	4	0	0	0	0	0	0
shear	1327	1327	151	474	261	129	4	0	0	0	0	0
simple	356	356	0	5	5	0	0	0	0	0	0	0
sphot	134	134	0	1	0	0	0	0	0	0	0	0
vortex	221	221	0	0	0	0	0	0	0	0	0	0
adm	1351	1345	154	17	1	0	97	0	0	52	52	0
arc2d	2821	2695	0	260	67	12	0	0	0	0	0	0
bdna	1415	1271	14	86	1	0	0	0	0	0	0	0
dyfesm	531	524	0	92	0	0	0	0	0	0	0	0
flo52	2683	2683	6	178	45	0	16	2	0	8	8	0
mdg	178	176	0	0	0	0	0	0	0	0	0	0
mg3d	456	456	54	2	0	0	0	0	0	36	36	0
ocean	108	107	8	1	1	0	4	0	0	8	4	0
qcd	198	148	0	31	2	0	0	0	0	0	0	0
spec77	1698	1647	272	37	0	0	12	12	12	0	0	0
spice	224	197	8	37	19	18	0	0	0	10	0	0
track	183	183	1	5	0	0	0	0	0	0	0	0
trfd	71	71	0	24	0	0	0	0	0	0	0	0
doduc	1619	1608	0	2	2	0	0	0	0	0	0	0
fpppp	254	254	118	3	2	0	1	0	0	0	0	0
matrix300	12	12	0	1	0	0	0	0	0	0	0	0
nasa7	973	965	0	290	18	0	5	0	0	0	0	0
tomcatv	204	204	0	149	50	1	26	16	16	0	0	0
eispack	13242	9866	0	3459	80	2	1389	0	0	0	0	0
linpack	1523	1200	87	344	0	0	113	3	1	0	0	0
Total	33380	29160	877	5619	564	166	1675	33	29	114	100	0
Average	1113	972	29	187	19	6	56	1	1	4	3	0

Table 3.12 SIV Application/Success/Independence Frequencies (Without Constant Propagation).

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
baro	412	35	0	0	0	0	0	0	0	17	17
euler	267	162	0	0	0	0	0	0	0	114	2
heat2d	269	79	19	17	11	1	0	0	2	1	0
linpackd	13	0	0	0	0	23	0	0	2	10	0
mhd2d	73	10	12	0	0	0	0	0	0	4	1
onedim	39	11	1	1	0	58	0	0	12	70	4
shear	179	66	284	18	0	84	0	0	0	4	1
simple	106	44	2	2	0	0	0	0	0	369	44
sphot	280	185	0	0	0	0	0	0	0	11	0
vortex	42	24	0	0	0	0	0	0	0	18	0
adm	632	164	13	13	12	0	0	0	0	139	21
arc2d	9501	3628	0	0	0	107	0	0	0	2868	454
bdna	85	15	2	2	0	119	0	0	1	454	14
dyfesm	1132	558	14	12	0	69	0	0	6	50	0
flo52	2410	996	0	0	0	77	0	0	0	52	10
mdg	464	27	0	0	0	0	0	0	2	20	0
mg3d	362	18	0	0	0	0	0	0	0	5	1
ocean	20	0	34	34	0	0	0	0	0	24	1
qcd	8144	6609	1	1	0	5	0	0	0	187	154
spec77	418	18	35	9	0	16	2	2	0	294	201
spice	4302	1995	0	0	0	4	0	0	9	2005	1966
track	242	77	1	0	0	4	0	0	0	15	0
trfd	6	0	10	0	0	8	2	0	0	12	0
doduc	795	314	0	0	0	0	0	0	0	58	0
fpppp	902	10	1	1	0	0	0	0	0	6	0
matrix300	3	0	0	0	0	1	0	0	0	0	0
nasa7	832	389	49	23	19	86	0	0	9	438	15
tomcatv	32	13	0	0	0	71	0	0	0	0	0
eispack	5445	185	1727	915	182	7366	320	296	958	7521	257
linpack	3271	561	6	6	4	326	1	1	49	882	431
Total	40678	16193	2211	1054	228	8425	325	299	1050	15648	3594
Average	1356	540	74	35	8	281	11	10	35	522	120

Table 3.13 Application/Success/Independence Frequencies (Without Constant Propagation).

manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

Table 3.14 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 were successful in eliminating one or more direction vectors.

ZIV and strong SIV tests dominate the results, accounting for 78% of the applications, 94% of the successes, and 96% of the proven independences. Symbolic constants were present in 32.3% of all successful tests and in 19% of all cases where independence was proven. The ZIV and all SIV tests combined were responsible for 86% of all test applications. 41% of those tests involved symbolic terms. 69% of those symbolic tests were ZIV, 12.6% were strong SIV, 13.4% were weak-zero SIV, 4.8% were weak-crossing SIV, and 0.04% were general SIV.

Table 3.15 shows the difference in dependence test frequencies when constant propagation was not used. In general, there were fewer successes when constant propagation was not used, even for the ZIV test, which had more applications. Only the results for the Delta test were unchanged. Table 3.16 shows the difference between Table 2.6 and Table 3.14. Only the general SIV test has a higher percentage of successes without constant propagation, and that improvement is misleading: there were no additional successes when constant propagation was not used, merely fewer applications, thus raising the percentage of success. The weak-crossing SIV test had the same number of applications but fewer successes, while the MIV test was both less used and less successful without constant propagation. The changes in the other tests were slight. Contrast this with Table 3.9, where the changes were much greater. These results indicate that constant propagation does contribute to the effectiveness of dependence testing, but that constant propagation is not solely or even primarily responsible for the differences between our original study and the new one.

Finally, Table 3.17 shows the number of direction vectors eliminated when constant propagation was and was not used. In general, there were fewer dependences eliminated when constant propagation was not used. This was not the case with *nasa7*, which had fewer dependences *without* constant propagation. This anomaly is probably the result of PFC's normalization phase, where the induction variable substitutor has replaced a formerly symbolic constant with an induction variable expression.

category	ZIV	SIV				MIV	Delta	symbolics used
		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	42.99	35.28	5.94	1.77	0.12	2.34	8.90	
% of all successful tests	33.42	60.19	1.16	0.07	0.21	2.18	0.67	32.30
% of all proven independences	91.01	4.93	0.93	0.16	0.00	1.28	1.68	20.20
% of apps that were successful	39.81	87.36	10.04	1.97	87.72	47.67	3.86	
% of apps that proved independence	39.81	2.63	2.95	1.73	0.00	10.31	3.55	
averaged over all programs								
% of all tests applied	39.06	47.60	4.71	0.70	0.38	1.86	3.38	
% of all successful tests	21.57	73.88	1.47	0.22	0.44	1.56	0.18	18.96
% of all proven independences	72.53	19.32	2.55	2.13	0.00	1.80	1.68	20.07
% of apps that were successful	27.91	94.76	23.14	14.72	70.00	65.13	2.34	
% of apps that proved independence	27.91	3.90	3.29	13.54	0.00	15.66	0.93	

Table 3.14 Comparison of Dependence Tests
(Without Constant Propagation).

	ZIV		SIV												MIV			Delta				symbolics used	
			strong			wk-zero			wk-cross			other											
	A	S/I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	P	S	I
Total	32	-54	-223	-296	-98	0	-1	0	0	-9	-9	-13	0	0	-31	-18	11	0	0	0	0	2213	86
Average	1	-2	-7	-10	-3	0	0	0	0	0	0	0	0	0	-1	-1	0	0	0	0	0	74	3

Table 3.15 Difference in Dependence Test Application/Success/Independence Frequencies Without Constant Propagation.

	ZIV	SIV				MIV	Delta	symbolics
category		strong	wk-zero	wk-cross	other			used
summed over all programs								
% of all tests applied	0.14	-0.15	0.01	0.00	-0.01	-0.03	0.02	
% of all successful tests	0.15	-0.14	0.01	-0.02	0.00	-0.02	0.01	4.78
% of all proven independences	0.46	-0.50	0.01	-0.05	0.00	0.07	0.01	0.65
% of apps that were successful	-0.16	-0.30	-0.02	-0.54	8.98	-0.14	0.00	
% of apps that proved independence	-0.16	-0.27	0.00	-0.54	0.00	0.63	0.00	
averaged over all programs								
% of all tests applied	0.69	-0.63	0.00	0.00	-0.02	-0.07	0.02	
% of all successful tests	-0.12	0.23	0.00	-0.04	0.00	-0.08	0.00	5.54
% of all proven independences	3.75	-2.81	0.00	-1.03	0.00	0.10	0.00	6.80
% of apps that were successful	-0.50	0.07	-0.01	-16.67	11.67	-5.25	0.00	
% of apps that proved independence	-0.50	-0.84	0.00	-16.67	0.00	1.32	0.00	

Table 3.16 Difference in Dependence Tests Without Constant Propagation.

	with const prop		no const prop		diff in	
program	dvs elim	dvs left	dvs elim	dvs left	dvs elim	max dvs
baro	1692	678	1577	793	115	2430
euler	1231	344	1209	366	22	1710
heat2d	1248	987	1248	987	0	2802
linpackd	156	160	148	168	8	342
mhd2d	13367	2554	13355	2566	12	19503
onedim	3690	1519	3690	1519	0	5784
shear	57256	12131	57256	12131	0	77343
simple	1578	496	1573	501	5	2118
sphot	7587	583	7587	583	0	11565
vortex	602	254	602	254	0	867
adm	13196	2023	13196	2023	0	15873
arc2d	29776	2889	29549	3116	227	32763
bdna	6332	5832	6332	5832	0	15060
dyfesm	5666	1718	5666	1718	0	8082
flo52	19469	1452	19469	1452	0	21054
mdg	3367	3639	2609	4199	758	9687
mg3d	37752	5305	37748	5309	4	48696
ocean	1119	2020	1119	2020	0	5736
qcd	101273	6126	101273	6126	0	207258
spec77	13849	4631	13612	4658	237	19818
spice	7402	8018	7402	8018	0	19596
track	2396	794	2324	810	72	4245
trfd	4202	954	4202	954	0	9171
doduc	5650	1611	5650	1611	0	7530
fpppp	1804	8726	1804	8726	0	124419
matrix300	40	8	40	8	0	48
nasa7	10986	1827	11058	1755	-72	13713
tomcatv	2629	814	2629	814	0	3459
eispack	73467	77250	73467	77250	0	190749
linpack	6044	12687	6044	12687	0	20397
Total	434826	168030	433438	168954	1388	901818
Average	14494	5601	14448	5632	46	130061

Table 3.17 Dependences Eliminated With and Without Constant Propagation.

3.5 Conclusions

We have seen that PFC's performance has improved in the last six years and that the improvement is only partially due to the use of interprocedural constant propagation. PFC was developed over a period of several years by dozens of people and was intended to be an experimental vehicle rather than a product. As a result, PFC does some things exceptionally well, but it can also fall over when it encounters something not anticipated by the developers. Our research has entailed a considerable implementation burden, but much of that effort was expended not in coding the new techniques, but in the robustification of extant parts of PFC and the integration of those parts with the new material.

That being the case, why was PFC used, particularly when an alternative, ParaScope, would seem to have been available? The answer is simple: PFC was the best dependence analysis tool we had. As troublesome as it has been, the quality of interprocedural analysis in PFC outstripped what was currently available in ParaScope and also what we (correctly) thought would be available for some time.

Our experiences have led us to two conclusions:

- If you already have a high quality robust tool, adding these techniques should not be too onerous.
- That is a rather large "if".

Chapter 4

Breaking and Proving Conditions

This chapter describes the creation and use of breaking and proving conditions, which allow us to take advantage of compiler-generated information at run-time. Section 4.1 introduces the concept of breaking and proving conditions and Section 4.2 discusses their implementation. Experimental results are described in Section 4.3, related work is presented in Section 4.4, and we conclude in Section 4.5.

4.1 Introduction

Consider the following loop.

```
DO I = 1, N
   A(5,I-1) = A(M,I)                     (S1)
ENDDO
```

Statement S1 has a loop-carried true dependence and a loop-carried antidependence on itself. If M were not equal to 5, neither of these dependences would exist, so the condition $\mathsf{M - 5} \neq 0$ is said to *break* both dependences. When a dependence involving a symbolic term cannot be disproved, a *breaking condition* for that dependence may be calculated. The breaking condition will always take the form

$$\mathsf{symbolic\_difference} \neq 0,$$

where symbolic_difference is the difference of the two subscript positions being compared. At code generation time, if there are no other dependences preventing vectorization or parallelization of the loop, the generated code will take the form:

```
IF breaking condition is satisfied
    THEN execute vectorized or parallelized loop
    ELSE execute serial version of loop
```

Thus, breaking conditions are run-time checks, generated from the information available during dependence analysis, that determine whether or not the optimized code is executed.

Proving Conditions

A *proving condition*, when satisfied, indicates that a possible dependence does in fact exist. Proving conditions are logical negations of breaking conditions.

In the following example, loop-carried true and anti dependences will exist when M= 5 and $2 \leq N + 10 \leq 2N$.

```
DO I = 1, N
   A(5,N-I+10) = A(M,I)                    (S1)
ENDDO
```

Proving conditions can be calculated in the following way. At each level k, symbolic_difference is assigned the value of the difference of the subscripts at level k and the proving condition for level k will be symbolic_difference = 0. The proving condition for the entire reference pair will be the conjunction of the proving conditions for the individual levels, since the subscripts must be equal at every level for the same memory location to be accessed by both references. The proving condition for the entire reference pair in this example would therefore be

$$\mathsf{M - 5 = 0} \textbf{ and } \mathsf{I_2 - N + I_1 - 10 = 0},$$

where I_1 and I_2 are two iterations of the loop. The condition

$$\mathsf{I_2 - N + I_1 - 10 = 0} \text{ is equivalent to } \mathsf{I_1 + I_2 = N + 10}.$$

We know that $1 \leq I_1, I_2 \leq N$, so $2 \leq I_1 + I_2 \leq 2N$, and we can replace the condition

$$\mathsf{I_1 + I_2 = N + 10} \text{ with } 2 \leq \mathsf{N + 10} \leq \mathsf{2N}.$$

Thus, the proving condition for the example above could be expressed as

$$\mathsf{M - 5 = 0} \textbf{ and } 2 \leq \mathsf{N + 10} \leq \mathsf{2N}.$$

Proving conditions can be useful in the detection of access anomalies. One method for detecting access anomalies is to maintain two lists for each shared variable: one, of the variable's readers, and the other, of its writers. Tags are used to detect when a previous or pending access conflicts with a current access. In effect, the sources and sinks of potential dependence edges in a parallel loop are annotated and a possible anomaly is reported when both endpoints of an edge can be executed concurrently.

Anomaly detection could be done more efficiently if proving conditions were used. A proving condition, derived during dependence analysis, would be tested at the beginning of the parallel loop. If the condition were met, then an anomaly would exist and there would be no need to annotate the loop and do any further work. If the condition were not met, then the breaking condition would be true and we would know that no anomaly could exist.

4.2 Implementation of Breaking and Proving Conditions

Breaking and proving conditions fall naturally out of the work already being done during dependence analysis, as they can be generated from the direction vectors which remain after testing. Since proving conditions are merely the logical negations of breaking conditions, only breaking conditions are explicitly dealt with in PFC.

4.2.1 Creating Breaking Conditions

During dependence analysis, individual dependence tests are invoked to determine whether two references are independent of each other. The tests calculate the difference between two references and try to determine whether that difference is nonzero.

In the past, if the value of the difference was unknown, dependence was assumed and no use was made of a piece of information we had acquired: namely, that the dependence *could* be broken if we could prove the difference were nonzero. Now, instead of discarding the information that dependence x does not exist when the difference is nonzero, we store that information in a breaking condition. This breaking condition will be passed to the code generator, and the information gathered during dependence analysis will ultimately be used at run-time.

Breaking conditions are only created under certain circumstances. First, the difference must contain a symbolic term. The word "symbolic" is applied somewhat loosely here, as it includes both "genuine" symbolic terms, i.e., N, and induction variables appearing as symbolic additive constants. Consider the following code fragment:

```
DO I = 1, 100                              (S1)
   A(L, I) =  ...
   DO J = 1, 100
     ...    = A(L, I + J + 1 - 1)          (S2)
   ENDDO
ENDDO
```

Even though two induction variables appear in the second subscript position in S2, the SIV test will be applied when comparing S1 and S2. Because S1 is outside the J loop, J should be treated as a symbolic additive constant rather than an induction variable. Thus, the breaking condition would be `J .NE. 0`.

The second restriction on the creation of breaking conditions is that the dependence distance must not be variable. Figure 2.1 in section 2.4.2 of Chapter 2 gives a geometric view of SIV tests. The lower part of that figure indicates the nature of the dependence distance for each test, making it clearly visible that the dependence distance varies for weak SIV tests, but not for strong ones.

Using the two restrictions stated above, breaking conditions are only created for ZIV and strong SIV tests when the difference contains a symbolic term. Since ZIV and strong SIV cases account for 80% of all cases tested and 36% of those tests contain symbolic terms, breaking conditions are being created for 29% of all tests.

4.2.2 Breaking Conditions and Code Generation

After a breaking condition has been generated, the dependence graph is annotated with that information. At this point, the number of dependences potentially eliminated can be readily calculated, but we cannot determine how many additional loops might be parallelized because the breaking conditions pertain to individual edges in the dependence graph, not to loops.

The dependence graph is divided into maximal strongly connected regions as a preliminary to the code generation phase of PFC. Nodes in the graph represent program statements, while edges represent dependences between statements.

For each strongly connected component or *piblock*, PFC tries to break as many recurrences as possible. Once recurrence breaking is done, code will be generated for each piblock, yielding the loops of the parallelized program.

Individual breaking conditions are generated during dependence analysis, but the composite conditions are created as part of the calculating-piblocks phase of the code generator. For each piblock, we collect the breaking conditions for all statements in the piblock and `AND` them together to form a composite breaking condition. Code will then be generated at most twice for each piblock in the graph: once, assuming the relevant recurrences in the piblock have been broken, and once, assuming they have not.

4.2.3 Breaking Conditions and Control Flow

To test a breaking or proving condition at the top of a loop, the variables involved in the condition must be loop-invariant. If they are not, then the condition should be tested just before executing the statement the condition pertains to. If the variables involved in the condition are loop-invariant, then there is no need to worry about the possibility of those variables having different values along different control flow paths within the loop; such an occurrence would contradict the definition of loopinvariance. Therefore, the only effect that control flow could have upon breaking or proving conditions would be that the statement the condition was derived for might or might not be executed, depending on which path in the loop one followed.

As an example, consider the application of proving conditions to access anomaly detection and suppose we have a proving condition which contains no loop-variant terms. Then we can test that condition at the head of the loop before annotation is done. Also suppose that there is some control flow within the loop such that the statement the condition is associated with is executed conditionally. If the proving condition is true but the statement is not executed, then we have annotated for a potential anomaly that would not have been encountered on this execution of the loop. Including a boolean test mimicking the control flow within the loop in the proving condition would prevent this unnecessary annotation but it would also destroy the if-and-only-if property of the condition. By not including control flow in breaking and proving conditions, we can test a single condition: if true, we know that a dependence exists; if false, we know that it cannot.

4.2.4 Trivial Breaking Conditions

Some breaking conditions may be considered *trivial*, as they can be proven or disproven during dependence analysis. For example, the breaking condition

```
IVAR1 .NE. 0
```

is known to be true, because normalized induction variables run from 1 to the normalized upper loop bound, i.e., they can never be less than one. Similarly, the breaking condition

```
IVAR1 - 1 .NE. 0
```

is false, because IVAR1 can be equal to 1.

All of the breaking conditions identified for our test suite were inspected to determine the prevalence of trivial breaking conditions. The results of that study are summarized in Figures 4.1 and 4.2. 32% of all breaking conditions generated were trivial, with 97% of those trivial conditions arising from SIV tests. Approximately 80% of all trivial conditions were true; only 20% were false.

4.2.5 Detecting Trivial Breaking Conditions

Removing trivial breaking conditions during dependence analysis will not affect the correctness of the code generated but should improve the efficiency of code genera-

category	trivially true	trivially false	must be tested	total
ZIV	36	35	4775	4846
SIV	2125	519	1075	3719
Total	2161	554	5850	8565

Figure 4.1 Categorization of Breaking Conditions.

% of trivial breaking conditions	trivially true	trivially false	all trivial conditions
ZIV trivial b.c.'s			
relative to ZIV trivial b.c.'s	50.7	49.3	100
relative to all trivial b.c.'s	1.33	1.29	2.62
SIV trivial b.c.'s			
relative to SIV trivial b.c.'s	80.4	19.6	100
relative to all trivial b.c.'s	78.3	19.1	97.4
all trivial b.c.'s			
relative to all trivial b.c.'s	79.6	20.4	100
relative to all b.c.'s	25.2	6.5	31.7

Figure 4.2 Percentage of Trivial Breaking Conditions.

tion and run-time testing by eliminating unnecessary cases. If we know a breaking condition must be true (or false), there is no need to generate and test code assuming that it is not.

The following algorithm, summarized in Figure 4.3, could be used to remove trivial conditions in PFC.

1. There must be no "genuine" symbolic terms, but induction variables masquerading as symbolic additive constants may be present. Constants may also be present, i.e., the breaking condition is of the form

   $$\mathsf{coeff1} * \mathsf{ivar1} + \mathsf{coeff2} * \mathsf{ivar2} + \ldots + \mathsf{const1} + \mathsf{const2} + \ldots \mathsf{.NE.\ 0.}$$

2. Sum all constants, i.e., $consts = const1 + const2 + \ldots$

3. If all of the coefficients are negative, multiply the entire condition by -1.

---

1. Consider only breaking conditions of the form

   `coeff1 * ivar1 + coeff2 * ivar2 + ... + const1 + const2 + ... .NE. 0.`

2. Sum all constant terms.
3. If all coefficients are negative, multiply breaking condition by -1.
4. For breaking conditions containing only one induction variable:
   (a) If *consts* $= 0$, breaking condition is TRUE.
   (b) If *consts* $\neq 0$, breaking condition is `coeff*ivar + consts .ne. 0.`
      i. if *coeff* and *consts* have same sign, breaking condition is TRUE.
      ii. if *coeff* = -*consts*, breaking condition is FALSE.
      iii. if *coeff* > -*consts*, breaking condition is TRUE.
      iv. otherwise, test condition at run-time.
5. For breaking conditions containing multiple induction variables:
   (a) If coefficients have same sign, then sum the coefficients.
   (b) If $|$*coeffs*$| > |$*consts*$|$, breaking condition is TRUE.
   (c) Otherwise, test condition at run-time.

Figure 4.3 Algorithm to Detect Trivial Breaking Conditions.

---

4. For breaking conditions containing only one induction variable:

   (a) If *consts* $= 0$, the breaking condition can be eliminated as TRUE, since the lowest possible ivar is 1. Normalization leaves us with induction variables that run from 1 to some upper bound N.

   (b) If *consts* $\neq 0$, breaking condition is

   `coeff*ivar + consts .ne. 0.`

   The proving condition would be `coeff*ivar .eq. -consts` and expand to

   `coeff*(lower bound for ivar)` $\leq$ `-consts` $\leq$ `coeff*(upper bound for ivar)`

If this proving condition is false, the corresponding breaking condition will be true, and vice versa.

   i. if *coeff* and *consts* have the same sign, the lower range is violated if both are positive and the upper range is violated if both are negative. For example, if *coeff* and *consts* are both positive, the proving condition would look like

      $$\text{coeff * 1} \leq \text{a negative number} \leq \text{coeff * upper bound}$$

      which is false, since *coeff* > 0. The breaking condition can be eliminated as TRUE.

   ii. if *coeff* $=$ -*consts*, the proving condition

      $$\text{coeff * 1} \leq \text{-consts} \leq \text{coeff * upper bound}$$

      will be trivially satisfied, since the upper bound must be at least one. Therefore, the breaking condition should be eliminated as FALSE.

   iii. if *coeff* $>$ -*consts*, the proving condition

      $$\text{coeff * 1} \leq \text{-consts} \leq \text{coeff * upper bound}$$

      will be false, as *coeff* cannot be $\leq$ -*consts*. The breaking condition should be eliminated as TRUE.

   iv. any single-ivar breaking conditions which do not fall into the above categories will have to be tested at run-time.

5. For breaking conditions containing multiple induction variables:

   (a) All coefficients must have the same sign. (The breaking condition must be tested at run-time if the coefficients have differing signs.)

   (b) Add the coefficients. If their absolute value is greater than the absolute value of *consts*, then the proving condition

   $$\text{coeff1 * 1} + \text{coeff2 * 1} + \text{...} \leq \text{-consts} < \text{coeff1*upper\_bnd1} + \text{coeff2*upper\_bnd2} + \ldots$$

   will be false, and the breaking condition can be eliminated as TRUE. Otherwise, the breaking condition will have to be tested at run-time.

4.3 Experimental Results

For this experiment, we used all of the programs in the test suite described in Chapter 2. Due to the limitations of PFC's code generator, we were unable to generate and run actual code. Instead, we measured the number of direction vectors which potentially could be eliminated using breaking conditions.

The test suite was executed twice, once assuming all breaking conditions could be satisfied, and once assuming that none of them could be satisfied. Normally, edges corresponding to direction vectors are added to the dependence graph, but no such edges were added when we assumed the breaking condition would be met at run-time. Thus, comparing the results between these two runs of the test suite gives an idea of the best performance we could expect using breaking conditions.

The results obtained when we assumed no breaking conditions could be met are given in Chapter 2. Here, we show the results obtained when we assume breaking conditions are satisfied.

Table 4.1, which details the basic characteristics of our test suite, is identical to Table 2.1 and is reproduced here for easy reference. Table 4.2 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. It is similar to Table 2.2, but there are fewer nonlinear subscripts reported in Table 4.2

Table 4.3 gives the number of indices for each subscript pair. 44.2% of all subscript pairs had zero indices, 46.9% had only one, 6.9% had two, 1.8% had three, and only 0.25% had four indices; none had more than four. Thus, 99.1% of all subscript pairs had either zero or one indices.

Tables 4.4 and 4.5 detail how often and how successfully each type of dependence test was employed. Table 4.4 is devoted to the various SIV tests while Table 4.5 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*. Note that the S and I columns are combined for the ZIV test because they are always identical. As in Chapter 2, no results are presented for the triangular MIV test.

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

program	type	lines	subrs
	RiCEPS		
baro	Shallow Water Atmosphere	1002	7
euler	1D Unsteady Euler	1200	14
heat2d	Heat Conduction System	336	2
linpackd	Linear Algebra Benchmark	797	11
mhd2d	2D MHD Equations	927	14
onedim	Eigenfunction/Eigenenergies	1016	16
shear	3D Turbulence	914	15
simple	2D Hydrodynamics	1892	18
sphot	Particle Transport	1144	7
vortex	Vortex Simulation	709	20
	Perfect		
adm (APS)	Pseudospectral Air Pollution	6105	97
arc2d (SRS)	2D Fluid Flow Solver	3965	39
bdna (NAS)	Molecular Dynamics of DNA	3980	43
dyfesm (SDS)	Structural Dynamics	7610	78
flo52 (TFS)	Transonic Inviscid Flow	1987	28
mdg (LWS)	Molecular Dynamics of Water	1238	16
mg3d (SMS)	Depth Migration	2812	28
ocean (OCS)	2D Ocean Simulation	4343	36
qcd (LGS)	Quantum Chromodynamics	2327	35
spec77 (WSS)	Weather Simulation	3885	65
spice (CSS)	Circuit Simulation	18524	128
track (MTS)	Missile Tracking	3735	34
trfd (TIS)	2 Electron Integral Transform	485	7
	SPEC		
doduc	Thermohydraulical Modelization	5334	41
fpppp	2 Electron Integral Derivative	2718	38
matrix300	Matrix Multiplications	439	6
nasa7	NASA Ames Fortran Kernels	1105	17
tomcatv	Mesh Generation	195	1
	Math Libraries		
eispack	Eigensystems Library	11886	76
linpack	Linear Algebra Library	7427	51
Total		100037	988
Average		3335	33

Table 4.1 Basic Program Characteristics.

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
baro	56	360	0	0	416	776	0	776	5
euler	106	294	0	0	400	694	0	694	8
heat2d	151	159	0	0	310	447	22	469	20
linpackd	52	36	0	0	88	74	50	124	25
mhd2d	280	196	21	0	497	711	24	735	232
onedim	297	209	0	0	506	567	148	715	47
shear	787	0	828	0	1615	2415	856	3271	368
simple	165	163	0	0	328	491	0	491	0
sphot	282	77	0	0	359	436	0	436	21
vortex	179	42	0	0	221	263	0	263	0
adm	642	160	521	0	1323	2501	24	2525	19
arc2d	168	405	4396	0	4969	13172	994	14166	5
bdna	1787	269	0	0	2056	2085	240	2325	409
dyfesm	268	866	7	7	1148	1897	152	2049	0
flo52	54	256	1624	0	1934	5284	154	5438	0
mdg	928	5	0	0	933	934	4	938	92
mg3d	749	52	133	40	974	1412	0	1412	491
ocean	439	29	0	0	468	497	0	497	322
qcd	7804	271	11	0	8086	8369	10	8379	0
spec77	1176	500	54	0	1730	2290	48	2338	50
spice	4529	51	0	0	4580	4605	26	4631	47
track	250	115	0	0	365	472	8	480	2
trfd	39	64	0	0	103	135	32	167	6
doduc	657	885	0	0	1542	2427	0	2427	11
fpppp	1023	108	0	0	1131	1239	0	1239	0
matrix300	0	8	0	0	8	14	2	16	0
nasa7	155	104	329	347	935	2263	475	2738	28
tomcatv	0	207	0	0	207	272	142	414	0
eispack	3001	13846	0	0	16847	12719	17974	30693	3391
linpack	3491	1084	0	0	4575	4897	762	5659	189
Total	29515	20821	7924	394	58654	74358	22147	96505	5788
Average	984	694	264	13	1955	2479	738	3217	193

Table 4.2 Dimensionality, Separability, and Linearity of Pairs Tested (With Breaking Conditions).

	subscript pair indices				
program	0	1	2	3	4
baro	380	315	0	0	0
euler	267	259	6	0	0
heat2d	254	131	23	0	0
linpackd	13	106	5	0	0
mhd2d	73	506	45	93	0
onedim	39	602	46	0	0
shear	179	2023	340	93	0
simple	106	361	2	0	0
sphot	280	135	21	0	0
vortex	42	221	0	0	0
adm	614	1508	24	8	0
arc2d	9475	3060	0	0	0
bdna	85	1807	418	0	0
dyfesm	1132	623	41	0	0
flo52	2402	2877	0	0	0
mdg	464	389	84	0	1
mg3d	362	884	104	0	0
ocean	20	276	108	93	0
qcd	8139	225	1	0	0
spec77	418	1788	99	3	0
spice	4302	312	12	0	0
track	242	189	7	0	0
trfd	5	95	64	2	0
doduc	795	1632	0	0	0
fpppp	902	258	18	0	1
matrix300	3	13	0	0	0
nasa7	832	1265	81	0	0
tomcatv	32	379	0	0	0
eispack	4607	17708	4462	1316	224
linpack	3164	2092	196	0	0
Total	39628	42039	6207	1608	226
Average	1321	1401	207	54	8

Table 4.3 Number of Subscript Pair Indices (With Breaking Conditions).

	SIV											
	strong			weak-zero			weak-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
baro	310	310	0	0	0	0	0	0	0	0	0	0
euler	251	251	7	2	0	0	4	0	0	0	0	0
heat2d	96	96	16	15	2	0	0	0	0	0	0	0
linpackd	81	81	16	0	0	0	0	0	0	0	0	0
mhd2d	384	384	1	12	0	0	4	4	4	0	0	0
onedim	468	468	52	92	8	4	0	0	0	0	0	0
shear	1327	1327	151	473	261	129	4	0	0	0	0	0
simple	356	356	0	5	5	0	0	0	0	0	0	0
sphot	134	134	0	1	0	0	0	0	0	0	0	0
vortex	221	221	0	0	0	0	0	0	0	0	0	0
adm	1342	1342	160	17	1	0	97	0	0	52	52	0
arc2d	2808	2808	13	247	68	12	0	0	0	0	0	0
bdna	1415	1415	158	86	1	0	0	0	0	0	0	0
dyfesm	531	531	7	92	0	0	0	0	0	0	0	0
flo52	2683	2683	6	170	37	0	16	2	0	8	8	0
mdg	382	382	127	0	0	0	0	0	0	0	0	0
mg3d	459	459	55	2	0	0	0	0	0	36	36	0
ocean	108	108	9	1	1	0	4	0	0	8	4	0
qcd	192	192	50	31	2	0	0	0	0	0	0	0
spec77	1717	1717	335	37	0	0	12	12	12	13	0	0
spice	221	221	32	35	19	18	0	0	0	10	0	0
track	183	183	1	5	0	0	0	0	0	0	0	0
trfd	71	71	0	24	0	0	0	0	0	0	0	0
doduc	1619	1619	11	2	2	0	0	0	0	0	0	0
fpppp	254	254	118	3	2	0	1	0	0	0	0	0
matrix300	12	12	0	1	0	0	0	0	0	0	0	0
nasa7	964	964	5	288	18	0	5	5	5	0	0	0
tomcatv	204	204	0	149	50	1	26	16	16	0	0	0
eispack	12155	12155	2966	3237	80	2	1389	0	0	0	0	0
linpack	1503	1503	398	321	0	0	113	3	1	0	0	0
Total	32451	32451	4694	5348	557	166	1675	42	38	127	100	0
Average	1082	1082	156	178	19	6	56	1	1	4	3	0

Table 4.4 Application/Success/Independence Frequencies for SIV Test (With Breaking Conditions).

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
baro	380	86	0	0	0	0	0	0	0	4	4
euler	267	162	0	0	0	0	0	0	0	117	6
heat2d	254	103	19	17	11	1	0	0	2	41	40
linpackd	13	0	0	0	0	11	0	0	2	18	16
mhd2d	73	10	12	0	0	0	0	0	0	0	0
onedim	39	12	1	1	0	45	0	0	12	91	53
shear	179	67	284	18	0	84	0	0	0	5	2
simple	106	45	2	2	0	0	0	0	0	4	0
sphot	280	195	0	0	0	0	0	0	0	21	10
vortex	42	24	0	0	0	0	0	0	0	18	0
adm	614	178	13	13	12	0	0	0	0	159	41
arc2d	9475	3622	0	0	0	107	0	0	0	1482	471
bdna	85	23	2	2	0	119	0	0	1	606	166
dyfesm	1132	558	14	12	0	63	0	0	6	57	7
flo52	2402	1000	0	0	0	77	0	0	0	60	18
mdg	464	27	0	0	0	0	0	0	2	67	29
mg3d	362	82	0	0	0	0	0	0	0	72	66
ocean	20	0	34	34	0	0	0	0	0	25	2
qcd	8139	6717	1	1	0	5	0	0	0	345	312
spec77	418	49	61	33	0	16	2	2	0	407	295
spice	4302	3178	0	0	0	4	0	0	4	3212	3173
track	242	93	6	5	0	4	0	0	0	31	16
trfd	5	1	10	0	0	8	2	0	0	13	1
doduc	795	314	0	0	0	0	0	0	0	69	11
fpppp	902	807	1	1	0	0	0	0	0	803	797
matrix300	3	0	0	0	0	1	0	0	0	0	0
nasa7	832	402	49	12	8	86	0	0	4	39	18
tomcatv	32	13	0	0	0	71	0	0	0	0	0
eispack	4607	1866	1697	906	173	5281	320	296	850	11819	4895
linpack	3164	1398	6	6	4	242	1	1	46	2030	1579
Total	39628	21032	2212	1063	208	6225	325	299	929	21615	12028
Average	1321	701	74	35	7	208	11	10	31	720	401

Table 4.5 Application/Success/Independence Frequencies (With Breaking Conditions).

columns in Table 4.5 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

The Delta test constraint intersection algorithm tested 6225 coupled groups exactly (57%). Propagation of distance constraints was applied in 929 cases (8.4%), converting MIV subscripts into SIV form in all but 33 cases. The Delta test thus managed to test 7121 coupled groups exactly (65%), using only constraint intersection and propagation of dependence distances.

Table 4.6 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 were successful in eliminating one or more direction vectors.

As before, ZIV and strong SIV tests dominate the results, accounting for 80% of all applications, 95% of all successes, and 97% of all proven independences. Few of the subscripts tested were MIV (5%), with 2.6% of all cases tested using the triangular MIV test. (All of the MIV numbers presented in Table 4.6 are for the normal MIV test only.) With breaking conditions, the prevalence of symbolic additive constants rose to 38% of all successful tests and 46% of all cases where independence was proven. 40% of all ZIV or SIV tests used symbolics: 70% were ZIV, 12% were strong SIV, 13% were weak-zero SIV, 5% were weak-crossing SIV, and 0.1% were general SIV.

Table 4.7 shows the difference between Table 2.3 and Table 4.3. For a few of the programs, most notably *eispack*, fewer subscript pair indices were reported when breaking conditions were used. Table 4.8 shows the drop in total number of test applications between Tables 2.4 and 2.5 and Tables 4.4 and 4.5, respectively. These decreases are to be expected as a side effect of this experiment, as we stopped testing for dependence after generating a breaking condition. The corresponding general increase in the number of successful tests and the number of times independence was proven is another by-product of our decision to declare success and cease testing upon discovering a breaking condition. It is mildly surprising to note that the weak-zero SIV test and the MIV test have fewer successes when breaking conditions are used, but these tests also have far fewer applications in that situation. By terminating dependence testing "prematurely", we have merely pre-empted the testing of those subscript pairs which would have been reported as successful.

	ZIV	SIV				MIV	Delta	symbolics used
category		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	43.96	36.00	5.93	1.86	0.14	2.45	6.91	
% of all successful tests	37.25	57.47	0.99	0.07	0.18	1.88	0.58	38.28
% of all proven independences	79.56	17.76	0.63	0.14	0.00	0.79	1.13	45.50
% of apps that were successful	53.07	100.00	10.42	2.51	78.74	48.06	5.22	
% of apps that proved independence	53.07	14.46	3.10	2.27	0.00	9.40	4.80	
averaged over all programs								
% of all tests applied	38.36	48.67	4.74	0.73	0.40	1.96	2.83	
% of all successful tests	25.82	69.96	1.42	0.26	0.42	1.50	0.16	20.13
% of all proven independences	67.65	27.01	1.63	2.91	0.00	0.60	0.21	34.45
% of apps that were successful	37.26	100.00	23.20	31.39	58.33	70.40	2.44	
% of apps that proved independence	37.26	9.99	3.41	30.20	0.00	14.32	1.03	

Table 4.6 Comparison of Dependence Tests (With Breaking Conditions).

	subscript pair indices				
program	0	1	2	3	4
baro	0	0	0	0	0
euler	0	0	0	0	0
heat2d	15	0	0	0	0
linpackd	0	0	0	0	0
mhd2d	0	0	0	0	0
onedim	0	5	8	0	0
shear	0	1	0	0	0
simple	0	0	0	0	0
sphot	0	0	0	0	0
vortex	0	0	0	0	0
adm	18	9	0	0	0
arc2d	26	26	0	0	0
bdna	0	0	0	0	0
dyfesm	0	0	0	0	0
flo52	8	8	0	0	0
mdg	0	0	0	0	0
mg3d	0	0	0	0	0
ocean	0	0	0	0	0
qcd	5	6	0	0	0
spec77	0	0	0	0	0
spice	0	5	0	0	0
track	0	0	0	0	0
trfd	1	0	0	0	0
doduc	0	0	0	0	0
fpppp	0	0	0	0	0
matrix300	0	0	0	0	0
nasa7	0	11	2	0	0
tomcatv	0	0	0	0	0
eispack	838	1339	67	0	20
linpack	107	53	0	0	0
Total	1018	1463	77	0	20
Average	34	49	3	0	1

Table 4.7 Decrease in Number of Subscript Pair Indices When Using Breaking Conditions.

	ZIV		SIV												MIV			Delta				symbolics used	
			strong			wk-zero			wk-cross			other											
	A	S/I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	P	S	I
Total	-1018	4785	-1152	2995	3719	-271	-8	0	0	0	0	0	0	0	-30	-9	-9	-2200	0	0	-121	8180	8520
Average	-34	160	-38	100	124	-9	0	0	0	0	0	0	0	0	-1	0	0	-73	0	0	-4	273	284

Table 4.8 Decrease in Test Frequencies When Using Breaking Conditions.

Table 4.9 shows the difference between Table 2.6 and Table 4.6. As expected, more applications were successful or proved independence when breaking conditions were used. The ZIV test was responsible for fewer indepedences when breaking conditions were used, but the strong SIV test showed a corresponding improvement. The increase in the number of tests where symbolic terms played a role increased dramatically when breaking conditions were used: 38% of all successful tests and 46% of the cases where independence was proven involved symbolics, as opposed to 28% and 20%, respectively, when breaking conditions were not employed. This increase shows how much could be gained by handling symbolic terms more effectively.

Table 4.10 lists the number of dependences eliminated both with and without the use of breaking conditions. Note that the sum of dependences eliminated and dependences remaining is not always the same with and without breaking conditions. For example, when *heat2d* is tested without breaking conditions, 1248 dependences are eliminated while 987 are added to the dependence graph, for a total of 2235 dependences. When the same program is tested with breaking conditions enabled, 1440 dependences are eliminated and 843 dependences remain, for a total of 2283 dependences. This is an artifact of the way dependences are counted in PFC. For a direction vector of $(*)$, there are three possible dependences, but if the dependence in question exists at an outer level, such that the direction vector is $(*,*)$, there are now nine possibilities to consider. Thus, any dependences eliminated at outer levels are weighted to take inner loops into account. The number of dependences reported is therefore partially dependent on *when* the elimination takes place. Since the use of breaking conditions tends to "discover" independence earlier, it is more likely that independence will be found at an outer level and/or the first subscript position tested, thus maximizing the number of dependences eliminated while minimizing the number of edges added to the dependence graph. The maximum number of possible dependences, which is $3^{\#levels}$, is usually far greater than the sum of dependences eliminated plus dependences actually added to the graph.

Finally, Table 4.11 shows the number of times breaking conditions were used, the total number of direction vectors (dependences) eliminated, and the percentage of all eliminated dependences and all successes which can be attributed to breaking conditions. Thus, 23% of all dependences eliminated and 14% of all successful tests were due to the use of breaking conditions. These statistics do include the breaking conditions we identified as trivial, but since only 6.5% of all breaking conditions were known to be false, these numbers should not be significantly skewed. Note that these

	ZIV	SIV				MIV	Delta	symbolics used
category		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	1.11	0.58	0.01	0.09	0.01	0.09	-1.98	
% of all successful tests	3.97	-2.86	-0.17	-0.01	-0.03	-0.31	-0.09	10.76
% of all proven independences	-11.00	12.32	-0.30	-0.07	0.00	-0.42	-0.54	25.94
% of apps that were successful	13.10	12.34	0.36	0.00	0.00	0.24	1.36	
% of apps that proved independence	13.10	11.56	0.15	0.00	0.00	-0.28	1.25	
averaged over all programs								
% of all tests applied	-0.01	0.44	0.03	0.02	0.00	0.03	-0.53	
% of all successful tests	4.13	-3.70	-0.05	0.00	-0.03	-0.15	-0.02	6.71
% of all proven independences	-1.13	4.87	-0.92	-0.26	0.00	-1.10	-1.47	21.18
% of apps that were successful	8.84	5.31	0.04	0.00	0.00	0.02	0.10	
% of apps that proved independence	8.84	5.24	0.12	0.00	0.00	-0.02	0.09	

Table 4.9 Increase in Dependence Tests When Using Breaking Conditions.

	w/o b.c.'s		with b.c.'s	
program	dvs elim	dvs left	dvs elim	dvs left
baro	1692	678	1692	678
euler	1231	344	1249	326
heat2d	1248	987	1440	843
linpackd	156	160	196	102
mhd2d	13367	2554	13367	2554
onedim	3690	1519	4093	1160
shear	57256	12131	57265	12122
simple	1578	496	1578	496
sphot	7587	583	8397	493
vortex	602	254	602	254
adm	13196	2023	13310	1945
arc2d	29776	2889	29821	2844
bdna	6332	5832	6812	5368
dyfesm	5666	1718	5677	1707
flo52	19469	1452	19477	1444
mdg	3367	3639	3616	3390
mg3d	37752	5305	38319	4986
ocean	1119	2020	1128	2011
qcd	101273	6126	102047	5352
spec77	13849	4631	14779	3917
spice	7402	8018	13459	4073
track	2396	794	2480	734
trfd	4202	954	4283	945
doduc	5650	1611	5683	1578
fpppp	1804	8726	98599	1155
matrix300	40	8	40	8
nasa7	10986	1827	11358	1695
tomcatv	2629	814	2629	814
eispack	73467	77250	96508	56269
linpack	6044	12687	9988	8775
Total	434826	168030	569892	128056
Average	14494	5601	18996	4269

Table 4.10 Dependences Eliminated With and Without Breaking Conditions.

program	b.c.'s used	dvs elim	% of elim	% of succs
baro	0	0	0.00	0.00
euler	4	18	1.44	0.97
heat2d	40	192	12.95	17.94
linpackd	16	40	16.95	18.18
mhd2d	0	0	0.00	0.00
onedim	53	417	9.58	9.93
shear	1	9	0.02	0.06
simple	0	0	0.00	0.00
sphot	10	810	9.65	3.04
vortex	0	0	0.00	0.00
adm	20	204	1.53	1.26
arc2d	26	468	1.54	0.39
bdna	152	480	5.73	9.68
dyfesm	7	11	0.18	0.60
flo52	8	24	0.12	0.21
mdg	29	249	6.89	7.06
mg3d	65	567	1.48	11.27
ocean	1	9	0.80	0.68
qcd	158	882	0.86	2.28
spec77	94	930	6.29	5.18
spice	1207	6057	44.57	35.17
track	16	84	3.39	5.69
trfd	1	81	1.21	0.96
doduc	11	33	0.58	0.57
fpppp	797	96795	92.69	74.28
matrix300	0	0	0.00	0.00
nasa7	18	648	5.62	1.27
tomcatv	0	0	0.00	0.00
eispack	4654	25905	21.97	24.79
linpack	1177	4083	36.10	37.60
Total	8565	138996	22.93	14.32
Average	286	4633	9.40	8.97

Table 4.11 Additional Dependences Eliminated Using Breaking Conditions.

totals and averages are for all direction vectors, not merely all the direction vectors possible in the cases where breaking conditions could be applied.

4.4 Related Work

4.4.1 Symbolic Analysis

A variety of methods for doing symbolic analysis have been proposed in the literature. Reif [66] uses the control flow graph to construct a mapping from program expressions to symbolic expressions for their value holding over all executions of the program. This *cover* is weaker than others that have been done, but the algorithm is much cheaper. Its cost is $O(l + a\alpha(a))$ bit vector operations, where l is the length of the program, a is the number of edges in the control flow graph, and α is Tarjan's function, which is almost linear.

Reif and Lewis [67] use a structure called the *global value graph* to compactly represent symbolic values and the flow of those values through the program. Their algorithm for symbolic evaluation discovers simple constants and minimal fixed point covers and is almost linear in the size of the global value graph. Like [66], this technique deals only with variables that have the same symbolic value along all paths in the the program's control flow graph. The global value graph is of size $O((\sigma a + l))$, where σ is the number of variables in the program, a is the number of edges in flow graph, and l is the length of the program.

Alpern, Wegman, and Zadeck [5] try to determine equality relations between program variables in the presence of control flow. Their approach uses value numbering and builds a *value graph* that represents the symbolic execution of the program. Determining which nodes in the value graph are congruent is an $O(E\log E)$ problem, where E is the number of edges in the *value graph*. Each assignment in the static single-assignment form of the program is a node in the value graph, which has two types of nodes: *executable function* nodes, which represent normal assignments, and ϕ *function* nodes, which represent join points in the control flow graph.

Cousot and Halbwachs [23] determine inequality as well as equality relations, using abstract interpretation. Their approach converts linear constraints into a convex polyhedron and tries to deduce assertions about the relationships between program variables from the semantics of the program. This method is intended to be applied to tasks such as array bound checking; precision will be lost when the numbers are reals instead of integers. The cost of their analysis is almost linear in the length of the program but exponential in the number of variables involved in the analysis.

Haghighat and Polychronopoulos [32] propose a framework for doing dependence analysis in the presence of symbolic terms, using abstract interpretation based on a lattice model. Essentially, they handle the simple symbolic cases that PFC takes care of: symbolic loop bounds, symbolic additive terms, and symbolic multipliers [1]. The lattice is used to do constraint propagation and inequalities are solved using the convex hull method of Cousot and Halbwachs.

Lichnewsky and Thomasset [54] evaluate constraints symbolically when it cannot be done numerically, using approximations to reduce the computational cost. A general decision procedure is employed only after the Banerjee-GCD test [9, 4, 10] has failed. The decision procedure uses the SUP-INF method for proving Presburger formulas, as described in Shostak [73]. The basic approach is to transform the integer problem into the real domain, solve it, and translate the results back into the integer domain. The complexity of this algorithm is $O(2^N)$, where N is the length of the formula.

Havlak [33] uses array section analysis, a special SSA form, and global value numbering as integral components of his analysis. His approach combines value numbering with graph rewriting and is particularly adept at representing and matching patterns of operators, unlike [32], whose strength lies in rewriting expressions and solving recurrences.

Blume and Eigenmann [12] describe a new dependence test, known as the Range Test, which proves independence by determining whether symbolic inequalities hold for a permutation of the loop nest. The Range Test is slower than Banerjee's Inequalities but faster on average than the Omega Test, although the Omega Test can break cross-iteration dependences in some cases where the Range Test cannot.

Some research has eschewed the more traditional memory-based notion of data dependence in favor of a value-based approach. Instead of testing every pair of array references which may access the same memory location, a value-based approach eliminates pairs separated by an intervening write, thus eliminating spurious dependences from consideration. Collard, Barthou, and Feautrier [19] present a method which gives exact information in some cases and which can handle nonlinear terms in loop bounds and IF conditions but not in array subscripts. Maslov [56] exactly computes value-based dependence relations for program fragments where all subscripts, loop bounds, and IF conditions are affine. The algorithm, which is lazy, also handles some nonlinear terms. Pugh and Wonnacott [64] handle more nonlinear constraints than Maslov's algorithm and their work also has the ability to relate terms in the dependence analyzer back to program expressions which can be more readily understood by the programmer.

Finally, Shen, Li, and Yew [72] present the results of an empirical study on array subscripts and data dependences. This study concludes that many of the subscripts contained symbolic terms with unknown values. An average of 33.97% of all subscripts were nonlinear, and of these, 85% to 100% of the nonlinear or partially linear subscripts were caused by an unknown variable. User assertions pertaining to the value of some symbolic variables were employed in an effort to reduce the number of nonlinear subscripts for six programs. Using user assertions, only 27.38% of one-dimensional array references and 14.7% of two-dimensional array references were nonlinear, as opposed to 47.44% of one-dimensional array references and 44.91% of two-dimensional array references when assertions were not used.

4.4.2 Run-time Parallelization

Polychronopoulos proposes a technique called *run-time dependence checking* (RDC) [60, 61, 62] that uses program information available at compile-time to generate code that resolves dependences at run-time. The goal of RDC is to determine at run-time whether a particular iteration of a loop depends on one or more previous iterations, and synchronize the loop iterations involved in dependences. A *dependence source vector* is created for each true dependence in the loop. Non-zero elements in the dependence source vector indicate the iterations of the loop that may be involved in a dependence. For each potential dependence source, a statement is inserted at the beginning of the target loop that causes the program to wait on the *synchronization vector* until the statements corresponding to the dependence sources have executed. Iterations which cannot be involved in a dependence are executed in parallel without any constraints. He gives no time complexity for the algorithm used to construct the dependence source vectors, but estimates that the storage requirement for the dependence source vectors is $O(kN)$, where k is the number of dependences and N is the number of loop iterations.

Saltz *et al* [70] describe a compiler that transforms a loop into two separate code segments, an *inspector* and an *executor*. The inspector reorders loop iterations into concurrent wavefronts, then the executor carries out the scheduled work. This method applies only to loop nests in which inter-iteration dependences do not depend on the results of computations carried out in the loop nest. If the run-time parallelization is being done for a distributed memory architecture, then the inspector phase will also determine the location of distributed array elements and calculate which communication calls will be needed to retrieve the data. Information about communication patterns is saved between executions in an effort to amortize the cost of the inspector.

Rauchwerger and Padua [65] present their Privatizing DOALL test, which identifies fully parallel loops at run-time and dynamically privatizes scalars and arrays. The test can be applied to any loop, regardless of control flow. The test can either be performed before the loop is executed, determining whether the loop can be executed as a DOALL, or it can be done while the loop is being speculatively executed as a DOALL. They conclude that the test should be applied in almost all cases because the cost of encountering a less-than-fully-parallel loop is minimal compared to the expected speedup.

Leung and Zahorjan [51] present an inspector-executor method with two possible approaches. The first, which targets inspector efficiency, achieves nearly linear speedup relative to a sequential execution of the inspector, but may produce a suboptimal schedule for the executor. The second approach emphasizes executor efficiency and is guaranteed to produce the best schedule, at the cost of a potentially slower inspector loop.

4.4.3 On-the-fly Anomaly Detection

Access anomalies, also referred to as *data races*, occur when one thread of a parallel program modifies a shared variable concurrently being accessed by another thread. Dinning and Schonberg [27] present a debugging algorithm known as *task recycling*, for detecting access anomalies at run-time. The expected space complexity is $O(V)$, but the worst-case bound is $O(TV)$, where T is the maximum number of threads in the program and V is the number of variables monitored. This *on-the-fly* approach detects anomalies by monitoring program execution. When a variable is accessed during execution, a check is made to see if the current access conflicts with a previous access. If a conflict does occur, then an error is reported. Their empirical estimate of the cost of monitoring is a 3-fold to 6-fold slowdown if every shared variable reference is monitored. This cost can be reduced by using static analysis or user assertions to limit the number of variables that are monitored.

Hood, Kennedy, and Mellor-Crummey [35] also use an on-the-fly detection method, but their algorithm is more efficient than the one presented in [27]. The overhead attributed to monitoring program execution is estimated to be less than 40%, and a single instrumented execution will either identify sources of schedule-dependent behavior or verify that all executions of the program on the same set of data compute the same result. Their approach uses dependence analysis to reduce the amount of run-time checking, and they also reduce the cost of detection by using a method that works for programs with no nested parallel constructs using only the structured synchronization primitives in a disciplined way.

4.5 Conclusions

Breaking conditions are run-time checks used to guard blocks of optimized code. Generating breaking conditions should be significantly less expensive than building the dependence source vectors used for run-time dependence checking [60, 61, 62]. While it is true that RDC will execute individual loop iterations in parallel if those iterations are not involved in any dependences and our method cannot, it is not clear how often such loops occur. The speedup obtained by executing part of a loop's iterations in parallel may not be enough to offset the cost of setting up the dependence source vectors. Similarly, we feel that the cost of an inspector-executor approach can be somewhat intimidating and that our approach, although possibly less effective in some cases, could be inexpensive enough and effective enough to merit its use.

Breaking conditions, when satisfied, indicate that potential data dependences do not exist. *Proving conditions* have the opposite meaning. The use of proving conditions could improve the efficiency of on-the-fly anomaly detection algorithms such as [27] and [35] by reducing the number of loops that must be annotated, as there would be no need to perform anomaly detection for loops whose proving conditions were true. If a proving condition contained variables that were not loop-invariant, then the condition could not be tested at the top of the loop. In such a case, the proving condition or its corresponding breaking condition would be tested immediately prior to the execution of the statement it pertained to.

How much do breaking conditions cost? That question cannot be answered without generating and running the optimized code, something we were unable to do. The bulk of the cost for breaking conditions would be incurred in the code generator, which has to generate code twice for every piblock in the program. At run-time, there would be some additional cost associated with testing the breaking conditions, but the larger cost could be in terms of space: the program could be up to twice the length of the same program without breaking conditions, which could be problematic if one were running a very large program on a limited machine. If there are no severe space limitations, however, calculating breaking conditions should be worth their cost, as 23% of all dependences eliminated in this experiment could be attributed to the use of breaking conditions.

Although this experiment shows the most improvement one could expect from the use of breaking conditions, it is worth remembering that 32% of all breaking conditions in this experiment were trivial. 80% of these trivial conditions were true, thus at least 25% of all breaking conditions generated in our experiment would be satisfied at run-time. In contrast, only 6.5% of all breaking conditions would definitely be false. These numbers show that we can derive some benefit from the *concept* of breaking conditions even without modifying our code generator or implementing run-time tests, simply by eliminating trivial breaking conditions during dependence analysis.

Chapter 5

User Assertions for Index Array References

Index arrays are sometimes used to simulate pointers in Fortran programs. This chapter discusses the use of assertions about index arrays to eliminate additional dependences. Section 5.1 describes the problem and suggests a few assertions, while Section 5.2 shows how the assertions could be implemented and utilized. Section 5.3 presents the results of an experiment using such assertions, Section 5.4 discusses some related work, and we conclude in Section 5.5.

5.1 Introduction

Consider the following loop.

```
DO I = 1, N
   A(INDEX(I)) =                     (S1)
         = A(INDEX(I))               (S2)
ENDDO
```

In the absence of any information about INDEX(I), loop-carried true, anti, and output dependences must be assumed between statements S1 and S2, because the values of INDEX(I) might overlap. This loop could be executed in parallel if run-time testing were used to verify that the values of INDEX(I) do not overlap. The overhead associated with such run-time testing might be avoided if we knew that INDEX(I) $\neq$ INDEX(J) for all I $\neq$ J. In that case, we could eliminate all loop-carried dependences on A between S1 and S2, and the loop could be run in parallel if there were no other loop-carried dependences on A.

We propose the following set of assertions, suggested in [57], which could be inserted into the source code by the user as directives. These assertions refer to orthogonal properties that an index array might possess.

1) *unique* -- `INDEX(I)` $\neq$ `INDEX(J) for all I` $\neq$ `J`
2) *monotone non-decreasing* -- `INDEX(I)` $\leq$ `INDEX(I+1)`
3) *monotone non-increasing* -- `INDEX(I)` $\geq$ `INDEX(I+1)`

Some of these properties can be combined to derive other meaningful properties:

```
1 and 2:  monotone increasing
1 and 3:  monotone decreasing
2 and 3:  INDEX(I) = INDEX(J) for all I, J
```

The following example illustrates the usefulness of index array assertions.

```
DO I = 1, N
   A(INDEX(I)) =                     (S1)
         = A(INDEX(I)+δ)             (S2)
ENDDO
```

Assume $\delta \neq 0$ and let I_1 and I_2 be values of I on two different iterations of the loop such that $I_1 < I_2$. Without assertions, the following dependences must be assumed.

edge	*type*	*dependence exists when*	
S1→S2	*true*	$\mathrm{INDEX}(I_1) = \mathrm{INDEX}(I_2) + \delta$	for $I_1 \leq I_2$
S2→S1	*anti*	$\mathrm{INDEX}(I_2) = \mathrm{INDEX}(I_1) + \delta$	for $I_1 \leq I_2$
S1→S1	*output*	$\mathrm{INDEX}(I_1) = \mathrm{INDEX}(I_2)$	for $I_1 \neq I_2$

Some of the dependences can be deleted when certain assertions are made.

1. Unique – $\mathrm{INDEX(I)} \neq \mathrm{INDEX(J)}$ for all $\mathrm{I} \neq \mathrm{J}$

   The output dependence can be deleted because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

2. Monotone Non-decreasing – $\mathrm{INDEX(I)} \leq \mathrm{INDEX(I+1)}$

   For $\delta > 0$:
   True dependence can be deleted because $\mathrm{INDEX}(I_1) < \mathrm{INDEX}(I_2) + \delta$.

   For $\delta < 0$:
   Anti dependence can be deleted because $\mathrm{INDEX}(I_2) > \mathrm{INDEX}(I_1) + \delta$.

3. Monotone Non-increasing – $\mathrm{INDEX(I)} \geq \mathrm{INDEX(I+1)}$

   For $\delta > 0$:
   Anti dependence can be deleted because $\mathrm{INDEX}(I_1) > \mathrm{INDEX}(I_2) + \delta$.

   For $\delta < 0$:
   True dependence can be deleted because $\mathrm{INDEX}(I_2) < \mathrm{INDEX}(I_1) + \delta$.

5.2 Implementing and Using Index Array Assertions

There are three basic tasks involved in handling directives which assert index array properties. First, the parser/scanner must be able to recognize directives and annotate the symbol table with that information. For example, the directive

`Cindexed INX unique`

will be treated as any other comment unless PFC's "accept index array directives" flag is set to true. When that flag is true, fields in the symbol table entry for the variable INX will be altered to indicate that INX is used as an index array and that its values are unique.

The second task which PFC must accomplish is the recognition of indirection in array references during the dependence analysis phase. During dependence analysis, subscript information is constructed for each reference before testing is done. One part of this information is the subscript reference's *type*. PFC currently recognizes three types of indirect references:

1) *indexed* `--- A(INX(x1) + x2)`
2) *symbolic_indexed* `- -- A(INX(x1) + s1)`
3) *indexed_symbolic_additive* `--- A(INX(s1) + c))`
`or A(INX(s1) + s2)`

where `A` is an array,
`INX` is an index array,
`x1` and `x2` are either constants or induction variables,
`c` is a constant, and
`s1` and `s2` are symbolic terms.

The third aspect of handling index array assertions is the actual use of these assertions during dependence testing. These assertions are needed only when comparing two indirect references which use the same index array. For example, if the comparison is between

A(INDEX1(I)) and A(J)

or between

A(INDEX1(I)) and A(INDEX2(I)),

the assertions cannot help us, but if we are comparing

A(INDEX(I)) and A(INDEX(J)),

then we should consult any information we have about the properties of INDEX. In this case, we may be able to eliminate direction vectors. If we have no information about the index array or are unable to use that information because we are not comparing two references using the same index array, all possible direction vectors must be assumed.

5.2.1 Algorithm for Comparing Indirect References

Our algorithm follows the basic ideas set forth in section 5.1. Assume we are comparing two references, A(INX(x1) + y1) and A(INX(x2) + y2), where A and INX are arrays, and x1, x2, y1, and y2 are a combination of symbolic terms, induction variables, and constants.

1. If x2 - x1 = 0, then let delta = y2 - y1. If delta $\neq$ 0:
    (a) If INX is "unique", there is no output dependence.
    (b) If INX is "monotone-nondecreasing":
        i. If delta > 0, there is no true dependence.
        ii. If delta < 0, there is no anti dependence.
    (c) If INX is "monotone-nonincreasing":
        i. If delta > 0, there is no anti dependence.
        ii. If delta < 0, there is no true dependence.
2. If x2 - x1 $\neq$ 0 and y2 - y1 = 0:
    (a) If x2 - x1 is a constant, let delta = x2 - x1.
        i. If INX is "unique", there is no output dependence.
        ii. If INX is "monotone-nondecreasing":
            A. If delta > 0, there is no true dependence.
            B. If delta < 0, there is no anti dependence.
        iii. If INX is "monotone-nonincreasing":
            A. If delta > 0, there is no anti dependence.
            B. If delta < 0, there is no true dependence.

5.2.2 Handling Index Arrays Passed as Parameters

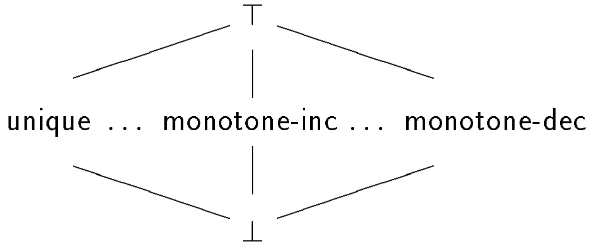
We encountered no programs where index arrays were passed as parameters to subroutines, but it is possible such programs exist. In that case, it would be useful if the properties of the index array in the callee could be deduced from the properties of the corresponding index array in the caller.

To accomplish this, information derived from index array assertions would be propagated during interprocedural analysis. A bottom-up tree walk of the call graph would be used to disseminate information about index array properties. The properties a callee "inherits" from its callers, once determined during interprocedural analysis, would be referred to in the normal way during dependence testing. Consider the following program fragment:

```
PROGRAM
      ...
Cindexed INX monotone-inc
      ...
CALL FOO(INX, ...)
      ...
SUBROUTINE FOO(JX, ...)
      ...
T(JX(I)) = T(JX(I) + 1)
      ...
```

If *foo* has only one caller, deriving the property of JX during interprocedural analysis will have the same effect as if the user had inserted a directive for JX. What happens when *foo* has more than one caller?

We could use a shallow lattice to represent possibile properties:



where ⊤ corresponds to "no properties known thus far", ⊥ is "conflicting properties or not an index array", and there is one broad middle level representing various properties. If an array has multiple properties, e.g., unique and monotone-decreasing, they must be declared at the same time; subroutines cannot inherit differing properties from different parent callers.

The meet function is defined as follows:

$$\begin{array}{ll} \top \bigcap \top = \top & \\ \top \bigcap \mathsf{property} = \mathsf{property} & \text{(lower to middle level)} \\ \mathsf{property1} \bigcap \mathsf{property2} = \bot & \text{(where } \mathit{property}_1 \neq \mathit{property}_2\text{)} \\ \bot \bigcap \mathsf{anything} = \bot & \end{array}$$

This lattice will be used when traversing the call graph during interprocedural analysis. The answer to "what happens to JX when *foo* has more than one caller?" depends on the properties of the index arrays in *foo's* callers. If the properties are harmonious, then JX will inherit some property from the middle level of the lattice. If not, the meet function will return bottom and we will have no information about JX's properties that can be utilized during dependence testing.

5.3 Experimental Results

Most of the programs used for this experiment were drawn from the Perfect or SPEC benchmarks. We encountered few test programs using index arrays to perform indirection, and the programs which did employ them tended to do so sparingly. *spice*, for example, only contains two explicitly indirect references. Most of its indirection is accomplished either by using a pointer as a base and adding offsets to it, or by using an intermediate assignment which obscures the indirect nature of the reference. For example, instead of

```
VALUE(LVN + NODPLC(ICSWPF + 1))
```

spice first performs the assignment

```
IORD = NODPLC(ICSWPF + 1)
```

and then says

```
VALUE(LVN + IORD)
```

Table 5.1 gives some basic information for the programs in our test suite: the number of lines and subroutines for each program are listed, together with a brief description of the program's application. These programs were executed twice, once with index array assertions and once without. First, we will present the results obtained without using assertions.

program	type	lines	subrs
bdna	Molecular Dynamics of DNA	3980	43
qcd	Quantum Chromodynamics	2327	35
sds	Structural Dynamics	7610	78
wave	Particle Transport	7520	92
mary	Microbial Biodegradation	3145	20
boast	Reservoir Simluation	8066	57
Total		32648	325
Average		5441	54

Table 5.1 Basic Program Characteristics.

Table 5.2 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. One-dimensional array references were the most common, accounting for 65.7% of all array references. 30.7% of the references were two-dimensional, 3.5% were three-dimensional, and only 0.07% were four-dimensional. None of the array references had more than four dimensions.

98% of the subscript pairs were separable; the remaining pairs were in coupled groups of size two. All of the programs had at least some coupled subscripts, unlike the test suite used in previous chapters.

PFC applied dependence tests 35793 times, accounting for 92% of all subscript pairs. Pairs were not tested if they were nonlinear (5.3%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Table 5.3 gives the number of indices for each subscript pair. 50.8% of all subscript pairs had zero indices, 46% had only one, 2.9% had two, 0.24% had three indices, and only 0.05% had four indices; none had more than four. Thus, 96.8% of all subscript pairs were either ZIV or SIV.

Tables 5.4 and 5.5 detail how often and how successfully each type of dependence test was employed. Table 5.4 is devoted to the various SIV tests while Table 5.5 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*. Note that the S and I columns are combined for the ZIV test because they are always identical. As in previous chapters, no results are presented for the triangular MIV test.

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
bdna	1787	269	0	0	2056	2085	240	2325	409
qcd	7804	271	11	0	8086	8369	10	8379	0
sds	268	866	7	7	1148	1897	152	2049	0
wave	1271	3765	692	0	5728	10725	152	10877	116
mary	1295	355	220	12	1882	2609	104	2713	292
boast	5941	3073	59	0	9073	12136	128	12264	1241
Total	18366	8599	989	19	27973	37821	786	38607	2058
Average	3061	1433	165	3	4662	6304	131	6434	343

Table 5.2 Dimensionality, Separability, and Linearity of Pairs Tested.

	subscript pair indices				
program	0	1	2	3	4
bdna	85	1807	418	0	0
qcd	8144	231	1	0	0
sds	1132	623	41	0	0
wave	4307	5851	38	0	0
mary	1257	886	399	1	0
boast	4112	7840	206	89	17
Total	19037	17238	1103	90	17
Average	3173	2873	184	15	3

Table 5.3 Number of Subscript Pair Indices.

	SIV											
	strong			wk-zero			wk-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
bdna	1415	1271	14	86	1	0	0	0	0	0	0	0
qcd	198	148	0	31	2	0	0	0	0	0	0	0
sds	531	524	0	92	0	0	0	0	0	0	0	0
wave	5118	4897	336	249	67	8	154	0	0	0	0	0
mary	639	630	0	207	16	0	7	4	1	1	0	0
boast	6144	5045	240	442	17	0	6	0	0	302	8	0
Total	14045	12515	590	1107	103	8	167	4	1	303	8	0
Average	2341	2086	98	184	17	1	28	1	0	50	1	0

Table 5.4 SIV Application/Success/Independence Frequencies.

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
bdna	85	15	2	2	0	119	0	0	1	454	14
qcd	8144	6609	1	1	0	5	0	0	0	187	154
sds	1132	558	14	12	0	69	0	0	6	50	0
wave	4307	2962	18	18	0	76	0	0	0	3023	49
mary	1257	414	106	11	2	36	0	0	8	72	0
boast	4112	58	2	2	0	61	0	0	3	1090	240
Total	19037	10616	143	46	2	366	0	0	18	4876	457
Average	3173	1769	24	8	0	61	0	0	3	813	76

Table 5.5 Dependence Test Application/Success/Independence Frequencies.

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 5.5 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

The Delta test constraint intersection algorithm tested 366 coupled groups exactly (93%). Propagation of distance constraints was applied in 18 cases (4.6%), converting MIV subscripts into SIV form in all cases. The Delta test thus managed to test 384 coupled groups exactly (98%), using only constraint intersection and propagation of dependence distances.

Table 5.6 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 were successful in eliminating one or more direction vectors.

Over all array reference pairs tested, most subscript pairs were ZIV (53%) or strong SIV (39%). Few of the subscripts tested were MIV (1%), with 0.6% of all cases tested using the triangular MIV test. (All of the MIV numbers presented in Table 5.5 are for the normal MIV test.) The ZIV and strong SIV tests combined for

	ZIV	SIV				MIV	Delta	symbolics used
category		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	53.19	39.24	3.09	0.47	0.85	0.40	1.02	
% of all successful tests	45.54	53.69	0.44	0.02	0.03	0.20	0.00	20.92
% of all proven independences	94.64	5.26	0.07	0.01	0.00	0.02	0.00	4.07
% of apps that were successful	55.77	89.11	9.30	2.40	2.64	32.17	0.00	
% of apps that proved independence	55.77	4.20	0.72	0.60	0.00	1.40	0.00	
averaged over all programs								
% of all tests applied	49.42	39.11	4.18	0.31	0.46	0.95	2.09	
% of all successful tests	37.72	61.08	0.46	0.06	0.42	0.00	18.07	18.07
% of all proven independences	76.68	23.16	0.04	0.04	0.00	0.08	0.00	22.10
% of apps that were successful	41.87	89.94	7.68	19.05	1.32	82.68	0.00	
% of apps that proved independence	41.87	1.91	0.54	4.76	0.00	0.31	0.00	

Table 5.6 Comparison of Dependence Tests.

most of the successful tests (92%), and the ZIV test accounted for most of the reference pairs proven independent (95%).

Table 5.7 shows the decrease in the number of dependence tests when index array assertions were used. Only a very small change is noticeable, because the assertions were used only in a few cases. The only change is in the relative success of each test, which has been skewed by the additional 271 "successes" obtained using index array assertions. The numbers of test applications/successes/independences is unaltered from those figures shown in Tables 5.4 and 5.5, because the indexed tests are counted separately, as either Indexed_ZIV, Indexed_SIV, or Indexed_MIV. All of the indexed references tested in this experiment were classified as Indexed_SIV.

Table 5.8 lists the number of dependences eliminated both with and without the use of index array assertions. The number of dependences eliminated is relatively small, but the assertions are not being tested that often and the assertion used for this experiment will only eliminate one dependence for each success.

Table 5.9 summarizes the outcome of the cases where we tried to use index array assertions. There are many ways in which an indexed test can fail, so we have listed them separately. First, one of the references might not be indexed. Second, both references might be indexed, but not by the same array. Third, the references might be indexed by the same array, but in different ways. For example, a reference categorized as symbolic_indexed would be incompatible with an indexed_symbolic_additive reference. Finally, even if all these preliminary checks are passed and we actually compare the references, the test may fail to eliminate any dependences.

Indexed_SIV subscripts were considered in 507 cases, representing 1.4% of all tests applied. 12% of these failed because one reference was not indexed, 5% failed because the references were indexed incompatibly, 27% failed during the actual application of the test, and 56% succeeded. 38% of the failures occurred during preliminary checks. A comparison was actually performed in 83% of the cases, and 67% of those applications were successful.

	ZIV	SIV				MIV	Delta	symbolics
category		strong	wk-zero	wk-cross	other			used
summed over all programs								
% of all tests applied	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
% of all successful tests	-0.52	-0.62	-0.01	0.00	0.00	0.00	0.00	-0.24
% of all proven independences	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
% of apps that were successful	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
% of apps that proved independence	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
averaged over all programs								
% of all tests applied	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
% of all successful tests	-0.19	-0.72	0.00	0.00	0.00	0.00	0.00	-0.33
% of all proven independences	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
% of apps that were successful	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
% of apps that proved independence	0.00	0.00	0.00	0.00	0.00	0.00	0.00	

Table 5.7 Decrease in Dependence Tests
When Using Index Array Assertions.

	w/o assertions		with assertions			
program	dvs elim	dvs left	dvs elim	dvs left	diff in dvs elim	max dvs
bdna	6332	5832	6364	5800	32	15060
qcd	101273	6126	101273	6126	0	207258
sds	5666	1718	5666	1718	0	8082
wave	24575	6006	24809	5772	234	31698
mary	19392	10503	19446	10492	54	51240
boast	37530	52623	37549	52618	19	126597
Total	194768	82808	195107	82526	339	439935
Average	32461	13801	32518	13754	56	173322

Table 5.8 Dependences Eliminated With and Without Index Array Assertions.

program	ref not indexed	different arrays	different inx types	tested, failed	tested, succeeded	total
bdna	0	0	20	96	32	148
qcd	2	0	0	0	0	2
sds	24	0	0	0	0	24
wave	0	0	0	0	234	234
mary	34	0	0	0	7	41
boast	0	0	6	43	9	58
Total	60	0	26	139	282	507
Average	10	0	4	23	47	84

Table 5.9 Categorization of Tests Involving Index Array Assertions.

5.4 Related Work

There have been a number of approaches to the general problem of pointer analysis [49, 36, 17, 48, 26, 37, 77, 30, 69]. Here, we restrict ourselves to a single aspect of the problem: indirection expressed using an index array, where the values of entries in the index array may possess special properties.

Polychronopoulos [60, 61] presents a technique called *subscript blocking* to deal with *subscripted subscripts*, which are references of the form A(*f*(I)), where *f* is a vector. He states that loops with subscripted subscripts are frequently found in numerical programs that manipulate sparse matrices. Blocking may be done at compile-time or at run-time, which is more expensive. His algorithm constructs *source domains*, which are groups of successive iterations that do not contain a dependence sink. All the iterations in source domains may be run in parallel.

5.5 Conclusions

User assertions can provide information about the values of variables that might not be obtainable through dependence analysis. We have presented three user assertions which will be used to describe the properties of index arrays. These assertions should increase the possibility that parallelism will be discovered, if it exists, by improving the quality of the dependence analysis performed. Like breaking conditions, employing user assertions should entail a much smaller overhead than that associated with run-time dependence checking.

In this experiment, we used an assertion which would eliminate one dependence for each success. Only a small amount of overall improvement was observed because there were a relatively small number of indexed statements in our test programs. However, 56% of all indexed references encountered were tested successfully.

Chapter 6

Partial Evaluation Using Input Data

This chapter investigates the use of compiling a program with its input, or *partial evaluation*, to reduce the number of symbolic terms in a program. The concept of partial evaluation is briefly covered in Section 6.1, and our implementation is described in Section 6.2. Experimental results are presented in Section 6.3, related work is presented in Section 6.4, and we conclude in Section 6.5.

6.1 Introduction

Perhaps the easiest way to ascertain the value of a symbolic constant is simply to read its initial value from the program's input file. Consider the following program fragment:

```
...
READ XMAX
...
MAXN = XMAX * 10
DO I = 1 TO MAXN
...
```

If we read the input file and discovered that XMAX is 20, we could then deduce that MAXN is 200.

Partial evaluation transforms a program by making use of information about the program's environment. When partial evaluation is used, a program must be recompiled when its input file is altered, even if the program itself has not changed. In a compilation system such as ParaScope [15], recompilation is considered expensive and avoided whenever possible. If partial evaluation were used in such a compilation system, reading in a different value for a variable would be considered an edit of that variable, and recompilation analysis [22] would be used to reduce the amount of recompilation.

6.2 Implementing and Using Partial Evaluation

Our method for carrying out the partial evaluation is straightforward. The program is interpreted, and an assignment is generated for each value read from the data file. This assignment will be inserted into the abstract syntax tree PFC uses to represent the program at the point between the read statement and the statement following the read in the original program. The values obtained for symbolic constants as a result of this interpretation will be propagated throughout the original program before dependence analysis takes place. Thus, partial evaluation should improve the performance of the dependence analysis phase by reducing the number of unknown symbolic constants it has to contend with. For example:

```
...
READ N, M
DO I = 1 TO N
      A(M, I) = ...
...
```

If the values for N and M were 100 and 50, respectively, the post-interpreter version of this program would read:

```
...
READ N, M
N = 100
M = 50
DO I = 1 TO N
      A(M, I) = ...
...
```

and after constant propagation, the same fragment would read

```
...
READ N, M
DO I = 1 TO 100
      A(50, I) = ...
...
```

In effect, the READ statement, which is unintelligible to our constant propagator, has been functionally (not literally) *replaced* by two simple assignment statements which can be capitalized on. Not every READ statement can be replaced, however. If the reads occur in linear code, i.e.,

```
...
READ X, Y
...
READ X
...
```

then we can replace them with impunity, as the constant propagator will have no difficulty catching the second definition of X. If, however, the read occurs in a loop, such as

```
...
DO I = 1 TO N
      READ X
      A(I) = X + I - 1
...
```

then we should not replace the READ with multiple assignments inside the loop. Even if we took the trouble to generate several boolean statements of the form

$$\mathsf{IF\ I} = \mathsf{K\ THEN\ X} = (k^{th}\ \textit{value}),$$

the constant propagator assumes that any scalar modified in a loop is non-constant. The most we can do is place an assignment corresponding to the last value read for X immediately after the loop.

We should also refrain from replacing *READ* statements for arrays, as the constant propagator only handles global scalars. Our basic algorithm is as follows:

1. Parse the program.
2. Do interprocedural summary analysis.
3. Perform inline expansion if desired. (We chose to expand only the main routine of the program.)
4. Interpret the program. Insert assignment statements for scalar reads outside of loops.
5. Do interprocedural constant propagation.
6. Analyze resulting program as normal.

Although we interpret the program after interprocedural summary analysis has been done, we need not recalculate the summary information before doing constant propagation. Any assignment statements we add to the program will kill definitions that are already being annihilated by read statements, so the MOD set will not be altered by our insertions.

Evaluating a Reduced Program

One optimization that was not performed due to time constraints was the use of dead-code elimination to trim a program to its bare essentials before interpretation is done. That could be done as follows:

1. Mark all of the program's READ statements as live.
2. Do dead-code elimination to remove all statements except the reads and the statements the reads depend upon.
3. Interpret the subprogram obtained using dead-code elimination. Instead of inserting assignments in the subprogram, add them to the original program.
4. Perform constant propagation and analyze the original program.

6.3 Experimental Results

Some of the test programs for this experiment were taken from the test suite used in Chapter 2. The programs tend to read most of their variables at the beginning of the program and then perform extensive computation.

The interpreter/interprocedural constant propagator/inline expander combination performed adequately for "toy" programs, but proved troublesome for "real" programs, so the results listed here were obtained by doing the insertions by hand. Thus, we have a good measure of how well our naive algorithm worked, but we do not know how much it would cost to do automatically.

Table 6.1 gives some basic information for the programs in our test suite: the number of lines and subroutines for each program are listed, together with a brief description of the program's application. These programs were executed twice, once with partial evaluation and once without. We will present the baseline results obtained without partial evaluation first, followed by a discussion of what happened when partial evaluation was used. After describing all of these tables in detail, we will present tables comparing the results of both executions.

program	type	lines	subrs
appsp	Nonlinear PDE Solver	4405	26
aps	Pseudospectral Air Pollution	6105	97
arc3d	3D Euler/Navier-Stokes	9815	84
bdna	Molecular Dynamics of DNA	3980	43
doduc	Thermohydraulical Modelization	5334	41
fpppp	2 Electron Integral Derivative	2718	38
hydro	2D Hydrodynamics	13057	36
sphot	Particle Transport	1144	7
track	Missile Tracking	3735	34
tuba	2D Turning Bands	479	3
Total		50772	409
Average		5077	41

Table 6.1 Basic Program Characteristics.

6.3.1 Results Without Partial Evaluation

Table 6.2 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. One-dimensional array references were the most common, accounting for 72.8% of all array references. 14.4% of the references were two-dimensional, 6.6% were three-dimensional, and 6.3% were four-dimensional. None of the array references had more than four dimensions.

98% of the subscript pairs were separable; the remaining pairs were in coupled groups of size two. 76% of the coupled subscripts were concentrated in one program, *arc3d*.

PFC applied dependence tests 61612 times, accounting for 94% of all subscript pairs. Pairs were not tested if they were nonlinear (1%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Table 6.3 gives the number of indices for each subscript pair. 68.8% of all subscript pairs had zero indices, 29.8% had only one, 1.4% had two, and only 0.01% had three indices; only one case (0.002%) had four. Thus, 98.6% of all subscript pairs were either ZIV or SIV.

Tables 6.4 and 6.5 detail how often and how successfully each type of dependence test was employed. Table 6.4 is devoted to the various SIV tests while Table 6.5 describes the other tests. For each test, these tables show the number of times a test

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
appsp	26464	215	0	0	26679	26840	54	26894	7
aps	642	160	521	0	1323	2501	24	2525	19
arc3d	870	3845	2422	2778	9915	25768	1170	26938	139
bdna	1787	269	0	0	2056	2085	240	2325	409
doduc	657	885	0	0	1542	2427	0	2427	11
fpppp	1023	108	0	0	1131	1239	0	1239	0
hydro	172	642	0	0	814	1456	0	1456	0
sphot	282	77	0	0	359	436	0	436	21
track	250	115	0	0	365	472	8	480	2
tuba	191	61	0	0	252	265	48	313	9
Total	32338	6377	2943	2778	44436	63489	1544	65033	617
Average	3234	638	294	278	4444	6349	154	6503	62

Table 6.2 Dimensionality, Separability, and Linearity of Pairs Tested.

	subscript pair indices				
program	0	1	2	3	4
appsp	26636	223	8	0	0
aps	632	1517	24	8	0
arc3d	12725	10869	330	0	0
bdna	85	1807	418	0	0
doduc	795	1632	0	0	0
fpppp	902	258	18	0	1
hydro	32	1420	4	0	0
sphot	280	135	21	0	0
track	242	189	7	0	0
tuba	17	271	25	0	0
Total	42346	18321	855	8	1
Average	4235	1832	86	1	0

Table 6.3 Number of Subscript Pair Indices.

	SIV											
	strong			weak-zero			weak-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
appsp	92	92	0	124	10	0	0	0	0	0	0	0
aps	1351	1345	154	17	1	0	97	0	0	52	52	0
arc3d	9605	9259	0	1205	322	25	56	0	0	0	0	0
bdna	1415	1271	14	86	1	0	0	0	0	0	0	0
doduc	1619	1608	0	2	2	0	0	0	0	0	0	0
fpppp	254	254	118	3	2	0	1	0	0	0	0	0
hydro	1420	1416	0	0	0	0	0	0	0	0	0	0
sphot	134	134	0	1	0	0	0	0	0	0	0	0
track	183	183	1	5	0	0	0	0	0	0	0	0
tuba	191	187	4	72	0	0	0	0	0	0	0	0
Total	16264	15749	291	1515	338	25	154	0	0	52	52	0
Average	1626	1575	29	152	34	2	15	0	0	5	5	0

Table 6.4 SIV Application/Success/Independence Frequencies.

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
appsp	26636	25349	0	0	0	27	0	0	0	0	0
aps	632	164	13	13	12	0	0	0	0	139	21
arc3d	12725	5605	194	120	0	548	0	0	0	3382	915
bdna	85	15	2	2	0	119	0	0	1	454	14
doduc	795	314	0	0	0	0	0	0	0	58	0
fpppp	902	10	1	1	0	0	0	0	0	6	0
hydro	32	0	4	4	0	0	0	0	0	243	0
sphot	280	185	0	0	0	0	0	0	0	11	0
track	242	77	6	5	0	4	0	0	0	15	0
tuba	17	0	24	0	0	0	0	0	0	7	4
Total	42346	31719	244	145	12	698	0	0	1	4315	954
Average	4235	3172	24	14	1	70	0	0	0	432	95

Table 6.5 Dependence Test Application/Success/Independence Frequencies.

was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*. Note that the S and I columns are combined for the ZIV test because they are always identical. As in previous chapters, no results are presented for the triangular MIV test.

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 6.5 show the number of times symbolic additive constants were manipulated in tests that (S) succeeded in eliminating direction vectors or (I) proved independence.

The Delta test constraint intersection algorithm tested 698 coupled groups exactly (90.4%). Propagation of distance constraints was applied in 1 case (0.9%), converting an MIV subscript into SIV form. The Delta test thus managed to test 699 coupled groups exactly (90.5%), using only constraint intersection and propagation of dependence distances.

Table 6.6 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 were successful in eliminating one or more direction vectors.

Over all array reference pairs tested, most subscript pairs were ZIV (69%) or strong SIV (26%). Few of the subscripts tested were MIV (0.7%), with 0.3% of all cases tested using the triangular MIV test. (All of the MIV numbers presented in Table 6.5 are for the normal MIV test only.) The ZIV and strong SIV tests combined for most of the successful tests (95%), while the ZIV test accounted for most of the reference pairs proven independent (99%).

Symbolic constants were present in 9% of all successful tests and in 10% of all cases where independence was proven. The ZIV and all SIV tests combined were responsible for 98% of all test applications. 13.7% of those tests involved symbolic terms. 89% of those symbolic tests were ZIV, 6.5% were strong SIV, 2.4% were weak-zero SIV, and 1.9% were weak-crossing SIV; none were general SIV.

	ZIV	SIV				MIV	Delta	symbolics
category		strong	wk-zero	wk-cross	other			used
summed over all programs								
% of all tests applied	68.73	26.40	2.46	0.25	0.08	0.40	1.13	
% of all successful tests	66.08	32.81	0.70	0.00	0.11	0.30	0.00	8.99
% of all proven independences	98.98	0.91	0.08	0.00	0.00	0.04	0.00	2.98
% of apps that were successful	74.90	96.83	22.31	0.00	100.00	59.43	0.00	
% of apps that proved independence	74.90	1.79	1.65	0.00	0.00	4.92	0.00	
averaged over all programs								
% of all tests applied	42.45	49.51	3.59	0.48	0.24	1.11	0.92	
% of all successful tests	25.49	73.43	0.31	0.00	0.33	0.43	0.00	10.13
% of all proven independences	67.50	32.05	0.05	0.00	0.00	0.40	0.00	18.99
% of apps that were successful	32.13	98.27	23.17	0.00	100.00	77.88	0.00	
% of apps that proved independence	32.13	6.15	0.23	0.00	0.00	13.19	0.00	

Table 6.6 Comparison of Dependence Tests.

6.3.2 Results With Partial Evaluation

Table 6.7 shows the number of array dimensions for each pair of array references tested, as well as the number of separable, coupled, and nonlinear subscript pairs found. Table 6.7 is quite similar to Table 6.2; only the number of nonlinear subscripts has changed.

PFC applied dependence tests 60961 times, accounting for 93% of all subscript pairs. Pairs were not tested if they were nonlinear (0.9%), or if tests on other subscripts in the same multidimensional array had already proven independence.

Table 6.8 gives the number of indices for each subscript pair, i.e., the number of indices which may need to be tested. 68.8% of all subscript pairs had zero indices, 29.8% had only one, 1.4% had two, and only 0.01% had three indices; only one case (0.002%) had four. These are the same percentages as were derived from Table 6.3, although there are slight differences between the two tables.

Tables 6.9 and 6.10 detail how often and how successfully each type of dependence test was employed. Table 6.9 is devoted to the various SIV tests while Table 6.10 describes the other tests. For each test, the tables show the number of times a test was (A) *applied*, (S) *successful* in eliminating at least one direction vector, and (I) able to prove *independence*.

The Delta test constraint intersection algorithm tested 678 coupled groups exactly (87.8%). Propagation of distance constraints was applied in 1 case (0.9%), converting MIV subscripts into SIV form. The Delta test thus managed to test 679 coupled groups exactly (87.9%), using only constraint intersection and propagation of dependence distances.

Table 6.11 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 were successful in eliminating one or more direction vectors.

Over all array reference pairs tested, most subscript pairs were ZIV (69%) or strong SIV (26%). Few of the subscripts tested were MIV (0.7%), with 0.3% of all cases tested using the triangular MIV test. The ZIV and strong SIV tests combined for most of the successful tests (99%), while the ZIV test accounted for most of the reference pairs proven independent (99%).

	array pairs tested					subscript pairs			nonlin
program	1D	2D	3D	4D	total	sep	coup	total	subs
appsp	26464	215	0	0	26679	26840	54	26894	7
aps	642	160	521	0	1323	2501	24	2525	19
arc3d	870	3845	2422	2778	9915	25768	1170	26938	139
bdna	1787	269	0	0	2056	2085	240	2325	409
doduc	657	885	0	0	1542	2427	0	2427	11
fpppp	1023	108	0	0	1131	1239	0	1239	0
hydro	172	642	0	0	814	1456	0	1456	0
sphot	282	77	0	0	359	436	0	436	0
track	250	115	0	0	365	472	8	480	2
tuba	191	61	0	0	252	265	48	313	8
Total	32338	6377	2943	2778	44436	63489	1544	65033	595
Average	3234	638	294	278	4444	6349	154	6503	60

Table 6.7 Dimensionality, Separability, and Linearity of Pairs Tested (With Partial Evaluation).

	subscript pair indices				
program	0	1	2	3	4
appsp	26636	223	8	0	0
aps	632	1517	24	8	0
arc3d	12263	10678	330	0	0
bdna	84	1805	421	0	0
doduc	795	1632	0	0	0
fpppp	902	258	18	0	1
hydro	32	1420	4	0	0
sphot	280	135	21	0	0
track	242	189	7	0	0
tuba	17	271	25	0	0
Total	41883	18128	858	8	1
Average	4188	1813	86	1	0

Table 6.8 Number of Subscript Pair Indices (With Partial Evaluation).

	SIV											
	strong			wk-zero			wk-cross			other		
program	A	S	I	A	S	I	A	S	I	A	S	I
appsp	92	92	0	124	10	0	0	0	0	0	0	0
aps	1351	1345	154	17	1	0	97	0	0	52	52	0
arc3d	9414	9076	1	1195	342	45	66	66	43	0	0	0
bdna	1413	1268	14	86	1	0	0	0	0	0	0	0
doduc	1619	1608	2	2	2	0	0	0	0	0	0	0
fpppp	254	254	118	3	2	0	1	0	0	0	0	0
hydro	1420	1416	0	0	0	0	0	0	0	0	0	0
sphot	134	134	0	1	1	0	0	0	0	0	0	0
track	183	183	1	5	0	0	0	0	0	0	0	0
tuba	191	187	15	72	0	0	0	0	0	0	0	0
Total	16071	15563	305	1505	359	45	164	66	43	52	52	0
Average	1607	1556	30	150	36	4	116	7	4	5	5	0

Table 6.9 SIV Application/Success/Independence Frequencies (With PE).

	ZIV		MIV			Delta				symbolics used	
program	A	S/I	A	S	I	A	S	I	P	S	I
appsp	26636	25349	0	0	0	27	0	0	0	0	0
aps	632	164	13	13	12	0	0	0	0	139	21
arc3d	12263	5621	194	120	0	528	0	0	0	2986	752
bdna	84	15	5	5	0	119	0	0	1	451	14
doduc	795	314	0	0	0	0	0	0	0	54	0
fpppp	902	10	1	1	0	0	0	0	0	5	0
hydro	32	0	4	4	0	0	0	0	0	243	0
sphot	280	185	21	21	0	0	0	0	0	11	0
track	242	77	6	5	0	4	0	0	0	15	0
tuba	17	0	25	25	0	0	0	0	0	1	0
Total	41883	31735	269	194	12	678	0	0	1	3905	787
Average	4188	3174	27	19	1	68	0	0	0	390	79

Table 6.10 Dependence Test Application/Success/Independence Frequencies (With Partial Evaluation).

	ZIV	SIV				MIV	Delta	symbolics used
category		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	68.70	26.36	2.47	0.27	0.09	0.44	1.11	
% of all successful tests	66.16	32.44	0.75	0.14	0.11	0.40	0.00	8.14
% of all proven independences	98.74	0.95	0.14	0.13	0.00	0.04	0.00	2.45
% of apps that were successful	75.77	96.84	23.85	40.24	100.00	72.12	0.00	
% of apps that proved independence	75.77	1.90	2.99	26.22	0.00	4.46	0.00	
averaged over all programs								
% of all tests applied	42.07	49.36	3.59	0.49	0.24	1.64	0.91	
% of all successful tests	25.15	71.87	0.36	0.04	0.33	2.25	0.00	9.45
% of all proven independences	67.31	32.12	0.09	0.08	0.00	0.40	0.00	7.53
% of apps that were successful	32.33	98.26	34.49	33.33	100.00	93.15	0.00	
% of apps that proved independence	32.33	6.74	0.42	21.72	0.00	11.54	0.00	

Table 6.11 Comparison of Dependence Tests (With Partial Evaluation).

Symbolic constants were present in 8% of all successful tests and in 9.5% of all cases where independence was proven. The ZIV and all SIV tests combined were responsible for 98% of all test applications. 12% of those tests involved symbolic terms. 90% of those symbolic tests were ZIV, 7.3% were strong SIV, 1.6% were weak-zero SIV, and 1.4% were weak-crossing SIV; none were general SIV.

6.3.3 Comparison of Results With and Without Partial Evaluation

Table 6.12 shows the difference between Table 6.3 and Table 6.8, representing the decrease in the number of subscript pair indices encountered when partial evaluation was used. *bdna* is somewhat unusual, in that the number of of pairs having two indices has increased by three while the number of pairs having zero or one indices has decreased by the same amount. The implication that we now are applying the MIV test where once the ZIV and SIV tests sufficed seems counterintuitive, as partial evaluation cannot be creating additional index variables out of thin air.

Table 6.13, which shows the change in the number of test applications, reveals that the number of MIV tests does increase when partial evaluation is used. A close inspection of the original statistics shows the increase can be attributed to three programs, *bdna*, *sphot*, and *tuba*. For *sphot* and *tuba* the increase in the number of normal MIV tests is exactly matched by a decrease in the number of nonlinear (MIV) cases, so the increase is readily understandable: partial evaluation has reduced a symbolic expression which formerly forced us to categorize a subscript as nonlinear.

This is not true, however, for *bdna*. The additional MIV tests for *bdna* arise as a side effect of PFC's normalization phase. Using partial evaluation, the upper bounds for some loops are discovered. When induction variable substitution takes place, a formerly symbolic expression in an array reference is replaced by an induction variable expression, e.g., `K` is replaced by `IVAR1*4`, and an expression like `A(K + IVAR2)` is replaced by `A(IVAR1*4 + IVAR2)`, where `K` is symbolic and `IVAR1` and `IVAR2` are index variables.

Table 6.14 shows the difference between Table 6.6 and Table 6.11. As indicated by Table 6.13, there were fewer applications of most dependence tests when partial evaluation was used, and fewer tests where symbolic terms played a role. This is as should be, as our goal was to assign constant values to symbolic terms. Dependence tests were more successful when partial evaluation was performed, with the exception

program	subscript pair indices 0	1	2	3	4
appsp	0	0	0	0	0
aps	0	0	0	0	0
arc3d	462	191	0	0	0
bdna	1	2	-3	0	0
doduc	0	0	0	0	0
fpppp	0	0	0	0	0
hydro	0	0	0	0	0
sphot	0	0	0	0	0
track	0	0	0	0	0
tuba	0	0	0	0	0
Total	463	193	-3	0	0
Average	46	19	0	0	0

Table 6.12 Decrease in Number of Subscript Pair Indices With Partial Evaluation.

Table 6.13 Decrease in Test Frequencies When Using Partial Evaluation

	ZIV		SIV												MIV			Delta				symbolics used	
			strong			wk-zero			wk-cross			other											
	A	S/I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	A	S	I	P	S	I
Total	-463	16	-193	-186	14	-10	21	20	10	66	43	0	0	0	25	49	0	-20	0	0	0	-410	-167
Average	-46	2	-19	-19	1	-1	2	2	1	7	4	0	0	0	2	5	0	-2	0	0	0	-41	-17

Table 6.13 Decrease in Test Frequencies When Using Partial Evaluation.

	ZIV	SIV				MIV	Delta	symbolics used
category		strong	wk-zero	wk-cross	other			
summed over all programs								
% of all tests applied	-0.03	-0.03	0.01	0.02	0.00	0.05	-0.02	
% of all successful tests	0.08	-0.36	0.04	0.14	0.00	0.10	0.00	-0.85
% of all proven independences	-0.24	0.04	0.06	0.13	0.00	0.00	0.00	-0.53
% of apps that were successful	0.87	0.01	1.54	40.24	0.00	12.69	0.00	
% of apps that proved independence	0.87	0.11	1.34	26.22	0.00	-0.46	0.00	
averaged over all programs								
% of all tests applied	-0.38	-0.15	0.00	0.00	0.00	0.53	0.00	
% of all successful tests	-0.34	-1.56	0.04	0.04	0.00	1.82	0.00	-0.68
% of all proven independences	-0.19	0.07	0.04	0.08	0.00	0.00	0.00	-11.45
% of apps that were successful	0.20	-0.01	11.32	33.33	0.00	15.26	0.00	
% of apps that proved independence	0.20	0.59	0.19	21.72	0.00	-1.65	0.00	

Table 6.14 Increase in Dependence Tests When Using Partial Evaluation.

of the strong SIV test, which declined very slightly. The weak-crossing SIV test enjoyed the most improvement, going from zero successful tests in our baseline experiment to a 40% success rate when partial evaluation took place.

Finally, Table 6.15 lists the number of direction vectors corresponding to dependences eliminated both with and without partial evaluation. The number of dependences eliminated seems small, especially when compared to the maximum number of direction vectors, yet even that amount was enough to dramatically improve the performance of the weak-crossing SIV test. The weak-zero SIV and MIV tests also improved significantly. There were no applications of the general SIV test, but reducing the number of symbolic terms would probably improve its performance also.

	w/o PE		with PE			
program	dvs elim	dvs left	dvs elim	dvs left	diff in dvs elim	max dvs
appsp	500091	5867	500091	5867	0	526239
aps	13196	2023	13196	2023	0	15873
arc3d	68736	14446	69024	14158	288	84591
bdna	6332	5832	6338	5829	6	15060
doduc	5650	1611	5652	1609	2	7530
fpppp3	1804	8726	1804	8726	0	124419
hydro	5704	984	5704	984	0	6714
sphot	7587	583	7646	578	59	11565
track	2396	794	2396	794	0	4245
tuba	5146	2345	5325	2216	179	10224
Total	616642	43211	617176	42784	534	806460
Average	61664	4321	61718	4278	53	180646

Table 6.15 Dependences Eliminated With and Without Partial Evaluation.

6.4 Related Work

Partial evaluation transforms a program by making use of information about the program's environment. Lombardi and Raphael [55] presented partial evaluation, sometimes referred to as currying, as an essential part of their *incremental computer*. This incremental computer evaluated expressions under the control of the available information context instead of executing predetermined commands. Since that time, partial evaluation has been used in a variety of applications, such as automatic theorem proving, pattern matching, syntax analysis, compiler generation, compiler generator generation, program transformation, the specialization of numerical Fortran programs, and even the specialization of specializers [28, 71, 75, 43, 68]. Recent work in dynamic compilation [20, 50, 7] owes some of its underlying principles to partial evaluation.

Berlin and Weise [11] use partial evaluation to convert a high-level program into a low-level, specialized one by taking advantage of information available at compile-time about the program's data structures. Essentially, their approach tries to eliminate procedure calls and data structure manipulations instead of optimizing them. They argue that this technique is particularly effective on scientific programs because such programs are *data independent*, i.e., the sequence of operations performed is not dependent on the results of the computation. At compile time, the program is run on data structures and placeholders are used to represent symbolic variables with unknown values. The actual value of variables is considered to be less essential than knowing their location within the data structures. The user must identify which loops to partially evaluate, and the size of the program can be a problem for this technique because loops are expanded when they are evaluated.

6.5 Conclusions

Some techniques for handling symbolic terms are efficient, but are effective only in simple cases. Other symbolic analysis methods are more effective , but can be far too expensive for use in a production compiler. Variables whose values are read from an input file at run-time pose a major obstacle for all of these methods. *Partial evaluation* can take advantage of information available at compile-time about programs' input data sets.

Berlin and Weise [11] do a more extensive partial evaluation, but their goal is different from ours: they want to specialize abstract programs by determining the shapes of the input data structures. Under their scheme, data manipulation operations and procedure calls will be performed at compile-time, when possible. When data values are not available, a list of instructions with placeholders for the unknown data values will be created and that list of instructions will be executed at run-time. Our approach, intended for programs that are much less abstract, is simpler, but could also be less expensive, especially if we interpreted a reduced program as indicated in section 6.2.

Our approach benefits programs which read values for variables used as loop bounds or array indices and is not applicable to other types of programs. Although our present method does not read arrays, index array assertions could be derived while partial evaluation was being done. If a user identified which arrays should be treated as index arrays, the properties of the arrays could be derived automatically, based on the input values read. For example, it would not be difficult to test whether an array A is monotonically non-decreasing while its values are being read in: the value read for A(k) would be compared to the value obtained for A(k-1), for $1 \leq k \leq N$, until we found an A(k)<A(k-1) or all N elements of the array had been read.

Our interprocedural constant propagator was only able to handle global scalars. Adding the ability to handle arrays should increase the number of dependences eliminated. Even in its present condition, however, the use of partial evaluation noticeably improved the performance of some dependence tests.

Chapter 7

Conclusions

We have shown that symbolic terms, which can hinder dependence analysis, appear frequently in numerical Fortran programs. Existing symbolic analysis methods cannot handle many symbolic terms and can be expensive. We have presented three techniques to enhance the effectiveness of dependence analysis by reducing the number of unknown symbolic terms and by making better use of the results of dependence analysis at run-time.

Breaking Conditions Breaking conditions are created by the dependence analyzer when the presence of symbolic terms make it impossible to either prove or disprove independence. Breaking conditions are passed along to the code generator, which creates fragments of optimized code guarded by those conditions. If a breaking condition is satisfied at run-time, then the optimized code can be executed.

Index Array Assertions Index array assertions are user-inserted directives describing any special properties of the index arrays used in a program. These directives can eliminate additional dependences by providing information that cannot be obtained using dependence analysis.

Partial Evaluation Partial evaluation can be used to inspect a program's input file and disseminate the actual values of symbolic variables before dependence analysis is done. Partial evaluation assists dependence analysis by reducing the number of unknown symbolic terms.

Of these three techniques, partial evaluation is the one which should be employed with the most discretion. Our experimental results indicate that partial evaluation is advantageous only for a certain category of programs, so it is in the user's best interest to apply partial evaluation only where it can be beneficial. Specifically, partial evaluation was useful when applied to programs that read in the values of scalar constants. Our constant propagator could not handle global arrays, so no partial evaluation was attempted for them.

The use of index array assertions is also limited by the nature of the program being optimized, but unlike partial evaluation, there is extremely little cost associated with applying this technique to the "wrong" sort of program. With partial evaluation, one must pay for the cost of interpretation whether or not it can do any good; with index array assertions, the only cost incurred if there are no index array references is that the scanner will briefly check each comment to see if it is actually a directive. The assertions suggested in this thesis can only eliminate one type of direction vector per test, whereas breaking conditions can eliminate all three, but index array assertions are probably cheaper, especially if properties are not being propagated around the call graph. This is not to say that breaking conditions are expensive, but rather that index array assertions are relatively cheap; the potential gain is smaller but so is the cost.

With the possible exception of extremely large programs, breaking conditions do not have the limited applicability of partial evaluation or index array assertions. Our experiments indicate that the use of breaking conditions was responsible for more than one-fifth of all direction vectors eliminated, which makes them a powerful tool indeed.

Let us return to the questions set forth in Chapter 1:

- How often do symbolic terms appear in dependence testing?
- How prevalent are index arrays and what properties do they frequently possess?
- How precise are dependence tests, i.e., how often can we prove that a dependence does or does not exist?
- How much improvement in the precision of dependence tests and in the execution time of the programs is observed when the techniques presented here are employed?

The first question is fairly simple to answer. For the experiment conducted in Chapter 2, 27.5% of all successful tests and 19.6% of the cases where independence was proven required the ability to handle symbolic additive constants. Those numbers rose to 38% and 46%, respectively, for our breaking conditions experiment, indicating that there are symbolic terms out there which PFC cannot grapple with using only traditional means.

The second question is a little more awkward, as we did not encounter that many programs explicitly using index arrays, and those we did find often lacked either data or documentation indicating how the index arrays might be used. Our search was not exhaustive, however, so that does not mean such programs do not exist. Some of the programs we did inspect appeared to be using their index arrays as perturbation arrays, so the *unique* assertion, at the least, should prove useful.

The third question is addressed to some extent in Chapter 2, Section 2.5.1, where we discuss the precision and complexity of the Delta test. Although the percentages varied, all of the experiments conducted for this thesis showed that most of the dependence tests applied were exact.

We were unable to generate and run code for our experiments, so the fourth question is partially unanswered. We did see an increase in the number of dependences eliminated when our techniques were used, but we have no accurate measure of gain or cost. For breaking conditions, the overall time spent on dependence tests dropped by 6.4%, with a 1.6% decrease in the average time required to perform a dependence test. For index array assertions, the total time spent on dependence tests increased by 0.3% and the average time to perform a test increased by 0.4%. For partial evaluation, the overall test time dropped by 0.2% while the average time to perform a test increased by 7%.

Future Work

The most obvious direction for future exploration is to generate and run the code optimized by these techniques. While we have shown that these techniques can assist dependence analysis, we do not yet have a precise idea of their cost.

It would also be interesting to investigate how well these techniques interact. Index array assertions and breaking conditions are unlikely to affect each other, as they are applied to different test cases. Breaking conditions are only calculated for the ZIV and strong SIV dependence tests, and indirect references would “flunk out” of those test cases because an indirect reference would never be categorized as “linear”.

Partial evaluation, on the other hand, could usefully complement either breaking conditions or index array assertions. The use of partial evaluation could increase the number of trivial breaking conditions, and might reduce the number of cases where index array reference comparisons have to be abandoned because we do not know the value of a symbolic term. Also, symbolic constants whose values have been discovered by partial evaluation but not exploited because they do not appear in loop bounds, etc., might be taken advantage of by another method, thus widening the applicability of partial evaluation.

Bibliography

[1] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Dept. of Computer Science, 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] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

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

[7] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, 21– May 1996.

[8] 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.

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

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

[11] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[12] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Supercomputing '94*. IEEE Computer Society, 1994.

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

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

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

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

[17] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

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

[19] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In Jeanne Ferrante and David Padua, editors, *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, volume 30(8), pages 92–101, August 1995.

[20] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.

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

[22] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^{\mathrm{n}}$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[23] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, January 1978.

[24] 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.

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

[26] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[27] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Seattle, WA, March 1990.

[28] Y. Futamura. Partial computation of programs. In E. Goto et alia, editor, *RIMS Symposia on Software Science and Engineering*, pages 1–35. Springer-Verlag, Kyoto, Japan, 1982.

[29] 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.

[30] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, 21–24 January 1996.

[31] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[32] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[33] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[34] 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.

[35] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[36] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[37] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

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

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

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

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

[42] 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.

[43] P. Kleinrubatscher, A. Kriegshaber, R. Zchling, and R. Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.

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

[45] 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.

[46] 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.

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

[48] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[49] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[50] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. *ACM SIGPLAN Notices*, 31(5):137–148, May 1996.

[51] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 28(7), pages 83–91, July 1993.

[52] 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.

[53] 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.

[54] 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.

[55] L.A. Lombardi and B. Raphael. Lisp as the language for an incremental compiler. In E.C. Berkeley and D.G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*. MIT Press, Cambridge, Massachusetts, 1964.

[56] Vadim Maslov. Lazy array data-flow dependence analysis. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–325, Portland, Oregon, January 1994.

[57] K. S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report TR91-162, Dept. of Computer Science, Rice University, December 1990.

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

[59] 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.

[60] C. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1986.

[61] C. Polychronopoulos. Advanced loop optimizations for parallel computers. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987. Springer-Verlag.

[62] C. Polychronopoulos. *Parallel Programming and Compilers.* KLUWER, 1988.

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

[64] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, Univ. of Maryland, November 1994.

[65] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Supercomputing '94: International conference — July 1994, Manchester*, pages 33–43, Manchester, UK, 1994. Univ of Manchester.

[66] John H. Reif. Symbolic programming analysis in almost linear time. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 76–83, Tucson, Arizona, January 1978.

[67] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 104–118, Los Angeles, California, January 1977.

[68] Erik Ruf and Daniel Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, July 1993.

[69] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, 21–24 January 1996.

[70] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[71] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganziger and N.D. Jones, editors, *Programs as Data Objects*, pages 236–256. Springer-Verlag, Copenhagen, Denmark, 1985.

[72] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[73] R. Shostak. On the sup-inf method for proving presburger formulas. *JACM*, 24(4):529–543, Oct 1977.

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

[75] R. Venken and B. Demoen. A partial evaluation system for prolog: Some practical considerations. *New Generation Computing*, 6:279–290, 1988.

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

[77] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6), pages 1–12, New York, NY, USA, June 1995.

[78] 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.

[79] 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.

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

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

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

[83] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.