ADVANCING DATA DEPENDENCE ANALYSIS IN PRACTICE

APPROVED BY SUPERVISING COMMITTEE:

_____
Kleanthis Psarris, Chair

_____
Rajendra V. Boppana, PhD

_____
Tom Bylander, PhD

_____
Kay A. Robbins, PhD

_____
Walter B. Richardson, PhD

Accepted: _____
Dean, Graduate School

ADVANCING DATA DEPENDENCE ANALYSIS IN PRACTICE

by

KONSTANTINOS KYRIAKOPOULOS, M.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2007

ACKNOWLEDGEMENTS

It has been quite a demanding task putting this material together over the years. I have personally grown and changed in the process of realizing what this academic achievement is about. What it takes to be recognized as an independent researcher and be awarded the degree of the doctor of philosophy. This would have not been possible without the continuous support and encouragement of my parents and the people who believed in me and helped me step up to the challenges. Many thanks go to Fotoula and Elias Kyriakopoulos, Haydar Tomas Sahin, and Liberta Quiroz.

I am especially grateful to my undergraduate advisor Prof. Antonios Panayotopoulos who inspired me to embark on this trip in academia. I would also like to thank the UTSA Department of Computer Science for supporting me financially and my PhD advisor Dr. Kleanthis Psarris for engaging me into a very interesting and challenging topic.

April 2007

ADVANCING DATA DEPENDENCE ANALYSIS IN PRACTICE

Konstantinos Kyriakopoulos, M.S.
The University of Texas at San Antonio, 2007

Supervising Professor: Kleanthis Psarris, Ph.D.

Optimizing and parallelizing compilers rely upon program analysis techniques to detect data dependences between program statements. Data dependence information captures the essential ordering constraints that need to be preserved in order to produce valid optimized and parallel code. However, source code for high performance computers is extremely complex to analyze and optimize. Most data dependence analysis tests can compute data dependence information for simple instances of the dependence problem. In more complex cases including triangular or trapezoidal loop regions, symbolic variables, multidimensional arrays with coupled subscripts, and if-statement conditions most tests ignore or simplify many of the problem constraints and thus introduce approximations. Furthermore, the majority of the data dependence tests have been designed to analyze problems with linear expressions even though non-linear expressions occur very often in practice. As a result, considerable amounts of potential parallelism remain unexploited.

We began our research by examining the reasons of inaccuracy in current dependence analysis. Based on our observations and experimental results, we developed new theory and methodology to overcome such limitations. In particular, we devised and implemented polynomial-time dependence analysis algorithms to handle complex instances of the problem and increase program parallelization. As a result, we introduced new dependence analysis techniques that can prove or disprove dependences in source code with non-linear and symbolic expressions, complex loop bounds, arrays with coupled subscripts, and if-statement constraints. In addition, our techniques can produce accurate and complete direction vector information, enabling the compiler to apply further transformations. Our method is based on the variable interval theory for multivariate functions that we developed as part of our research. We incorporated our techniques into a new data dependence analysis tool, the NLVI-Test, which we implemented as part of the PLATO library. To validate our method we performed an extensive experimental evaluation and comparison against the I-Test, the Omega test, and the Range test in the Perfect benchmarks, the SPEC benchmarks, and the Lapack library. The experimental results show that our dependence analysis tool is accurate, efficient and more effective in program parallelization than all the other dependence tests. The improved parallelization results into higher speedups and better program execution performance in several benchmarks.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction to Data Dependence Analysis

Optimizing compilers rely on data dependence analysis techniques for the automatic detection of implicit parallelism in sequential programs. Most high level optimizations involve parallelization, vectorization, and other code motion transformations. Data dependence analysis is the key to optimization and detection of implicit parallelism in sequential programs. It provides the compiler with the necessary information to perform valid transformations such as those for improving memory locality, load balancing, and efficient scheduling [2], [3]. Furthermore, data dependence information is essential for detecting loop iterations that can be executed in parallel on multi-processor and vector-processor architectures. Data dependence relations capture the essential ordering constraints among the statements in a program that have to be preserved in any compiler transformation to ensure the validity of the generated code.

In the following sections we will briefly discuss the data dependence problem and several algorithms that have been proposed in literature to address the problem.

1.1 The Data Dependence Problem

The dependence problem in general, is defined between two statements in the source code that perform memory access. Particularly, there exists a dependence relation between two such statements if and only if both statements access the same memory location and one of them performs a write operation. We denote the data dependence relation between two statements, as $S_1 \delta S_2$ if statement S_1 precedes statement S_2 in the order of execution. There are three types of data dependence:

- *Flow dependence*: When statement S_1 writes on a memory location first and S_2 reads that memory location later. It is denoted as $S_1 \delta^f S_2$.

- *Anti-dependence*: When statement S_1 reads a memory location first and S_2 writes on that memory location later. It is denoted as $S_1 \delta^a S_2$.

1

- *Output dependence*: When statement S_1 writes on a memory location first and later S_2 writes again on the same memory location. It is denoted as $S_1 \, \delta^o \, S_2$.

These memory accesses in conventional sequential languages are perrformed either with scalar variables, array variables, or pointer references. Data dependence analysis for scalar variables has been studied extensively and can be solved with data flow analysis [2]. Data flow analysis includes techniques such as induction variable substitution [40], constant propagation [10], and other transformations. The data dependence problem for pointers is difficult to handle in the general case because it deals with memory accesses to dynamic data structures and it constitutes an altogether separate research area [23].

Our research we focuses on the problem of data dependence for array references. Particularly we are concerned with array references inside loop nests, since most of the execution time of programs is spend in loop iterations and because the data dependence information for loops, is very important for automatic parallelization and vectorization. Furthermore, many loop transformations [3] that are based upon the data dependence information.

1.2 Data Dependence in Loops

Inside the loop body, each statement can be executed many times. Data dependences can flow from any instance of execution of a statement to any other statement instance. Figure 1-1 displays a general loop structure with two array references. Most of the loops in real code like those depicted in Figure 1-1 have stride of one. When that is not the case, most compilers will normalize the loop. If a compiler does not perform normalization, each data dependence test can be easily modified to take the stride into account. The nest of a statement is defined as the set of all enclosing loops. The common nest of two or more statements is defined by the intersection of their individual nests. In order for a data dependence relation to exist between two statements, there have to be statement instances that access the same memory location. Each statement instance is defined by a set of values of the enclosing loop indices that determine the specific iterations that the instance is executed. If there exist values of the loop indices such that at two distinct iterations the two statements instances access the same memory location through an array reference, then data dependence exists. The data dependence problem translates into the following integer system of equations

$$
\begin{aligned}
f_1(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) &= g_1(I'_1, I'_2, \ldots, I'_n, I_{n+1}, I_t) \\
f_2(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) &= g_2(I'_1, I'_2, \ldots, I'_n, I_{n+1}, I_t) \\
&\vdots \\
f_p(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) &= g_p(I'_1, I'_2, \ldots, I'_n, I_{n+1}, I_t)
\end{aligned}
\tag{1-1}
$$

```
for I₁ = P₁ to Q₁ do
  for I₂ = P₂ to Q₂ do
    •
      •
    for Iₙ = Pₙ to Qₙ do
        for Iₙ₊₁ = Pₙ₊₁ to Qₙ₊₁ do
          •
            •
          for Iₛ = Pₛ to Qₛ do
            •
              •
```

S_1 : $A[f_1(I_1, I_2, …, I_n, I_{n+1}, …, I_s), …, f_p(I_1, I_2, …, I_n, I_{n+1}, …, I_s)] = …$

```
              •
            endfor
              •
              •
          endfor
          for Iₛ₊₁ = Pₛ₊₁ to Qₛ₊₁ do
            •
              •
            for Iₜ = Pₜ to Qₜ do
              •
                •
```

S_2 : $… = A[g_1(I_1, I_2, …, I_n, I_{s+1}, …, I_t), …, g_p(I_1, I_2, …, I_n, I_{s+1}, …, I_t)]$

```
                •
              endfor
                •
                •
            endfor
        endfor
          •
        •
  endfor
endfor
```

$\text{nest}(S_1) = (I_1, I_2, …, I_n, I_{n+1}, …, I_s)$

$\text{nest}(S_2) = (I_1, I_2, …, I_n, I_{s+1}, …, I_t)$

$\text{nest}(S_1, S_2) = (I_1, I_2, …, I_n)$
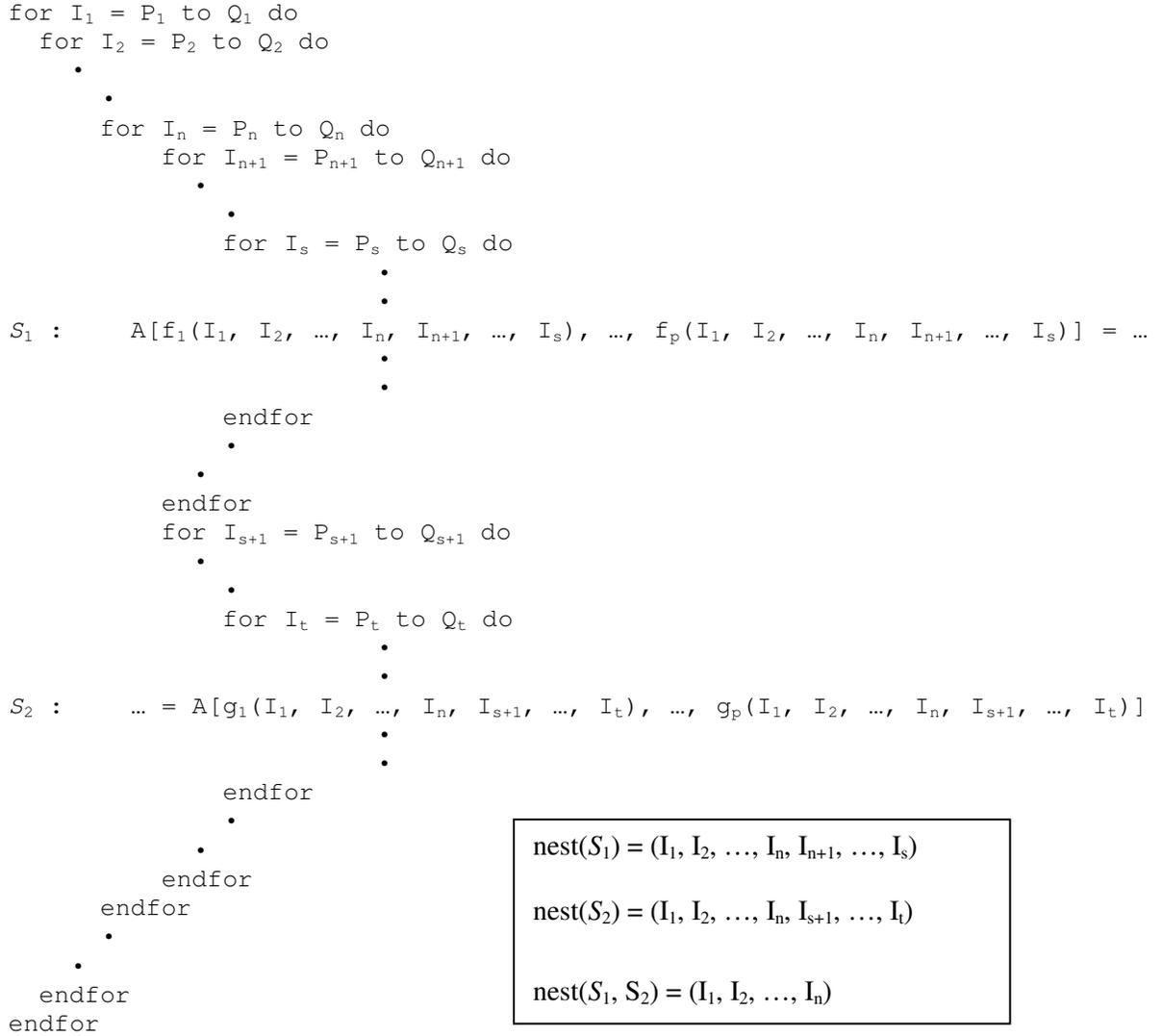
Figure 1-1: General l loop structure

subject to the following constraints:

$$P_k \le I_k, I'_k \le Q_k, \; 1 \le k \le n,$$
$$P_k \le I_k \le Q_k, \quad n + 1 \le k \le t$$

Every P_k and Q_k represents the lower and upper bounds of each loop index respectively. If the system in (1-1) has an integer solution, then there exist values for the loop index variables that access the same memory location and, therefore, data dependence exists between the statements. For single dimensional arrays, there is only one equation to be tested. When testing multidimensional arrays, we say that a subscript position is *separable* [12] if each loop index does not occur in more than one subscripts. If two different subscripts contain the same loop index, we say they are *coupled* [20].

When all subscripts are separable we can test each subscript separately, since the equations in the system are independent. This is known as *subscript-by-subscript testing* and results into testing a single equation for integer solutions each time. However, this method introduces a conservative approximation for multidimensional arrays with coupled subscripts, since it does not guarantee the existence of a simultaneous integer solution for the system of equations.

1.3 Distance and Direction Vectors

Data dependence relations are annotated with information about the relative iterations in which the related (dependent) instances occur. Such information is stored in distance or direction vectors. When two statements are data dependent, then there exist two vectors of values of the loop index variables for which the two array references access the same memory location. If the two statements share n common loops, namely loops $I_1, I_2, \ldots, I_n$, we represent the values for which data dependence exists with two vectors:

$$\boldsymbol{i} = (i_1, i_2, \ldots, i_n) \qquad \text{and} \qquad \boldsymbol{i}' = (i'_1, i'_2, \ldots, i'_n)$$

These values specify iterations of the common loops for which the executed statements S_1 and S_2 will be referred as instances $S_1[\boldsymbol{i}]$ and $S_2[\boldsymbol{i}']$, respectively. Suppose that $\boldsymbol{i} \leq \boldsymbol{i}'$ (in lexicographic order) this means that the dependence relation originates from $S_1[\boldsymbol{i}]$ to $S_2[\boldsymbol{i}']$ since the instance of S_1 is executed before the instance of S_2. We define the *distance vector* of a data dependence relation to be the always lexicographically positive vector:

$$\boldsymbol{d} = \boldsymbol{i}' - \boldsymbol{i} \tag{1-2}$$

If $\boldsymbol{i}' \leq \boldsymbol{i}$, then the dependence exists from $S_2[\boldsymbol{i}']$ to $S_1[\boldsymbol{i}]$ and the distance vector is defined as the negative of (1-2). Most loop transformations including parallelization and vectorization require only the sign of the elements in the distance vector. The sign provides information about the direction of the dependence as well as the loop that carries the dependence. This information can be summarized using the direction vector. A *direction vector* of a data dependence relation is defined as the symbolic vector:

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_n) \qquad \text{where} \quad v_k \in \{<, =, >\}, 1 \leq k \leq n \text{ and}$$

$$v_k = \begin{cases} < & \text{if } i_k < i'_k \\ = & \text{if } i_k = i'_k \\ > & \text{if } i_k > i'_k \end{cases} \tag{1-3}$$

When we test for data dependence, we usually need to find all direction vectors for which dependence exists between the various execution instances of S_1 and S_2. This is done by introducing into the data dependence system in (1-1) additional constraints, imposed by the direction vector. When the direction in a particular loop level is not of interest, we denote the direction as '*'. Thus, when we check if dependence exists for a particular direction vector, we let each $v_k \in \{<, =, >, *\}$.

We define the *direction vector dependence problem under direction vector $v = (v_1, v_2, \ldots, v_n)$* to be the question of whether the system of equations:

$$f_1(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) = g_1(I_1', I_2', \ldots, I_n', I_{s+1}, I_t)$$
$$f_2(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) = g_2(I_1', I_2', \ldots, I_n', I_{s+1}, I_t) \qquad\qquad (1\text{-}4)$$
$$\vdots$$
$$f_p(I_1, I_2, \ldots, I_n, I_{n+1}, I_s) = g_p(I_1', I_2', \ldots, I_n', I_{s+1}, I_t)$$

has an integer solution subject to the following constraints.

$$P_k \leq I_k\ v_k\ I_k' \leq Q_k, \qquad 1 \leq k \leq n$$
$$P_k \leq I_k \leq Q_k, \qquad\quad n + 1 \leq k \leq t,$$

where $v_k \in \{<, =, >, *\}$ and $I_k < I_k'$, $I_k = I_k'$, and $I_k > I_k'$ have obvious meanings and $I_k * I_k'$ is always true.

A data dependence relation between two statement instances in two different iterations of a loop is called *loop carried dependence*. A data dependence relation between two statement instances that are executed within the same iteration for all common loops is called *loop independent dependence*. A dependence is carried by a loop at level i if and only if there exists a direction vector v where $v_i = $ '<' and $v_k = $ '=' for $k < i$. A loop that carries no dependences can be executed in parallel. The distance vector was first used by Kuck and Muraoka [20], and the notion of direction vector was introduced by Wolfe [37].

1.4 Data Dependence Tests

The data dependence problem is at least as hard as integer programming, and thus, it can not be efficiently solved in general. Nonetheless, in some cases we can provide a solution in polynomial time, for certain instances of the problem. A number of data dependence tests are proposed in the literature [4], [22], [19], [39], [31], [6]. In each test there are different tradeoffs between accuracy and efficiency. Data dependence tests always approximate on the conservative side; i.e., dependence is assumed if independence cannot be proved. In this way, no unsafe parallel program is produced. Other techniques using a combination of simpler tests have been proposed and were proved to work

well in practice [14], [24]. The issues of data dependence analysis and how different tests deal with them, is the focus of Chapter 2.

1.4.1 The Banerjee Test

The Banerjee test [4] is the most commonly used data dependence test. It is simple in concept and implementation, and quite efficient. It is based on the intermediate value theorem and can prove or disprove the existence of a real solution in a single linear equation subject to the loop bound constraints. This theorem applies to every real function that is continuous in a region $\mathfrak{R}$ in the real domain.

Theorem 1-1:

Consider the equation

$$f(\boldsymbol{x}) = c, \qquad \boldsymbol{x} \in \mathfrak{R} \subset \mathbf{R}^n$$

and $L \in f(\mathfrak{R})$, $U \in f(\mathfrak{R})$ such that $L \le c \le U$. Then there exists a real solution to the equation satisfying the constraints of $\mathfrak{R}$.

The theorem states that the function takes all intermediate values between a minimum and a maximum if they are realized by the function. The test can applied to linear equations of the following form

$$a_1 x_1 + a_2 x_3 + \ldots + a_n x_n = a_0 \tag{1-5}$$
$$P_k \le x_k \le Q_k, \qquad \text{for } 1 \le k \le n$$

The minimum and maximum values of the left-hand expression in (1-5) can be determined as

$$L = \sum_{k=1}^{n} (a_k^+ P_k - a_k^- Q_k), \qquad U = \sum_{k=1}^{n} (a_k^+ Q_k - a_k^- P_k)$$

where

$$a^+ = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{and} \qquad a^- = \begin{cases} -a & \text{if } a < 0 \\ 0 & \text{otherwise} \end{cases}$$

When $L \le a_0 \le U$ the Banerjee test assumes dependence since there exists a real solution to the equation subject to the bounds. If the above inequality does not hold, then there is no real solution and therefore no integer solution subject to the bounds.

When additional constraints are imposed by a direction vector, the extreme values become tighter. When the direction for loop variable I_k is star (*), then there is no constraint and the extreme values of each term $a_k I_k$ appearing in an equation of problem (1-4) are

6

$$\text{min} = a_k^+ P_k - a_k^- Q_k, \qquad\qquad \text{max} = a_k^+ Q_k - a_k^- P_k$$

When the direction is less (<), then there is an extra constraint imposed on the two variables namely $I_k < I'_k$. Suppose that the coefficients of these two variables are b_k and c_k, respectively. The extreme values of the two terms, $b_k I_k' + c_k I_k'$, are:

$$\text{min} = (b_k + c_k)P_k - (b_k^- - c_k)^+(Q_k - P_k - 1) + c_k$$
$$\text{max} = (b_k + c_k)P_k + (b_k^+ + c_k)^+(Q_k - P_k - 1) + c_k$$

When the direction is greater (>), i.e. $I_k > I'_k$, the analogy is obvious and the bounds can be derived by switching the variables. When the direction is equal (=) then the extreme values are:

$$\text{min} = (b + c)^+ P_k - (b + c)^- Q_k$$
$$\text{max} = (b + c)^+ Q_k - (b + c)^- P_k$$

The Banerjee test is a very approximate data dependence test. It considers one single subscript of a multidimensional array reference at a time and does not test for a simultaneous solution across subscripts. It detects only real solutions and, therefore, can never prove data dependences. Also it requires loops with constant bounds and ignores constraints imposed by loop bounds that are expressions of other loop indices (*triangular* and *trapezoidal* bounds) or other variables that appear in other problem constraints (*symbolic* bounds). An extension to the Banerjee test is the triangular Banerjee test [4], which was developed with respect to triangular and trapezoidal regions. Yet this test introduces another approximation because the bounds derived from the algorithm, are not always tight especially when testing for direction vectors. In addition, the Banerjee test does not consider if-statement conditions and cannot analyze non-linear expressions. However, the efficiency of the Banerjee test at disproving dependences makes it one of the most common tests used in parallelizing compilers.

In most compilers, the Banerjee test is not applied alone. Usually if the extreme values fail to disprove the data dependence, the GCD test is applied. The GCD (greatest common divisor) test is based upon a theorem of elementary number theory, which states that a linear equation has an integer solution if and only if the greatest common divisor of the coefficients on the left-hand side of the equation divides the right hand side constant [18].

Theorem 1-2:

There is an integer solution to the equation in (1-5) iff $\gcd(a_1, a_2, \ldots, a_n)$ divides a_0.

Unfortunately, the GCD test cannot prove dependence either, because it ignores the bounds completely. It can only disprove dependences and cannot work with direction vectors. In addition, the

number of dependences it can disprove is limited, because in most cases there exist terms with coefficient of one, which will cause the divisibility condition to be satisfied.

Although the Banerjee-GCD test is an approximate test it has been shown [29] that under certain conditions that occur frequently in practice, the Banerjee-GCD test produces exact answers.

1.4.2 The I-Test

The I-Test [19], [30] is based on and enhances the Banerjee test. Whereas the Banerjee test is unable to distinguish real from integer solutions, the I-Test can conclusively prove or disprove the existence of integer solutions (and hence dependences) in many cases. The I-Test is based on the observation that most of the real solutions predicted by the Banerjee test were in fact integer solutions. This insight led to the development of a set of conditions, which if met, meant that a given linear expression achieved every integer value between the minimum and maximum values calculated by the Banerjee test. These *accuracy conditions* [29] state the necessary and sufficient relationship between the coefficients of loop index variables and the range of values they realize, in order to guarantee that every integer value between the extreme values is achievable. In practice, these conditions are met so frequently that the I-Test in most cases constitutes a linear time exact test for single dependence equations. The *I*-Test is based on the notion of the integer interval equation:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = [L, U] \tag{1-6}$$
$$P_k \le x_k \le Q_k, \quad \text{for } 1 \le k \le n$$

An integer interval equation is used to denote the set of all ordinary linear equations with constant terms the integers between L and U. The interval equation in (1-6) has an integer solution if and only if at least one of the equations in the set has an integer solution, subject to the constraints. In other words there is a solution iff there exist values for the variables of the expression on the left-hand-side subject to constraints for which the value realized by the expression is between L and U. Any ordinary linear equation can be written as an interval equation. The equation and constraints in (1-5) are equivalent to

$$a_1x_1 + a_2x_3 + \ldots + a_nx_n = [a_0, a_0] \tag{1-7}$$
$$P_k \le x_k \le Q_k, \quad \text{for } 1 \le k \le n$$

The *I*-Test is applied starting with equations in the form of (1-7), and it is based on the following theorem.

Theorem 1-3:

If $|a_n| \le U - L + 1$, then the constrained interval equation in (1-6) is integer solvable iff the constrained interval equation

$$a_1 x_1 + a_2 x_2 + \ldots + a_{n-1} x_{n-1} = [L - a_n^+ Q_k + a_n^- P_k, \; U - a_n^+ P_k + a_n^- Q_k]$$
$$P_k \le x_k \le Q_k, \quad \text{for } 1 \le k \le n - 1$$

is integer solvable.

Theorem 1-1 describes a transformation which, when applicable, eliminates a term from the left-hand side of the equation and increases the length of the integer interval on the right hand side. If no such term exists then the following theorem may be applied.

Theorem 1-4:

If $d = \gcd(a_1, a_2, \ldots, a_n)$ then the constraint interval equation in (1-6) is integer solvable iff the constrained interval equation

$$a_1/d \, x_1 + a_2/d \, x_2 + \ldots + a_n/d \, x_n = [\lceil L/d \rceil, \lfloor U/d \rfloor]$$
$$P_k \le x_k \le Q_k, \quad \text{for } 1 \le k \le n$$

is integer solvable.

If $\lceil L/d \rceil > \lfloor U/d \rfloor$, then there is no solution to the interval equation regardless of the constraints. This is the Interval-Equation GCD test, and it can often disprove the dependence. Theorem 1-4 is used to produce small enough coefficients that satisfy the inequality of Theorem 1-3.

Theorem 1-3 is applied until there are no terms on the left-hand side. The *I*-Test can now determine whether there is an integer solution or not. If the integer interval on the right-hand side includes zero, then a solution exists, otherwise there is no integer solution subject to the constraints. The bounds of the final integer interval are the extreme values of the expression that are computed by the Banerjee test. If Theorem 1-3 does not apply, even after Theorem 1-4 has been applied, the *I*-Test continues to move terms on the right-hand side hoping to break the dependence like the Banerjee test would. If this does not happen, the *I*-Test becomes inconclusive.

When a direction vector imposes additional constraints, then pairs of terms on the left-hand side become coupled and need to be handled together. This means that Theorem 1-3 cannot apply to terms that are constrained by a direction. When the direction is star (*) or equal (=) then there is no restriction in the application of the theorem because in the first case there is no constraint and in the second case the two variables can be substituted by one. The problem comes with the less (<) direction. The greater (>) direction can be expressed with a less direction (<) by switching variables. In the general case when a direction vector is imposed on an interval equation, the interval equation can be written in following form:

9

$$\sum_{k=1}^{n} a_k x_k + \sum_{k=n+1}^{m} (b_k y_k + c_k z_k) = [L, U] \tag{1-8}$$

$$P_k \leq x_k \leq Q_k, \qquad \text{for } 1 \leq k \leq n$$
$$P_k \leq y_k < z_k \leq Q_k, \qquad \text{for } n+1 \leq k \leq m$$

Variables y_k and z_k represent the loop index variables with the less than direction constraint, and variables x_k represent the loop index variables with star or equal direction constraint. Theorem 1-3 can be extended to terms that are coupled by the less than direction constraints by the following theorem:

Theorem 1-5:

Let

$$t_m = \begin{cases} \max (|b_m|, |c_m|) & \text{if } b_m c_m > 0 \\ \max (\min (|b_m|, |c_m|), |b_m + c_m|) & \text{if } b_m c_m > 0 \end{cases}$$

If $t_m \leq U - L + 1$, then the interval equation in (1-8) is integer solvable iff the interval equation

$$\sum_{k=1}^{n} a_k x_k + \sum_{k=n+1}^{m-1} (b_k y_k + c_k z_k) = [L - (b_m^+ + c_m)^+ (Q_m - P_m - 1) - (b_m + c_m)P_m - c_m,$$

$$U + (b_m^- - c_m)^+ (Q_m - P_m - 1) - (b_m + c_m)P_m - c_m]$$

$$P_k \leq x_k \leq Q_k, \qquad \text{for } 1 \leq k \leq n$$
$$P_k \leq y_k < z_k \leq Q_k, \qquad \text{for } n+1 \leq k \leq m-1$$

has integer solution subject to the constraints:

Theorem 1-5 is applied the same way, moving coupled terms to the right-hand side reducing the number of terms on the left-hand side and increasing the length of the integer interval.

Both Theorem 1-3 and Theorem 1-5 applied when necessary together with Theorem 1-4 constitute the core of the Direction Vector *I*-Test.

The *I*-Test is an extension to the Banerjee test but it also tests when certain conditions hold that could lead to an exact answer. When these conditions are met, the maybes of the Banerjee test become definite yes answers. Nevertheless, the *I*-Test is not just another way of applying the Banerjee test. The *I*-Test can break a dependence relation even if both Banerjee and GCD tests fail to break it. In addition the *I*-Test subsumes the GCD test in a very natural way and most of the times we avoid the expensive computation of the greatest common divisor. However, the I-Test is also applied on a subscript by subscript basis in the case of multidimensional array references. If dependence is disproved when considering any of the subscripts then there can be no dependence. However, if all the individual tests produce yes answers, dependence is known to exist only if the subscripts are separable. The I-Test inherits all of the benefits of the Banerjee test, including efficiency and ability

to provide direction vector information. Similarly to the Banerjee test I-Test requires constant loop bounds and can be applied only to linear subscript.

1.4.3 The Omega Test

Omega test [31] is an exact dependence test and it constitutes a general-purpose integer constraint satisfaction algorithm. It is based on the Fourier-Motzkin variable elimination (FMVE) [9]. In addition the test applies an extension of FMVE which can guarantee the existence of integers.

The input to the Omega test is a set of constraints. Those can be equalities or inequalities resulting from the subscript expressions, the iteration index bounds or the if-statement conditions. The Omega test performs a series of variable elimination operations trying to reduce the problem to a number of equivalent sub-problems, which can be solved recursively. First, the Omega test normalizes all the constraints by dividing all coefficients by the greatest common divisor d of each constraint. Thus in equations, it may produce coefficients of one or break the dependence through the GCD test, and in inequalities it can tighten the constraint by $\lfloor a_0/d \rfloor$ for every inequality of the following form.

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq a_0$$

In the next step it eliminates all equations but not in the trivial way, i.e., substituting an equality by two inequalities; in fact when this situation occurs the Omega test is doing the opposite and replaces the two inequalities by an equation. The idea behind the test is variable elimination, so if a term has a coefficient with absolute value of one then the equation is being substituted to the rest of the constraints by eliminating both the equation and the variable. If this is not the case then the Omega test applies the following transformation. Consider the equation in (1-5). First find the absolute less coefficient, a_i, and set $m = |a_i| + 1$. Then add a new variable y and produce the constraint:

$$my = \sum_{k=1}^{n} (a_k \ \hat{\text{mod}} \ m)x_k$$

where $a \ \hat{\text{mod}} \ b = a - b\lfloor a/b + 1/2 \rfloor$. It follows that $a_i \ \hat{\text{mod}} \ m = -\text{sign}(a_i)$ and therefore the coefficient of a_i will have absolute value of one. So the variable x_i is substituted in all constraints including the original equation in (1-5). It turns out, that all terms of the equation after the substitution are divisible by m. This procedure, if applied repeatedly, will decrease the value of coefficients in the equation until a coefficient with absolute value of one can be found and the system can be reduced by one equation.

The next step is quite expensive. The Omega test performs a Fourier-Motzkin projection [11] of the problem constraints on the dimension of a variable that is eliminating. The variable elimination is performed on pairs of inequalities of the following form

$$\left.\begin{array}{c} \alpha x \leq u \\ v \leq bx \end{array}\right\} \tag{1-9}$$

where a, b are the coefficients of x in the two inequalities, and u, v are expressions that may involve other variables too. The elimination of x from (1-9) using the FMVE technique results into the inequality

$$av \leq bu \tag{1-10}$$

This process is applied for all constraints similar to those in (1-9). The set of new constraints produced is referred to as *real shadow*. Intuitively, the elimination of a variable by FMVE may be viewed as projecting an n-dimensional polyhedron onto an n-1 dimensional surface. If the resulting "real shadow" contains no integers, then the original object contains no integers, and the test reports that no solutions exist. Systems (1-9) and (1-10) are equivalent in the real domain but not in the integer. This is because while the object may not include integer points, its projection can actually cast a shadow over integer points.

For that reason a new way of projecting the two inequalities in (1-9) was used which guarantees that if the shadow of an object has integer points then there exist integer points in the original object. This new type of shadow is referred as *dark shadow* and is derived from (1-9) as:

$$bu - av \geq (a-1)(b-1) \tag{1-11}$$

The dark shadow of a region defined by a set of constraints, guarantees that for every integer point in it there exists at least one integer point in the original region. Note that if a or b are equal to one then the real shadow in (1-10) and the dark shadow in (1-11) are identical. This type of projection is called *exact* and it is very desirable because the number of constraints produced is minimized.

The real shadow is used in order to disprove a solution to the system. Since if there are no integers in it there can be no integer points in the original region. The dark shadow is used to prove a solution to the system. Since if there are integer points in it then there also exist integer points in the original region. When the existence of solution cannot be proved or disproved it is a situation referred as Omega's nightmare. What the Omega test does, in this case, is that it takes a perfect enumeration of the space between the real and the dark shadow. In particular it is searching for every value of $0 \leq i \leq$

($ab - a - b$)/a, to see whether there exists integer solution to the original set of constraints augmented by the equality $bx = v + i$. It has been claimed [31] that this situation only rarely occurs in practice.

The Omega test with a simple modification can produce direction vectors and distance vectors for every dependence problem. By adding the definition of the dependence distance into the set of constraints and protecting these variables until their value is resolved, the Omega test can find the sign of the dependence distance or its actual value if constant. In that case the original problem (1-1) is augmented by equations of the form:

$$\Delta_k = I'_k - I_k$$

The Omega test is a powerful test. It can be used in a lot of problems other than data dependence analysis. In the data dependence problem though, it can handle a lot of cases. The Omega test can handle array references with coupled subscripts because it tests for a simultaneous solution. It can handle triangular, trapezoidal and symbolic bounds because it is using the very powerful FMVE technique. It can also handle nested if-statements because it can take into account any type of linear constraint. It can derive direction vectors and direction distances in some cases. It has also been extended to compute value-based dependences [32] unlike all other data dependence tests. The only limitations to the Omega test are non-linear constraints. The test utilizes functions symbols [33] to express non-affine terms but those do not capture the actual non-linear patterns in most cases.

Nevertheless, the advantages of Omega test are shadowed by its high cost. The Omega test has very costly initialization and requires a lot of memory to hold all these sub-problems. Most of all, the Omega test grows exponential in many ways. The whole procedure is exponential regarding the number of variables. It performs recursively an exponential operation such as the FMVE. It even sometimes has to take a perfect enumeration of the solution space to determine whether a solution exists or not. There exist however cases where the Omega test can be restricted to a provably polynomial-time complexity [35]. For all these reasons it may not be suitable as a data dependence tool in real source code, where the number of problems that have to be solved is extremely high and efficient data dependence testing is required.

1.4.4 The Range Test

The Range Test [6] arose from need to address the issue of non-linear expressions. None-linear expressions are very common in scientific applications. Many of them are due to the actual source code while many other are due to compiler transformations, especially induction variable recognition

[8]. These non-linear expressions have been proven to hide parallelism that traditional dependence analysis techniques cannot expose [12].

In Range test, a data dependence carried by a particular loop is disproved by showing that the range of elements (memory locations) accessed by an iteration of that loop does not overlap with the range of elements accessed by other iterations. In the process of proving that, the Range test compares the symbolic expressions of the array subscripts. Utilizing the range propagation technique [7] it can prove such inequalities efficiently. Given a nest of loops $(i_1, \ldots, i_n)$ which are subject to a set of constraints in a region $\Re \subseteq Z^n$ and a subscript function f, the Range test defines following functions.

Definition 1-1:

$f_j^{\min}$ and $f_j^{\min}$ are functions of the first j loop variables such that.

$$f_j^{\min}(i_1, \ldots, i_j) \leq \min\{ f(i_1', \ldots, i_n'): (i_1', \ldots, i_n') \in \Re, i_1' = i_1, \ldots, i_j' = i_j \}$$
$$f_j^{\max}(i_1, \ldots, i_j) \geq \max\{ f(i_1', \ldots, i_n'): (i_1', \ldots, i_n') \in \Re, i_1' = i_1, \ldots, i_j' = i_j \}$$

The functions above return the minimum or the maximum value of f for given iterations of the j-th outermost loops. Based upon those functions the Range test utilizes the following three theorems to disprove dependences for particular direction vectors. Let $\Re_j \subseteq Z^j$ be the region defined by the constraints on variables $i_1, \ldots, i_j$.

Theorem 1-6:

If $f_j^{\max}(i_1, \ldots, i_j) < g_j^{\min}(i_1, \ldots, i_j)$ for all $(i_1, \ldots, i_j) \in \Re_j$ then there can be no dependence between two array references $\mathbf{A}(f(i_1, \ldots, i_n))$ and $\mathbf{A}(g(i_1, \ldots, i_n))$ with direction vector v of the form $v_1 = $ '=', $\ldots$, $v_j = $ '='

The above theorem provides a condition under which the j outmost loops of nest do not carry any dependence.

Theorem 1-7:

If $g_j^{\min}(i_1, i_2, \ldots, i_j)$ is increasing for variable i_j and $f_j^{\max}(i_1, \ldots, i_j) < g_j^{\min}(i_1, \ldots, i_j + 1)$ for all $(i_1, \ldots, i_j) \in \Re_j$ then there can be no dependence from $\mathbf{A}(f(i_1, \ldots, i_n))$ to $\mathbf{A}(g(i_1, \ldots, i_n))$ with direction vector v of the form $v_1 = $ '=', $\ldots$, $v_{j-1} = $ '=' , $v_j = $ '<'.

For decreasing functions there exists a similar theorem.

Theorem 1-8:

If $g_j^{\min}(i_1, i_2, \ldots, i_j)$ is decreasing for variable i_j and $f_j^{\max}(i_1, \ldots, i_j) < g_j^{\min}(i_1, \ldots, i_j - 1)$ for all $(i_1,\ldots, i_j)$ $\in \Re_j$ then there can be no dependence from $\mathbf{A}(g((_1, \ldots, i_n))$ to $\mathbf{A}(f(i_1, \ldots, i_n))$ with direction vector $\boldsymbol{v}$ of the form $\boldsymbol{v}_1 = \,$'$=$', $\ldots, \boldsymbol{v}_{j\text{-}1} = \,$'$=$' , $\boldsymbol{v}_j = \,$'$<$'.

Theorem 1-7 can disprove a dependence carried by the *j*-th loop from $\mathbf{A}(f((i_1, \ldots, i_n))$ to $\mathbf{A}(g((i_1, \ldots, i_n))$ while Theorem 1-8 can disprove a dependence carried by the *j*-th loop from $\mathbf{A}(g((i_1, \ldots, i_n))$ to $\mathbf{A}(f((i_1, \ldots, i_n))$.

The above theorems can be applied to disprove certain direction vectors. More direction vectors can be disproved by logically permuting the loops and replying the theorems. The Range test is using the data dependence information computed at each step in order to determine a valid loop permutation.

It is clear that the Range test is based on the notion of extreme value computation like the Banerjee test. However it is using a simplified heuristic to compute the direction vector information rather then solving each direction vector problem independently like the Banerjee test would. This technique even though it is far more efficient, it has the disadvantage of not computing complete direction vector information. This is likely to prevent certain compiler transformations that utilize complete direction vector information such as loop interchange [3].Furthermore the Range test is a subscript-by-subscript test and does not take into account the rest of the array subscripts when testing for data dependence. However it takes into account trapezoidal and symbolic bounds even though it can never provide a conclusive answer in case the dependence cannot be disproved.

1.4.5 Other Work in Dependence Analysis

Maydan et al [24] proposed a small set of efficient algorithms, each one exact for special case inputs, combined with FMVE as an expensive back up test. They show empirically that they derive an exact answer in all cases on the Perfect benchmarks. Their experiments obviously exclude dependence problems with nonlinear constraints, since neither of the proposed tests can accurately handle them.

Goff et al [14] propose a dependence testing scheme based on classifying pairs of subscripts. This approach is based on the fact that most array references in scientific programs are fairly simple. Efficient and exact tests are presented for certain classes of commonly occurring array references involving zero index variables (ZIV) and single index variables (SIV). In case of multiple index variable (MIV) subscripts their techniques rely on the GCD and Banerjee tests. The I-Test is exact and more general than all special exact SIV cases.

There also exist other important data dependence tests that are not described. The λ test [22] is a very intuitive test that extends the ideas behind the Banerjee test to handle multidimensional array references with coupled subscripts, but it is only able to prove real solutions. The Power Test [39] has a lot in common with the Omega test. They both use the Fourier-Motzkin variable elimination method to prove consistency or inconsistency among the dependence problem constraints. A significant difference is that the Power Test is using simplified constraints after performing the Extended GCD test [4], and that it does not force itself to produce an exact answer when this is going to be expensive.

Work by the Polaris group [8] and the SUIF group [16] have contributed state of the art compiler technology that can help extract inherent program parallelism. However their work is not primarily focused on dependence analysis. Several studies have acknowledged the limitations of existing data dependence analysis techniques and they have shown that non-linear symbolic analysis is necessary in order to uncover parallelism in scientific applications [12], [15],

Work in array access analysis has proposed new methodology that can represent memory access patterns through array access descriptors even when the array subscripts are non-linear [26]. This theory has lead to the development of new a framework for dependence analysis based on array access regions. However their method does not cover several of the non-linear array access patterns often found in practice.

Recent work on data dependence analysis focuses on loop variants that appear in source code and which can not be inducted to closed form expressions by traditional induction variable substitution algorithms. These theoretical studies propose techniques that can recognize non-linear patterns based on the monotonic evolution of the variables [41], value evolution graphs [34], or on chains of recurrences algebra [13], [5]. Non-linear tests like the NLVI-Test can take advantage of such techniques to uncover even more parallelism.

Chapter 2

Issues in Data Dependence Analysis

When implementing a data dependence analysis algorithm all issues that may result to inconclusive answers should be considered. Ideally a data dependence problem is defined between two array references with linear and uncoupled subscripts, embedded in a nest of loops with constant upper and lower bounds. However, this ideal case is far from truth in practice. Each data dependence test has its own restrictions and limitations so inexact answers may arise due to different reasons. In the following sections we detail all the different cases for which a data dependence test might return an inexact answer. We discuss the effects of constraints and source code patterns such as loop variant variables, non-linear and symbolic expressions, if statement conditions and multidimensional array subscripts. We also discuss the effects of complex expressions in loop bounds, array subscripts and if-statement conditions. In addition we present simple techniques which may address these issues and increase data dependence accuracy.

2.1 Loop Variant Variables

Loop variant variables are programming variables whose values are updated inside the loop nest. Their value depends on the values of the enclosing loop indices. The issue of loop variants extends beyond data dependence analysis. Compiler techniques such as induction variable substitution [40], or CR analysis [13], can express the loop variant variables as a function of the loop indices and can replace all occurrences of these variables in the source code. Such transformations are not always possible and very often when applied they result into non-linear expressions. In some cases however a dependence test may be able to resolve problems with loop variants accurately. Consider the following example.

Example 2-1:

```
      for I = 1 to N do
            for J = 1 to M[I] do
  S1:               A[I, K + 2*J] = …
  S2:               … = A[I, K + 2*J + 1]
            endfor
            K = 2*K + 1
      endfor
```

In the above example the value of array reference *M*[*I*] and the value of variable *K* change inside the loop nest. The value changes in each iteration of loop *I*, but it remains the same for all iterations of loop *J*. So the level of variance for array reference *M*[*I*] and variable *K* is the level of loop *I*. From the dependence equation of the first subscript we determine that dependence can only exist with direction vector (=, *). Within the same iteration of *I*, all values of *M*[*I*] and all values of *K* are equal. For that reason the terms of variable *K* in the second subscript can be eliminated from the dependence equation, since they appears on each side. After this elimination the dependence can be broken in this case.

For each loop variant expression we can determine the innermost loop that the value of the expression depends upon. This is the level of variance for that expression. All occurrences of that expression for direction vectors of the form ($=_1$, …, $=_l$, *, ..*), where *l* is the level of variance, are identical. The above technique can help determine whether loop variant terms that appear in the dependence problem are equal and therefore can be simplified through algebraic operations. This may result into elimination of the loop variants from the dependence system, which can help resolve the dependence problem accurately. We incorporated this technique in all of the data dependence tests to handle some cases of loop variant variables.

2.2 Non-Linear Expressions

Scientific source code tends to be quite complex as it usually expresses challenging mathematical problems for non-linear models. In addition, compiler transformations such as expression propagation, loop normalization and induction variable recognition may result into non-linear expressions that appear in loop bounds and array subscripts. Consider the following example.

Example 2-2:

```
      for I = 1 to N do
  S1:      A[I*N + 1] = …
  S2:              … = A[I]
      endfor
```

In the above example the first subscript of array *A* has a non-linear term *I*N*. Because of this term the values of the first array subscript are always greater than *N* + 1.Without taking into account this restriction a dependence test cannot determine that the maximum value of the second subscript is always smaller than the minimum value of the first subscript and therefore no dependence exists. Most data dependence algorithms simply ignore non-linear constraints and very often loose in accuracy. The vast majority of data dependence tests, including the Banerjee test, the I-Test and the Omega test, focus on dependence analysis for linear expressions. When a dependence test cannot analyze a non-linear expression it can treat it as loop variant variable using the technique describe in the previous subsection. Dependence tests such as the Range test can analyze any type of non-linear expression using ranges.

2.3 If-Statement Conditions

Another reason that may result to inexact answers is the IF and IF-ELSE statements. If-statements may contain conditions that can disprove the dependence if taken into account.

The general case of if-statements is hard to handle. Traditional dependence tests, such as the Banerjee test and the I-Test can somewhat deal with if-statements by examining the variables that appear in the conditional part of the if-statement. If these variables do not appear in any other constraints of the dependence problem and they are not loop variants, i.e. they are not updated inside the loop, then we can simply ignore them without introducing an approximation. In the above case, if the two array references tested for data dependence belong to the if-part and else-part respectively, we claim independence since only one of them executes for all loop iterations. If the variables in the conditional part of the if statement appear in other constraints of the dependence problem or they are contain loop variant variables, then the test can still ignore the if-constraints but cannot indicate a definite dependence. In this case an answer of maybe is reported due to the fact that the if-statement constraints have been ignored. The Omega test can include the conditions of if-statements into the dependence system, because it can handle all linear integer constraints. Consider the following example.

Example 2-3:
```
      for I = 1 to 10 do
            if (I < 5) then
S₁:                A[I] = …
            else
S₂:                … = A[I + 1]
            endif
      endfor
```

The Banerjee test, the I-Test and the Range test ignore the constraint ($I < 5$) in the if-statement and report a maybe answer. The Omega test on the other hand is able to disprove the dependence in this case.

2.4 Coupled Subscripts

In case of multidimensional arrays with coupled subscripts, data dependence tests that rely on subscript-by-subscript testing, such as the Banerjee test, the I-Test and the Range Test are not able to provide an exact positive answer, since they cannot detect a simultaneous solution for all equations. Consider the following example.

Example 2-4:
```
     for I = 1 to 100 do
S1:       A[I, I] = …
S2:       … = A[I, I + 1]
     endfor
```

In the above example subscripts are coupled and subscript-by-subscript testing will indicate possible dependence when no dependence exists. Nonetheless, there exist data dependence problems where even though the subscripts are coupled the equations of the dependence system are independent for certain direction vectors. Consider the following example.

Example 2-5:
```
     for I = 1 to 20 do
       for J = 1 to 20 do
S1:        A[I + J , I] = …
S2:           … = A[I + J + 1 , I]
       endfor
     endfor
```

Even though the subscripts are coupled, for all (=, *) direction vectors the equations of the dependence system do not share any common variables. In this case the I-Test is able to prove the dependence. Thus, coupling between dependence equations should be checked for every direction vector. Coupled subscripts do not introduce an approximation in the Omega test, since it takes all equations into consideration at once.

2.5 Complex Loop Bounds

Many data dependence tests, including the Banerjee test and the I-Test, make the assumption that loops have constant upper and lower bounds. In practice though, there exist many loops with

20

triangular bounds and bounds that are expressions of symbolic variables. The ability to handle complex loop regions is the main distinction between a symbolic and non-symbolic analysis.

Both the Banerjee test and the I-Test take as input a set of equations that include loop index variables and any other program variables. Each variable has a lower and an upper bound. We have enhanced the implementation of the I-Test and the Banerjee test by augmenting the upper and lower bound information of each variable with a state flag. The state of a bound can be either exact or inexact. An exact state means that the bound is constant. An inexact state means that the bound is an expression of other loop indices (triangular bounds) or other variables (symbolic bounds). We can approximate triangular and symbolic bounds with constant values in order for the Banerjee test and the I-Test to be applied. In this case the state flag indicates that we have made an approximation and both tests lose in accuracy, because we simplified a constraint in the problem.

There are many cases though that we can provide a definite answer even in the presence of triangular or symbolic bounds. Consider the following example.

Example 2-6:
```
      for I = 1 to 5 do
        for J = 1 to I + 1 do
  S1:        A[I + J] = …
  S2:        … = A[I + J + 10]
        endfor
      endfor
```

In Example 2-6, the upper bound of *J* can be approximated by the extreme value of the expression *I* + 1, which is 6. Even though the state of the upper bound of *I* in this case is inexact, we can still break the dependence. This approximation is better than assuming that the bound is infinity which would not have broken the dependence in this case.

If a loop bound involves symbolic variables that do not appear anywhere else, then it is safe to assume that the state of the bound is exact without making an approximation. Its value is either minus or plus infinity depending on whether it is a lower or an upper bound respectively. Consider the following example

Example 2-7:
```
      for I = 1 to N do
  S1:        A[I] = …
  S2:        … = A[I + 100]
      endfor
```

In Example 2-7, since the variable *N* does not appear in any other constraints, the bound of *I* can be considered to have an exact state and value plus infinity. In the implementation its value can be substituted by a large constant. If dependence exists for that constant then there exists dependence for

a value of *N*. If the dependence is broken then there cannot be a solution to the system for any value of *N* less than this constant. Since this constant is very large, we can safely assume that dependence does not exist for any value of *N*. In the example above, note that dependence does not exist for values of *N* less or equal to 100. Yet no technique at compile time can determine whether the dependence will exist until the program is executed and the value of *N* is determined at run time. Since a program may be executed several times and for certain values of *N* the two array references will access the same memory location, the I-Test in this case returns a positive yes answer. The Omega test in a similar manner will simply eliminate *N* from the dependence problem constraints and return a yes answer as well.

All symbolic variables appearing in the subscripts are assumed to have bounds with exact state and values minus and plus infinity, if they do not appear in any loop bound expressions. Since there is no constraint on them they can take any integer value possible. Consider the following example.

Example 2-8:
```
     for I = 1 to 100 do
S1:       A[I] = …
S2:       … = A[I + N]
     endfor
```

In Example 2-8, *N* appears as a variable in the dependence equation with bounds of minus and plus infinity. In this case the I-Test can provide a definite yes answer since dependence exists for values of *N* less than 100.

However, symbolic variables that appear both in loop bounds and arrays subscripts introduce approximations. Consider the following example.

Example 2-9:
```
     for I = 1 to N do
S1:       A[I] = …
S2:       … = A[I + N]
     endfor
```

In Example 2-9, *N* appears in both the upper bound of *I* and in the subscript of *A*. In this case the upper bound of *I* has an inexact state with value plus infinity. The I-Test cannot break the dependence in this case, because of the bound approximation.

Triangular and symbolic bounds is not an issue for the Omega test and the Range test since they are symbolic tests and can cope with complex loop regions.

2.6 Testing for Integer Solutions

Data dependence tests introduce an approximation when they test for real solutions as opposed to integer solutions. The Banerjee test, for instance, can only detect real solutions and therefore can never indicate a definite dependence. The Range test also cannot prove dependences. The I-Test on the other hand can prove integer solutions as long as a set of conditions, called accuracy conditions [29], is satisfied. The Omega test always tests for integer solutions. In fact in some cases when an integer solution is hard to prove the algorithm is choosing to perform a perfect enumeration of the space where an integer solution may be found each time recursively solving several instances of the Omega problem. In this case the Omega test becomes exponential in the order of magnitude of the coefficients and this situation is known as "the Omega test nightmare" [31]. It has been proven [25] that when the accuracy conditions of the I-Test fail the Omega test nightmare is inevitable. Consider the following example.

Example 2-10:
```
      for I = 1 to 6 do
  S₁:        A[I + 1] = …
  S₂:        … = A[7*I − 6]
      endfor
```

In the above example the Banerjee test and the Range test fail to disprove the existence of an integer solution subject to the loop bound constraints. The Banerjee and Range tests will return a false positive "maybe" answer when no dependence exists. Both the I-Test and the Omega test are able to disprove the dependence in this case.

Chapter 3

Dependence Analysis for Complex Loop Regions

In this Chapter we extend the ideas behind the I-Test to handle complex loop regions. We propose a set of polynomial-time techniques that can detect data dependences in loops with triangular or trapezoidal loop bounds and symbolic variables, subject to any direction vector. In addition, in the cases of multidimensional arrays with coupled subscripts, we introduce the equation propagation technique that can minimize or eliminate the coupling between subscripts and thus it can reduce the data dependence problem to a set of equations that can be tested independently to obtain exact data dependence information. We also propose simple techniques that incorporate if-statement constraints into the dependence problem and be able to conclusively prove or disprove dependences. We incorporated all these techniques into a new data dependence algorithm, which is termed the Variable Interval Test or VI-Test.

The rest of the Chapter is organized as follows. In Section 3.1 we present the theory behind the variable interval test. In Section 3.2 we describe how we apply the theory in practice to handle loops with trapezoidal bounds subject to direction vectors. We also show how the VI-Test can handle problems involving symbolic variables, we present the equation propagation technique, which is used to eliminate or minimize coupling between subscripts and we introduce techniques to handle simple cases of if-statements. Finally in Section 3.3 we analyze the VI-Test algorithm and its complexity.

3.1 Variable Integer Interval Theory

3.1.1 Basic Definitions

In this section we provide the basic definitions and some simple theorems that support the variable interval theory.

Definition 3-1:

As an integer interval [*L*, *U*], we define the set of all integer numbers between *L* and *U* including the bounds.

$$[L, U] = \begin{cases} \{L, L + 1,\ldots, U\} & \text{if } L \leq U \\ \varnothing & \text{otherwise} \end{cases}$$

For example the integer interval [1, 4] denotes the set {1, 2, 3, 4}.

Definition 3-2:

Given an integer region $\Re \subseteq Z^n$ and two functions *L* and *U* from $\Re$ to Z, we define the variable integer interval, denoted [*L*(**x**), *U*(**x**)], as the union of all integer intervals for all values $\mathbf{x}_i$ of **x** in $\Re$.

$$[L(\mathbf{x}), U(\mathbf{x})] = \bigcup_{\mathbf{x}_i \in \Re} [L(\mathbf{x}_i), U(\mathbf{x}_i)]$$

A variable integer interval is contiguous if the union of the integer intervals in Definition 3-2 is equal to a single integer interval.

For example the variable integer interval [*X*, 3*X*], $1 \leq X \leq 3$ is equal to the union of the integer intervals [1, 3] U [2, 6] U [3, 9] which is equal to the integer interval [1, 9] or the set of integers {1, 2, …, 9}.

Definition 3-3:

Given an integer region $\Re \subseteq Z^n$ and three functions *F*, *L*, and *U* from $\Re$ to Z, we define as variable interval equation any equation of the following form

$$F(\mathbf{x}) = [L(\mathbf{x}), U(\mathbf{x})]$$

The above equation is said to be integer solvable in $\Re$ iff there exist a value of **x**, $\mathbf{x}_0 \in \Re$ such that

$$L(\mathbf{x}_0) \leq F(\mathbf{x}_0) \leq U(\mathbf{x}_0)$$

Theorem 3-1:

The variable interval equation $F(\mathbf{x}) + G(\mathbf{x}) = [L(\mathbf{x}), U(\mathbf{x})]$ is integer solvable subject to a set of constraints on **x** in $\Re$, iff the variable interval equation $F(\mathbf{x}) = [L(\mathbf{x}) - G(\mathbf{x}), U(\mathbf{x}) - G(\mathbf{x})]$, is integer solvable subject to the same set of constraints.

Proof: The proof is straightforward from Definition 3-3.

∎

The above theorem basically states that you can move terms from one side of the equation to the other.

Before we proceed with the main part of our theory we provide some additional definitions, regarding our notation.

Definition 3-4:

We define the positive and negative part of an integer a respectively as:

$$a^+ = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad a^- = \begin{cases} -a & \text{if } a < 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 3-5:

Given an integer function $F(\boldsymbol{x})$ defined in a region $\Re \subseteq Z^n$ we define as $\min(F(\boldsymbol{x}))$ and $\max(F(\boldsymbol{x}))$ the minimum and maximum values realized by the function in $\Re$.

As an example consider $F(X) = aX$, $P \leq X \leq Q$, then $\min(F(X)) = a^+P - a^-Q$ and $\max(F(X)) = a^+Q - a^-P$. For instance, if $F(X) = 3X + 1$, where $1 \leq X \leq 5$, then $\min(F(X)) = 4$ and $\max(F(X)) = 16$.

3.1.2 Contiguous Variable integer interval

In this section we investigate the conditions under which a variable integer interval is equivalent to a single contiguous integer interval. First we examine a simple variable integer interval of the form $[L + \kappa X, U + \lambda X]$, where $P \leq X \leq Q$

Theorem 3-2:

The variable integer interval $[L + \kappa X, U + \lambda X]$, where $P \leq X \leq Q$ is equal to the integer interval

$$[L + \kappa^+P - \kappa^-Q, U + \lambda^+Q - \lambda^-P],$$

iff $\kappa\lambda \leq 0$, or $\kappa\lambda > 0$ and $U - L + (\lambda - \kappa)^+P - (\lambda - \kappa)^-Q + 1 \geq \min(|\kappa|, |\lambda|)$.

Proof: See Appendix.

∎

Example 3-1:

The variable integer intervals $[4 - 2X, 7 + X]$ and $[4 + 2X, 7 + 3X]$, where $1 \leq X \leq 5$ are contiguous. In the first interval $\kappa = -2$, $\lambda = 1$, $L = 4$, $U = 7$, $P = 1$, $Q = 5$. Since $-2 \times 1 \leq 0$, it follows that $[4 - 2X, 7 + X]$ is equal to $[4 - 2 \times 5, 7 + 1 \times 5] = [-6, 12]$. In the second interval $\kappa = 2$, $\lambda = 3$, $L = 4$, $U = 7$, $P = 1$, $Q = 5$. Since $2 \times 3 > 0$ and $7 - 4 + (3 - 2)^+ \times 1 - (3 - 2)^- \times 5 + 1 = 5 \geq 2 = \min(|2|, |3|)$, it follows that $[4 + 2X, 7 + 3X]$ is equal to $[4 + 2 \times 1, 7 + 3 \times 5] = [6, 22]$.

We can extend Theorem 3-2 to more complex variable integer intervals with the following theorem.

Theorem 3-3:

Consider the variable integer interval $[L(\boldsymbol{x}) + \kappa X, U(\boldsymbol{x}) + \lambda X]$, subject to a set of constraints on $\boldsymbol{x}$ in $\mathfrak{R}$ and the constraint $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$, where X does not appear in any of the constraints in $\mathfrak{R}$.

If $\kappa\lambda \leq 0$, or $\kappa\lambda > 0$ and $\min(U(\boldsymbol{x}) - L(\boldsymbol{x}) + (\lambda - \kappa)^{+} P(\boldsymbol{x}) - (\lambda - \kappa)^{-} Q(\boldsymbol{x}) + 1) \geq \min(|\kappa|, |\lambda|)$, then the above variable integer interval is equal to $[L(\boldsymbol{x}) + \kappa^{+} P(\boldsymbol{x}) - \kappa^{-} Q(\boldsymbol{x}), U(\boldsymbol{x}) + \lambda^{+} Q(\boldsymbol{x}) - \lambda^{-} P(\boldsymbol{x})]$, subject to the same constraints on $\boldsymbol{x}$ in $\mathfrak{R}$ and the constraint $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$

Proof: See Appendix.

∎

Example 3-2:

Consider the variable integer interval $[4 + 2X_1 + X_2, 9 + X_1 + 2X_2]$, where $1 \leq X_1 \leq 5$, $0 \leq X_2 \leq X_1 - 1$.

In this example $\kappa = 1$, $\lambda = 2$, $L(\boldsymbol{x}) = 4 + 2X_1$, $U(\boldsymbol{x}) = 9 + X_1$, $P(\boldsymbol{x}) = 0$, $Q(\boldsymbol{x}) = X_1 - 1$. According to Theorem 3-3, since $1 \times 2 = 2 > 0$ and $\min((9 + X_1) - (4 + 2X_1) + (2 - 1)^{+} \times 0 - (2 - 1)^{-}(X_1 - 1) + 1) = \min(-X_1 + 6) = 1 \geq 1 = \min(|1|, |2|)$, it follows that the original variable integer interval can be reduced to $[4 + 2X_1 + 1^{+} \times 0 - 1^{-}(X_1 - 1), 9 + X_1 + 2^{+}(X_1 - 1) - 2^{-} \times 0] = [4 + 2X_1, 7 + 3X_1]$, $1 \leq X_1 \leq 5$, $0 \leq X_1 - 1$. It turns out from Example 3-1 that the variable integer interval $[4 + 2X_1 + X_2, 9 + X_1 + 2X_2]$, where $1 \leq X_1 \leq 5$, $0 \leq X_2 \leq X_1 - 1$ is contiguous and equal to the integer interval $[6, 22]$.

Theorem 3-4:

Consider the following variable interval equation

$$F(\boldsymbol{x}) = [L(\boldsymbol{x}) + \kappa X, U(\boldsymbol{x}) + \lambda X]$$

subject to a set of constraints on $\boldsymbol{x}$ in $\mathfrak{R}$ and the constraint $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$, where X does not appear in any of the constraints in $\mathfrak{R}$. If $\kappa\lambda \leq 0$, or $\kappa\lambda > 0$ and

$$\min(U(\boldsymbol{x}) - L(\boldsymbol{x}) + (\lambda - \kappa)^{+} P(\boldsymbol{x}) - (\lambda - \kappa)^{-} Q(\boldsymbol{x}) + 1) \geq \min(|\kappa|, |\lambda|),$$

then the above equation is integer solvable iff the variable interval equation

$$F(\boldsymbol{x}) = [L(\boldsymbol{x}) + \kappa^{+} P(\boldsymbol{x}) - \kappa^{-} Q(\boldsymbol{x}), U(\boldsymbol{x}) + \lambda^{+} Q(\boldsymbol{x}) - \lambda^{-} P(\boldsymbol{x})]$$

is integer solvable subject to the same constraints on $\boldsymbol{x}$ in $\mathfrak{R}$ and the constraint $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$.

Proof: The proof is straightforward from Theorem 3-3.

∎

3.1.3 Basic Transformations

The following theorem provides the basic transformation in the VI-Test. This transformation is repeatedly applied and eliminates one variable at a time from an interval equation and thus reducing the size of the problem.

Theorem 3-5:

Consider the following variable interval equation

$$F(x) + aX = [L(x) + bX, U(x) + cX]$$

subject to a set of constraints on x in $\Re$ and the constraint $P(x) \le X \le Q(x)$, where X does not appear in any of the constraints in $\Re$. If $(b - a)(c - a) \le 0$, or $(b - a)(c - a) > 0$ and

$$\min(U(x) - L(x) + (c - b)^+ P(x) - (c - b)^- Q(x) + 1) \ge \min(|b - a|, |c - a|),$$

then the equation above is integer solvable iff the variable interval equation

$$F(x) = [L(x) + (b - a)^+ P(x) - (b - a)^- Q(x), U(x) + (c - a)^+ Q(x) - (c - a)^- P(x)]$$

is integer solvable subject to the same constraints on x in $\Re$ and the constraint $P(x) \le Q(x)$.

Proof: Follows directly from Theorem 3-1 and Theorem 3-4.

$\blacksquare$

Example 3-3:

Consider the variable interval equation

$$X_1 + X_2 + X_3 = [10 - 2X_3, 15 - X_2], \text{ where } 1 \le X_1 \le 5, 0 \le X_2 \le X_1 - 1, X_1 \le X_3 \le X_1 + X_2$$

We apply Theorem 3-5 in order to eliminate variable X_3.

In this example $a = 1, b = -2, c = 0, L(x) = 10, U(x) = 15 - X_2, P(x) = X_1, Q(x) = X_1 + X_2$.

Since $(-2 - 1)(0 - 1) = 3 > 0$ and $\min((15 - X_2) - 10 + (0 - (-2))^+ X_1 - (0 - (-2))^- (X_1 + X_2) + 1) = \min(2X_1 - X_2 + 6) = \min(2X_1 - (X_1 - 1) + 6) = \min(X_1 + 7) = 8 \ge 1 = \min(|-2 - 1|, |0 - 1|)$,

we can eliminate variable X_3 and replace the original variable interval equation with the following

$$X_1 + X_2 = [10 + (-2 - 1)^+ X_1 - (-2 - 1)^- (X_1 + X_2), 15 - X_2 + (0 - 1)^+ (X_1 + X_2) - (0 - 1)^- X_1] =$$
$$[10 - 3X_1 - 3X_2, 15 - X_1 - X_2], \text{ where } 1 \le X_1 \le 5, 0 \le X_2 \le X_1 - 1, X_1 \le X_1 + X_2$$

Note that the last inequality constraint is implied by the second inequality and so it is redundant in this case.

The applicability of Theorem 3-5 depends on the size of the coefficients in the variable interval equation. In case the size of the coefficients is not small enough, the following transformation can reduce the size of the coefficients in the variable interval equation, potentially enabling the application of Theorem 3-5.

Theorem 3-6:

Consider a linear variable interval equation of the following form

$$a_1X_1 + \ldots + a_nX_n = [L + b_1X_1 + \ldots + b_nX_n, U + c_1X_1 + \ldots + c_nX_n]$$

Let $d = \gcd(a_1 - b_1, \ldots, a_n - b_n, a_1 - c_1, \ldots, a_n - c_n)$. Consider an integer sequence $v_i = b_i$ or $v_i = c_i$, $1 \leq i \leq n$. The above linear variable interval equation is integer solvable subject to any constraints iff the following linear variable interval equation

$$(a_1 - v_1)/dX_1 + \ldots + (a_n - v_n)/dX_n = [\lceil L/d \rceil + (b_1 - v_1)/dX_1 + \ldots + (b_n - v_n)/dX_n,$$
$$\lfloor U/d \rfloor + (c_1 - v_1)/dX_1 + \ldots + (c_n - v_n)/dX_n]$$

is integer solvable subject to the same constraints.

Proof: See Appendix.

■

After the transformation of Theorem 3-6 has been applied we check if the variable interval equation is still feasible. If max ($\lfloor U/d \rfloor - \lceil L/d \rceil + (c_1 - b_1)/dX_1 + (c_2 - b_2)/dX_2 + \ldots + (c_n - b_n)/dX_n$) < 0, then no integer solution exists for the above linear variable interval equation subject to the problem constraints, since the variable integer interval in the right hand side of the variable interval equation is empty

3.2 The Variable Interval Test in Data Dependence Analysis

In the following subsections we demonstrate how we can apply the variable integer interval theory in order to handle loops with trapezoidal bounds and symbolic variables when testing for data dependence under any direction vector constraints.

3.2.1 Handling Trapezoidal Regions

Trapezoidal loop regions are nested loops with bounds that are linear functions of the outer loop indices. These regions are difficult to handle and only computationally expensive techniques such as the Fourier-Motzkin variable elimination algorithm can properly analyze them. The case of

trapezoidal loop bounds is a generalization of the case of triangular loop bounds. Trapezoidal loops in the general case have the following form:

```
for I₁ = p₁,₀ to q₁,₀ do
  for I₂ = p₂,₀ + p₂,₁*I₁ to q₂,₀ + q₂,₁*I₁ do
    •
    •
    for Iₙ = pₙ,₀ + pₙ,₁*I₁ + … + pₙ,ₙ₋₁*Iₙ₋₁ to qₙ,₀ + qₙ,₁*I₁ + … + qₙ,ₙ₋₁*Iₙ₋₁
    do
S₁:       A[b₀ + b₁*I₁ + … + bₙ*Iₙ] = …
S₂:                       … = A[c₀ + c₁*I₁ + … + cₙ*Iₙ]
    endfor
    •
    •
  endfor
endfor
```

For the above trapezoidal loop region the data dependence problem between two instances $(I_1, I_2, \ldots, I_n)$ of S_1 and $(I'_1, I'_2, \ldots, I'_n)$ of S_2 can be formulated as follows:

$$\sum_{i=1}^{2n} a_i X_i = a_0 \tag{3-1}$$

$$p_{k,0} + \sum_{i=1}^{k-1} p_{k,i} X_{2i-1} \quad \leq X_{2k-1} \leq q_{k,0} + \sum_{i=1}^{k-1} q_{k,i} X_{2i-1} \tag{3-2}$$

$$p_{k,0} + \sum_{i=1}^{k-1} p_{k,i} X_{2i} \quad \leq X_{2k} \leq q_{k,0} + \sum_{i=1}^{k-1} q_{k,i} X_{2i} \quad , \qquad 1 \leq k \leq n,$$

where $X_{2k-1} = I_k$ and $X_{2k} = I'_k$, are two instances of the same loop iteration variable, and $a_{2k-1} = b_k$, $a_{2k} = -c_k$, for $k = 1, 2, \ldots, n$ and $a_0 = c_0 - b_0$. Alternatively, we can simplify the notation of the inequality constraints in (3-2) as follows:

$$\begin{aligned} P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq Q_{2k-1}(\boldsymbol{x}) \qquad & 1 \leq k \leq n \qquad (3\text{-}3) \\ P_{2k}(\boldsymbol{x}) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x}) \qquad & \boldsymbol{x} = (X_1, X_2, \ldots, X_{2n-1}, X_{2n}), \end{aligned}$$

where

$$P_{2k-1}(\boldsymbol{x}) = p_{k,0} + \sum_{i=1}^{k-1} p_{k,i} X_{2i-1} \quad , \qquad Q_{2k-1}(\boldsymbol{x}) = q_{k,0} + \sum_{i=1}^{k-1} q_{k,i} X_{2i-1}$$

$$P_{2k}(\boldsymbol{x}) = p_{k,0} + \sum_{i=1}^{k-1} p_{k,i} X_{2i} \quad , \qquad Q_{2k}(\boldsymbol{x}) = q_{k,0} + \sum_{i=1}^{k-1} q_{k,i} X_{2i}$$

In order to apply the VI-Test algorithm we rewrite equation (3-1) in the following trivial form of a variable interval equation:

$$\sum_{i=1}^{2n} a_i X_i = [a_0, a_0] \qquad\qquad (3\text{-}4)$$

Starting with an equation of the above form we repeatedly apply Theorem 3-5, each time reducing the number of variables in the system by one. Note that Theorem 3-5 requires that the variable to be eliminated does not appear in any constraints other than the equation and the constraints defining its bounds. This means that we must proceed by eliminating variables from the highest to the lowest index. Thus, the first variable to be eliminated should be either X_{2n} or X_{2n-1}. However, if there exist other variables that do not appear in loop bound expressions they can be eliminated in any order. The order of elimination is a partial order and can be determined by applying a graph algorithm as illustrated in 3.3. If we have successfully eliminated all variables, then we end up with zero in the left-hand side of the variable interval equation and with an integer interval on the right-hand side. If the integer interval includes zero, then the algorithm concludes that there exists an integer solution to the equation in (3-1) subject to the constraints in (3-2). Furthermore, the linear expression in the left-hand side of the equation realizes every integer value between its minimum and its maximum. If zero lies outside the integer interval, we conclude that no integer solution exists to the equation in (3-1) subject to the constraints in (3-2).

One of the conditions that have to be met, according to Theorem 3-5, in order to successfully eliminate all variables in the equation is the following.

Accuracy Condition 1:

For every variable X_i, eliminated from the variable interval equation

$$F(x) + aX = [L(x) + bX, U(x) + cX]$$

subject to a set of constraints on x in $\Re$ and the constraint $P(x) \leq X \leq Q(x)$, where X does not appear in any of the constraints in $\Re$ the following inequalities need to be satisfied:

$(b-a)(c-a) \leq 0$, or $(b-a)(c-a) > 0$ and
$\min(U(x) - L(x) + (c-b)^{+}P(x) - (c-b)^{-}Q(x) + 1) \geq \min(|b-a|, |c-a|)$

After we eliminate variable X, according to Theorem 3-5, the constraint $P(x) \leq Q(x)$ still remains in the system. This constraint complicates the problem because it may involve variables that appear in other constraints too, making it impossible to reapply Theorem 3-5 for those variables. We would like this constraint to be eliminated together with the variable X. In order to do so we need to prove this

31

inequality for all values of x subject to the constraints in (3-3). In general when comparing two such expressions to prove that $P(x) \leq Q(x)$, it suffices to show that the maximum value realized by $P(x)$ is less or equal than the minimum value realized by $Q(x)$, i.e. max($P(x)$) $\leq$ min($Q(x)$). A more efficient way to perform this comparison is to compute the minimum value of their difference, min($Q(x)$ – $P(x)$). If this minimum value is greater or equal than zero, then we conclude that $Q(x)$ is always greater or equal than $P(x)$. Alternatively we could compute max($P(x)$ – $Q(x)$) and check if this value is less or equal than zero.

In order to compute the minimum value realized by an expression we apply a variable substitution algorithm. Each variable in the expression is substituted by its lower bound if its coefficient is positive or by its upper bound if its coefficient is negative. The maximum value realized by an expression is computed similarly. The details of the variable substitution algorithm are described in Section 3.3.

In summary in order to accurately eliminate a variable from a variable interval equation, the following additional accuracy condition has to be met.

Accuracy Condition 2:

For every variable X, eliminated from the variable interval equation

$$F(x) + aX = [L(x) + bX, U(x) + cX]$$

subject to a set of constraints on x in $\Re$ and the constraint $P(x) \leq X \leq Q(x)$, where X does not appear in any of the constraints in $\Re$ the following inequality needs to be satisfied:

$$\min(Q(x) - P(x)) \geq 0$$

What Accuracy Condition 2 states, is that the upper bound of each loop variable should always be greater or equal than the lower bound. In most cases the above inequality will hold because in actual source code each loop executes at least once for each iteration of the outermost loops. In the unlike event that this condition is not satisfied, we will see in Section 3.2.3 how we can handle this case. We illustrate the application of the VI-Test algorithm in trapezoidal regions with the following example:

Example 3-4:

```
for I = 1 to 10 do
    for J = 10 - I to 2*I + 7 do
S₁:    A[3*I + J] = …
S₂:            … = A[J - 2*I + 20] + …
    endfor
endfor
```

There is data dependence between the two array references if the following system has an integer solution:

$$3X_1 + 2X_2 + X_3 - X_4 = 20$$
$$1 \leq X_1 \leq 10, \qquad\qquad 1 \leq X_2 \leq 10,$$
$$10 - X_1 \leq X_3 \leq 2X_1 + 7, \qquad 10 - X_2 \leq X_4 \leq 2X_2 + 7.$$

where X_1, X_2 are instances of the loop variable I and X_3, X_4 are instances of the loop variable J.

We write the above equation in its variable interval form:

$$3X_1 + 2X_2 + X_3 - X_4 = [20, 20]$$

The only variables that may be eliminated, according to Theorem 3-5, are X_3 and X_4. We may start with X_4.

In this case $a = -1$, $b = 0$, $c = 0$, $L(\boldsymbol{x}) = 20$, $U(\boldsymbol{x}) = 20$, $P(\boldsymbol{x}) = 10 - X_2$, $Q(\boldsymbol{x}) = 2X_2 + 7$.

Checking Accuracy Condition 1: $(0 - (-1))\,(0 - (-1)) = 1 > 0$, and

$\min(20 - 20 + 0(10 - X_2) - 0(2X_2 + 7) + 1) = 1 \geq 1 = \min(|0 - (-1)|, |0 - (-1)|)$.

Checking Accuracy Condition 2: $\min((2X_2 + 7) - (10 - X_2)) = \min(3X_2 - 3) = 0 \geq 0$.

Since both accuracy conditions are satisfied we can indeed eliminate X_4 and the variable interval equation is transformed into:

$$3X_1 + 2X_2 + X_3 = [20 + (0 - (-1))\,(10 - X_2),\ 20 + (0 - (-1))\,(2X_2 + 7)] \Leftrightarrow$$
$$3X_1 + 2X_2 + X_3 = [-X_2 + 30,\ 2X_2 + 27]$$

We continue by eliminating variable X_3.

In this case, $a = 1$, $b = 0$, $c = 0$, $L(\boldsymbol{x}) = -X_2 + 30$, $U(\boldsymbol{x}) = 2X_2 + 27$, $P(\boldsymbol{x}) = 10 - X_1$, $Q(\boldsymbol{x}) = 2X_1 + 7$.

Checking Accuracy Condition 1: $(0 - 1)(0 - 1) = 1 > 0$, and

$\min((2X_2 + 27) - (-X_2 + 30) + 0(\,10 - X_1) - 0(\,2X_1 + 7) + 1) = \min(3X_2 - 2) = 1 \geq 1 = \min(|0 - 1|, |0 - 1|)$.

Checking Accuracy Condition 2: $\min((2X_1 + 7) - (10 - X_1)) = \min(3X_1 - 3) = 0 \geq 0$.

Since both conditions are satisfied we can eliminate X_3 and the derived variable interval equation is as follows:

$$3X_1 + 2X_2 = [(-X_2 + 30) + (0 - 1)(2X_1 + 7),\ (2X_2 + 27) + (0 - 1)(\,10 - X_1)] \Leftrightarrow$$
$$3X_1 + 2X_2 = [-2X_1 - X_2 + 23,\ X_1 + 2X_2 + 17]$$

Next we eliminate variable X_2.

In this case, $a = 2$, $b = -1$, $c = 2$, $L(\boldsymbol{x}) = -2X_1 + 23$, $U(\boldsymbol{x}) = X_1 + 17$, $P(\boldsymbol{x}) = 1$, $Q(\boldsymbol{x}) = 10$.

Checking Accuracy Condition 1: $(-1 - 2)(2 - 2) = 0 \leq 0$.

Checking Accuracy Condition 2: $\min(10 - 1) = 9 \geq 0$

Since both conditions are satisfied the new variable interval equation derived after eliminating X_2 is:

$3X_1 = [(-2X_1 + 23) + (-1 - 2)) \times 10, (X_1 + 17) + 0] \Leftrightarrow$

$3X_1 = [-2X_1 - 7, X_1 + 17]$

Finally, we eliminate variable X_1.

In this case, $a = 3$, $b = -2$, $c = 1$, $L(\boldsymbol{x}) = -7$, $U(\boldsymbol{x}) = 17$, $P(\boldsymbol{x}) = 1$, $Q(\boldsymbol{x}) = 10$.

Checking Accuracy Condition 1: $(-2 - 3)(1 - 3) = 10 > 0$, and

$\min(17 - (-7) + (1 - (-2)) \times 1 + 1) = 28 \geq 2 = \min(|-2 - 3|, |1 - 3|)$

Checking Accuracy Condition 2: $\min(10 - 1) = 9 \geq 0$

After the final elimination we end up with:

$0 = [-7 + (-2 - 3) \times 10, 17 + (1 - 3) \times 1] = [-57, 15]$

The VI-Test concludes that an integer solution to the variable interval equation subject to the constraints indeed exists. The values $-(15 - 20) = 5$, and $-(-57 - 20) = 77$ are the extreme values of the expression in the left-hand side of the original equation. Furthermore, the expression realizes every integer value between those values.

3.2.2 Handling Trapezoidal Regions with Direction Vectors

Compiler transformations for program optimization and parallelization require additional information with regard to the relative loop iterations in which the related (dependent) instances occur. Such information is summarized in direction vectors [37]. When testing for data dependence subject to direction vectors, additional constraints ($<$, $=$, or $>$) are introduced in the dependence system between any two instances of the same loop iteration variable. We will examine each relation separately.

First consider the less "$<$" direction imposed on variables X_{2k-1}, X_{2k}, the two instances of the same loop iteration variable I_k. The constraints in (3-3) become:

$$\begin{aligned} P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq Q_{2k-1}(\boldsymbol{x}) \\ P_{2k}(\boldsymbol{x}) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x}) \\ X_{2k-1} < X_{2k} \end{aligned} \qquad (3\text{-}5)$$

In the above case the direction vector constraint introduces additional bounds for either variable X_{2k-1} or X_{2k}. We can incorporate the last inequality in the other constraints in one of the following two ways:

$$P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq Q_{2k-1}(\boldsymbol{x}) \qquad\qquad P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq \min(Q_{2k-1}(\boldsymbol{x}), X_{2k}-1)$$
$$\max(P_{2k}(\boldsymbol{x}), X_{2k-1}+1) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x}) \quad \text{OR} \quad P_{2k}(\boldsymbol{x}) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x})$$

Each case provides a different order of elimination for the pair of variables X_{2k-1}, X_{2k}. In the fist case we eliminate variable X_{2k} first, since it does not appear in any other constraints as Theorem 3-5 requires. In the second case we eliminate variable X_{2k-1} first for the same reason. However, the above constraints contain the *min* or the *max* functions that are hard to handle when performing algebraic operations. We would like to eliminate those functions in order to enable the application of Theorem 3-5. In the first case we can eliminate the *max* function, if $X_{2k-1}+1$ is always greater or equal than $P_{2k}(\boldsymbol{x})$. By proving the inequality $P_{2k}(\boldsymbol{x}) \leq P_{2k-1}(\boldsymbol{x})+1$, it follows that $P_{2k}(\boldsymbol{x}) \leq X_{2k-1}+1$, since $P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1}$. Therefore, in this case $X_{2k-1}+1$ is the tightest lower bound for variable X_{2k}. Note that the inequality $P_{2k}(\boldsymbol{x}) \leq P_{2k-1}(\boldsymbol{x})+1$ is always true when the lower bound is constant (i.e. triangular loop bounds). Most compilers, including the Polaris compiler, perform normalization before the data dependence analysis phase so all trapezoidal bounds are transformed into triangular.

In the unlike case where normalization is not performed and the inequality $P_{2k}(\boldsymbol{x}) \leq P_{2k-1}(\boldsymbol{x})+1$ is not satisfied we may consider the second set of constraints. In this case we can eliminate the *min* function, if $X_{2k}-1$ is always less or equal than $Q_{2k-1}(\boldsymbol{x})$. By proving the inequality $Q_{2k-1}(\boldsymbol{x}) \geq Q_{2k}(\boldsymbol{x})-1$, it follows that $Q_{2k-1}(\boldsymbol{x}) \geq X_{2k}-1$, since $Q_{2k}(\boldsymbol{x}) \geq X_{2k}$. Here $X_{2k}-1$ is the tightest upper bound for variable X_{2k-1}. Note that the inequality $Q_{2k-1}(\boldsymbol{x}) \geq Q_{2k}(\boldsymbol{x})-1$ is always true when the upper bound is loop invariant. In case that neither of the inequalities $P_{2k}(\boldsymbol{x}) \leq P_{2k-1}(\boldsymbol{x})+1$, $Q_{2k-1}(\boldsymbol{x}) \geq Q_{2k}(\boldsymbol{x})-1$ is satisfied, we will see in Section 3.2.3 how we can satisfy either of them by restricting the data dependence problem domain.

Depending on which of the above two inequalities is satisfied, the variable bounds can be defined as follows:

$$P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq Q_{2k-1}(\boldsymbol{x}) \qquad\qquad P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq X_{2k}-1$$
$$X_{2k-1}+1 \leq X_{2k} \leq Q_{2k}(\boldsymbol{x}) \qquad \text{OR} \qquad P_{2k}(\boldsymbol{x}) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x})$$

In the first case the constraint $X_{2k-1}+1 \leq Q_{2k}(\boldsymbol{x})$, derived from the bounds of X_{2k}, introduces an additional upper bound on variable X_{2k-1}. In the second case the constraint $P_{2k-1}(\boldsymbol{x}) \leq X_{2k}-1$, derived from the bounds of X_{2k-1}, introduces an additional lower bound on variable X_{2k}. In order for the VI-

$v_k = $ "<"			
Conditions	**Bounds**	**Conditions**	**Bounds**
$P_{2k}(x) \le P_{2k-1}(x) + 1$		$Q_{2k-1}(x) \ge Q_{2k}(x) - 1$	
$Q_{2k-1}(x) \ge Q_{2k}(x) - 1$	$P_{2k-1}(x) \le X_{2k-1} \le Q_{2k}(x)-1$ $X_{2k-1} + 1 \le X_{2k} \le Q_{2k}(x)$	$P_{2k}(x) \le P_{2k-1}(x) + 1$	$P_{2k-1}(x) \le X_{2k-1} \le X_{2k} - 1$ $P_{2k-1}(x) + 1 \le X_{2k} \le Q_{2k}(x)$
$Q_{2k-1}(x) \le Q_{2k}(x) - 1$	$P_{2k-1}(x) \le X_{2k-1} \le Q_{2k-1}(x)$ $X_{2k-1} + 1 \le X_{2k} \le Q_{2k}(x)$	$P_{2k}(x) \ge P_{2k-1}(x) + 1$	$P_{2k-1}(x) \le X_{2k-1} \le X_{2k} - 1$ $P_{2k}(x) \le X_{2k} \le Q_{2k}(x)$
$v_k = $ ">"			
Conditions	**Bounds**	**Conditions**	**Bounds**
$P_{2k-1}(x) \le P_{2k}(x) + 1$		$Q_{2k}(x) \ge Q_{2k-1}(x) - 1$	
$Q_{2k}(x) \ge Q_{2k-1}(x) - 1$	$X_{2k} + 1 \le X_{2k-1} \le Q_{2k-1}(x)$ $P_{2k}(x) \le X_{2k} \le Q_{2k-1}(x) - 1$	$P_{2k-1}(x) \le P_{2k}(x) + 1$	$P_{2k}(x) + 1 \le X_{2k-1} \le Q_{2k-1}(x)$ $P_{2k}(x) \le X_{2k} \le X_{2k-1} - 1$
$Q_{2k}(x) \le Q_{2k-1}(x) - 1$	$X_{2k} + 1 \le X_{2k-1} \le Q_{2k-1}(x)$ $P_{2k}(x) \le X_{2k} \le Q_{2k}(x)$	$P_{2k-1}(x) \ge P_{2k}(x) + 1$	$P_{2k-1}(x) \le X_{2k-1} \le Q_{2k-1}(x)$ $P_{2k}(x) \le X_{2k} \le X_{2k-1} - 1$
$v_k = $ "="			
Conditions	**Bounds**	**Conditions**	**Bounds**
$P_{2k}(x) \le P_{2k-1}(x)$		$P_{2k}(x) \ge P_{2k-1}(x)$	
$Q_{2k}(x) \ge Q_{2k-1}(x)$	$P_{2k-1}(x) \le X_{2k-1} \le Q_{2k-1}(x)$	$Q_{2k}(x) \ge Q_{2k-1}(x)$	$P_{2k}(x) \le X_{2k-1} \le Q_{2k-1}(x)$
$Q_{2k}(x) \le Q_{2k-1}(x)$	$P_{2k-1}(x) \le X_{2k-1} \le Q_{2k}(x)$	$Q_{2k}(x) \le Q_{2k-1}(x)$	$P_{2k}(x) \le X_{2k-1} \le Q_{2k}(x)$

Figure 3-1: Conditions to choose bounds for variables X_{2k-1}, X_{2k} under a direction v_k.

Test algorithm to be more accurate we incorporate the derived bounds into the original bounds. The resulting bounds will be the following.

$$P_{2k-1}(x) \le X_{2k-1} \le \min(Q_{2k-1}(x), Q_{2k}(x) - 1) \qquad P_{2k-1}(x) \le X_{2k-1} \le X_{2k} - 1$$
$$X_{2k-1} + 1 \le X_{2k} \le Q_{2k}(x) \qquad \text{OR} \qquad \max(P_{2k}(x), P_{2k-1}(x) + 1) \le X_{2k} \le Q_{2k}(x)$$

We would like to eliminate the *min* and *max* functions in each case in order to enable the application of Theorem 3-5. In the fist case we compare $Q_{2k-1}(x)$, $Q_{2k}(x) - 1$ and choose the tightest upper bound. Similarly in the second case we compare $P_{2k}(x)$, $P_{2k-1}(x) + 1$ and choose the tightest lower bound. After all the variable bounds have been selected, the VI-Test algorithm proceeds as before by eliminating one variable at a time from the interval equation while checking for Accuracy Condition 1 and Accuracy Condition 2.

The case of the greater ">" direction is similar. In the case of an equal "=" direction the two variables are substituted by one, lets say X_{2k-1}, in all problem constraints. Now variable X_{2k-1} has two lower bounds, $P_{2k-1}(x)$ and $P_{2k}(x)$, and two upper bounds $Q_{2k-1}(x)$ and $Q_{2k}(x)$. Again we would like

to choose the tightest for lower and upper bounds by comparing them with each other. In general, depending on what inequalities are satisfied the bounds can be chosen accordingly for any given direction. The table in Figure 3-1 depicts all the conditions that need to be satisfied in order to accurately determine the bounds for a pair of variables X_{2k-1}, X_{2k}, given a direction vector relation $X_{2k-1} \, v_k \, X_{2k}$.

Throughout the application of the VI-Test a number of inequalities between two expressions need to be checked for all possible values of the variables in the data dependence problem domain. These comparisons are performed as described in Section 3.2.1. In many cases we are able to prove these inequalities because in practice, when a direction v_k is refined (i.e. other than star "*"), then all directions v_i, for $i < k$ of the outer loops are also refined. This is because the algorithm employees the direction vector hierarchy [9] refining from the outermost to the innermost loop. Since constraints on the outer loop variables have already been imposed, we can more easily determine the relationship between any two expressions which are functions of those variables. If we are not able to satisfy any of the inequalities, then we indicate that we have made an approximation in case we can not disprove the dependence.

Example 3-5:

Using the same loop as in Example 3-4 we test for data dependence subject to direction vector $(<, <)$. The data dependence problem in this example results into the following system:

$3X_1 + 2X_2 + X_3 - X_4 = 20$

$1 \leq X_1 \leq 10,$	$1 \leq X_2 \leq 10,$	$X_1 < X_2,$
$10 - X_1 \leq X_3 \leq 2X_1 + 7,$	$10 - X_2 \leq X_4 \leq 2X_2 + 7,$	$X_3 < X_4.$

We fist determine the bounds for the variables.

$v_1 = \text{"}<\text{"}, \ P_1(\pmb{x}) = 1, \ P_2(\pmb{x}) = 1, \ Q_1(\pmb{x}) = 10, \ Q_2(\pmb{x}) = 10$

        Checking Condition $P_2(\pmb{x}) \leq P_1(\pmb{x}) + 1$:
        $\max(P_2(\pmb{x}) - P_1(\pmb{x}) - 1) = \max(1 - 1 - 1) = -1 \leq 0$

        Checking Condition $Q_1(\pmb{x}) \geq Q_2(\pmb{x}) - 1$:
        $\min(Q_1(\pmb{x}) - Q_2(\pmb{x}) + 1) = \min(10 - 10 + 1) = 1 \geq 0$

        In this case the bounds are:
            $1 \leq X_1 \leq 9,$
            $X_1 + 1 \leq X_2 \leq 10$

$v_2 = \text{"}<\text{"}, \ P_3(\pmb{x}) = 10 - X_1, \ P_4(\pmb{x}) = 10 - X_2, \ Q_3(\pmb{x}) = 2X_1 + 7, \ Q_4(\pmb{x}) = 2X_2 + 7$

        Checking Condition $P_4(\pmb{x}) \leq P_3(\pmb{x}) + 1$:
        $\max(P_4(\pmb{x}) - P_3(\pmb{x}) - 1) = \max((10 - X_2) - (10 - X_1) - 1) = \max(X_1 - X_2 - 1) =$
        $\max(X_1 - (X_1 + 1) - 1) = -2 \leq 0$

Checking Condition $Q_3(x) \geq Q_4(x) - 1$:
$\min(Q_3(x) - Q_4(x) + 1) = \min((2X_1 + 7) - (2X_2 + 7) + 1) = \min(2X_1 - 2X_2 + 1) = \min(2X_1 - 2 \times 10 + 1) = \min(2X_1 - 19) = \min(2 \times 1 - 19) = -17 \ngeq 0$

Checking Condition $Q_3(x) \leq Q_4(x) - 1$
$\max(Q_3(x) - Q_4(x) + 1) = \max((2X_1 + 7) - (2X_2 + 7) + 1) = \max(2X_1 - 2X_2 + 1) = \max(2X_1 - 2(X_1 + 1) + 1) = -1 \leq 0$

In this case the bounds are:
$$10 - X_1 \leq X_3 \leq 2X_1 + 7$$
$$X_3 + 1 \leq X_4 \leq 2X_2 + 7$$

So the above equation in its variable interval form and the final constraints are as follows:

$3X_1 + 2X_2 + X_3 - X_4 = [20, 20]$
$1 \leq X_1 \leq 9,$ $X_1 + 1 \leq X_2 \leq 10$
$10 - X_1 \leq X_3 \leq 2X_1 + 7,$ $X_3 + 1 \leq X_4 \leq 2X_2 + 7$

We start by eliminating variable X_4.

In this case $a = -1$, $b = 0$, $c = 0$, $L(x) = 20$, $U(x) = 20$, $P(x) = X_3 + 1$, $Q(x) = 2X_2 + 7$.

Checking Accuracy Condition 1: $(0 - (-1))(0 - (-1)) = 1 > 0$, and

$\min(20 - 20 + 0(X_3 + 1) - 0(2X_2 + 7) + 1) = 1 \geq 1 = \min(|0 - (-1)|, |0 - (-1)|)$.

Checking Accuracy Condition 2: $\min((2X_2 + 7) - (X_3 + 1)) = \min(2X_2 - X_3 + 6) =$

$\min(2X_2 - (2X_1 + 7) + 6) = \min(2X_2 - 2X_1 - 1) = \min(2(X_1 + 1) - 2X_1 - 1) = 1 \geq 0$

After eliminating X_4 the variable interval equation is transformed into:

$3X_1 + 2X_2 + X_3 = [20 + (0 - (-1))(X_3 + 1), 20 + (0 - (-1))(2X_2 + 7)] \Leftrightarrow$
$3X_1 + 2X_2 + X_3 = [X_3 + 21, 2X_2 + 27]$

We continue by eliminating variable X_3.

In this case $a = 1$, $b = 1$, $c = 0$, $L(x) = 21$, $U(x) = 2X_2 + 27$, $P(x) = 10 - X_1$, $Q(x) = 2X_1 + 7$.

Checking Accuracy Condition 1: $(1 - 1)(0 - 1) = 0 \leq 0$.

Checking for Accuracy Condition 2: $\min((2X_1 + 7) - (10 - X_1)) = \min(3X_1 - 3) = 0 \geq 0$.

After eliminating X_3 the variable interval equation is transformed into:

$3X_1 + 2X_2 = [21 - 0, (2X_2 + 27) + (0 - 1)(10 - X_1)] \Leftrightarrow$
$3X_1 + 2X_2 = [21, X_1 + 2X_2 + 17]$

The algorithm proceeds by eliminating X_2 and X_1 in that order. After terminating the resulting variable interval equation would be $0 = [-26, 15]$. The VI-Test concludes that a solution to the equation subject to constraints indeed exists. The values $-(15 - 20) = 5$, and $-(-26 - 20) = 46$ are the extreme

values of the expression in the left-hand side of the original equation. Furthermore, the expression realizes every integer value between those values, in the domain defined by the loop bounds and the direction vector constrains.

3.2.3 Relaxing the Accuracy Conditions

When applying the VI-Test algorithm to test for data dependence in loops with trapezoidal bounds a set of conditions have to be met in order to provide an exact answer. Each of these conditions entails that some inequality on a linear expression must be satisfied. In order to check any of these conditions we compute the minimum or the maximum value of the expression and check whether the inequality holds for that value, as illustrated in Section 3.2.1. It may be the case that some of these inequalities are not satisfied. In that case we may be able to restrict the problem domain in order to satisfy the inequality and possibly come up with an exact answer at the end.

Let us consider a condition requiring that the minimum of an expression is greater than a constant value. In the final steps of the variable substitution algorithm for the computation of the minimum value of the expression all the variables remaining are those that have constant bounds. In data dependence problems for trapezoidal loops there is at least one variable that has constant lower and upper bounds. After all variables, except those with constant bounds, have been eliminated the initial condition is reduced to an inequality of the following form:

$$\min(\sum_{i=1}^{m} a_i X_i) \geq c, \qquad P_i \leq X_i \leq Q_i, \qquad 1 \leq i \leq m.$$

If the minimum value of the left hand side expression is less than c, then we may be able to restrict the bounds of the variables in this expression in order to satisfy the inequality. The minimum value of the expression in this case needs to be increased by the following constant:

$$c - \sum_{i=1}^{m} (a_i^+ P_i - a_i^- Q_i).$$

In order to achieve this we must find positive integers $s_1, s_2, \ldots, s_m$ such that

$$\sum_{i=1}^{m} |a_i| s_i \geq c - \sum_{i=1}^{m} (a_i^+ P_i - a_i^- Q_i), \qquad s_i \leq Q_i - P_i$$

In this case we restrict the bounds of each variable X_i as follows

$$P_i' = \begin{cases} P_i + s_i & \text{if } a_i > 0 \\ P_i & \text{otherwise} \end{cases} \quad \text{and} \quad Q_i' = \begin{cases} Q_i - s_i & \text{if } a_i < 0 \\ Q_i & \text{otherwise} \end{cases}$$

39

and the original inequality will be satisfied in the restricted domain. The case of maximum is handled similarly. There may be several values s_i that satisfy the inequality. We are looking for the values that result into the minimum increase or decrease in the original bounds. We can find this assignment by sorting the terms of the expression by the absolute values $|a_i|$ of the coefficients in decreasing order and applying a greedy algorithm restricting the bounds of each variable until the condition is met. If we cannot find such an assignment for s_i the corresponding condition fails and the test becomes inexact.

When we restrict the bounds of a variable we lose some precision. In this case two problems are created. One problem is created for the actual bounds and one for the restricted bounds. The problem for the restricted bounds is a subset of the original problem. If a solution exists for the problem with the restricted bounds then a solution also exists for the original problem. In practice two variable integer intervals are created for each of the problems. When we apply the algorithm, the Accuracy Condition 1 and Accuracy Condition 2 are tested for the restricted bounds because only in this case we can give a definite yes answer. When the algorithm terminates two constant integer intervals remain in the right hand side. The first interval has the bounds that would be computed if we applied the simple substitution algorithm using the original bounds. These bounds may not be extreme values for the expression in the left hand side of the equation. If zero is outside this interval then no solution exists to the system and therefore there is no dependence. The second interval represents a set of consecutive integer values that the expression on the left-hand side of the equation realizes. If zero lies in that interval then a solution indeed exists. If zero is outside the second interval but inside the first interval then the VI-Test becomes inconclusive and reports a maybe answer.

Example 3-6:

Consider the same loop as in Example 3-4. We test for data dependence subject to direction vector (>, <). After selecting the bounds for the variables, in a similar manner as in Example 3-5, the data dependence problem results into the following system:

$3X_1 + 2X_2 + X_3 - X_4 = [20, 20]$
$X_2 + 1 \leq X_1 \leq 10, \qquad\qquad 1 \leq X_2 \leq 9$
$10 - X_1 \leq X_3 \leq X_4 - 1, \qquad\qquad 10 - X_2 \leq X_4 \leq 2X_2 + 7$

We start by eliminating variable X_3.

In this case $a = 1$, $b = 0$, $c = 0$, $L(x) = U(x) = 20$, $P(x) = 20 - X_1$, $Q(x) = X_4 - 1$.

Checking Accuracy Condition 1: $(0 - 1)(0 - 1) = 1 > 0$, and

$\min(20 - 20 + 0(10 - X_1) - 0(X_4 - 1) + 1) = 1 \geq 1 = \min(|0 - 1|, |0 - 1|)$.

Checking Accuracy Condition 2: $\min((X_4 - 1) - (10 - X_1)) = \min(X_1 + X_4 - 11) =$

$\min(X_1 + (10 - X_2) - 11) = \min(X_1 - X_2 - 1) = \min((X_2 + 1) - X_2 - 1) = 0 \geq 0$

After eliminating X_3 the variable interval equation is transformed into:

$3X_1 + 2X_2 - X_4 = [20 + (0 - 1)(X_4 - 1), 20 + (0 - 1)(10 - X_1)] \Leftrightarrow$

$3X_1 + 2X_2 - X_4 = [-X_4 + 21, X_1 + 10]$

We proceed by eliminating variable X_4.

In this case $a = -1$, $b = -1$, $c = 0$, $L(\boldsymbol{x}) = 21$, $U(\boldsymbol{x}) = X_1 + 10$, $P(\boldsymbol{x}) = 10 - X_2$, $Q(\boldsymbol{x}) = 2X_2 + 7$.

Checking Accuracy Condition 1: $(-1 - (-1))(0 - (-1)) = 0 \leq 0$.

Checking Accuracy Condition 2: $\min((2X_2 + 7) - (10 - X_2)) = \min(3X_2 - 3) = 0 \geq 0$

After eliminating X_4 the variable interval equation is transformed into:

$3X_1 + 2X_2 = [21 + 0, X_1 + 10 + (0 - (-1))(2X_2 + 7)] \Leftrightarrow$

$3X_1 + 2X_2 = [21, X_1 + 2X_2 + 17]$

Next variable to be eliminated is X_1.

In this case $a = 3$, $b = 0$, $c = 1$, $L(\boldsymbol{x}) = 21$, $U(\boldsymbol{x}) = 2X_2 + 17$, $P(\boldsymbol{x}) = X_2 + 1$, $Q(\boldsymbol{x}) = 10$.

Checking Accuracy Condition 1: $(0 - 3)(1 - 3) = 6 > 0$, and

$\min((2X_2 + 17) - 21 + (1 - 0)(X_2 + 1) + 1) = \min(3X_2 - 2) = 1 \not\geq 2 = \min(|0 - 3|, |1 - 3|)$.

Note that Accuracy Condition 1 fails here and also the variable interval GCD transformation (Theorem 3-6) fails because the gcd is 1. However, we can satisfy the condition by restricting the lower bound of variable X_2. The new bounds will be $(1, 2) \leq X_2 \leq 9$. The original bound remains the same equal to 1 and the restricted bound is 2.

Checking Accuracy Condition 2: $\min(10 - (X_2 + 1)) = \min(-X_2 + 9) = 0 \geq 0$

Now we eliminate X_1:

$2X_2 = [21 + (0 - 3) \times 10, 2X_2 + 17 + (1 - 3)(X_2 + 1)] \Leftrightarrow$

$2X_2 = [-9, 15]$

In the final elimination of X_2 all conditions of accuracy are satisfied. Variable X_2 has two sets of bounds that will produce two integer intervals:

$0 = ([-27, 13], [-27, 11])$

The VI-Test concludes that a solution to the equation subject to constraints indeed exists, since zero lies within the bounds of the second interval.

3.2.4 Handling Symbolic Variables

Symbolic variables are program variables which are not loop indices and they may appear in subscript expressions and loop bounds. When testing for data dependence in this case we should determine whether there exist values for the loop index variables and the symbolic variables satisfying the constraints of the problem. The data dependence problem with symbolic variables is equivalent to the trapezoidal data dependence problem. The symbolic variables appear in the data dependence system and we set their lower and upper bounds to minus and plus infinity respectively, unless we can derive a tighter bound from the other problem constraints. The plus and minus infinity in the actual implementation of the test can be substituted by a very large positive and a very small negative value. Consider the following example:

Example 3-7:
```
      for   I = 1 to 2*N do
S:          A[I + N] = A[I - N] + …
      endfor
```

Note that the loop does not iterate for values of *N* less than one and, therefore, we adjust the lower bound of variable *N* accordingly. The data dependence system is as follows:

$2N + X_1 - X_2 = [0, 0]$
$1 \le N \le +\infty$
$1 \le X_1 \le 2N, \quad 1 \le X_2 \le 2N$

The VI-Test eliminates all variables and terminates with $0 = [-\infty, -1]$. This indicates that no data dependence exists. Most of the conventional data dependence tests that do not perform symbolic expression operations would fail to disprove that dependence.

3.2.5 Handling Coupled Subscripts

Coupled array subscripts introduce equations in the dependence system that share common variables. Therefore, considering one equation at a time when testing for data dependence introduces approximations. In this section we introduce a simple technique that seems to work well in practice when coupled subscripts occur in the dependence problem even if they are non-linear. The algorithm proceeds by propagating each equation into the rest of the equations, starting from the one with the fewest terms. The propagation of one equation into a second one is similar to Gaussian elimination, i.e. we multiply the two equations by a constant and add them together producing a new equation. If the new equation has fewer terms than the second one, then the new equation replaces the original equation and the algorithm proceeds with the next pair of equations. This method can eliminate the

coupling of variables across equations or can reduce the coupling to a minimum number of variables. Consider the following loop:

Example 3-8:
```
        for I = 1 to 10 do
           for J = 1 to I do
S:            A[2*I + J, I] = A[2*J, J] + …
            endfor
        endfor
```

The data dependence problem results into the following system:

$2X_1 + X_3 - 2X_4 = 0$
$X_1 - X_4 = 0$
$1 \le X_1 \le 10, \qquad 1 \le X_2 \le 10$
$1 \le X_3 \le X_1, \qquad 1 \le X_4 \le X_2$

By propagating the second equation into the first, multiplying by -2 and 1 respectively, and adding them together we produce a new equation $X_3 = 0$. The new equation will replace the original equation and the dependence system will be transformed as follows:

$X_3 = 0$
$X_1 - X_4 = 0$
$1 \le X_1 \le 10, \qquad 1 \le X_2 \le 10$
$1 \le X_3 \le X_1, \qquad 1 \le X_4 \le X_2$

The fist equation now does not have a solution subject to the bounds of X_3, and thus we conclude that no dependence exists. Subscript by subscript testing without applying the equation propagation would have failed to disprove the dependence in this case. When the coupling between equations cannot be eliminated by equation propagation, the VI-Test generates and tests additional equations according to the lambda test algorithm [22]. According to this method for every two coupled equations we consider the linear combinations between the two equations that eliminate at least one of their common terms. This method cannot be used to prove integer solutions in the system but it can help disprove additional dependences.

3.2.6 Handling If-Statements

In addition to the subscript equations and the loop bound inequalities, the constraints derived from the conditional part of enclosing *if* and *else* statements need to be taken into account in order to accurately test for data dependence. Frequently if-statement conditions are loop variant and contain variables that appear in the loop bounds and the array subscripts. The conditions of if-statements are logical expressions and may contain any type of relational or logical operator. In this case a

dependence test must be able to analyze logical expressions and solve the integer constraint satisfaction problem in its most general form.

The Omega test can take into account logical and relational operators in if-statement conditions. However, this is only limited to linear integer constraints and it comes with a significant cost in efficiency as will see in Section 5.3. The VI-Test algorithm can handle simple cases of if-statement constraints, which are often found in source code. If the two array statements tested for data dependence belong to the if-part and else-part respectively and the if-statement condition is loop invariant, then the algorithm determines independence since only one of the two array statements executes for all iterations of the enclosing loops. Otherwise, the algorithm checks if the variables in the conditional part of the if-statement also appear in the loop bound expressions or the array subscripts. If this is the case then the constraints are added to the dependence system or else they are ignored.

For each equality constraint added in the dependence system:

- If the equality does not include loop variables and has a linear term, then we solve the equation for that term's variable and substitute it in the rest of the dependence system.
- In any other case a new equation is added and tested in the dependence system. In this case the equation propagation technique described in Section 3.2.5 may propagate the equality in the rest of the problem.

For each inequality constraint added in the dependence system:

- If the inequality involves a single variable then the bounds of that variable are updated.
- Otherwise, the VI-Test checks whether the inequality is implied or contradicts the rest of the dependence constraints by computing the minimum and the maximum of the inequality expression. If the inequality contradicts the other constraints, then the algorithm returns independence. If the inequality is implied by the other constraints, then the algorithm can accurately ignore it.
- If the inequality neither contradicts nor is implied by the other constraints, then the VI-Test may restrict the problem domain, as described in Section 3.2.3, in order to satisfy the inequality.
- If all the above fail then the VI-Test becomes inaccurate.

The above techniques are simple but they work well in practice and can increase the accuracy of the VI-Test in cases with if-statement constraints, as we will see in Section 5.2. Consider the following example:

Example 3-9:

```
for I = 1 to 200 do
  if (I < 100) then
      A[I] = …
  else
      A[I + 1] = …
  endif
endfor
```

Taking into account the if-statement constraints in addition to the loop bounds and array subscripts, the data dependence problem results into the following system:

$$X_1 - X_2 = 1$$
$$1 \leq X_1 \leq 99, \qquad 100 \leq X_2 \leq 200$$

The VI-Test can easily determine that no dependence exists given the updated bounds of the loop variables. Ignoring the if-statement condition would have failed to disprove the dependence in this case.

3.3 The Algorithm

In this section we describe the VI-Test algorithm. Before the application of the VI-Test a data dependence problem is created for a pair of array references within a nest of loops and a nest of if-statements. After the dependence problem is created a sequence of direction vectors are tested on the problem to verify the existence of data dependence. First the bounds of each variable are set based on the loop bounds, the if-statement constraints, and the direction vector information as described in Sections 3.2.1, 3.2.2, and 3.2.6. The lower and upper bounds of each variable can be expressions of other variables, i.e. trapezoidal or symbolic bounds. All variables with constant bounds have two values for each bound. One is the original constant loop bound and the other is the restricted bound. Initially these are equal. The values of the restricted bounds change every time we restrict the problem domain in order to meet the accuracy conditions, if necessary, as described in Section 3.2.3. In the next step of the algorithm the VI-Test creates the variable interval equations based on the subscript information and the equality constraints in the if-statements as described in Sections 3.2.1 and 3.2.6. If there is more than one equation in the dependence system we apply the equation propagation technique trying to eliminate or reduce the coupling between equations as described in Section 3.2.5. If coupling cannot be eliminated the algorithm generates additional equations according to the lambda test algorithm. The algorithm then proceeds in a subscript-by-subscript fashion testing each equation individually.

45
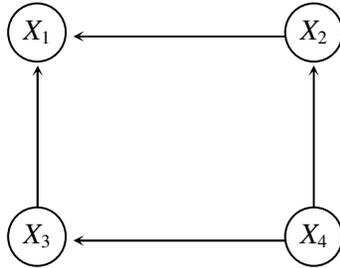
Figure 3-2: Example of a variable precedence graph

The main part of the algorithm is a process of variable elimination applied on each variable interval equation. Before we eliminate a variable we need to check Accuracy Condition 1 and Accuracy Condition 2. This requires the computation of the minimum or the maximum value of an expression using the variable substitution algorithm. The variable substitution algorithm for the computation of the minimum value of an expression proceeds by substituting each variable with its lower bound if the coefficient of the variable is positive or with its upper bound if the coefficient of the variable is negative. For the computation of the maximum value the process is similar. After all variables have been substituted the remaining constant value is the minimum or the maximum value of the expression. There is a simple rule regarding the order of substitution when computing the minimum or maximum: only variables that do not appear in other bounds may be substituted first. In the actual implementation of the algorithm we define the order by using a directed acyclic graph (DAG) that we call variable precedence graph. Each variable, appearing in the expression or a bound of a variable, represents a node in the graph. For every variable X and Y where Y appears in the lower or upper bound of X there exists an arc in the graph from X to Y. The algorithm proceeds by first substituting a variable whose graph node that has no incoming arcs. Because of this property when the variable is substituted it will not appear in any of the remaining bounds. The variable is subsequently eliminated from the graph and the process of substitution is repeated. The graph construction requires $O(n^2)$ time for n variables and the application of the variable substitution algorithm has worst case $O(n^2)$ time complexity.

A variable precedence graph is also used during the variable elimination process from the variable interval equation. The graph defines a partial ordering in which the variables may be eliminated. Figure 3-2 displays the variable precedence graph for the equation and inequalities of Example 3-5. In this example the variables can be eliminated only in the order of X_4, X_3, X_2, X_1 or X_4, X_2, X_3, X_1. In general there may be more than one valid sequence of elimination, since the ordering of the variables is partial. The whole procedure takes $O(n^3)$ for the elimination of all variables.

Finally, when the VI-Test algorithm terminates we are going to have two integer intervals on the right hand side of the equation and zero on the left. The interval resulting from the restricted bounds represents an interval in which the expression realizes every integer value. If zero is included in that interval then an integer solution exists subject to the problem constraints. The interval resulting from the original bounds of variables indicates a lower and an upper bound for the expression. If zero is outside this interval then no solution exists. If zero is inside the interval resulting from the original bounds, but outside the restricted interval, the VI-Test is inconclusive. Once the VI-Test determines whether dependence exists or not for the current direction vector, the algorithm proceeds with the next direction vector according the direction vector hierarchy [9].

Chapter 4

Dependence Analysis for Non-Linear Expressions

Data dependence analysis has been studied extensively and many algorithms were proposed in the literature. Traditional dependence analysis techniques utilized in commercial compilers focus on disproving dependences between array statements that are enclosed in a nest of loops [4], [14], [24]. As we discussed in previous chapters, these techniques oversimplify the dependence problem by ignoring complex loop bounds, array subscripts with coupled variables, if-statement constraints, and testing for real as opposed to integer solutions. Furthermore, most data dependence tests can only be applied when the array subscripts and the loop bounds are linear functions of the loop indices. In many cases the additional parallelism that would have been revealed through non-linear symbolic analysis remains unexploited [15].

Several dependence analysis techniques that were developed to address some of these issues have improved data dependence accuracy, but have not significantly increased program parallelization. Such tests are discussed in Section 1.4. The I-Test [19], [30] is an extension of the Banerjee test [4] and it can provide the conditions [29] that when satisfied the existence of a real solution in the data dependence system also implies the existence of an integer solution. It is very efficient and when enhanced with other simple techniques described in Chapter 2 it has been proven to be fairly accurate in practice. The Omega test [31], [32] constitutes an integer constraint satisfaction algorithm and therefore it can handle any type of linear constraints, including complex loop bounds, multi-dimensional arrays with coupled subscripts, and if-statement constrains. The Range test [6] is based on the notion of overlapping ranges of array subscripts access, which is similar to the extreme value computation of the Banerjee test, but in addition it can be applied in both linear and non-linear functions.

On other hand all data dependence tests have their limitations. The I-Test although extremely practical for real compilers it simplifies many problem constraints. It examines only one subscript at a

time, it requires constant loop bounds, and it ignores if-statements and non-linear expressions. The Omega test can handle most of these cases, but it cannot analyze non-linear expressions. Furthermore, it has worst-case exponential time complexity and it is inefficient in practice. The Range test, even though it is a practical test, it still ignores coupled subscript and if-statement constraints. In addition it does not provide complete direction vector information, since it only returns the level of the loop that carries the dependence and therefore it assumes more dependent direction vectors than they actually exist. This limitation prohibits important compiler transformations such as loop interchange [37]. Also the Range test cannot detect integer solutions to the dependence problem, like the I-Test and the Omega test do, and therefore it does not provide a conclusive answer when the dependence cannot be disproved.

In this chapter we present new data dependence analysis techniques that combine the advantages of existing techniques and at the same time overcome their limitations. Our method is based on a set of polynomial time algorithms that can accurately resolve non-linear and symbolic expressions, handle complex loop regions, take into account coupling in array subscripts, and consider simple if-statement constraints in the dependence problem. It provides accurate and complete direction vector information and it can conclusively determine whether a dependence exists or not, even in source codes with non-linear and symbolic expressions. Our techniques have been incorporated into a new data dependence analysis tool that we term Non-Linear Variable Interval Test or NLVI-Test for sort. The NLVI-Test is based on the theory of variable intervals which was introduced in Chapter 3 but extended for non-linear functions. It utilizes efficient techniques to handle coupled subscripts and if-statement constraints. It has been implemented for integer and rational multivariable polynomials, which capture the vast majority of array subscripts and loop bound expressions found in actual source code. The NLVI-Test is part of the PLATO library (Programming Language Analysis Translation and Optimization) and it can be easily ported to any optimizing and parallelizing compiler. The PLATO library provides an infrastructure for data dependence analysis and implementation of linear and polynomial expression arithmetic. The library also includes implementations of several other dependence tests including the Banerjee test and the I-Test it is available on the web (http://www.cs.utsa.edu/~plato).

The rest of the chapter is organized as follows. In Section 4.1 we establish the theoretical foundation of the NLVI-Test. We extend the definition of the variable integer interval to non-linear functions and we discuss how it can be simplified through a process of variable elimination. In Section 4.2 we describe how we apply the theory in practice to solve a data dependence problem for any type of direction vector. We illustrate how we can compute extreme values on non-linear

expressions, which is a procedure utilized by the NLVI-Test algorithm. We also show how we can restrict the problem domain to satisfy conditions that guarantee the existence of an integer solution. We also discuss the issues of data dependence analysis for problems with integer and rational polynomial expressions, which are very often found in actual source code. We discuss how the equation propagation technique can be incorporated in the NLVI-Test to eliminate or reduce coupling in multi-dimensional array subscripts and how we can deal with simple cases of if-statements. In Section 4.3 we describe the NLVI-Test algorithm and analyze its complexity. In Section 4.4 we provide examples from real source code to showcase the applicability of NLVI-Test in practice.

4.1 Generalized Variable Interval Theory

The theory of variable intervals was first introduced and applied to linear functions in Chapter 3. In this section we extend the theory of the variable integer intervals to multivariable non-linear functions, which constitutes the foundation of our dependence test. We present definitions, theorems and the intuition behind them, as well as several examples to demonstrate their application.

Consider all basic definitions and theorems in Section 3.1.1. For the main part of our theory we investigate the conditions under which a variable integer interval is contiguous. We first consider the simple case of a single variable integer interval:

$$[L(X), U(X)], \text{ where } P \leq X \leq Q$$

In this case functions L and U are from region $\Re = \{P, P + 1, \ldots, Q\} \subseteq Z$ to Z. According to Definition 3-2 the above variable integer interval is the union of the integer intervals $[L(X_i), U(X_i)]$, for all $X_i \in \Re$. We consider the monotonicity of the bound functions L and U. In case they have opposite monotonicity, then one bound of the variable integer interval will be increasing and the other will be decreasing for consecutive values of variable X. In this case either of two consecutive integer intervals $[L(X_i), U(X_i)]$ and $[L(X_i + 1), U(X_i + 1)]$, for X_i between P and $Q - 1$ will contain the other. For instance, if L is decreasing and U is increasing, then the integer interval $[L(X_i), U(X_i)]$ is contained into $[L(X_i + 1), U(X_i + 1)]$. On the other hand, if L is increasing and U is decreasing, then the integer interval $[L(X_i), U(X_i)]$ contains $[L(X_i + 1), U(X_i + 1)]$. In either case the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, consists of a union of integer intervals containing one another, so the largest interval will contain all of them and therefore their union is a contiguous integer interval. When the bound functions of the variable integer interval have the same monotonicity, the union of all integer intervals is contiguous if the union of each pair of non-empty consecutive integer intervals is contiguous. If L and U are both increasing functions the union of two consecutive integer intervals
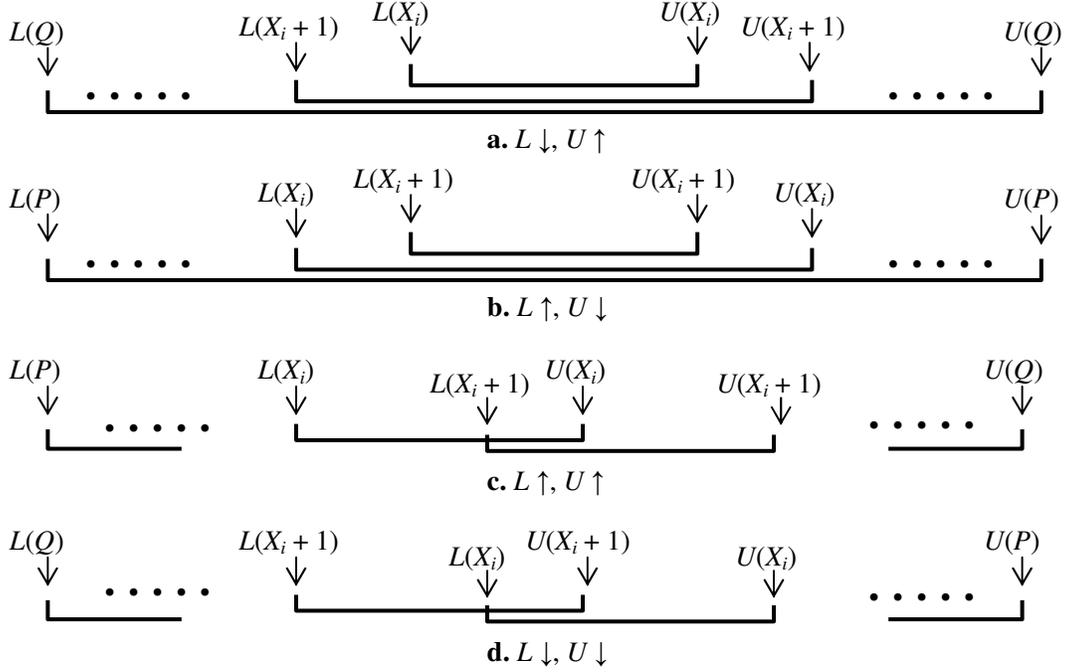
Figure 4-1: Conditions for simple variable interval contiguity

$[L(X_i), U(X_i)]$ and $[L(X_i + 1), U(X_i + 1)]$ is contiguous and equal to $[L(X_i), U(X_i + 1)]$ if $L(X_i + 1) \leq U(X_i) + 1$. Similarly if L and U are both decreasing the union of two consecutive integer intervals $[L(X_i + 1), U(X_i + 1)]$ and $[L(X_i), U(X_i)]$ is contiguous and equal to $[L(X_i + 1), U(X_i)]$ if $L(X_i) \leq U(X_i + 1) + 1$. The conditions for contiguity and the intuition behind them are depicted in Figure 4-1 and they are summarized by the following theorem.

Theorem 4-1

Given a single variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$:

a. If L is decreasing and U increasing, then $[L(X), U(X)]$ is equal to the integer interval $[L(Q), U(Q)]$.

b. If L is increasing and U decreasing, then $[L(X), U(X)]$ is equal to the integer interval $[L(P), U(P)]$.

c. If L and U are increasing and $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $[L(X), U(X)]$ is equal to the integer interval $[L(P), U(Q)]$.

d. If L and U are decreasing and $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $[L(X), U(X)]$ is equal to the integer interval $[L(Q), U(P)]$.

Proof: See Appendix.

■

Example 4-1

Consider the single variable integer interval $[2X - 1, 3X^3 + X]$, where $1 \leq X \leq 5$. In this case $L(X) = 2X - 1$, $U(X) = 3X^3 + X$, $P = 1$ and $Q = 5$. Function L is obviously increasing and function U is also increasing for values of X between 1 and 5. In this case we compute $U(X_i) - L(X_i + 1) + 1 = (3X_i^3 + X_i) - (2(X_i + 1) - 1) + 1 = 3X_i^3 - X_i$. We can easily verify that $3X_i^3 - X_i$ is positive for all values of X_i between 1 and 4. Therefore, according to Theorem 4-1, the variable integer interval $[2X - 1, 3X^3 + X]$, where $1 \leq X \leq 5$, is contiguous and equal to the integer interval $[2 \times 1 - 1, 3 \times 5^3 + 5] = [1, 380]$.

We use the above theorem as the basis for a transformation that can reduce a multivariate integer interval into a simple integer interval by eliminating one variable at a time. Before we introduce this transformation we define the min and max functions and the property of monotonicity for multivariate integer functions.

Definition 4-1

Consider a function $F(\boldsymbol{x}, X)$ from Z^{n+1} to Z, where $\boldsymbol{x}$ is a vector of n variables and X is an additional variable. The multivariable function F is *increasing* for variable X, if $F(\boldsymbol{x}_i, X_j) \leq F(\boldsymbol{x}_i, X_j + 1)$ for any $\boldsymbol{x}_i \in Z^n$ and for any $X_j \in Z$.

Similarly we can define a *decreasing* multivariate function for any given variable. The monotonicity of a multivariate function can also be defined in any region that is subset of Z^{n+1} for values of $(\boldsymbol{x}, X)$ in that region. If the region is finite, which is the case for all loop regions, we can prove that F is increasing by showing that $\min(F(\boldsymbol{x}, X + 1) - F(\boldsymbol{x}, X)) \geq 0$. A more efficient way to prove that F is increasing is to show that the first partial derivative of F for X is greater or equal than zero in the region, i.e. $\min(dF(\boldsymbol{x}, X)/dX) \geq 0$. Similarly we can prove that a function F is decreasing by showing that $\max(F(\boldsymbol{x}, X + 1) - F(\boldsymbol{x}, X)) \leq 0$ or $\max(dF(\boldsymbol{x}, X)/dX) \leq 0$.

Example 4-2

Consider the functions $L(X_1, X_2) = 2X_1X_2 - 1$ and $U(X_1, X_2) = 3X_1X_2 + X_1$, where $1 \leq X_1 \leq 5$, $1 \leq X_2 \leq X_1^2$. In order to determine the monotonicity of the two functions for variable X_2, we compute the first partial derivative for X_2 for each function. In this case, $dL(X_1, X_2)/dX_2 = 2X_1$ and $dU(X_1, X_2)/dX_2 = 3X_1$. Both derivative functions are positive for values of X_1 between 1 and 5, so L and U are increasing functions for variable X_2 in the region defined by their constraints.

We now consider the multivariable integer interval $[L(\boldsymbol{x}, X), U(\boldsymbol{x}, X)]$ where $\boldsymbol{x}$ is a variable vector in a region $\Re \subseteq Z^n$. The region $\Re$ is defined by a set of constrains on the variables of vector $\boldsymbol{x}$. Variable X appears only in one additional constraint $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$. The region $\{\boldsymbol{x} \in \Re: P(\boldsymbol{x}) \leq Q(\boldsymbol{x})\}$ can be enumerated and in addition, since it reflects a loop region in our case, it is also finite. If we apply

Theorem 4-1 to each single variable integer interval $[L(x_i, X), U(x_i, X)]$ for all values x_i in the enumeration of region $\{x \in \mathfrak{R}: P(x) \leq Q(x)\}$ then we can derive the following theorem:

Theorem 4-2

Given a variable integer interval $[L(x, X), U(x, X)]$ subject to a set of constraints on x in $\mathfrak{R}$ and the constraint $P(x) \leq X \leq Q(x)$, where X does not appear in any of the constraints in $\mathfrak{R}$:

a. If L is decreasing and U is increasing for variable X, then $[L(x, X), U(x, X)]$ is equal to the variable integer interval $[L(x, Q(x)), U(x, Q(x))]$, where x in $\mathfrak{R}$ and $P(x) \leq Q(x)$.

b. If L is increasing and U is decreasing for variable X, then $[L(x, X), U(x, X)]$ is equal to the variable integer interval $[L(x, P(x)), U(x, P(x))]$, where x in $\mathfrak{R}$ and $P(x) \leq Q(x)$.

c. If L and U are increasing for variable X and $\min(U(x, X) - L(x, X + 1) + 1) \geq 0$, then $[L(x, X), U(x, X)]$ is equal to the variable integer interval $[L(x, P(x)), U(x, Q(x))]$, where x in $\mathfrak{R}$ and $P(x) \leq Q(x)$.

d. If L and U are decreasing for variable X and $\min(U(x, X + 1) - L(x, X) + 1) \geq 0$, then $[L(x, X), U(x, X)]$ is equal to the variable integer interval $[L(x, Q(x)), U(x, P(x))]$, where x in $\mathfrak{R}$ and $P(x) \leq Q(x)$.

Proof: See Appendix.

∎

Example 4-3

Consider the variable integer interval $[2X_1X_2 - 1, 3X_1X_2 + X_1]$, where $1 \leq X_1 \leq 5$, $1 \leq X_2 \leq X_1^2$. We can initially eliminate X_2, because it is the only variable that appears in a single constraint. In this case $x = \langle X_1 \rangle$, $X = X_2$, $L(x, X) = 2X_1X_2 - 1$, $U(x, X) = 3X_1X_2 + X_1$, $P(x) = 1$ and $Q(x) = X_1^2$. From Example 4-2, we know that both functions L and U are increasing for variable X_2. In this case we compute $U(x, X) - L(x, X + 1) + 1 = (3X_1X_2 + X_1) - (2X_1(X_2 + 1) - 1) + 1 = X_1X_2 - X_1 + 2$. By bounding this expression from below, we can prove that $X_1X_2 - X_1 + 2 \geq X_1 \times 1 - X_1 + 2 \geq 2 \geq 0$. Therefore $\min(U(x, X) - L(x, X + 1) + 1) \geq 0$ and $[2X_1X_2 - 1, 3X_1X_2 + X_1]$, according to Theorem 4-2, is equal to $[2X_1 \times 1 - 1, 3X_1 \times X_1^2 + X_1] = [2X_1 - 1, 3X_1^3 + X_1]$, for $1 \leq X_1 \leq 5$, $1 \leq X_1^2$. It turns out from Example 4-1 that we can eliminate variable X_1 as well and therefore our original variable integer interval $[2X_1X_2 - 1, 3X_1X_2 + X_1]$, where $1 \leq X_1 \leq 5$, $1 \leq X_2 \leq X_1^2$ is equal to the integer interval $[1, 380]$.

Theorem 4-2 provides a transformation mechanism that reduces a variable integer interval into a simpler variable integer interval by eliminating one variable. The same mechanism can be applied in a variable interval equation.

Theorem 4-3

Given the following variable interval equation

$$F(\pmb{x}) = [L(\pmb{x}, X),\ U(\pmb{x}, X)]$$

subject to a set of constraints on $\pmb{x}$ in $\Re$ and the constraint $P(\pmb{x}) \leq X \leq Q(\pmb{x})$, where X does not appear in function F or any of the constraints in $\Re$:

a. If L is decreasing and U is increasing for variable X, then the above equation is integer solvable iff the variable interval equation $F(\pmb{x}) = [L(\pmb{x}, Q(\pmb{x})),\ U(\pmb{x}, Q(\pmb{x}))]$ is integer solvable subject to the same constraints on $\pmb{x}$ in $\Re$ and the constraint $P(\pmb{x}) \leq Q(\pmb{x})$.

b. If L is increasing and U is decreasing for variable X, then the above equation is integer solvable iff the variable interval equation $F(\pmb{x}) = [L(\pmb{x}, P(\pmb{x})),\ U(\pmb{x}, P(\pmb{x}))]$ is integer solvable subject to the same constraints on $\pmb{x}$ in $\Re$ and the constraint $P(\pmb{x}) \leq Q(\pmb{x})$.

c. If L and U are increasing for variable X and $\min(U(\pmb{x}, X) - L(\pmb{x}, X + 1) + 1) \geq 0$, then the above equation is integer solvable iff the variable interval equation $F(\pmb{x}) = [L(\pmb{x}, P(\pmb{x})),\ U(\pmb{x}, Q(\pmb{x}))]$ is integer solvable subject to the same constraints on $\pmb{x}$ in $\Re$ and the constraint $P(\pmb{x}) \leq Q(\pmb{x})$.

d. If L and U are decreasing for variable X and $\min(U(\pmb{x}, X + 1) - L(\pmb{x}, X) + 1) \geq 0$, then the above equation is integer solvable iff the variable interval equation $F(\pmb{x}) = [L(\pmb{x}, Q(\pmb{x})),\ U(\pmb{x}, P(\pmb{x}))]$ is integer solvable subject to the same constraints on $\pmb{x}$ in $\Re$ and the constraint $P(\pmb{x}) \leq Q(\pmb{x})$.

Proof: Straightforward from Theorem 4-2 and Definition 3-3:

∎

The above theorem provides a way to eliminate a variable from an interval equation. In order to eliminate variable X from an interval equation we move the terms involving X to the right hand side of the equation using Theorem 3-1 and then apply Theorem 4-3.

The applicability of Theorem 4-3 depends on the size of the coefficients in the variable interval equation. In case the size of the coefficients is not small enough, the following transformation can reduce the size of the coefficients in a variable interval equation, potentially enabling the application of Theorem 4-3

Theorem 4-4

Consider a variable interval equation of the following form

$$F(\pmb{x}) = [L(\pmb{x}) + L_0,\ U(\pmb{x}) + U_0]$$

where expressions $F(\pmb{x})$, $L(\pmb{x})$, $U(\pmb{x})$ do not contain any constant terms. Let d be the greatest common divisor of the coefficients of all the terms in expressions $F(\pmb{x})$, $L(\pmb{x})$, $U(\pmb{x})$. The above variable interval equation is integer solvable subject to any constraints iff the following variable interval equation

$$F(\pmb{x})/d = [L(\pmb{x})/d + \lceil L_0/d \rceil,\ U(\pmb{x})/d + \lfloor U_0/d \rfloor]$$

54

is integer solvable subject to the same constraints.

Proof: See Appendix.

∎

After the transformation of Theorem 3-6 has been applied we check if the variable interval equation is still feasible. If max $((U(\boldsymbol{x})/d + \lfloor U_0/d \rfloor) - (L(\boldsymbol{x})/d + \lceil L_0/d \rceil)) < 0$, then no integer solution exists for the above variable interval equation subject to the problem constraints, since the variable integer interval on the right hand side is empty.

4.2 Application in Data Dependence Analysis

As we have already discussed, that the data dependence analysis problem for arrays can be reduced to the integer constraint satisfaction problem [24]. In particular, testing for data dependence translates into solving a system of constraints consisting of a set of equations and inequalities derived from the array subscripts, from the loop bounds, and from other problem constraints. In many cases the system of constraints is non-linear. In this section we demonstrate how the variable interval theory can be applied to solve the data dependence problem for the general non-linear case.

4.2.1 Data Dependence Testing

A simple loop nest has the following form:

```
for I₁ = p₁ to q₁ do
  for I₂ = p₂(I₁)to q₂(I₁) do
    for I₃ = p₃(I₁, I₂) to q₃(I₁, I₂) do
        •
        •
        for Iₙ = pₙ(I₁, I₂, …,Iₙ₋₁) to qₙ(I₁, I₂, …,Iₙ₋₁) do
S₁:         A[f(I₁, I₂, …,Iₙ)] = …
S₂:                     … = A[g(I₁, I₂, …,Iₙ)]
        endfor
        •
        •
    endfor
  endfor
endfor
```

For the above loop nest the data dependence problem between two instances $(I_1, I_2, \ldots, I_n)$ of S_1 and $(I_1', I_2', \ldots, I_n')$ of S_2 can be formulated as follows:

$$
\begin{aligned}
&f(I_1, I_2, \ldots I_n) = g(I_1', I_2', \ldots, I_n') \\
&p_k(I_1, I_2, \ldots, I_{k-1}) \leq I_k \leq q_k(I_1, I_2, \ldots, I_{k-1}) \\
&p_k(I_1', I_2', \ldots I_{k-1}') \leq I_k' \leq q_k(I_1', I_2', \ldots I_{k-1}'), \qquad 1 \leq k \leq n.
\end{aligned}
\tag{4-1}
$$

For simplicity we substitute X_{2k-1} for I_k and X_{2k} for I_k'. In addition we denote the bounds as follows:

$$P_{2k-1}(\boldsymbol{x}) = p_k(X_1, X_3, \ldots, X_{2k-1}), \qquad Q_{2k-1}(\boldsymbol{x}) = q_k(X_1, X_3, \ldots, X_{2k-1})$$
$$P_{2k}(\boldsymbol{x}) = p_k(X_2, X_4, \ldots, X_{2k}), \qquad Q_{2k}(\boldsymbol{x}) = q_k(X_2, X_4, \ldots, X_{2k})$$

where $\boldsymbol{x} = (X_1, X_2, \ldots X_{2n})$. Finally by defining function $F(\boldsymbol{x}) = f(X_1, X_3, \ldots, X_{2n-1}) - g(X_2, X_4, \ldots, X_{2n})$ the data dependence problem in (4-1) can be expressed by the following equivalent system of constraints:

$$F(\boldsymbol{x}) = 0$$
$$P_{2k-1}(\boldsymbol{x}) \leq X_{2k-1} \leq Q_{2k-1}(\boldsymbol{x}) \qquad\qquad 1 \leq k \leq n \qquad\qquad (4\text{-}2)$$
$$P_{2k}(\boldsymbol{x}) \leq X_{2k} \leq Q_{2k}(\boldsymbol{x}) \qquad\qquad \boldsymbol{x} = (X_1, X_2, \ldots, X_{2n}),$$

The NLVI-Test starts with the equation in (4-2) and converts it to the trivial variable interval equation $F(\boldsymbol{x}) = [0, 0]$. Subsequently the process of variable elimination begins. The algorithm proceeds by moving the terms of the variable to be eliminated to the right hand side of the equation according to Theorem 3-1 and then applies Theorem 4-3. This process is repeated until all variables have been eliminated. Since Theorem 4-3 requires that the variable to be eliminated does not appear in any constraints other than those defining its bounds, we need to start eliminating variables from the highest to the lowest index. Thus, the first variable to be eliminated is either X_{2n} or X_{2n-1}. Other variables may also be eliminated first if they do not appear in any other problem constraints. The order of variable elimination is a partial order and can be determined by applying a graph algorithm. According to Theorem 4-3, one of the conditions to successfully eliminate a variable from the interval equation is the following:

Accuracy Condition 1

For every variable X eliminated from the variable interval equation

$$F(\boldsymbol{x}) = [L(\boldsymbol{x}, X), U(\boldsymbol{x}, X)]$$

subject to a set of constraints on $\boldsymbol{x}$ in $\Re$ and the constraint $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$, where X does not appear in function F or any of the constraints in $\Re$, the bound functions L, U should have opposite monotonicity for variable X or else

- if they are both increasing, then $\min(U(\boldsymbol{x}, X) - L(\boldsymbol{x}, X+1) + 1) \geq 0$
- if they are both decreasing, then $\min(U(\boldsymbol{x}, X+1) - L(\boldsymbol{x}, X) + 1) \geq 0$

Note that after variable X is eliminated, according to Theorem 4-3, the constraint $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$ still remains in the system. This constraint complicates the problem because it may involve variables that appear in other constraints too, making it impossible to reapply Theorem 4-3 for those variables. We would like this constraint to be eliminated together with variable X. We can achieve that if we show

that the inequality is valid for all values of x in the problem domain by testing the following condition.

Accuracy Condition 2

For every variable X eliminated from the variable interval equation

$$F(x) = [L(x, X), U(x, X)]$$

subject to a set of constraints on x in $\Re$ and the constraint $P(x) \leq X \leq Q(x)$, where X does not appear in function F or any of the constraints in $\Re$, the following inequality needs to be satisfied

$$\min(Q(x) - P(x)) \geq 0$$

Accuracy Condition 2 is very frequently met in practice because each loop usually iterates for all values of the outermost loops.

At the end of the algorithm the variable interval equation will consist of zero on the left-hand side and an integer interval on the right. If zero lies inside the integer interval and all accuracy conditions have been satisfied, then the NLVI-Test concludes that an integer solution exists to the equation subject to the problem constraints in (4-2) and there is data dependence. If zero is outside the integer interval, then no solution exists and there is no dependence. If any of the accuracy conditions are not satisfied, then the algorithm still proceeds with variable elimination but it can not conclusively prove data dependence.

4.2.2 Dependence Testing Under Direction Vectors

Compiler transformations for program optimization and parallelization require additional information with regard to the relative loop iterations in which the related (dependent) instances occur. Such information is summarized in direction vectors [37]. When testing for data dependence subject to direction vectors, additional constraints ($<$, $=$, or $>$) are introduced in the dependence system between any two instances of the same loop iteration variable. We will examine each relation separately.

First consider the less "$<$" direction imposed on variables X_{2k-1}, X_{2k}, the two instances of the same loop iteration variable I_k. The inequality constraints in (4-2) are as follows.

$$\begin{aligned}
P_{2k-1}(x) &\leq X_{2k-1} \leq Q_{2k-1}(x) \\
P_{2k}(x) &\leq X_{2k} \ \ \leq Q_{2k}(x) \\
X_{2k-1} &< X_{2k}
\end{aligned} \qquad (4\text{-}3)$$

Note that direction vector constraint in (4-3) imposes additional bounds on both variables. Theorem 4-3 requires a single upper and lower bound for each variable. Taking into account the additional inequality the bounds of the two variables can be expressed in the following two ways.

$$P_{2k-1}(\pmb{x}) \leq X_{2k-1} \leq \min(Q_{2k-1}(\pmb{x}), Q_{2k}(\pmb{x}) - 1) \qquad P_{2k-1}(\pmb{x}) \leq X_{2k-1} \leq \min(Q_{2k-1}(\pmb{x}), X_{2k} - 1)$$
$$\max(P_{2k}(\pmb{x}), X_{2k-1} + 1) \leq X_{2k} \leq Q_{2k}(\pmb{x}) \qquad \text{OR} \qquad \max(P_{2k}(\pmb{x}), P_{2k-1}(\pmb{x}) + 1) \leq X_{2k} \leq Q_{2k}(\pmb{x})$$

Note that each case defines a different order of variable elimination. In first case variable X_{2k} is eliminated before variable X_{2k-1} and in the second case the opposite. Because *min* and *max* function are difficult to handle, when performing algebraic operations the NLVI-Test determines which of the bounds is tighter by comparing directly or indirectly the two bounds.

The case of the greater ">" direction is similar. For the equal relation ('=') the two variables are substituted by one. In this case the new variable has to satisfy the bound constraints of both X_{2k-1} and X_{2k}. We determine which of the two bounds is the tightest for both upper and lower bounds by comparing them. The algorithm for comparing two non-linear expressions is discussed in the next subsection. The conditions for bound selection are the same like those in Chapter 3 and are summarized in the table of Figure 3-1.

4.2.3 Computation of Minimum and Maximum Values

The computation of minimum and maximum values is a procedure utilized throughout the NLVI-Test algorithm. In particular it is used to compare two expressions and prove that one is less or equal than the other. In order to prove an inequality of the form $P(\pmb{x}) \leq Q(\pmb{x})$ in a region defined by a set of constraints, it suffices to show that $\min(Q(\pmb{x}) - P(\pmb{x})) \geq 0$, or $\max(P(\pmb{x}) - Q(\pmb{x})) \leq 0$.

For the computation of the minimum or maximum value of an expression a simple variable substitution algorithm is employed. Note that in some cases this algorithm will not return the actual extreme value but rather a bound of the expression, which also suffices for our purposes. For the computation of the minimum value, each variable in the expression is substituted with its lower bound if the expression is increasing for that variable or with its upper bound if the expression is decreasing. For the computation of the maximum value the process is similar. The procedure for computing the minimum or maximum involves the determination of the monotonicity of the expression with respect to each variable that is substituted. Furthermore, the procedure for determining the monotonicy of an expression involves the computation of the minimum or the maximum of the difference function or the first partial derivative of that expression. Note that each procedure calls the other recursively each time reducing the degree of the problem until a halting condition is met when the recursion ends. The level of recursion depends on the type of function. For

polynomials, as we discuss later in Section 4.2.5, the maximum recursion level is the degree of the polynomial. This is because either differencing or partial differentiation reduces the degree of the polynomial by one.

The order of substitution in the expression is defined by a directed acyclic graph (DAG) that we call variable precedence graph. Every variable in the expression and bounds corresponds to a node in the graph. For each variable Y appearing in the lower or upper bound of another variable X an arc exists in the graph from node X to node Y. Any topological sort of the DAG is a valid order for variable substitution. The precedence graph is also used to determine a valid order of variable elimination from the variable interval equation in the main part of the NLVI-Test algorithm described in Section 4.3.

Example 4-4

Consider computation of $\min(X_1 X_2 - X_1 + 2)$, where $1 \leq X_1 \leq 5$, $1 \leq X_2 \leq X_1^2$, in Example 4-3. The variable precedence graph in this example has two nodes, namely X_1 and X_2, and one arc from X_2 to X_1. The order of substitution in this case is variable X_2 first and variable X_1 second. Assume $F_2(X_1, X_2) = X_1 X_2 - X_1 + 2$. The algorithm attempts to determine whether F_2 is increasing or decreasing for variable X_2. Subsequently it computes the first partial derivative $F_{2.2}(X_1, X_2) = dF_2(X_1, X_2)/dX_2 = X_1$. Next the algorithm attempts to determine whether $F_{2.2}$ is positive or negative therefore it calls itself recursively in order to check if $\min(X_1) \geq 0$ or $\max(X_1) \leq 0$. In order to compute $\min(F_{2.2}(X_1, X_2))$ the algorithm will compute the first partial derivative $F_{2.2.1}(X_1, X_2) = dF_{2.2}(X_1, X_2)/dX_1 = 1$, which in this case is a positive constant. The recursion ends here and the algorithm concludes that $F_{2.2}$ is increasing for variable X_1. As a result variable X_1 is substituted by its lower bound so $\min(F_{2.2}(X_1, X_2)) = 1$. From this the algorithm concludes that $F_{2.2}$ is positive and therefore function F_2 is increasing for variable X_2. In the next step the algorithm will substitute X_2 with its lower bound in F_2 reducing the expression to $F_1(X_1) = X_1 \times 1 - X_1 + 2 = 2$. Since F_1 is constant the algorithm terminates here returning as $\min(X_1 X_2 - X_1 + 2)$ the value 2.

4.2.4 Relaxing the Accuracy Conditions

Occasionally some of the accuracy conditions or some of the bound selection conditions will not be satisfied. In this case we may be able to restrict the problem domain in order to satisfy the conditions and possibly provide an exact answer upon termination of the algorithm. All conditions require that some inequality is satisfied for all possible values of the variables in the problem domain. In order to prove the inequality, as we have already discussed, we compute the minimum of an expression and check if its value is greater or equal than zero or the maximum of an expression and

check if its value is less or equal than zero. If we restrict the problem domain the minimum value of an expression will increase and the maximum value will decrease. We may be able to restrict the problem domain just enough so that we can satisfy the inequality and meet the condition that has failed. We refer to this process as *relaxing the accuracy conditions*. We restrict the problem domain by tightening the bounds of the variables that have constant bounds. In practice we create a new problem which is defined in a subset of the original problem domain. If an integer solution exists for the sub-problem then an integer solution exists for the original problem.

Let us consider a condition requiring that the minimum of an expression is greater than a constant value. In the final steps of the variable substitution algorithm for the computation of the minimum value of the expression all the variables remaining are those that have constant bounds. In data dependence problems for trapezoidal loops there is at least one variable that has constant lower and upper bounds. After all variables, except those with constant bounds, have been eliminated the initial condition is reduced to an inequality of the following form:

$$\min(F_m(X_1, X_2, \ldots, X_m)) \ge c, \qquad P_i \le X_i \le Q_i, \qquad 1 \le i \le m.$$

Continuing the variable substitution algorithm the following expressions will be created as part of the variable substitution process:

$$F_m(X_1, X_2, \ldots, X_{m-1}, X_m), F_{m-1}(X_1, X_2, \ldots, X_{m-1}), \ldots, F_2(X_1, X_2), F_1(X_1), F_0.$$

Since the condition has failed it follows that $F_0 < c$. The procedure for relaxing the condition starts with expression F_1. We attempt to determine a value X_1^0 such that $F_1(X_1^0) \ge c$ and $P_1 \le X_1^0 \le Q_1$. If F_1 is increasing for variable X_1 then for all $X_1 \ge X_1^0$ it follows that $F_1(X_1) \ge c$. Similarly if F_1 is decreasing for variable X_1 then for all $X_1 \le X_1^0$ it follows that $F_1(X_1) \ge c$. In order to determine the value X_1^0 we solve the equation $F_1(X_1) = c$. If the equation is linear then the solution is straightforward. For non-linear equations we use the Newton-Raphson numerical method [1]. Because we are only interested in integers we only need the integer portion of the root. Given the functions are monotonic, this can be easily determined within the first few iterations of the Newton-Raphson method. If no such value exists between P_1 and Q_1 then we restrict the bounds of variable X_1 within one value i.e. Q_1 if F_1 is increasing or P_1 if F_1 is decreasing. We substitute that value in all expressions F_2 through F_m and continue in the same manner with F_2 which now has only one free variable, namely X_2.

If the procedure is successful we will find a solution X_i^0 such that $P_i \le X_i^0 \le Q_i$ for expression F_i. In this case we will define new bounds on X_i as follows:

$$P'_i = \begin{cases} X_i^0 & \text{if } F_i \text{ increasing} \\ P_i & \text{if } F_i \text{ decreasing} \end{cases} \quad \text{and} \quad Q'_i = \begin{cases} Q_i & \text{if } F_i \text{ increasing} \\ X_i^0 & \text{if } F_i \text{ decreasing} \end{cases}$$

For each variable X_j, $j < i$ for which we had to restrict the bounds within one value we have:

$$P'_j = \begin{cases} Q_j & \text{if } F_j \text{ increasing} \\ P_j & \text{if } F_j \text{ decreasing} \end{cases} \quad \text{and} \quad Q'_j = \begin{cases} Q_j & \text{if } F_j \text{ increasing} \\ P_j & \text{if } F_j \text{ decreasing} \end{cases}$$

The new bounds of the variables now satisfy the condition.

In the actual algorithm implementation each variable has two values for each bound. The regular bounds derived from the original loop bounds and the restricted bounds. They are initially equal. Each time we relax a condition by restricting the problem domain we tighten the restricted bounds of some variables. All accuracy conditions are tested on the restricted bounds because only for those we can give an exact positive answer to the dependence problem. When the NLVI-Test algorithm terminates, instead of one integer interval on the right hand side of the equation we are going to have two integer intervals. The large integer interval will result from the regular bounds and a smaller integer sub-interval will result from the restricted bounds. If zero lies outside the larger integer interval then no solution exists. If zero lies inside the smaller integer interval then an integer solution exists. If zero is inside the larger interval but outside the smaller then the NLVI-Test becomes inconclusive and reports a maybe answer.

4.2.5 Polynomial and Rational Polynomial Expressions

Non-linear expressions appear frequently in loop bounds, array subscripts and if-statement conditions. In most cases non-linear expressions can be represented by an integer or rational multivariate polynomial. Rational expressions usually appear as a result of the induction variable substitution and loop normalization transformations. In our dependence tool we implement the NLVI-Test for integer and rational polynomial expressions. Integer polynomial expressions present the following advantages:

   a. They capture the vast majority of non-linear expressions in source code.
   b. All basic operations are well defined and closed under polynomials.
   c. Computing extreme values is straightforward, since the monotonicity can be easily determined through the first partial derivative or difference.
   d. The computation of the partial derivative or difference of a polynomial for a variable is easy to compute and decreases the degree of the polynomial for that variable by one.

For rational polynomial expressions there exist a few more issues that need to be taken into consideration:

o **Rational polynomial addition/subtraction**: Accurate addition or subtraction of rational polynomials requires factorization of their denominators. However, this is only important for simplification of the computed expression. Addition or subtraction between two rational polynomials can performed as follows:

$$P_1(x)/Q_1(x) \pm P_2(x)/Q_2(x) = (P_1(x)Q_2(x) \pm P_2(x)Q_1(x))/(Q_1(x)Q_2(x))$$

Even though the above method may result into large rational expressions this is a very rare case for expressions that appear in actual data dependence problems. A simple method that factors out common terms suffices for the simplification in practice e.g.: $4X^3/(2X^2 + 2X) = 2X^2/(X + 1)$

o **Variables that have infinite bounds**: Such cases can be easily addressed in integer polynomial expressions. For example if an integer polynomial is increasing and a variable has an upper bound of $+\infty$ then the computation of the maximum value should also return $+\infty$. This is not necessarily the case for rational polynomials. Consider the following two expressions:

$$P_1(X) = -1/X, \quad P_2(X) = X/(X + 1)$$

The above rational polynomials are both increasing. However, when x goes to infinity the first polynomial converges to 0 and the second to 1. Limits theory of calculus has been incorporated into the NLVI-Test algorithm in order to resolve such cases.

o **Rational Polynomial Derivative**: According to theory the first partial derivative of a rational expression $P(x)/Q(x)$ for a variable X is as follows:

$$d(P(x)/Q(x))/dX = \frac{(dP(x)/dX)Q(x) - (dQ(x)/dX)P(x)}{Q(x)^2}$$

The derivative is used to determine the monotonicity of the expression. The algorithm attempts to verify whether the first partial derivative is positive or negative. In order to do so it suffices to determine only the sign of the numerator since the denominator is always positive. This approach helps to significantly reduce the cost and the complexity of the procedure.

In addition to the above issues rational expressions introduce additional problems when it comes to actual source code expressions. The additional issues arise from the fact that a rational expression is treated by computers at run time as integer division rather than a fractional number. Integer division of two integers a, b denoted by $a \div b$ is defined to be $\lfloor a/b \rfloor$ if $a/b \geq 0$ or $\lceil a/b \rceil$ if $a/b < 0$. Note that $b(a \div b)$ is not necessarily equal to a so many of the properties of the rational numbers do not apply to

integer division performed by computers. However, by exploiting some properties of integer division we can represent such expressions as rational numbers and perform exact data dependence analysis without loosing in precision, in most of the cases. Consider the following properties of integer division:

$$
\begin{aligned}
&\text{if } a/b \geq 0 && \text{then} && a/b - 1 \leq a \div b \leq a/b \\
&\text{if } a/b \leq 0 && \text{then} && a/b \leq a \div b \leq a/b + 1 && \text{(4-4)} \\
&\text{and always} && && a/b - 1 \leq a \div b \leq a/b + 1
\end{aligned}
$$

The above properties can be applied when a rational expression appears as a bound of a variable. Suppose that the upper bound of a variable X is the rational expression $P(x)/Q(x)$. Also assume that that the lower bound of variable X is a positive constant. The above rational expression evaluates to some positive integer through integer division when the code is executed. We also know from the properties in (4-4) that $P(x)/Q(x) - 1 \leq P(x) \div Q(x) \leq P(x)/Q(x)$. By setting the restricted bound of variable X equal to $P(x)/Q(x) - 1$ and the regular bound to equal to $P(x)/Q(x)$ we know that if the problem has integer solution for values of X within the restricted bounds then the original problem has integer solutions too. In addition, if the problem has no solution for the regular bounds then the original problem has no solution either. In this case we can use the restricted and regular bounds to address the issue of integer division by marginally restricting and expanding the problem domain.

In order to illustrate the application of the NLVI-Test we consider the following example.

Example 4-5

```
for I = 1 to N do
    for J = 1 to I*I do
S:     A[I] = A[(I - 1)*N + J] + …
    endfor
endfor
```

Suppose that we want to test the two references of array A for data dependence under the direction vector $(<, *)$. The data dependence problem in this example results into the following system:

$$X_1 - NX_2 - X_4 + N = 0$$
$$-\infty \leq N \leq +\infty$$
$$1 \leq X_1 \leq N, \qquad\qquad 1 \leq X_2 \leq N,$$
$$1 \leq X_3 \leq X_1^2, \qquad\qquad 1 \leq X_4 \leq X_2^2$$

where X_1 and X_2 are instances of loop I, X_3 and X_4 are instances of loop J, and N is a free variable. The vector of all variable $x = (X_1, X_2, X_3, X_4, N)$.

Because of the direction relation imposed on variables X_1 and X_2 their bounds need to be updated according to the table in Figure 3-1..
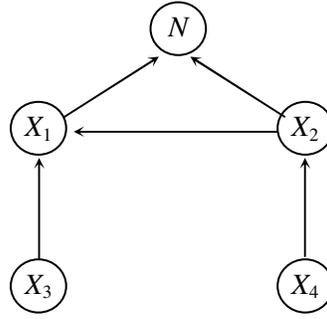
Figure 4-2: Variable precedence graph of Example 4-5

$v_1 =$ "<", $P_1(\boldsymbol{x}) = 1$, $P_2(\boldsymbol{x}) = 1$, $Q_1(\boldsymbol{x}) = N$, $Q_2(\boldsymbol{x}) = N$

Checking Condition $P_2(\boldsymbol{x}) \leq P_1(\boldsymbol{x}) + 1$:
$\max(P_2(\boldsymbol{x}) - P_1(\boldsymbol{x}) - 1) = \max(1 - 1 - 1) = -1 \leq 0$

Checking Condition $Q_1(\boldsymbol{x}) \geq Q_2(\boldsymbol{x}) - 1$:
$\min(Q_1(\boldsymbol{x}) - Q_2(\boldsymbol{x}) + 1) = \min(N - N + 1) = 1 \geq 0$

In this case the bounds are:
$$1 \leq X_1 \leq N - 1,$$
$$X_1 + 1 \leq X_2 \leq N$$

Note that from the new bounds of X_1 we can easily derive that $N \geq 2$. Also there is no direction imposed on variables X_3 and X_4 so their bounds remain the same. The dependence equation in its variable interval form and the final constraints are as follows:

$X_1 - NX_2 - X_4 + N = [0, 0]$
$2 \leq N \leq +\infty$
$1 \leq X_1 \leq N - 1$, $\qquad X_1 + 1 \leq X_2 \leq N$,
$1 \leq X_3 \leq X_1^2$, $\qquad 1 \leq X_4 \leq X_2^2$.

The variable precedence graph for the above constraints is depicted in Figure 4-2. The graph defines the order for variable elimination. Note that there is more than one valid order of elimination, namely X_4, X_3, X_2, X_1, N or X_3, X_4, X_2, X_1, N or X_4, X_2, X_3, X_1, N. We start with variable X_4 and move it to the right hand side:

$X_1 - NX_2 + N = [X_4, X_4]$

In this case $L(\boldsymbol{x}, X_4) = X_4$, $U(\boldsymbol{x}, X_4) = X_4$, $P(\boldsymbol{x}) = 1$, $Q(\boldsymbol{x}) = X_2^2$.

Checking Accuracy Condition 1: L and U are both increasing for X_4, since their first partial derivative is a positive constant. Because they are both increasing we need to compute $\min(U(\boldsymbol{x}, X_4) - L(\boldsymbol{x}, X_4 + 1) + 1) = \min(X_4 - (X_4 + 1) + 1) = 0 \geq 0$.

Checking Accuracy Condition 2: $\min(Q(\boldsymbol{x}) - P(\boldsymbol{x})) = \min(X_2{}^2 - 1) \geq (X_1 + 1)^2 - 1 =$

$X_1{}^2 + 2X_1 \geq 1^2 + 2\times 1 = 3 \geq 0.$

Both conditions are satisfied so we can eliminate X_4 as follows:

$X_1 - NX_2 + N = [1, X_2{}^2]$

We proceed by moving variable X_2 to the right hand side.

$X_1 + N = [NX_2 + 1, X_2{}^2 + NX_2]$

In this case $L(\boldsymbol{x}, X_2) = NX_2 + 1$, $U(\boldsymbol{x}, X_2) = X_2{}^2 + NX_2$, $P(\boldsymbol{x}) = X_1 + 1$, $Q(\boldsymbol{x}) = N$.

Checking Accuracy Condition 1: The monotonicity can be determined by computing the first partial derivative of each function for variable X_2. The partial derivatives are $dL(\boldsymbol{x}, X_2)/dX_2 = N$ and $dU(\boldsymbol{x}, X_2)/dX_2 = 2X_2 + N$. It can be easily shown that they both realize positive values. Therefore L and U are increasing so in this case we compute $\min(U(\boldsymbol{x}, X_2) - L(\boldsymbol{x}, X_2 + 1) + 1) = \min((X_2{}^2 + NX_2) - (N(X_2 + 1) + 1) + 1) = \min(X_2{}^2 - N) \geq (X_1 + 1)^2 - N = X_1{}^2 + 2X_1 - N + 1 \geq 1 + 2 - N + 1 = 4 - N \geq -\infty \ngeq 0.$

So Accuracy Condition 1 here fails. At this point the algorithm employs the restriction domain technique to relax the condition. Note that for $N \leq 4$ the condition is satisfied, therefore by tightening the bounds of N we can define new bounds as $2 \leq N \leq (4, +\infty)$. The upper bound of N has two distinct values, the restricted bound and the regular bound.

Checking Accuracy Condition 2: $\min(Q(\boldsymbol{x}) - P(\boldsymbol{x})) = \min(N - (X_1 + 1)) = \min(N - X_1 - 1) \geq$

$N - (N - 1) - 1 = 0 \geq 0.$

Now we can eliminate X_2 and the variable interval equation is transformed into:

$X_1 + N = [N(X_1 + 1) + 1, N^2 + NN] \Leftrightarrow$
$X_1 + N = [NX_1 + N + 1, 2N^2]$

Next variable to be eliminated is variable X_1.

$N = [NX_1 - X_1 + N + 1, 2N^2 - X_1]$

In this case $L(\boldsymbol{x}, X_1) = NX_1 - X_1 + N + 1$, $U(\boldsymbol{x}, X_1) = 2N^2 - X_1$, $P(\boldsymbol{x}) = 1$, $Q(\boldsymbol{x}) = N - 1$.

Checking Accuracy Condition 1: In order to determine the monotonicity we compute the first partial derivative of L and U for variable X_1. In our case $dL(\boldsymbol{x}, X_1)/dX_1 = N$ and $dU(\boldsymbol{x}, X_1)/dX_1 = -1$. Note that the first is always positive and the second always negative. Therefore since function L is increasing while function U is decreasing Accuracy Condition 1 is satisfied.

Checking Accuracy Condition 2: $\min(Q(\boldsymbol{x}) - P(\boldsymbol{x})) = \min((N - 1) - 1) = \min(N - 2) \geq 2 - 2 = 0 \geq 0.$

After eliminating X_1 the variable interval equation is as follows:

$N = [N - 1 + N + 1, 2N^2 - 1] \Leftrightarrow$
$N = [2N, 2N^2 - 1]$

Finally, we eliminate variable N. First we move it to the right hand side of the equation.

$0 = [N, 2N^2 - N - 1]$

In this case $L(\boldsymbol{x}, N) = N$, $U(\boldsymbol{x}, N) = 2N^2 - N - 1$, $P(\boldsymbol{x}) = 2$, $Q(\boldsymbol{x}) = (4, +\infty)$.

Checking Accuracy Condition 1: We can easily determine that both L and U are increasing for N therefore $\min(U(\boldsymbol{x}, N) - L(\boldsymbol{x}, N+1) + 1) = \min(2N^2 - N - 1 - (N + 1) + 1) = \min(2N^2 - 2N - 1) \geq 2 \times 2^2 - 2 \times 2 - 1 = 3 \geq 0$.

Checking Accuracy Condition 2: $\min(Q(\boldsymbol{x}) - P(\boldsymbol{x})) = \min(4 - 2) = 2 \geq 0$.

Note that the accuracy conditions are tested only for the restricted bounds. In this case both conditions are satisfied. Also note that variable N has two values for upper bound. These two values will produce two distinct integer intervals as follows:

$0 = ([2, +\infty], [2, 27])$

Because zero is not included in the first interval the algorithm concludes that no integer solution to the system and therefore no data dependence exists under the specified direction vector. The second interval defines a region where the expression of the equation on the left hand side realizes every value. If zero lied in that interval we would conclude that a definite dependence exists but that is not the case here.

4.2.6 Addressing Coupled Subscripts and If-Statement Constraints

We address the issue of coupled subscripts and if-statement constraints with same techniques described in Section 3.2.5 and Section 3.2.6. Those techniques can be used in non-linear expressions as well. The algorithm of the equation propagation proceeds by propagating each equation into the rest of the equations, starting from the one with the fewest terms. The propagation of one equation into a second one is similar to Gaussian elimination, i.e. we multiply the two equations by a constant and add them together producing a new equation. If the new equation has fewer terms than the second one, then the new equation replaces the original equation and the algorithm proceeds with the next pair of equations. This method can eliminate the coupling of variables across equations or can reduce the coupling to a minimum number of variables. In case of if-statements our algorithm takes into account the constraints of the conditional part of the if-statements into the dependence problem. Consider the following loop:

Example 4-6

```
if(N > 1) then
  for I = 1 to N do
    for J = 1 to I*I do
      A[2*N*I – J + N, N*I – J + 1] = A[2*J, J] + …
    endfor
  endfor
endif
```

The data dependence problem results into the following system:

$2NX_1 - X_3 - 2X_4 + N = 0$
$NX_1 - X_3 - X_4 = -1$
$2 \le N \le +\infty$
$1 \le X_1 \le N,$ $\qquad\qquad$ $1 \le X_2 \le N$
$1 \le X_3 \le X_1^2,$ $\qquad\qquad$ $1 \le X_4 \le X_2^2$

By propagating the second equation into the first, multiplying by -2 and 1 respectively, and adding them together we produce a new equation $N + X_3 = 2$. The new equation will replace the first equation in the dependence system. Because of the lower bounds of N and X_3 the above equation does not have a solution and thus we conclude that no dependence exists. Subscript by subscript testing without applying the equation propagation and without taking into consideration the if-statement constraint would have failed to disprove the dependence in this case.

4.3 Algorithm and Complexity

The NLVI-Test algorithm follows exactly the same steps as the VI-Test algorithm described in Section 3.3. Before the application of the NLVI-Test a data dependence problem is created for the pair of array references within a nest of loops and a nest of if-statement constraints. After the dependence problem is created a sequence of direction vectors are tested on the problem to verify the existence of data dependence. The direction vector sequence can follow either the dependence level or the full direction vector hierarchy.

First the bounds of each variable are set based on the loop bounds, the inequalities found in the if-statement conditions, and the direction vector constraints as described in Sections 4.2, 4.2.2 and 4.2.6 respectively. The lower and upper bounds of each variable can be expressions of other variables, i.e. trapezoidal or symbolic bounds. All variables have two values for each bound. One is the regular bound and the other is the restricted bound and they are initially these are equal. The values of the restricted bounds change every time we restrict the problem domain in order to meet the accuracy conditions, if necessary, as described in Section 4.2.4. After the bounds are selected and each pair of equal variables is substituted by a single variable, the data dependence equations are constructed from

the array subscripts and the equality constraints found in if-statements. If there is more than one equation in the dependence system we apply the equation propagation technique as described in Section 4.2.6 trying to eliminate or reduce the coupling. If coupling cannot be eliminated the algorithm generates additional equations according to the lambda test algorithm. Next the NLVI-Test proceeds in a subscript-by-subscript fashion testing each equation individually.

In the main part of the algorithm the test proceeds with variable elimination. A DAG is constructed based on the problem constraints similar to the DAG of Example 4-5 in Figure 4-2. Any topological sort of the DAG is a valid order of elimination. Each time a variable is being eliminated the NLVI-Test checks both Accuracy Condition 1 and Accuracy Condition 2 using the minimum and maximum computation described in Section 4.2.3. If any of the conditions fail then the variable interval GCD test is applied in Theorem 4-4. If that fails too then the domain of the problem is restricted by tightening the restricted bounds of some variables as described in Section 4.2.4. Upon termination, the algorithm compares zero against the two constant integer intervals that remain on the right hand side and decides whether a dependence exists or not.

As we already discussed, checking for a condition involves the computation of a minimum or a maximum of an expression. The computation proceeds through a series of variable substitutions. Assuming that the substitution of a single variable with an expression of n variables requires $O(S(n))$ time, then the substitution of n variables would requite $O(nS(n))$. Since all variables are tested for Accuracy Condition 1 and Accuracy Condition 2 the total time for the elimination of n variables would be $O(n^2S(n))$. The function $S(n)$ depends on the type of expressions that the algorithm is applied to.

In integer and rational polynomials the substitution involves raising expressions into powers in each of the terms where the substituted variable appears. The process of raising an expression into a power d involves $O(\log(d))$ multiplications. Each multiplication requires n^2 time so the cost of raising an expression into a power d would be $O(\log(d)n^2)$. In a polynomial of degree d we can have at most d terms where each variable may appear so for each substitution we need to compute at most d powers. Thus, the time required to compute all the powers would be $O(d\log(d)n^2)$. After the powers are computed they are multiplied with each of the polynomial terms that contained the substituted variable and added together. Assuming we can have at most d such term this process takes $O(dn)$ time. Thus the total time for the computation of a minimum or a maximum is $S(n) = O(d\log(d)n^2 + dn)$ This bring the total time complexity of the NLVI-Test to $O(d\log(d)n^4 + dn^3)$ where d is the degree of the polynomial and n the number of variables.

```
DO n0 = 1, n, 1                             DO k = 2, lmi-1, 1
 t(n0) = t(n0) + (-c(n0))*c0                 DO i3 = 1, mm(k), 1
 t(n+n0) = t(n+n0) + (-c(n0))*c1              DO i2 = 1, mm(k), 1
 t(n0+2*n) = t(n0+2*n) + (-c(n0))*c2           DO i1 = 1, mm(k), 1
 t(n0+3*n) = t(n0+3*n) + (-c(n0))*c3            u(-1+ir(k)+i1-mm(k)-mm(k)**2+
 t(n0+4*n) = t(n0+4*n) + (-c(n0))*c4    *         mm(k)*i2+i3*mm(k)**2) = 0.D0
 t(n0+5*n) = t(n0+5*n) + (-c(n0))*c5           ENDDO
ENDDO                                         ENDDO
                                            ENDDO
```

a: BDNA of Perfect   **c:** MGRID of SPEC

```
DO mrs = 1, (num*(num+1))/2, 1              DO j = 3, n-1, 1
 DO mi = 1, num, 1                           DO i = 2, n-1, 1
  DO mj = 1, mi, 1                            r = aa(i,j)*d(i,j-1)
   xrsij(mj+(mi**2-mi-num-num**2+             d(i,j)   =    1.D0/(dd(i,j)-aa(i,j-
*   mrs*num+mrs*num**2)/2) = xij(mj)   1)*r)
  ENDDO                                       rx(i,j) = rx(i,j)-rx(i,j-1)*r
 ENDDO                                        ry(i,j) = ry(i,j)-ry(i,j-1)*r
ENDDO                                        ENDDO
                                            ENDDO
```

b: TRFD of Perfect   **d:** TOMCATV of SPEC

Figure 4-3: Simplified loop examples from the Perfect and SPEC benchmarks

4.4 Real Code Examples

In this section we provide examples from real scientific benchmarks in order to demonstrate the practical importance of the NLVI-Test and identify cases where it enhances program parallelization and execution performance compared to other tests. The examples are in Fortran 77 code and were selected from the Perfect and SPEC benchmarks. Simplified versions of the loop-nests are depicted in Figure 4-3 after inter-procedural constant propagation and induction variable recognition has been performed.

The first example is from BDNA of the Perfect benchmarks. The loop is taken from the CORR56 subroutine and it is displayed in Figure 4-3.a. The upper bound of loop variable *n0* is the symbolic variable *n* that also appears in the subscripts of array *t*. Traditional data dependence tests like the Banerjee test and the I-Test do not perform symbolic analysis and therefore they cannot disprove the dependences between the instances of the array *t* accesses. The NLVI-Test as well as the Omega test and the Range test can handle symbolic loop bounds and in this case they can parallelize the loop. Parallelization of this loop results into additional speedup in BDNA as we will see in Section 5.5. The loop-nest in Figure 4-3.b is from the OLDA subroutine in the TRFD Perfect benchmark. Because of induction variable recognition the loop bounds and the array subscripts become non-linear. The I-Test and the Omega test cannot analyze non-linear expressions and therefore they cannot disprove the dependence in this case. The NLVI-Test as well as the Range test can analyze the non-linear

69

expressions and as a result they can parallelize the outermost loop. Without proper parallelization of this particular loop-nest, the TRFD benchmark does not yield any speedup.

Next consider the loop-nest in Figure 4-3.c. This example is from MGRID of the SPEC benchmarks. In this loop-nest we can see that for equal values of the loop variable k all occurrences of the array values $mm(k)$ are equal. The NLVI-Test with non-linear dependence analysis can disprove the dependences within the same iteration of loop k, i.e. for the (=, *, *, *) direction vector, and parallelize loop $i3$. The Range test cannot disprove such type of dependences because the range propagation technique does not consider direction vector constraints. In this case the Range test computes inaccurate ranges on the expressions of $mm(k)$ and as a result it cannot parallelize the loop. In general the range propagation technique is not as accurate as the polynomial expression arithmetic that is implemented in the NLVI-Test. There exist numerous examples of loops in the Perfect and SPEC benchmarks that are parallelized by the NLVI-Test and not by the Range test, but only few actually contribute to program execution performance. Parallelization of this particular loop, however, does in fact increase the speedup of MGRID as we will see in Section 5.5. The I-Test and the Omega test do not perform non-linear analysis and therefore they cannot parallelize the loop in Figure 4-3.c. Finally consider the loop-nest in Figure 4-3.d. The code appears in the TOMCATV benchmark. In this example all the dependences are carried by the outer loop and none by the inner loop. In addition all dependences occur only for a (<, =) direction vector. The Range test only computes the level of the dependence instead of the complete direction vector. Therefore, it determines dependences with (<, *) direction vector and results into parallelization of loop i in its current position. Complete direction vector information computed by the NLVI-Test, as well as the I-Test and the Omega test, enables the loop interchange transformation to move the parallel loop i in the outermost position where it yields additional speedup in the TOMCATV benchmark.

Chapter 5

Experimental Results and Conclusions

For our experiments we compared our dependence analysis techniques against the I-Test, the Omega test, and the Range test, which represent the current state of the art in data dependence analysis. In this chapter we present the empirical results on how accurate, efficient, and effective the examined data dependence tests are in practice.

We performed our experiments using the Perfect Club Benchmarks, the SPEC benchmarks, and the Lapack library. The Perfect benchmarks are a collection of thirteen scientific and engineering FORTRAN 77 programs that are representative of high performance applications. For our experiments we used all 13 programs, namely ADM, SPICE, QCD, MDG, TRACK, BDNA, OCEAN, DYFESM, MG3D, ARC2D, FLO52, TRFD, and SPEC77. The SPEC benchmark suite is a set of programs written in various programming languages that can be used to compare computers in terms of execution performance. We selected all 10 FORTRAN 77 programs from the SPEC 95 and SPEC 2000 benchmarks suites, namely APPLU, APSI, HYDRO2D, MGRID, SU2COR, SWIM, TOMCATV, TURB3D, WAVE5, and WUPWISE. Lapack is a linear algebra library for high-performance computers. It includes a collection of FORTRAN 77 subroutines for solving the most common problems in numerical linear algebra, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. Its core consists of a standard set of Basic Linear Algebra Subprograms (BLAS). Lapack is the successor to the older Linpack and Eispack libraries. All codes used for our experiments are computationally intense programs that are representative of applications executed on high-performance computers.

5.1 Implementation details

We conducted our experiments using the Polaris compiler that was developed at the University of Illinois at Urbana Champaign and Purdue University [8]. Polaris is a restructuring compiler that automatically parallelizes FORTRAN 77 programs for execution on shared memory multiprocessors.

Polaris produces parallel code in various formats including OpenMP that we used for our experiments.

We have debugged and extended the Polaris compiler in order to compile our benchmarks and produce the required statistics. The I-Test, the VI-Test and the NLVI-Test have been implemented in C++ along with other dependence tests as part of the PLATO library. In particular, we have implemented the VI-Test for linear expression and the NLVI-Test for integer and rational multivariate polynomials. Consequently, we have created the necessary classes and data structures to support arithmetic of linear and polynomial expressions including addition, multiplication, and power operators as well as and function composition and derivative computation. We ported the PLATO library into Polaris and implemented the interface for the I-Test, the VI-Test, and the NLVI-Test. The interface can be used to conduct data dependence analysis by using the dependence level hierarchy that computes only the level of the dependence, or by using the direction vector hierarchy [9] that computes complete direction vector information.

The Omega test was developed as part of the Omega project in a C++ library [32] and it has been ported in the Polaris compiler. We have extended the interface of the Omega test in the Polaris compiler to take into account if-statement constraints in the dependence problem. Finally the only implementation of the Range tests exists as part of the Polaris compiler and it is dependent on the compiler's internal program representation. In addition, it is based on the range propagation technique, which is only implemented in Polaris. This implementation of the Range test as opposed to a stand alone library makes it hard to port into other compilers.

Before applying any dependence test, constant propagation, induction variable recognition, reduction, inline expansion, scalar and array privatization, loop normalization and several other transformations are applied by Polaris [8]. We have also implemented, as an extension to Polaris, a simple loop interchange routine that can move, when legal, the parallel loops to the outermost position, where they can yield better execution performance on shared memory multiprocessors.

In the rest of the chapter we present our experimental results in terms of accuracy, efficiency, and effectiveness in parallelization and program execution performance in comprehensive charts and tables. At the end of the chapter we present our conclusions.

5.2 Data Dependence Accuracy Results

For our dependence accuracy results we considered array references inside a common nest of perfectly nested or non-perfectly nested loops, since only those can directly affect parallelization. Two such array references with at least one of them being a "write" into memory constitute a data
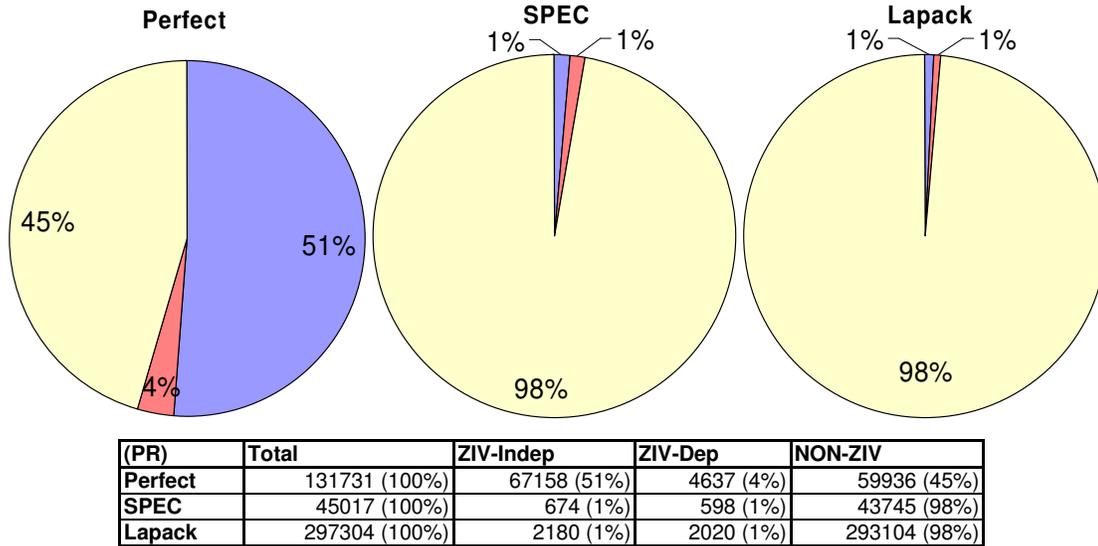
Figure 5-1: Percentage of ZIV problems in Perfect, SPEC and Lapack.

(PR)	Total	ZIV-Indep	ZIV-Dep	NON-ZIV
Perfect	131731 (100%)	67158 (51%)	4637 (4%)	59936 (45%)
SPEC	45017 (100%)	674 (1%)	598 (1%)	43745 (98%)
Lapack	297304 (100%)	2180 (1%)	2020 (1%)	293104 (98%)

dependence problem. When the dependence exists, an arc is created in the data dependence graph between the statements that contain the two array references. When these two array references are commonly nested in n loops, then a total of 3^n direction vectors are considered. Each arc of the data dependence graph stores the set of all possible direction vectors. This information helps the compiler decide which loops carry the dependence and which loops may be executed in parallel.

In the I-Test, the VI-Test, the NLVI-Test and the Omega Test we employed the direction vector hierarchy to determine the direction vectors. According to this procedure the compiler starts testing for dependence with (*, *, …, *) direction vector and recursively refines the direction vectors by testing for (<, *, …, *), (=, *, …, *), and (>, *, …, *) as described in [9]. When we reach at the leaves of the tree, where all direction vectors are totally refined, we store the dependence relations if they exist. The direction vector hierarchy has exponential time complexity. The Range Test follows the dependence level hierarchy. The difference is that the algorithm recursively refines only direction vectors of the form (=, *, …, *) and therefore it has linear time complexity.

In our experiments, private variables are not taken into consideration and are excluded from our results. We also separate the dependence problems for arrays with constant subscripts from the dependence problems for arrays with variable subscripts. The former can be resolved by simply comparing the values of the constant subscripts to determine whether dependence exists or not. This test is known as the constant or ZIV (zero index variables) test [14]. The later require a more elaborate data dependence test.

73

		Test	Total	Independent	Dependent	Maybe
	PR	I-Test	59936 (100%)	17947 (30%)	4689 (8%)	37276 (62%)
		Omega	59936 (100%)	21232 (35%)	6487 (11%)	32217 (54%)
		Range	59936 (100%)	17615 (29%)	N/A	42321 (71%)
		VI-Test	59936 (100%)	18469 (31%)	5526 (9%)	35941 (60%)
		NLVI-Test	59936 (100%)	19033 (32%)	6973 (12%)	33930 (57%)
	DV	I-Test	665304 (100%)	427422 (64%)	7981 (1%)	229901 (35%)
		Omega	665304 (100%)	450261 (67%)	11699 (2%)	203344 (31%)
		Range	665304 (100%)	404716 (61%)	N/A	260588 (39%)
		VI-Test	665304 (100%)	433455 (65%)	10159 (2%)	221690 (33%)
		NLVI-Test	665304 (100%)	446934 (67%)	14517 (2%)	203853 (31%)

Figure 5-2: Data dependence accuracy in the Perfect benchmarks

Figure 5-1 displays how many dependence problems were found in each benchmark and how many were resolved by the ZIV test. We see that the number of problems resolved by the ZIV test (ZIV problems) in the Perfect benchmarks account for 55% of the total problems, with 51% found independent and 4% found dependent. In the SPEC benchmarks and the Lapack library the problems resolved by the ZIV test are very few. They only account for 2% of the total problems, with 1% independent and 1% dependent. The rest of the problems are analyzed further by each data dependence test.

5.2.1 Dependence Accuracy in the Perfect Benchmarks

For the data dependence problems involving arrays with variable subscripts we applied each of the data dependence tests separately. Figure 5-2 summarizes the accuracy results for each data dependence test in the Perfect benchmarks. Both data dependence problems (PR) and direction vector (DV) results are reported. The charts and tables illustrate how many of the total dependence problems and the total direction vectors each test found independent, how many were proved dependent, and how many times each test returned an inconclusive answer. Note that the Range test can never

conclusively prove data dependences and thus it reports an inconclusive answer when the dependence cannot be disproved.

From the experimental results in the Perfect benchmarks we observe that the Omega test has the best overall accuracy with 35% disproved and 10% proved dependent problems. The Omega test also detected 67% independent and 1% dependent direction vectors. The NLVI-Test found 32% independent and 12% dependent problems and 67% independent and 1% dependent direction vectors. The difference in accuracy between the Omega test and the NLVI-Test is only by 3% and it is due exclusively to the if-statement constraints that the NLVI-Test was not able to resolve. If it was not for the if-statement constraints the NLVI-Test would have surpassed the Omega test in data dependence accuracy. Even though the if-statement constraints contribute to the increased accuracy of the Omega test, over the other tests, they do not contribute nearly anything in program parallelization as we will see in Section 5.4. The NLVI-Test test was able to break 1% (32% vs. 31%) more data dependence problems and 2% (67% vs. 65%) more direction vector problems than the linear VI-Test. In addition, the NLVI-Test test was able to prove about 3% (12% vs. 9%) more non-linear data dependent problems than the VI-Test. The I-Test and the Range test follow in accuracy, with the I-Test disproving 30% of the dependence problems as opposed to 29% for the Range test. In addition, the I-Test can prove dependence in 8% of the problems, where the Range test cannot. The NLVI-Test in this case even though it is more accurate than the I-Test and the Range test, it does not break as many dependences as the Omega test.

From the direction vector results we observe that the percentage of independence in all tests has significantly increased. This is because even if two array references are data dependent most of their direction vectors can be found independent. The Omega test was able disprove almost as many direction vector as the NLVI-Test and not many more than the other tests except, for the Range test. When it comes to direction vectors the I-Test disproved 64% of all possible direction vectors, while the Range test disproved only 61% since it does not compute complete direction vector information. Because of this limitation the Range test assumes more dependent directions vectors than they actually exist.

In Figure 5-3 we present the actual reason or combination of reasons for any inexact (maybe) result in the Perfect benchmarks. The LV column counts the number of dependence problems that involved loop variant variables and the dependence could not be disproved. The NL column counts the number of problems with non-linear expressions that each test could not handle and were not proved independent. Such expressions may appear in the subscripts, loop bounds, or if-statement conditions. The kind of non-linear expressions that the NLVI-Test cannot analyze are exponential expressions,

min/max functions, and other user defined functions. The IF column shows the number of problems for which additional constraints due to if-statements were ignored and the dependence could not be disproved. The CE column counts the number of dependence problems that include equations with coupled variables. Equations with coupled variables arise from array references with coupled subscripts or from if-statement equalities, in the case of the VI-Test and the NLVI-Test. The UB column counts the number of problems that involve variables with unknown or inexact bounds. Finally, the last column, NI count the number of problems where an integer solution could not be determined in the dependence problem. A data dependence test may result to an inexact answer because of more than one of the above reasons. In our experiments we report all the applicable reasons for which a data dependence test returned a maybe answer. Therefore, the sum of the numbers in the six columns of the table in Figure 5-3 exceeds the number in the Maybe column of the table in Figure 5-2. We present the results for both data dependence problems (PR) and direction vectors (DV).

The main reason of inaccuracy for all tests is expressions that contain loop variant variables. They account for 44% of the total dependence problems (25% total direction vectors) for the I-Test, 45% of the total dependence problems (24% total direction vectors) for the Omega test, and about 48% of the problems (26% direction vectors) for the VI-Test and NLVI-Test. The number of problems with loop variants is higher for the VI-Test, the NLVI-Test and the Omega test because they take into consideration if-statement constraints that may contain additional loop variants. Induction variable recognition is used to express some of the loop variants as functions of the loop indices. When the transformation is successful non-linear expressions may appear in the array subscripts, loop bounds and if-statement conditions. The number of problems that resulted in a maybe answer due to non-linear constrains account for 25% of the total dependence problems (21% of the total direction vectors) for the I-Test, 23% of the total dependence problems (21% of the total direction vectors) for the Omega test, and 26% of the total dependence problems (23% of the total direction vectors) for the VI-Test. Many of those non-linear expressions are produced by compiler transformations, such as induction variable substitution, loop normalization, and expression propagation. Even though these transformations can help discover more parallelism in scientific applications, they may also result into non-linear expressions. Note, that a data dependence test may break a dependence relation even if there exist loop variant variables or non-linear constraints in the problem. This is why the number of maybes due to loop variants and non-linear constraints is different for each dependence test. The NLVI-Test with non-linear dependence analysis has significantly reduced the number of maybe
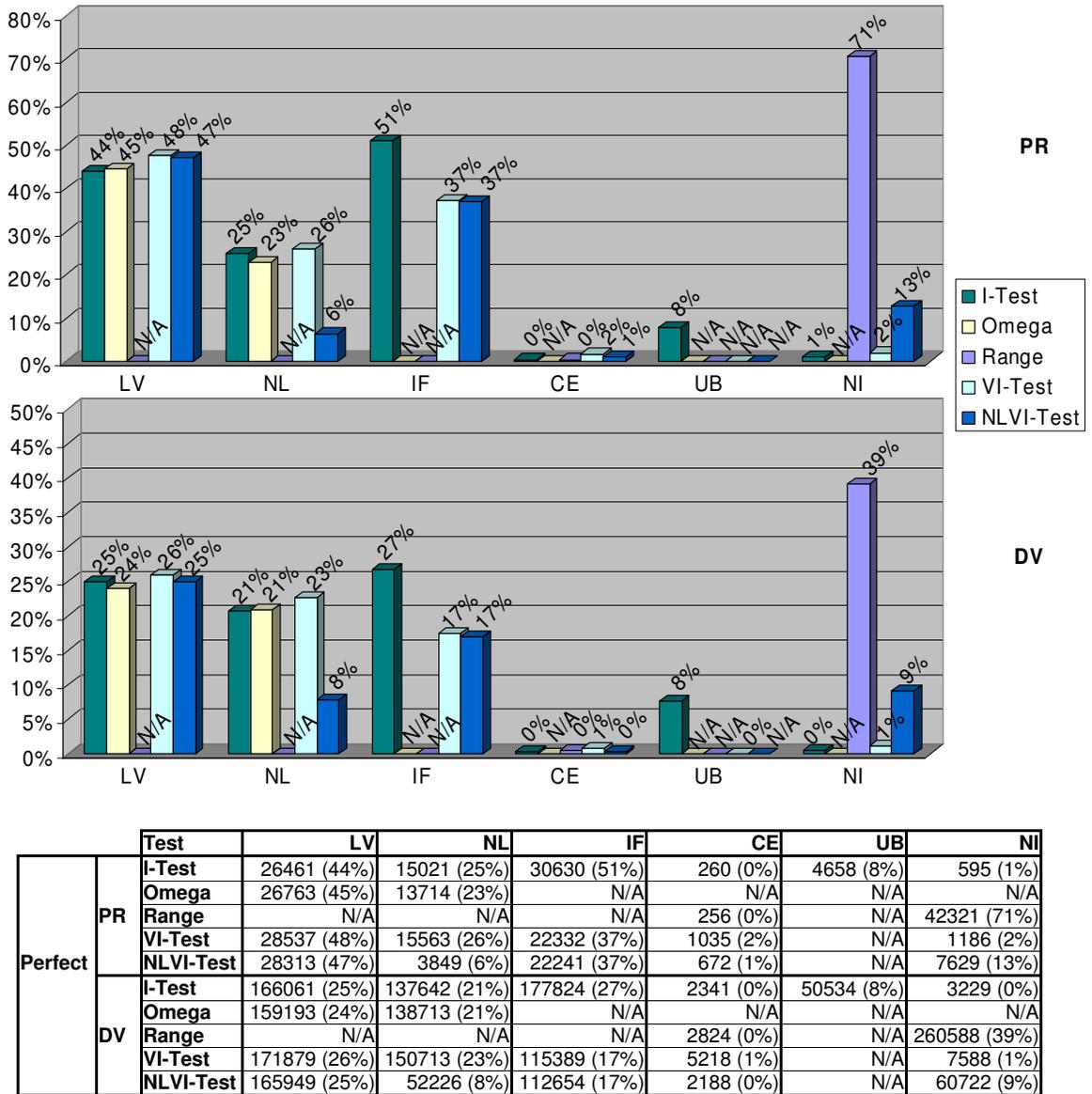
Figure 5-3: Reasons of inaccuracy in the Perfect benchmarks

		Test	LV	NL	IF	CE	UB	NI
Perfect	**PR**	**I-Test**	26461 (44%)	15021 (25%)	30630 (51%)	260 (0%)	4658 (8%)	595 (1%)
		Omega	26763 (45%)	13714 (23%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	256 (0%)	N/A	42321 (71%)
		VI-Test	28537 (48%)	15563 (26%)	22332 (37%)	1035 (2%)	N/A	1186 (2%)
		NLVI-Test	28313 (47%)	3849 (6%)	22241 (37%)	672 (1%)	N/A	7629 (13%)
	DV	**I-Test**	166061 (25%)	137642 (21%)	177824 (27%)	2341 (0%)	50534 (8%)	3229 (0%)
		Omega	159193 (24%)	138713 (21%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	2824 (0%)	N/A	260588 (39%)
		VI-Test	171879 (26%)	150713 (23%)	115389 (17%)	5218 (1%)	N/A	7588 (1%)
		NLVI-Test	165949 (25%)	52226 (8%)	112654 (17%)	2188 (0%)	N/A	60722 (9%)

answers due to non-linear expressions to 6% of the total dependence problems and 8% of the total direction vectors.

The most important reason of inaccuracy for the I-Test, is the if-statement constraints, which account for 51% of the total dependence problems (27% of the total direction vectors). Many of them are the result of transformations such as induction variable substitution and control flow normalization. The VI-Test and the NLVI-Test, with the simple techniques described in Section 3.2.6,

have significantly reduced the inaccuracy due to if-statement constraints even though the percentage remains still high. This is 37% for dependence problem and 17% for direction vectors.

Loops with unknown (triangular or symbolic) bounds that resulted into inexact answers for the I-Test, account for 8% of the total dependence problems and 8% of the total direction vectors. Unknown bounds is not a reason of inaccuracy for the VI-Test or the NLVI-Test. However, the limited number of problems with complex loop bounds does not significantly increase the accuracy of either test.

The Range test cannot prove dependences and this is a reason of inaccuracy in 71% of the total dependence problems and 39% of the total direction vectors. The I-Test returned a maybe answer in only 1% of the problems and very few direction vectors, because it detected real solutions instead of integer ones. This only occurs when the accuracy conditions of the I-Test fail, while the dependence could not be disproved. The VI-Test and the NLVI-Test returned a maybe answer in 2% and 13% respectively of the total problems and 1% and 9% of the total direction vectors. This occurred when their corresponding accuracy conditions failed and the dependence could not be disproved.

Finally, we note that coupled subscripts do not present a significant reason of inaccuracy in the Perfect benchmarks even though the percentage for the VI-Test and the NLVI-Test is a bit higher accounting for about 1% of the problems and 1% of the direction vectors. This percentage has slightly increased because of if-statement equalities that have been included in the system as equations.

5.2.2 Dependence Accuracy in the SPEC Benchmarks

The results for the SPEC benchmarks follow a similar pattern, Figure 5-4. The Omega test is the most accurate test with 52% proved independent and 8% proved dependent of the total dependence problems. The NLVI-Test follows with 50% independent and 8% dependent. The difference here is also due to the if-statement constraints. Without them the NLVI-Test would have reported better accuracy results than the Omega test. The VI-Test is almost as accurate as the NLVI-Test in the SPEC benchmarks with 50% independent and 8% dependent problems. The I-Test and the Range test found 48% independent problems, while 7% of the problems were proved dependent by the I-Test. When it comes to direction vectors the NLVI-Test performs as well as the Omega test disproving 68% and proving 1% of the total direction vectors. The VI-Test and the I-Test comes also very close with 67% independent and 1% dependent direction vectors. The Range test in the SPEC benchmarks can disprove only 53% of all possible direction vectors and therefore significantly lacks in accuracy compared to the other tests.

		Test	Total	Independent	Dependent	Maybe
SPEC	PR	I-Test	43745 (100%)	21287 (48%)	2951 (7%)	19507 (45%)
		Omega	43745 (100%)	22978 (52%)	3494 (8%)	17273 (39%)
		Range	43745 (100%)	21184 (48%)	N/A	22561 (52%)
		VI-Test	43745 (100%)	21553 (50%)	3279 (7%)	18913 (43%)
		NLVI-Test	43745 (100%)	21976 (50%)	3538 (8%)	18231 (42%)
	DV	I-Test	1034319 (100%)	695748 (67%)	7142 (1%)	331429 (32%)
		Omega	1034319 (100%)	706144 (68%)	9852 (1%)	318323 (31%)
		Range	1034319 (100%)	551212 (53%)	N/A	483107 (47%)
			1034319 (100%)	699061 (67%)	9312 (1%)	325946 (32%)
		NLVI-Test	1034319 (100%)	702708 (68%)	9915 (1%)	321696 (31%)

Figure 5-4: Data dependence accuracy in the SPEC benchmarks

The chart and table in Figure 5-5 summarize the reasons of inaccuracy of the dependence tests in the SPEC benchmarks. The main reason of inaccuracy is still the loop variant variables. The percentage of dependence problems that contained loop variants while the dependence could not be disproved ranges from 37% to 39%. In terms of direction vector problems the percentage ranges between 30% and 31%. The number of dependence problems that resulted into maybe answers because of unresolved non-linear expressions is 35% for the I-Test and the VI-Test and 32% for the Omega test as it managed disprove some of them. In terms of direction vector problems, the unresolved non-linear expressions account for about 30% of the maybe answers. The NLVI-Test reduces the percentage of maybes due to non-linear expressions down to 23% for dependence problems and 20% for direction vectors. The remaining unresolved cases are mostly exponential and min/max functions.

In terms of if-statement constraints the VI-Test and the NLVI-Test reduce the percentage of inaccuracy from 28% to 24% compared to the I-Test results. In addition the VI-Test and the NLVI-Test eliminate the issue of unknown loop bounds which constitutes a considerable reason of inaccuracy for the I-Test in 14% of the dependence problems and 12%.of the direction vectors. The
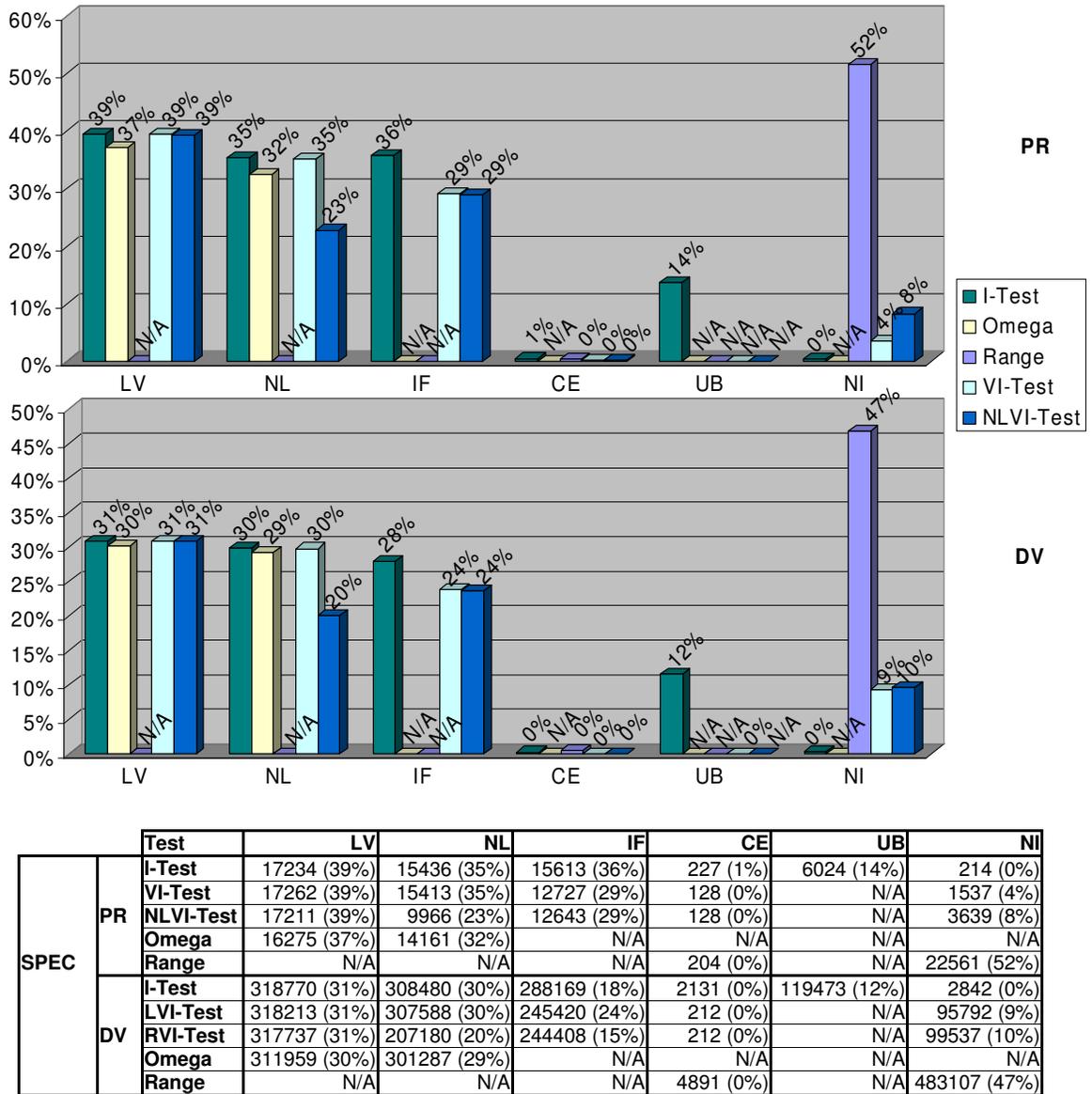
Figure 5-5: Reasons of inaccuracy in the SPEC benchmarks

		Test	LV	NL	IF	CE	UB	NI
SPEC	PR	I-Test	17234 (39%)	15436 (35%)	15613 (36%)	227 (1%)	6024 (14%)	214 (0%)
		VI-Test	17262 (39%)	15413 (35%)	12727 (29%)	128 (0%)	N/A	1537 (4%)
		NLVI-Test	17211 (39%)	9966 (23%)	12643 (29%)	128 (0%)	N/A	3639 (8%)
		Omega	16275 (37%)	14161 (32%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	204 (0%)	N/A	22561 (52%)
	DV	I-Test	318770 (31%)	308480 (30%)	288169 (18%)	2131 (0%)	119473 (12%)	2842 (0%)
		LVI-Test	318213 (31%)	307588 (30%)	245420 (24%)	212 (0%)	N/A	95792 (9%)
		RVI-Test	317737 (31%)	207180 (20%)	244408 (15%)	212 (0%)	N/A	99537 (10%)
		Omega	311959 (30%)	301287 (29%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	4891 (0%)	N/A	483107 (47%)

ability of the VI-Test and the NLVI-Test to handle complex loop regions, is what primarily contributes to their increased dependence accuracy in the SPEC benchmarks. Dependence problems with coupled constraints are not a reason of inaccuracy in the SPEC benchmarks as was the case in the Perfect benchmarks.

Finally, there were very few cases where the accuracy conditions of the I-Test failed. The VI-Test and the NLVI-Test were not able to prove integer solutions to the dependence problem in 4% and 8% of the total dependence problems (9% and 10% of the total direction vectors) respectively. The Range
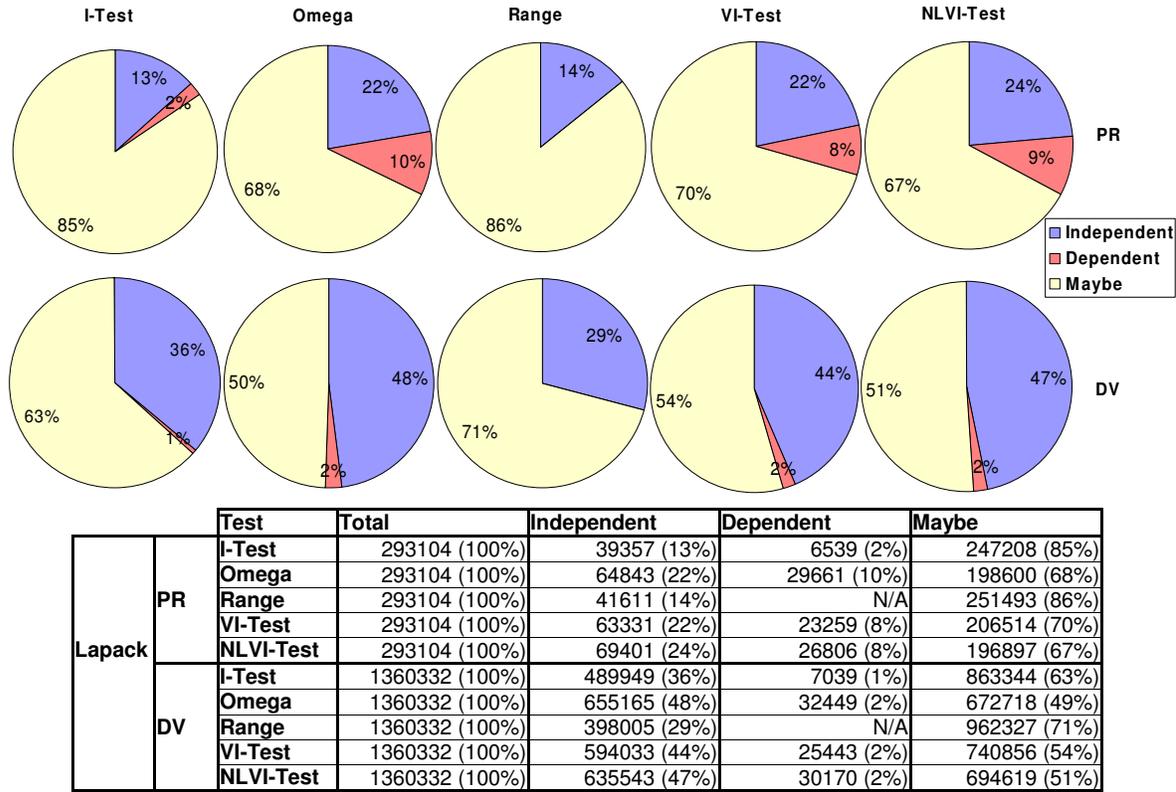
Figure 5-6: Data dependence accuracy in the Lapack library

		Test	Total	Independent	Dependent	Maybe
Lapack	**PR**	I-Test	293104 (100%)	39357 (13%)	6539 (2%)	247208 (85%)
		Omega	293104 (100%)	64843 (22%)	29661 (10%)	198600 (68%)
		Range	293104 (100%)	41611 (14%)	N/A	251493 (86%)
		VI-Test	293104 (100%)	63331 (22%)	23259 (8%)	206514 (70%)
		NLVI-Test	293104 (100%)	69401 (24%)	26806 (8%)	196897 (67%)
	DV	I-Test	1360332 (100%)	489949 (36%)	7039 (1%)	863344 (63%)
		Omega	1360332 (100%)	655165 (48%)	32449 (2%)	672718 (49%)
		Range	1360332 (100%)	398005 (29%)	N/A	962327 (71%)
		VI-Test	1360332 (100%)	594033 (44%)	25443 (2%)	740856 (54%)
		NLVI-Test	1360332 (100%)	635543 (47%)	30170 (2%)	694619 (51%)

test cannot prove dependence which was the case in all 52% of the total dependence problems (47% of the total direction vectors) that could not be disproved.

5.2.3 Dependence Accuracy in the Lapack Library

The experimental results we obtained from the Lapack library are displayed in Figure 5-6. Unlike the Perfect and SPEC benchmarks, there is a large number of loops with unknown bounds, coupled subscripts, and if-statements, which constitute the primary reason of the big difference in accuracy between the Omega test and the I-Test.

From the experimental results we observe that the NLVI-Test has the best overall accuracy with 24% disproved and 9% proved dependent problems. The NLVI-Test test also detected 47% independent and 2% dependent direction vectors. The Omega test follows in data dependence accuracy with 22% independent and 10% dependent problems and 48% independent and 2% dependent direction vectors. The results indicate a significant improvement in accuracy by applying the VI-Test and the NLVI-Test over the I-Test. The VI-Test alone has reached the accuracy of the

Omega test in data dependence problems by disproving 22% and proving 8% of the total. Looking at the direction vector results we notice that the Omega test can break 4% more than the VI-Test.

The Range test and the I-Test are in the last place, with the Range test disproving 14% of the dependence problems as opposed to 13% for the I-Test. In addition the I-Test can prove dependence in only 2% of the problems, where the Range test cannot. When it comes to direction vectors the I-Test disproved 36% of all possible direction vectors, while the Range test disproved only 29% since it does not compute complete direction vector information. Because of this limitation the Range test assumes more dependent directions vectors than they actually exist. The percentage of inaccuracy for the Range test and the I-Test is very high, reaching 86% and 85% respectively compared to 68% for the Omega test and 76% for the NLVI-Test. The difference is higher than what we observed in the Perfect and SPEC benchmarks and demonstrates the degree of improvement that the VI-Test and the NLVI-Test can achieve in data dependence analysis.

In Figure 5-7 we present the experimental results measuring the reasons of inaccuracy in the Lapack library. We can see that the main reason of inaccuracy for all data dependence tests is the loop variant variables in the problem constraints. They account for 69% of the total dependence problems (58% of the total direction vectors) for the I-Test, 65% of the total dependence problems (48% of the total direction vectors) for the Omega test, 65% of the total dependence problems (52% of the total direction vectors) for the VI-Test and 63% of the total dependence problems (50% of the total direction vectors) for the NLVI-Test.. The number of problems that resulted in a maybe answer due to non-linear constrains account for 21% of the total dependence problems (19% of the total direction vectors) for the I-Test, 24% of the total dependence problems (18% of the total direction vectors) for the Omega test, 23% of the total dependence problems (20% of the total direction vectors) for the VI-Test and 12% of the total dependence problems (10% of the total direction vectors) for the NLVI-Test. We can see that the NLVI-Test has significantly reduced the amount of unresolved non-linear expressions by 11% compared to the VI-Test. The remaining percentage is mostly due to min/max functions appearing in the source code expressions. In the Lapack library we observed a high percentage of inexact answers due to if-statements for the I-Test, the VI-Test and the NLVI-Test. These account for 68%, 34% and 35% of the total dependence problems (54%, 29% and 28% of the total direction vectors) respectively. The VI-Test and NLVI-Test, with the simple techniques described in Section 3.2.6, has significantly reduced the percentage of problems that produced inexact answers because of if-statements as apposed to the I-Test.

		Test	LV	NL	IF	CE	UB	NI
Lapack	**PR**	**I-Test**	201666 (69%)	62062 (21%)	199306 (68%)	97892 (33%)	118794 (41%)	0 (0%)
		Omega	188774 (64%)	69282 (24%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	97396 (33%)	N/A	251493 (86%)
		LVI-Test	189630 (65%)	68054 (23%)	100589 (34%)	68286 (23%)	N/A	47222 (16%)
		RVI-Test	185046 (63%)	35152 (12%)	101465 (35%)	65788 (22%)	N/A	45232 (15%)
	DV	**I-Test**	792840 (58%)	256360 (19%)	728052 (54%)	319766 (24%)	484572 (36%)	0 (0%)
		Omega	650842 (48%)	250386 (18%)	N/A	N/A	N/A	N/A
		Range	N/A	N/A	N/A	359010 (26%)	N/A	962327 (71%)
		LVI-Test	713898 (52%)	267860 (20%)	395395 (29%)	294184 (22%)	N/A	231469 (17%)
		RVI-Test	680196 (50%)	131178 (10%)	387111 (28%)	283974 (21%)	N/A	195378 (14%)

Figure 5-7: Reasons of inaccuracy in the Lapack library

In addition, there is a high percentage of inexact answers due to unknown bounds for the I-Test, which is 41% of the total dependence problems and 36% of the total direction vectors. Unlike the results in the Perfect and SPEC benchmarks, coupled subscripts were a significant reason of inaccuracy for the I-Test in the Lapack library reaching 33% of the total dependence problems and 26% of the total direction vectors.Because of the techniques described in 2.4, the VI-Test and NLVI-Test reduced the number of inaccurate results, due to coupled subscripts, to about 22% of the total dependence problems and 21% of the total direction vectors. The significant percentage of problems

with non-linear expressions, if-statements, unknown bounds, and coupled subscripts, give the opportunity to NLVI-Test to improve the overall accuracy over the I-Test and even exceed the accuracy of the Omega test in terms of dependence problems.

Finally, the accuracy conditions of the I-Test never failed and integer solutions where always proved, while that was not the case in 16% of the dependence problems and 17% of direction vectors for the VI-Test, 15% of the dependence problems and 14% of direction vectors for the NLVI-Test, since their accuracy conditions are more restrictive than those of the I-Test. These percentages may seem high but what it really means is that, given the problem constraints the NLVI-Test conclusively proved or disproved integer solutions in 85% of the problems.

5.3 Efficiency Results

The efficiency of the dependence analysis phase and its percentage of the total compilation time are very important to ensure that a dependence test is practical for use in real compilers. We measured the efficiency of our dependence analysis tool and compared it with the efficiency of the other data dependence tests in the Perfect benchmarks, the SPEC benchmarks, and the Lapack library. We calculate the cost of dependence analysis and we analyze the tradeoff between accuracy and efficiency by the comparing average time per dependence problem for each test. In addition, we measure the impact of data dependence analysis on the compilation process and we display the time and percentage of the dependence analysis phase against the total compilation time. For each data dependence algorithm we present the total compilation time, the time required building the data dependence graphs for all source code, and the time spent in each data dependence test individually. The results gathered are indicative of how practical each data dependence test is in high performance compilers. The efficiency experiments were conducted on a Sun UltraSPARC-IIIi with 1 GHz CPU and 1 GB main memory.

Figure 5-8 summarizes the efficiency results in the Perfect benchmarks. The first chart displays the average time per dependence problem. It also reports the average time required for each test to break a dependence, the average time to prove a dependence and the average time in case of inexact answers. In the Perfect benchmarks the I-Test and the Range test were the most efficient dependence tests. Their average cost per dependence problem was about 1 msec and 2 msec respectively. The VI-Test was also very efficient taking on average about 6 msec per dependence problem. The NLVI-Test was a bit slower taking on average about 18 msec per dependence problem. For the Omega test, the average cost per dependence problem was about 83 msec, which is about two orders of magnitude the cost of the I-Test. Note, that the Omega test was faster when it was accurate. It took on average 25
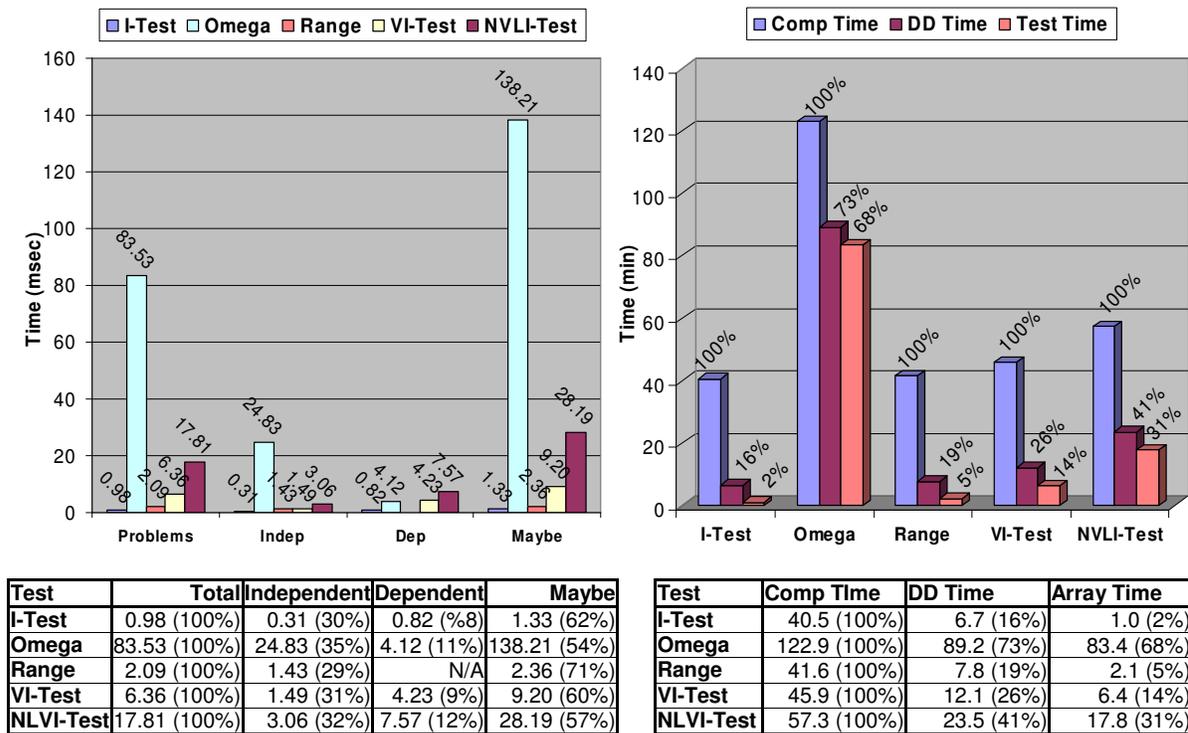
Figure 5-8: Time per dependence problem and compilation efficiency in the Perfect benchmarks

Test	Total	Independent	Dependent	Maybe
I-Test	0.98 (100%)	0.31 (30%)	0.82 (%8)	1.33 (62%)
Omega	83.53 (100%)	24.83 (35%)	4.12 (11%)	138.21 (54%)
Range	2.09 (100%)	1.43 (29%)	N/A	2.36 (71%)
VI-Test	6.36 (100%)	1.49 (31%)	4.23 (9%)	9.20 (60%)
NLVI-Test	17.81 (100%)	3.06 (32%)	7.57 (12%)	28.19 (57%)

Test	Comp TIme	DD Time	Array Time
I-Test	40.5 (100%)	6.7 (16%)	1.0 (2%)
Omega	122.9 (100%)	89.2 (73%)	83.4 (68%)
Range	41.6 (100%)	7.8 (19%)	2.1 (5%)
VI-Test	45.9 (100%)	12.1 (26%)	6.4 (14%)
NLVI-Test	57.3 (100%)	23.5 (41%)	17.8 (31%)

msec when it broke a dependence, and only 4 msec when it proved a dependence. However, all the other tests were even more efficient when they were accurate. The NLVI-Test took 3 msec on average to break a dependence which is still substantially faster than the Omega test. The time jumps to 138 msec per dependence problem on average when the Omega test was inaccurate. It is obvious that the Omega test pays a high price for the additional dependences it can break.

The second chart in Figure 5-8 displays the data dependence analysis time and the data dependence testing time as a percentage of the total compilation time. The data dependence analysis time is the time taken to build the data dependence graph for all arrays and scalar variables in each benchmark. The data dependence testing time is the time taken exclusively by each test to compute the dependence information for arrays in each benchmark. The I-Test, which is the most efficient test, took 1.0 minute to analyze all data dependence problems and direction vectors in the Perfect benchmarks, which is 2% of the total compilation time. The Range test was also very efficient taking 2.1 minutes or 5% of the total compilation time in the Perfect benchmarks. The dependence testing time for all thirteen Perfect benchmarks accounts for 14% of the compilation time with the VI-Test. This is a little over 6 minutes. The NLVI-Test spent 18 minutes analyzing all data dependence problems and direction vectors in the Perfect benchmarks which accounts for 31% of time

Test	Total	Independent	Dependent	Maybe
I-Test	2.0 (100%)4	0.25 (49%)	1.57 (7%)	4.06 (45%)
Omega	133.46 (100%)	1.40 (53%)	6.29 (8%)	334.86 (39%)
Range	2.06 (100%)	0.28 (48%)	N/A	3.74 (52%)
VI-Test	10.89 (100%)	0.70 (50%)	7.04 (%7)	23.16 (43%)
NLVI-Test	21.11 (100%)	1.30 (50%)	11.11 (8%)	46.95 (42%)

Test	Comp TIme	DD Time	Array Time
I-Test	14.0 (100%)	2.8 (20%)	1.5 (11%)
Omega	109.9 (100%)	98.6 (90%)	97.3 (89%)
Range	14.1 (100%)	2.8 (20%)	1.5 (11%)
VI-Test	20.5 (100%)	9.2 (45%)	7.9 (39%)
NLVI-Test	28.0 (100%)	16.7 (60%)	15.4 (55%)

Figure 5-9: Time per dependence problem and compilation efficiency in the SPEC benchmarks

compilation time. This percentage jumped to 68% when the Omega test was applied. The Omega test took about 1 hour and 23 minutes of dependence analysis time in the Perfect benchmarks.

The Perfect benchmarks compiled in about 41 minutes using either the I-Test or the Range test and 46 minutes when using the VI-Test. The compilation time for the NLVI-Test was 57 minutes which is higher but still quite acceptable. However, the Perfect benchmarks required 2 hours and 3 minutes of compilation time when using the Omega test.

The efficiency result for the SPEC benchmarks are summarize in Figure 5-9. The I-Test and the Range test were still the most efficient tests, taking on average 2 msec per dependence problem. The VI-Test and the NLVI-Test trail in efficiency with 11 msec and 21 msec per dependence problem respectively. The efficiency of the Omega test in the SPEC benchmarks was poor. It took on average 133 msec per dependence problem. It is surprising how much inefficient the Omega test becomes when handling more complex problems. There is an average cost of 1 msec and 6 msec per dependence problem for the cases where the Omega test was exact and an average cost of 335 msec for the cases where it was inexact. This occured in 39% of the problems and it significantly affected the efficiency of the Omega test. In the SPEC benchmarks the I-Test and the Range test took 1.5 minutes, which is 11% of the total compilation time. The VI-Test took 8 minutes which is 39% of the

Test	Total	Independent	Dependent	Maybe
I-Test	0.76 (100%)	0.32 (13%)	0.43 (2%)	0.84 (84%)
Omega	33.03 (100%)	2.05 (22%)	1.39 (10%)	47.87 (68%)
Range	1.25 (100%)	0.60 (14%)	N/A	1.35 (86%)
VI-Test	4.86 (100%)	1.74 (22%)	2.18 (8%)	6.11 (70%)
NLVI-Test	7.78 (100%)	2.60 (24%)	3.53 (9%)	10.18 (67%)

Test	Comp TIme	DD Time	Array Time
I-Test	32.4 (100%)	9.9 (30%)	3.7 (11%)
Omega	189.9 (100%)	167.5 (88%)	161.4 (85%)
Range	34.8 (100%)	12.2 (35%)	6.1 (18%)
VI-Test	52.2 (100%)	29.8 (57%)	23.7 (45%)
NLVI-Test	66.4 (100%)	44.1 (66%)	38.0 (57%)

Figure 5-10: Time per dependence problem and compilation efficiency in the Lapack library

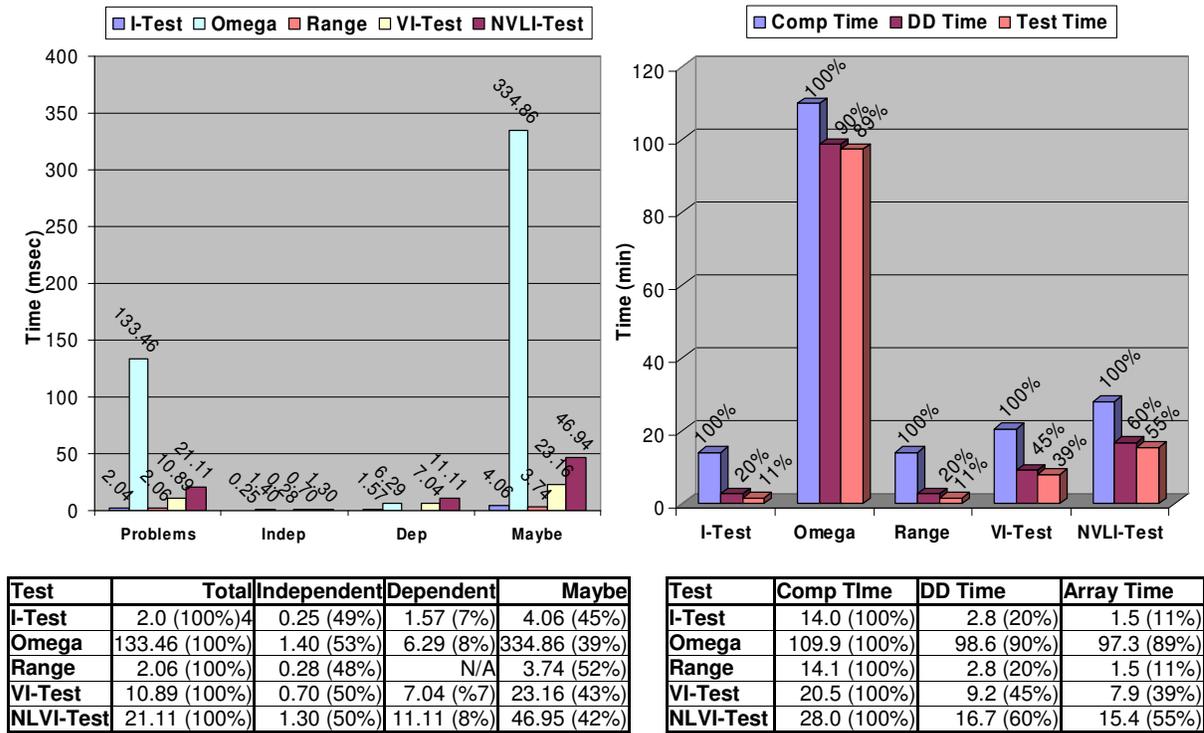total compilation time. The NLVI-Test spent 15 minutes analyzing all data dependence problems and direction vectors in the the SPEC benchmarks, which accounts for 55% of the total compilation time. The Omega test took one 1 hour and 37 minutes of data dependence testing time in the SPEC benchmarks, which is 89% of the total compilation time.

The SPEC benchmarks compiled in 14 minutes with either the I-Test or the Range test. The VI-Test and the NLVI-Test resulted into 24 min and 28 min of compilation time in the SPEC benchmarks respectively. The time required when using the Omega test was 1 hour and 50 minutes which translates into a 670% impact on compilation time compared to the I-Test or the Range test.

Similar conclusions are drawn from the results in the Lapack library. The average time per dependence problem here was less than in the Perfect and SPEC benchmarks. The most efficient test was the I-Test taking on average less than 1 msec to solve a data dependence problem. The Range test was also very efficient with a little over 1 msec on average per dependence problem. The VI-Test and the NLVI-Test were much faster in the Lapack library as well, taking on average 5 msec and 8 msec per dependence problem respectively. The Omega test was again much less efficient than the other tests. Considering at the timing analysis we can see that the Omega test was fast when it was exact,

taking on average about 2.0 msec and 1.4 msec to disprove or prove dependences. When the Omega test was inexact, the cost changed dramatically to about 48 msec per dependence problem on average.

In terms of compilation efficiency the I-Test required 3.7 minutes to analyze all data dependence problems and direction vectors in the Lapack library, which is 12% of the total compilation time. The Range test took 6.1 minutes or 18% of the total compilation time. The VI-Test took 24 minutes or 45% and the NLVI-Test spent 38 minutes analyzing all data dependence problems and direction vectors in the Lapack library, which accounts for 57% of the total compilation time. The Omega test took about 2 hours and 41 minutes which accounts for 85% of the total compilation time. The impact of the NLVI-Test on the total compilation time was only 104% more than with the I-Test while the impact of the Omega test was over 480%. The compilation of the Lapack library including the BLAS package required 32 minutes with the I-Test, 35 minutes with the Range test, 52 minutes with the VI-Test, and 66 minutes with the NLVI-Test. On the other hand it took 3 hours and 10 minutes of compilation time when the Omega test was used.

In summary, our results indicate a 6 times increase in cost for the VI-Test compared to the I-Test and about one order of magnitude increase in cost for the NLVI-Test. With respect to the Omega Test, the VI-Test and the NLVI-Test are about one order of magnitude more efficient. In general when the Omega is exact it is also efficient, but that is not the case when it returns a maybe answer. The VI-Test and the NLVI-Test also take more time when they are inexact, but they are far more efficient than the Omega Test. It is obvious that there is a significant overhead in the compilation efficiency when using the Omega test. Especially, since the cost of the data dependence analysis phase is a significant percentage of the total compilation time.

The NLVI-Test is substantially faster than the Omega test. It is not as efficient as the I-Test and the Range test, but the impact of the NLVI-Test on the total compilation is quite acceptable. The higher cost of the NLVI-Test over the I-Test is justified and attributed to expensive operations on polynomial expressions, which are required for the NLVI-Test to disprove or prove dependences for non-linear constraints as well as the techniques it utilizes to handle coupled subscripts and if-statement constraints. The extremely high cost of the Omega test is attributed to exponential time techniques that the algorithm employs, such as the Fourier-Motzkin variable elimination and its integer extensions, as well as a very expensive enumeration technique that is known as the Omega test nightmare. Since the Omega test cannot perform non-linear dependence analysis, its cost is probably not well justified. It should be noted that the Range test appears surprisingly fast, which is due to a number of reasons. The Range test is using the internal representation of Polaris, so it does not have to parse and reconstruct a dependence problem. Also it relies on the range propagation
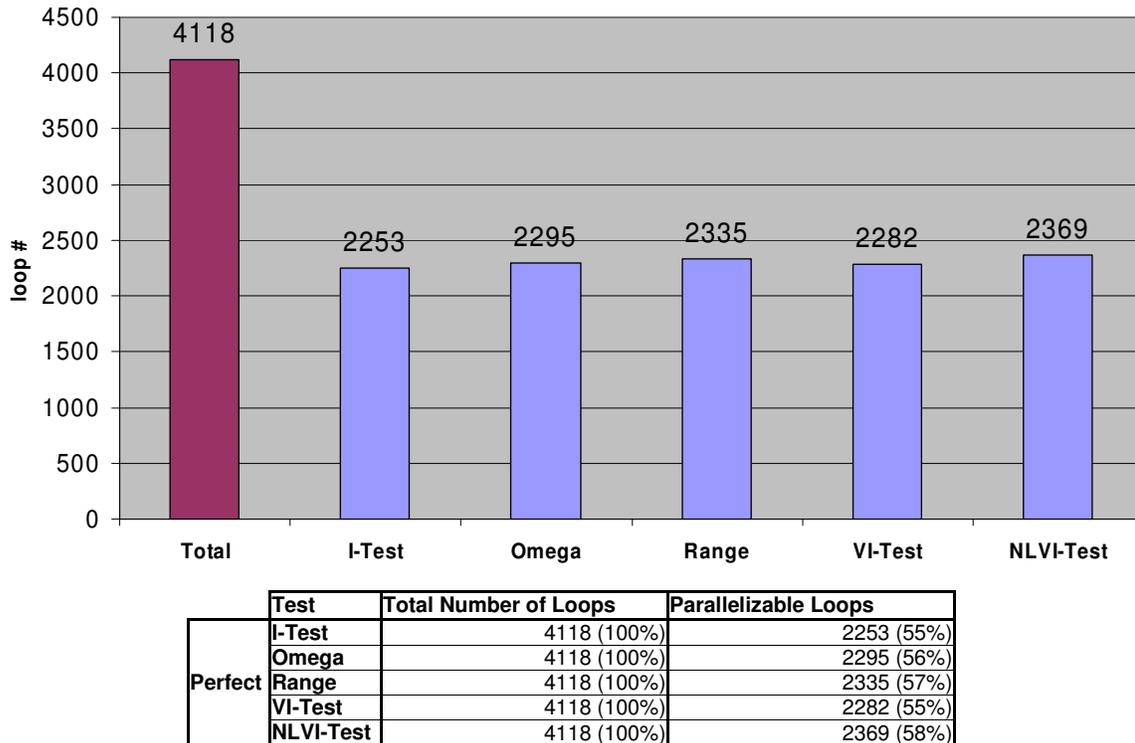
Figure 5-11: Parallelizable loops in the Perfect benchmarks

Test	Total Number of Loops	Parallelizable Loops
Perfect	I-Test	4118 (100%)	2253 (55%)
	Omega	4118 (100%)	2295 (56%)
	Range	4118 (100%)	2335 (57%)
	VI-Test	4118 (100%)	2282 (55%)
	NLVI-Test	4118 (100%)	2369 (58%)

technique that is performed in a previous compilation step by Polaris. Even though range propagation is used in a few other compilation phases, it is not a mainstream technique and it is not implemented in any other compiler. Also the Range test does not take into account if-statement constraints or coupled subscripts. More importantly the Range test does not compute complete direction vector information, but only the level of the loop that carries the dependence. Therefore, it tests only a fraction of all possible direction vectors as the results in Section 5.2 indicate. The efficiency of the NLVI-Test, when it follows the dependence level instead of the direction vector hierarchy is comparable to the efficiency of the Range test in all benchmarks. Finally, the Range test does spend time trying to conclusively prove data dependences like the other tests do.

5.4 Loop Parallelization Results

Even though data dependence accuracy is a fair comparison metric between data dependence tests, it is only sufficient from a theoretical point of view. The actual benefits of employing powerful data dependence analysis techniques are more evident from the amount of parallelism that they can uncover in a program. This can be expressed in terms of the number of loops that can be parallelized.

	Test	Total Number of Loops	Parallelizable Loops
	I-Test	1729 (100%)	1040 (60%)
	Omega	1729 (100%)	1069 (62%)
SPEC	Range	1729 (100%)	1094 (63%)
	VI-Test	1729 (100%)	1062 (61%)
	NLVI-Test	1729 (100%)	1163 (67%)
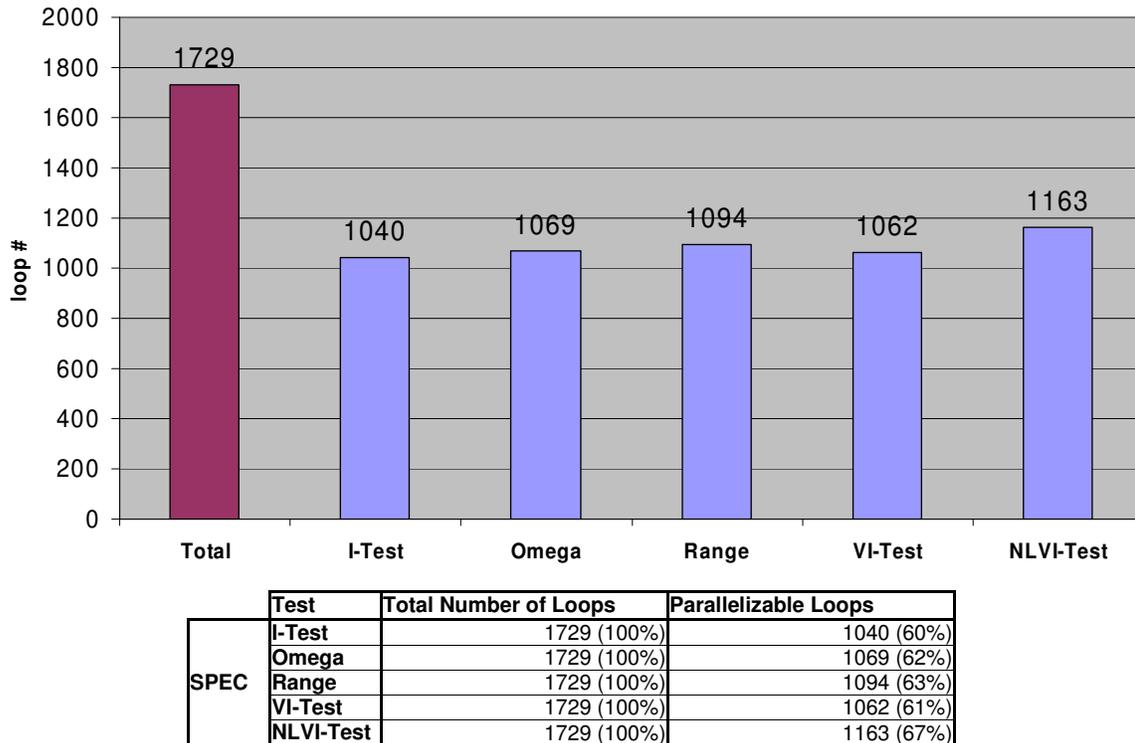
Figure 5-12: Parallelizable loops in the SPEC benchmarks

In our experiments we measured the total number of loops that were found parallelizable by each data dependence test.

Figure 5-11 summarizes the number of loops parallelizable for the Perfect benchmarks. The charts and tables display the number of loops parallelized by each test. According to the results the NLVI-Test provides the highest degree of parallelization. In the Perfect benchmarks the NLVI-Test determined that 2369 out of 4118 loops or 58% can be safely parallelized. The Range test found 2335 parallelizable loops, which is 57% of the total. The Omega test found 2295 parallelizable loops or 56% of the total. Finally the VI-Test and the I-Test found 2282 and 2253 parallelizable loops respectively or 55%. Despite the fact that the Omega test disproved 5% more dependence problems (3% more direction vectors) it was able to parallelize only 1% more loops more than the I-Test.

In the SPEC benchmarks, Figure 5-12, the difference between the NLVI-Test and the other test is much higher. The NLVI-Test determined 1163 parallelizable loops, which accounts for 67% of the total 1729 loops. The Range test found 1094 parallelizable loops or 63% of the total. The Omega test, the VI-Test and the I-Test follow with 1069 or 62%, 1062 or 61% and 1040 or 60% parallelizable

	Test	Number of Loops	Parallelizable Loops
	I-Test	7768 (100%)	4195 (54%)
	Omega	7768 (100%)	4367 (56%)
Lapack	**Range**	7768 (100%)	4377 (56%)
	LVI-Test	7768 (100%)	4363 (56%)
	RVI-Test	7768 (100%)	4425 (57%)
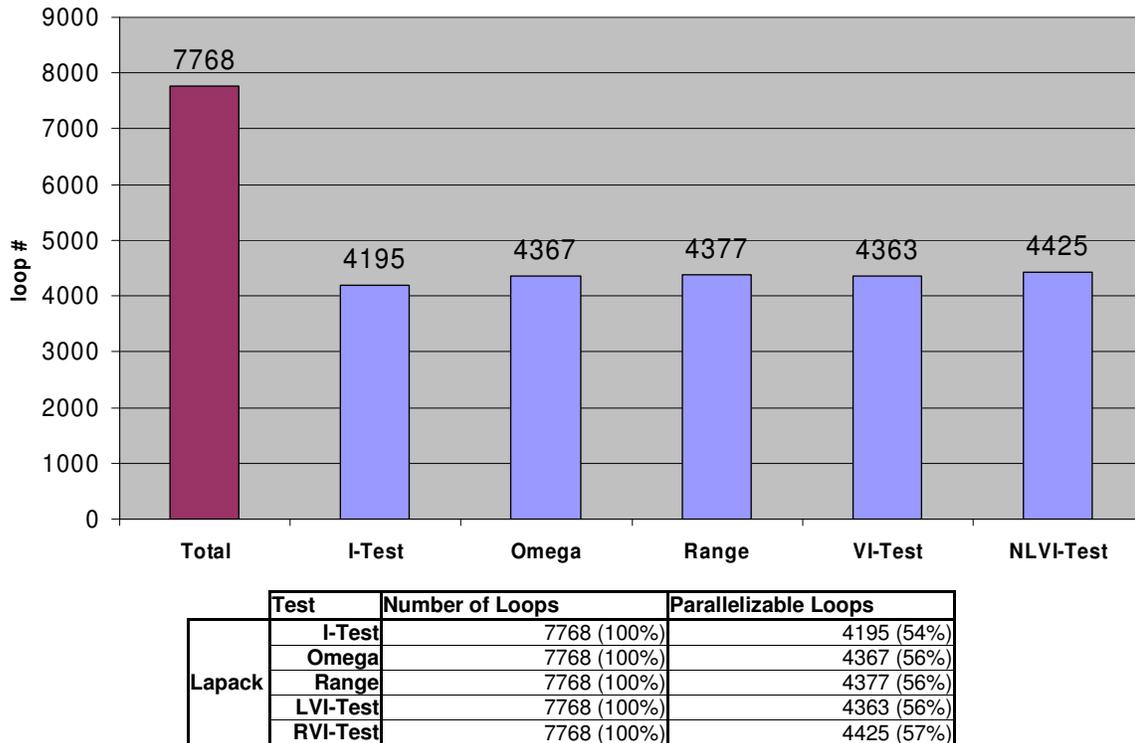
Figure 5-13: Parallelizable loops in the Lapack Library

loops respectively. Non-linear dependence analysis gives a significant advantage to the NLVI-Test in the SPEC benchmarks over the linear expression dependence tests.

In the Lapack library, Figure 5-13, the NLVI-Test was still the most effective dependence test in terms of loop parallelization. The NLVI-Test determined that 4425 out of 7768 loops or 57% can be safely parallelized. The Range test found 4377 parallelizable loops, which is 56% of the total. The Omega test and the VI-Test found 4367 and 4363 parallelizable loops respectively, which is also 56% of the total. Finally the I-Test found 4195 parallelizable loops or 54%. In the Lapack library the increased accuracy of the VI-Test and especially of the NLVI-Test paid off in getting the best possible result in terms of loop parallelization.

In addition to dependence testing, the degree of parallelization also relies on the code transformations that are applied before the data dependence analysis phase. Transformations such as constant and expression propagation, code inlining, and induction variable recognition may reveal non-linear patterns that can be used in data dependence analysis to obtain more accurate results and increase parallelization. Even though Polaris performs such transformations, more powerful program

91

analysis techniques could better assist a non-linear dependence test, like the NLVI-Test, in extracting even more parallelism from source code.

5.5 Execution Performance Results

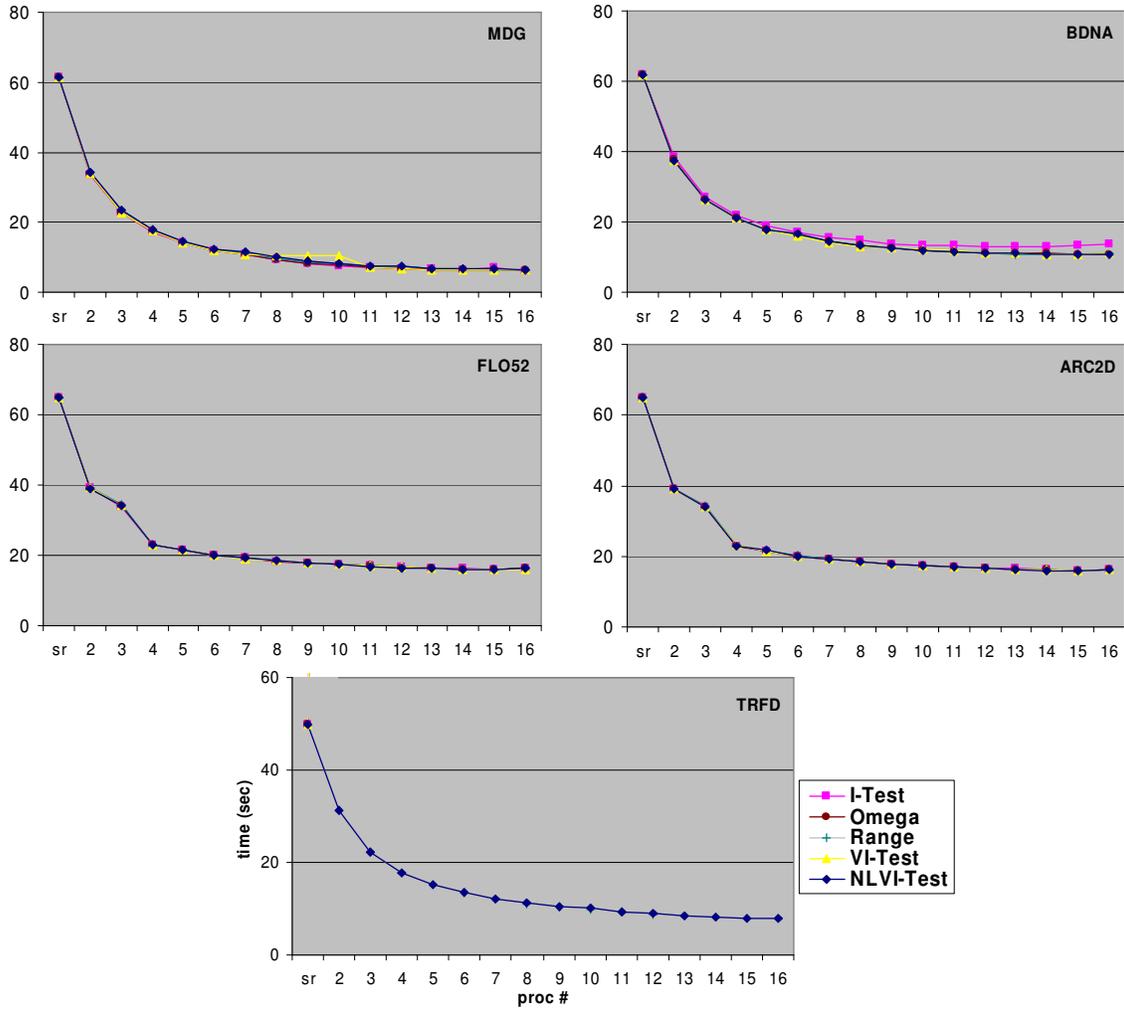The number of parallelizable loops is a good metric of program parallelization. However, it does not necessarily reflect the actual execution performance of a program. It is often the case that only few of the loops found parallelizable actually account for the overall speedup. In some cases loop parallelization may even result in a slow-down due to excessive communication and synchronization cost. The final execution performance depends on a number of factors, in addition to data dependence analysis, including all the other compilation phases, the parallel library used, the operating system, and the actual machine.

In order to further assess the impact of each data dependence test on program execution performance and speedup we compiled and run the Perfect and SPEC benchmarks on a shared memory multiprocessor. For each data dependence test the Polaris compiler converted the Fortran 77 source code into Fortran 90 with parallel OpenMP directives. Subsequently, we used the native f90 compiler to generate machine code for parallel execution on an 8 dual-processor Sun Fire server with 16 processing cores of 1.2GHz each. The execution times and speedups presented in Figure 5-14 and Figure 5-15 are for those benchmarks only for which loop parallelization produced speedup over the serial execution time. Also only the results for the dependence tests that managed to improve the execution time are displayed for each benchmark. Execution time results for the benchmarks for which the dependence tests were not able to effectively parallelize are not included.

The results for the Perfect benchmarks are summarized in Figure 5-14 Only 5 out of the 13 programs were effectively parallelized by either test. The MDG benchmark was effectively parallelized by all tests, and produced a speedup of about 9.7 to 9.9 on 16 processing elements. MDG can be parallelized with proper reduction recognition and variable privatization [12]. The BDNA benchmark was also effectively parallelized by all tests. The VI-Test, the NLVI-Test, the Omega test and the Range test produced an additional speedup over the I-Test with a factor of 5.7 versus 4.5 on 16 processing elements. The additional speedup resulted from parallelizing the loop in Figure 4-3.a, which has complex loop bounds that the I-Test could not analyze. In ARC2D and FLO52 benchmarks all tests produced the similar speedups of 5.5 and 4.0 respectively. Finally, in the TRFD benchmark only the NLVI-Test and the Range test with non-linear dependence analysis effectively parallelized key loops, Figure 4-3.b, and produced a speedup of 6.4 on 16 processing elements. Neither the I-Test, nor the VI-Test nor the Omega test produced any speedup in this benchmark.

Figure charts: MDG, BDNA, FLO52, ARC2D, TRFD (time (sec) vs proc #) with legend: I-Test, Omega, Range, VI-Test, NLVI-Test

Program	Test	serial	2-proc	sp	4-proc	sp	6-proc	sp	8-proc	sp	10-proc	sp	12-proc	sp	14-proc	sp	16-proc	sp
MDG	I-Test	61.37	33.59	1.8	17.30	3.5	11.84	5.2	9.19	6.7	7.62	8.1	6.88	8.9	6.41	9.6	6.22	9.9
	Omega	61.37	33.75	1.8	17.38	3.5	11.92	5.1	9.20	6.7	7.63	8.0	6.80	9.0	6.48	9.5	6.33	9.7
	Range	61.37	34.24	1.8	17.87	3.4	12.33	5.0	9.66	6.4	8.08	7.6	7.30	8.4	6.64	9.2	6.42	9.6
	VI-Test	61.37	33.73	1.8	17.38	3.5	11.89	5.2	10.38	5.9	13.18	4.7	10.24	6.0	6.76	9.1	6.33	9.7
	NLVI-Test	61.37	34.28	1.8	17.84	3.4	12.34	5.0	12.14	5.1	8.10	7.6	7.32	8.4	6.75	9.1	6.32	9.7
BDNA	I-Test	61.74	38.62	1.6	22.01	2.8	17.10	3.6	14.69	4.2	13.43	4.6	13.06	4.7	13.08	4.7	13.68	4.5
	Omega	61.74	37.76	1.6	21.09	2.9	16.22	3.8	13.27	4.7	11.92	5.2	11.23	5.5	10.93	5.6	10.88	5.7
	Range	61.74	37.57	1.6	21.14	2.9	16.34	3.8	13.12	4.7	11.96	5.2	11.21	5.5	10.60	5.8	10.98	5.6
	VI-Test	61.74	37.54	1.6	21.04	2.9	15.96	3.9	13.00	4.7	12.62	4.9	12.06	5.1	11.11	5.6	10.91	5.7
	NLVI-Test	61.74	37.53	1.6	20.94	2.9	16.51	3.7	13.21	4.7	11.98	5.2	11.17	5.5	10.90	5.7	10.81	5.7
ARC2D	I-Test	51.36	29.41	1.7	16.56	3.1	13.15	3.9	11.49	4.5	10.51	4.9	10.00	5.1	9.41	5.5	9.34	5.5
	Omega	51.36	29.88	1.7	16.76	3.1	13.28	3.9	11.45	4.5	10.42	4.9	10.02	5.1	9.18	5.6	9.25	5.6
	Range	51.36	29.49	1.7	16.54	3.1	13.15	3.9	11.51	4.5	10.44	4.9	10.01	5.1	10.06	5.1	9.35	5.5
	VI-Test	51.36	29.56	1.7	16.55	3.1	13.13	3.9	11.53	4.5	11.03	4.7	10.51	4.9	10.00	5.1	9.33	5.5
	NLVI-Test	51.36	29.49	1.7	16.68	3.1	13.23	3.9	11.59	4.4	10.50	4.9	10.00	5.1	9.31	5.5	9.36	5.5
FLO52	I-Test	64.92	39.13	1.7	23.00	2.8	20.08	3.2	18.30	3.5	17.40	3.7	16.66	3.9	16.20	4.0	16.23	4.0
	Omega	64.92	39.03	1.7	22.89	2.8	19.96	3.3	18.27	3.6	17.30	3.8	16.45	3.9	16.07	4.0	16.15	4.0
	Range	64.92	39.20	1.7	23.07	2.8	20.14	3.2	18.38	3.5	17.41	3.7	16.53	3.9	16.10	4.0	16.19	4.0
	VI-Test	64.92	39.12	1.7	23.06	2.8	20.06	3.2	18.47	3.5	17.65	3.7	17.36	3.7	16.53	3.9	16.09	4.0
	NLVI-Test	64.92	39.06	1.7	23.00	2.8	20.02	3.2	18.40	3.5	17.33	3.7	16.47	3.9	15.97	4.1	16.19	4.0
TRFD	Range	49.82	31.24	1.6	17.79	2.8	13.39	3.7	11.12	4.5	9.77	5.1	8.79	5.7	8.19	6.1	7.75	6.4
	NLVI-Test	49.82	31.24	1.6	17.83	2.8	13.49	3.7	11.20	4.4	10.15	4.9	8.89	5.6	8.26	6.0	7.79	6.4

Figure 5-14: Execution time and speedup of five of the Perfect benchmarks

The results for the SPEC benchmarks are summarized in Figure 5-15 Here 5 out of the 10 total benchmarks produced speedup. In HYDRO2D all tests produced speedup. The NLVI-Test and the Range test however managed to yield better speedup on 16 processing elements with 6.8 compared to 5.8 for the I-Test, the VI-Test and the Omega test. In MGRID all tests produced speedup, but the NLVI-Test alone in this case increased the speedup from 3.4 and 3.6 to 4.7 on 16 processing elements. This additional speedup is due to the parallelization of the loops in Figure 4-3.c. In SU2COR all dependence tests effectively parallelize the benchmark and produced a speedup of about 3.8. The next benchmark SWIM is a fairly simple program and easily parallelizable. Therefore, all test produced an impressive super-linear speedup of 35 times faster than the serial execution. In the last and final benchmark TOMCATV all benchmarks managed to parallelize the code. The I-Test, the VI-Test, the NLVI-Test and the Omega test with the loop interchange transformation performed on the loops in Figure 4-3.d produced additional speedup over the Range test. Even though initially the Range test outperformed the other tests on fewer processors, when more processors were utilized the other tests obtained a speedup of 2.7 as opposed to 1.9 on 16 processing elements. In this example we can see that more accurate direction vector information can result into improved execution performance.

In summary the NLVI-Test achieved the highest program execution performance among all other data dependence tests in all benchmarks. In particular the NLVI-Test produced significantly better speedup than both the Omega test and the I-Test in 3 out of the 10 parallelizable Perfect and SPEC benchmarks. It also outperformed the Range test in 2 of the benchmarks. Taking into account that the Range test was designed to parallelize all the key loops in the Perfect benchmarks [12], we can conclude that the NLVI-Test surpassed the effectiveness of the Range test and produced the highest degree of parallelization and the best execution times in both the Perfect and SPEC benchmarks. In addition to data dependence analysis, the program execution performance depends on a number of factors including the compiler, the parallel library used, the operating system, and the actual machine. Even though the Polaris compiler implements many novel techniques it still remains a research prototype tool. In many benchmarks an even higher speedup could be obtained if a more sophisticated compiler was used to take advantage of the increased number of parallelizable loops that the NLVI Test detected.
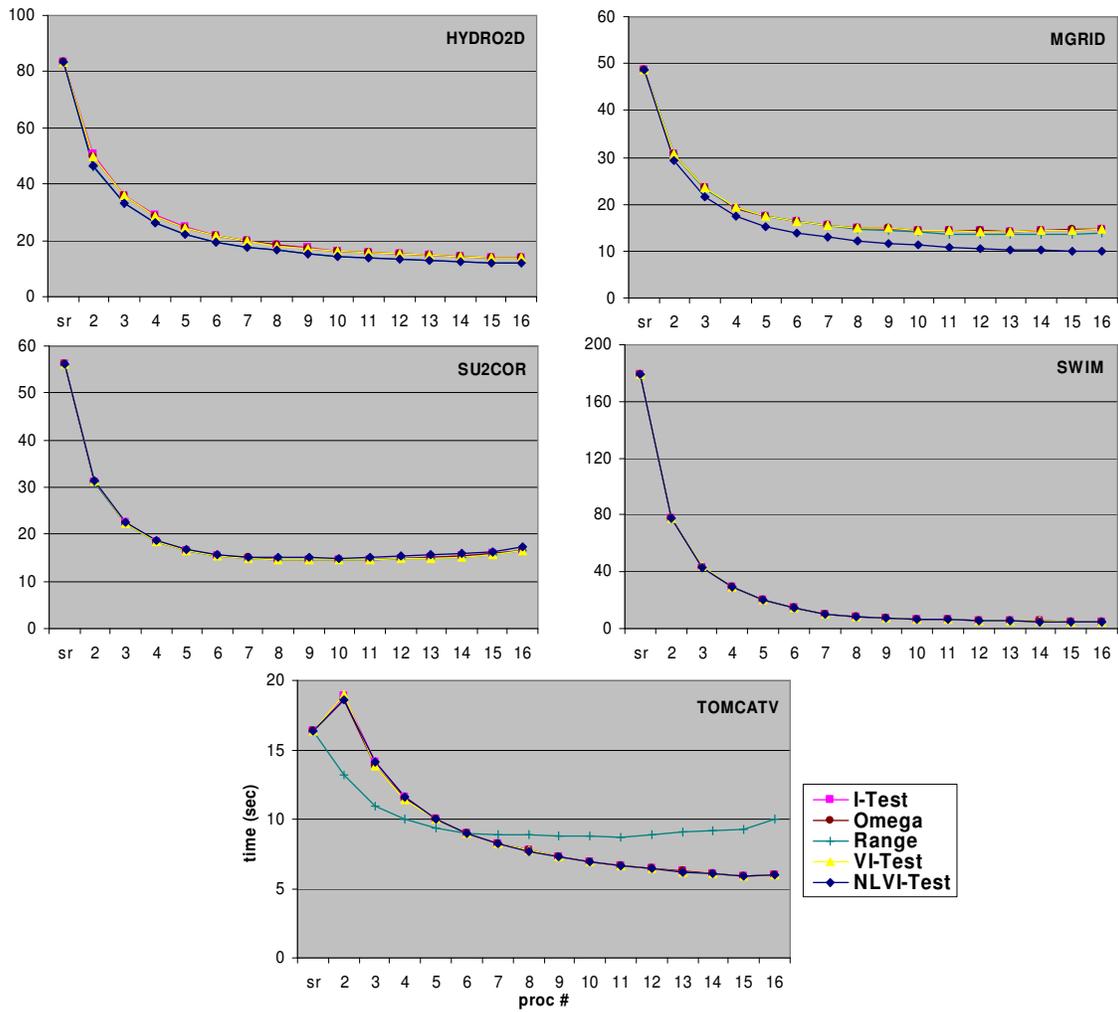
Figure 5-15: Execution time and speedup of five of the SPEC Benchmarks

Program	Test	serial	2-proc	sp	4-proc	sp	6-proc	sp	8-proc	sp	10-proc	sp	12-proc	sp	14-proc	sp	16-proc	sp
HYDRO2D	I-Test	83.20	50.71	1.6	28.95	2.9	21.80	3.8	18.29	4.5	16.25	5.1	15.23	5.5	14.30	5.8	14.03	5.9
	Omega	83.20	49.82	1.7	28.74	2.9	21.74	3.8	18.37	4.5	16.32	5.1	15.16	5.5	14.27	5.8	13.99	5.9
	Range	83.20	46.27	1.8	26.19	3.2	19.57	4.3	16.38	5.1	14.34	5.8	13.23	6.3	12.27	6.8	11.86	7.0
	VI-Test	83.20	49.77	1.7	28.71	2.9	21.64	3.8	18.19	4.6	16.14	5.2	15.04	5.5	14.07	5.9	13.78	6.0
	NLVI-Test	83.20	46.39	1.8	26.30	3.2	19.51	4.3	16.43	5.1	14.41	5.8	13.29	6.3	12.36	6.7	11.97	7.0
MGRID	I-Test	48.66	30.66	1.6	19.16	2.5	16.26	3.0	14.91	3.3	14.39	3.4	14.30	3.4	14.45	3.4	14.77	3.3
	Omega	48.66	30.62	1.6	19.18	2.5	16.22	3.0	14.91	3.3	14.48	3.4	14.27	3.4	14.46	3.4	14.74	3.3
	Range	48.66	30.77	1.6	19.27	2.5	16.18	3.0	14.65	3.3	13.97	3.5	13.67	3.6	13.62	3.6	13.78	3.5
	VI-Test	48.66	30.78	1.6	19.31	2.5	16.27	3.0	14.88	3.3	14.35	3.4	14.19	3.4	14.31	3.4	14.53	3.3
	NLVI-Test	48.66	29.27	1.7	17.37	2.8	13.85	3.5	12.11	4.0	11.20	4.3	10.63	4.6	10.29	4.7	9.86	4.9
SU2COR	I-Test	56.13	31.16	1.8	18.32	3.1	15.50	3.6	14.54	3.9	14.60	3.8	14.82	3.8	15.35	3.7	16.57	3.4
	Omega	56.13	31.23	1.8	18.44	3.0	15.54	3.6	14.65	3.8	14.66	3.8	14.92	3.8	15.45	3.6	16.63	3.4
	Range	56.13	31.23	1.8	18.37	3.1	15.39	3.6	14.46	3.9	14.48	3.9	14.75	3.8	15.25	3.7	16.40	3.4
	VI-Test	56.13	31.24	1.8	18.39	3.1	15.52	3.6	14.56	3.9	14.51	3.9	14.77	3.8	15.27	3.7	16.45	3.4
	NLVI-Test	56.13	31.32	1.8	18.59	3.0	15.79	3.6	15.03	3.7	14.98	3.7	15.40	3.6	15.91	3.5	17.24	3.3
SWIM	I-Test	178.76	77.44	2.3	29.36	6.1	14.35	12.5	8.45	21.2	6.76	26.4	5.79	30.9	5.04	35.5	4.68	38.2
	Omega	178.76	77.46	2.3	29.22	6.1	14.32	12.5	8.48	21.1	6.77	26.4	5.80	30.8	5.07	35.3	4.69	38.1
	Range	178.76	77.56	2.3	29.30	6.1	14.28	12.5	8.46	21.1	6.78	26.4	5.80	30.8	5.02	35.6	4.69	38.1
	VI-Test	178.76	77.74	2.3	29.37	6.1	14.29	12.5	8.41	21.3	6.74	26.5	5.79	30.9	5.06	35.3	4.65	38.4
	NLVI-Test	178.76	77.66	2.3	29.34	6.1	14.33	12.5	8.43	21.2	6.76	26.4	5.78	30.9	5.02	35.6	4.65	38.4
TOMCATV	I-Test	16.32	18.88	0.9	11.61	1.4	8.95	1.8	7.72	2.1	6.93	2.4	6.42	2.5	6.08	2.7	5.98	2.7
	Omega	16.32	18.60	0.9	11.45	1.4	9.00	1.8	7.72	2.1	6.93	2.4	6.44	2.5	6.08	2.7	5.99	2.7
	Range	16.32	13.22	1.2	10.00	1.6	8.97	1.8	8.86	1.8	8.79	1.9	8.91	1.8	9.15	1.8	10.01	1.6
	VI-Test	16.32	18.99	0.9	11.44	1.4	8.99	1.8	7.74	2.1	6.93	2.4	6.43	2.5	6.09	2.7	6.00	2.7
	NLVI-Test	16.32	18.59	0.9	11.59	1.4	8.98	1.8	7.69	2.1	6.93	2.4	6.42	2.5	6.08	2.7	6.01	2.7

5.6 Conclusions

Several studies have shown that scientific source code is far more complex than the cases traditional data dependence analysis techniques can handle. Advances in data dependence analysis have improved dependence accuracy, but without much success in increasing program parallelization. The goal of this work is to improve data dependence analysis in practice. Designers of optimizing and parallelizing compilers should consider all issues related to data dependence accuracy, efficiency, and effectiveness in program parallelization and execution performance of the generated code. Our research was primarily focused on these issues.

We examined the limitations of existing data dependence analysis techniques in actual source code. We verified that loops with triangular or trapezoidal bounds and loops with symbolic variables frequently occur in practice. We also verified that if-statement conditions can significantly limit the accuracy of dependence testing, if not taken into account. Coupling of variables between multidimensional array subscripts, can also present a major reason of inaccuracy especially in linear algebra libraries like the Lapack library. Traditional data dependence analysis techniques such as the Banerjee test and the I-Test, tend to ignore or simplify many of the problem constraints and therefore significantly lack in accuracy. On the other hand, tests that can handle general type of constraints like the Omega test, tend to be expensive and at times can exhibit unpredictable behavior as far as efficiency is concerned. More importantly, we concluded that analyzing non-linear expressions and patterns, which very frequently appear in source code, can significantly improve parallelization. Most data dependence tests ignore such non-linear constraints and therefore additional amount of parallelism remains unexploited.

In this work we propose new data dependence analysis techniques that overcome the limitations of the existing dependence tests in practice. As a first step in our research, we developed a new polynomial-time data dependence algorithm, the VI-Test, which extends the ideas behind the I-Test to complex loop regions and can accurately handle loops with trapezoidal bounds and symbolic variables. The algorithm is based on the variable interval theory and can be applied to both simple and direction vector dependence problems. In addition, we implemented the equation propagation technique to address the issue of multidimensional arrays with coupled subscripts and we incorporated the lambda equations into the algorithm. We also implemented efficient techniques to include simple cases of if-statement constrains into the dependence problem. As a second step we extended the theory of variable intervals to non-linear expressions. Our dependence testing method, which we term NLVI-Test, is based on the variable interval theory for multivariate functions and a set

of efficient techniques that can resolve complex instances of the dependence problem. In particular our algorithm can conclusively prove or disprove data dependences, in the presence of non-linear expressions and in problems with complex loop bounds, multidimensional array subscripts with coupled variables and enclosing if-statements. It can produce accurate and complete direction vector information and it has polynomial time complexity. The problem of data dependence analysis translates into an integer constraint satisfaction problem which is NP-complete. In this work have developed theory and techniques that can accurately solve many instances of the integer constraint satisfaction problem in polynomial time even for the non-linear case.

We have implemented the proposed techniques, including the VI-Test and the NLVI-Test, in a comprehensive data dependence analysis tool and we have made it part of the PLATO library. The PLATO library implements linear, polynomial and rational polynomial arithmetic and it includes implementation of several other data dependence tests. It is available through the web (http://www.cs.utsa.edu/~plato) with instructions of how it can be integrated in a compiler.

In order to measure the impact of data dependence analysis on the actual compilation process we evaluated the dependence analysis phase using several metrics. We performed an extensive empirical evaluation comparing our dependence analysis tool against the I-Test, the Omega test and the Range test. For our experiments we used the Perfect benchmarks, the SPEC benchmarks and the Lapack library. We measured the accuracy and the reasons that can result into inaccurate answers for each test. We determined the efficiency of each data dependence test by computing the average time to solve a data dependence problem, the total compilation time, and the time taken to resolve all data dependences in each benchmark. Finally, we measured the impact of data dependence analysis on loop parallelization and on the actual execution time and speedup of each benchmark on a multiprocessor.

In terms of accuracy we verified that the Omega test, which has exponential time complexity, is more accurate than the polynomial-time I-Test and Range Test. In scientific applications like the Perfect and SPEC benchmarks this difference is not substantial especially when it comes to direction vector problems. On the other hand in scientific libraries like the Lapack library, the Omega test takes advantage of its ability to handle more complex instances of the dependence problem and performs better than the I-Test. The reason for that difference is due to loops with unknown bounds, coupled subscripts and if-statement conditions. We proved that our techniques can significantly increase the data dependence accuracy, approaching or even exceeding the accuracy of the Omega test. We also saw that the accuracy of the Range test, in terms of direction vector problems, is compromised by the fact that it does not compute all applicable direction vectors but only the level of the dependence.

In terms of efficiency we verified that the cost of the Omega test is high compared to the other tests and that it significantly impacts the overall compilation time. By analyzing the tradeoff between accuracy and efficiency, we discovered that the Omega test is efficient when it is accurate, but on the other hand, it becomes very inefficient when dealing with complex dependence problems. We saw that data dependence testing can significantly impact the compiler's efficiency, since it can take up most of the total compilation time. This was always the case with the Omega test. We verified that the dependence analysis phase constitutes a major part of the compilation process and therefore it is essential to employ efficient data dependence analysis techniques in real compilers. The Omega test in its current implementation seems to be impractical for production compilers. The VI-Test and the NLVI-Test, on the other hand, even though they are slower than the I-Test, they are both much faster than the Omega test and more practical in terms of compilation efficiency.

In terms of effectiveness in loop parallelization we proved that our techniques offer the highest degree of parallelization among all data dependence tests. In our experiments we detected more parallel loops then any other data dependence test at a very reasonable computation cost. The Omega test, even though it was more accurate than the other tests, it did not significantly increase the degree of parallelization. The VI-Test virtually eliminates the difference in loop parallelization between the I-Test and the Omega test and the NLVI-Test further improves loop parallelization beyond any other test. The Omega test can be applied with the hope of breaking more dependences but without actually affecting program parallelization. Finally, with regard to program execution performance, which is the best metric in evaluating any compiler technique, we saw that our dependence analysis tool outperformed all the other dependence tests and produced the best execution time and speedup on a multiprocessor for each benchmark. We verified that the increased parallelization can result into improved execution and speedup in real benchmarks.

The information regarding the relative accuracy, efficiency, and effectiveness of each dependence test demonstrates the impact of data dependence analysis on the compilation process and the execution performance of the generated code. These results can help researchers and practitioners implement the most appropriate dependence analysis techniques in order to develop more efficient and effective compilers. It is our conclusion that in order to achieve the highest degree of optimization and improve the parallel execution time of programs, a data dependence test must have the ability to accurately analyze complex instances of the dependence problem, including those with non-linear and symbolic expressions. The data dependence algorithm must also be efficient and portable into any compiler implementation. Our proposed method fulfills these requirements and constitutes an effective tool for advanced data dependence analysis.

Appendix

Proof of Theorems

Proof of Theorem 3-2

According to Definition 3-2 we will prove that the union of all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P]$.

Case I: $\kappa\lambda \leq 0$.

Let us assume that $\kappa \leq 0$ and $\lambda \geq 0$, then $L + \kappa X$ is a decreasing function and $U + \lambda X$ is an increasing function. If there exist X_0 such that $L + \kappa X_0 \leq U + \lambda X_0$, then for all X_1, X_2 such that $Q \geq X_2 \geq X_1 \geq X_0$,

$$[L + \kappa X_1, U + \lambda X_1] \subseteq [L + \kappa X_2, U + \lambda X_2] \subseteq \ldots \subseteq [L + \kappa Q, U + \lambda Q]$$

This is true since $L + \kappa X_2 \leq L + \kappa X_1$ and $U + \lambda X_1 \leq U + \lambda X_2$, because of the monotonicity of the bound functions. In this case each interval contains all the previous intervals. Thus, the final integer interval contains all of them and therefore the union of all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is the integer interval $[L + \kappa Q, U + \lambda Q]$, which is equal to $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P]$ for $\kappa \leq 0$ and $\lambda \geq 0$. If no such X_0 exists, then each integer interval $[L + \kappa X_i, U + \lambda X_i] = \varnothing$, where $P \leq X_i \leq Q$, and also $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P] = \varnothing$

In both cases the union of all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P]$.

For the case of $\kappa \geq 0$ and $\lambda \leq 0$ we can prove similarly that the union of all integer intervals is the integer interval $[L + \kappa P, U + \lambda P]$, which is also equal to $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P]$ in this case.

Case II: $\kappa\lambda > 0$ and $U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^- Q + 1 \geq \min(|\kappa|, |\lambda|)$

First we will prove that the condition of the hypothesis guarantees that all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, are non-empty. From the hypothesis we can derive the following:

$$U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^- Q + 1 \geq 1.$$

The expression on the left hand side of the above inequality is the extreme lower value of the function $U - L + (\lambda - \kappa)X + 1$, where $P \leq X \leq Q$. Therefore:

$$U - L + (\lambda - \kappa)X + 1 \geq 1, \text{ for all } X \text{ such that } P \leq X \leq Q.$$

The expression on the left hand side is the length function $l(X)$ of the variable integer interval $[L + \kappa X, U + \lambda X]$. Since $l(X) \geq 1$, each integer interval $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is non-empty. Let us now consider the following two distinct cases and two sub-cases for each case.

Case II.1: $\kappa > 0$ and $\lambda > 0$

In this case both bounds functions $L + \kappa X$ and $U + \lambda X$ are increasing, so the integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, have the following form in increasing order of their bounds:

$$[L + \kappa P, U + \lambda P], \ldots, [L + \kappa X_i, U + \lambda X_i], [L + \kappa(X_i + 1), U + \lambda(X_i + 1)], \ldots, [L + \kappa Q, U + \lambda Q]$$

Case II.1.a: $\kappa > 0$ and $\lambda > 0$ and $\kappa \geq \lambda$

In this case the condition in the hypothesis is equivalent to $U - L + (\lambda - \kappa)Q + 1 \geq \lambda$. For every two consecutive integer intervals $[L + \kappa X_i, U + \lambda X_i]$ and $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$, where $P \leq X_i \leq Q - 1$, since $X_i + 1 \leq Q$, we can derive from the hypothesis:

$$U - L + (\lambda - \kappa)(X_i + 1) + 1 \geq \lambda$$
$$\Leftrightarrow \quad L + \kappa(X_i + 1) \leq U + \lambda X_i + 1$$

The above condition guaranties that for the above pair of consecutive integer intervals the lower bound of the second is less or equal than the upper bound of the first plus one. Therefore, their union constitutes the integer interval $[L + \kappa X_i, U + \lambda(X_i + 1)]$.

Case II.1.b: $\kappa > 0$ and $\lambda > 0$ and $\kappa < \lambda$

In this case the condition in the hypothesis is equivalent to $U - L + (\lambda - \kappa)P + 1 \geq \kappa$. For every two consecutive integer intervals $[L + \kappa X_i, U + \lambda X_i]$ and $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$, where $P \leq X_i \leq Q - 1$, since $P \leq X_i$, we can derive from the hypothesis:

$$U - L + (\lambda - \kappa)X_i + 1 \geq \kappa$$
$$\Leftrightarrow \quad L + \kappa(X_i + 1) \leq U + \lambda X_i + 1$$

The above condition, like in the previous sub-case, guaranties that the union of the above two consecutive integer intervals constitutes the integer interval $[L + \kappa X_i, U + \lambda(X_i + 1)]$.

In conclusion for the case of $\kappa > 0$ and $\lambda > 0$ the union of each two consecutive integer intervals $[L + \kappa X_i, U + \lambda X_i]$ and $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$, where $P \leq X_i \leq Q - 1$, is the integer interval $[L + \kappa X_i, U + \lambda(X_i + 1)]$. By induction we can prove that the union of all integer intervals

100

$[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is the integer interval $[L + \kappa P, U + \lambda Q]$, which is equal to the integer interval $[L + \overset{+}{\kappa}P - \overset{-}{\kappa}Q, U + \overset{+}{\lambda}Q - \overset{-}{\lambda}P]$ in this case.

Case II.2: $\kappa < 0$ and $\lambda < 0$

In this case both bounds functions $L + \kappa X$ and $U + \lambda X$ are decreasing, so the integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, have the following form in increasing order of their bounds:

$$[L + \kappa Q, U + \lambda Q], \ldots, [L + \kappa(X_i + 1), U + \lambda(X_i + 1)], [L + \kappa X_i, U + \lambda X_i], \ldots, [L + \kappa P, U + \lambda P]$$

Case II.2.a: $\kappa < 0$ and $\lambda < 0$ and $\kappa \geq \lambda$

In this case the condition in the hypothesis is equivalent to $U - L + (\lambda - \kappa)Q + 1 \geq -\kappa$. For every two consecutive integer intervals $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$ and $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q - 1$, since $X_i + 1 \leq Q$, we can derive from the hypothesis:

$$U - L + (\lambda - \kappa)(X_i + 1) + 1 \geq -\kappa$$
$$\Leftrightarrow \quad L + \kappa X_i \leq U + \lambda(X_i + 1) + 1$$

The above condition, guaranties that the union of the above two consecutive integer intervals constitutes the integer interval $[L + \kappa(X_i + 1), U + \lambda X_i]$.

Case II.2.b: $\kappa < 0$ and $\lambda < 0$ and $\kappa < \lambda$

In this case the condition in the hypothesis is equivalent to $U - L + (\lambda - \kappa)P + 1 \geq -\lambda$. For every two consecutive integer intervals $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$ and $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q - 1$, since $P \leq X_i$, we can derive from the hypothesis:

$$U - L + (\lambda - \kappa)X_i + 1 \geq -\lambda$$
$$\Leftrightarrow \quad L + \kappa X_i \leq U + \lambda(X_i + 1) + 1$$

The above condition, similarly, guaranties that the union of the above two consecutive integer intervals constitutes the integer interval $[L + \kappa(X_i + 1), U + \lambda X_i]$.

In conclusion for the case of $\kappa < 0$ and $\lambda < 0$ the union of each two consecutive integer intervals $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$ and $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q - 1$, is the integer interval $[L + \kappa(X_i + 1), U + \lambda X_i]$. By induction we can prove that the union of all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is the integer interval $[L + \kappa Q, U + \lambda P]$, which is equal to the integer interval $[L + \overset{+}{\kappa}P - \overset{-}{\kappa}Q, U + \overset{+}{\lambda}Q - \overset{-}{\lambda}P]$ in this case.

(*only-if*)

Assume that the variable integer interval $[L + \kappa X, U + \lambda X]$, where $P \leq X \leq Q$, is equal to the integer interval $[L + \overset{+}{\kappa}P - \overset{-}{\kappa}Q, U + \overset{+}{\lambda}Q - \overset{-}{\lambda}P]$. If $\kappa\lambda > 0$, we will prove that $U - L + (\lambda - \kappa)\overset{+}{P} - (\lambda - \kappa)\overset{-}{Q} + 1$

$\geq \min(|\kappa|, |\lambda|)$. According to Definition 3-2 and the hypothesis the union of all integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L + \kappa^+ P - \kappa^- Q, U + \lambda^+ Q - \lambda^- P]$

Case 1: $\kappa > 0$ and $\lambda > 0$

In this case both bounds functions $L + \kappa X$ and $U + \lambda X$ are increasing, so the integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, have the following form in increasing order of their bounds:

$$[L + \kappa P, U + \lambda P], \ldots, [L + \kappa X_i, U + \lambda X_i], [L + \kappa(X_i + 1), U + \lambda(X_i + 1)], \ldots, [L + \kappa Q, U + \lambda Q]$$

Since the union of all integer intervals constitutes an integer interval then the union of every two consecutive integer intervals $[L + \kappa X_i, U + \lambda X_i]$ and $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$, where $P \leq X_i \leq Q - 1$, must also constitute an integer interval. Therefore, the lower bound of the second interval must be less or equal than the upper bound of the first interval plus one, i.e.:

$$L + \kappa(X_i + 1) \leq U + \lambda X_i + 1$$
$$\Leftrightarrow \quad U - L + (\lambda - \kappa)X_i + 1 \geq \kappa$$

Since the above inequality holds for all values of X_i between P and $Q - 1$, then it must hold for the minimum value of the expression on the left hand side. Therefore:

$$U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^-(Q - 1) + 1 \geq \kappa$$
$$\Leftrightarrow \quad U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^- Q + 1 \geq \kappa - (\lambda - \kappa)^-$$

The expression on the right hand side of the above inequality is equal to $\min(|\kappa|, |\lambda|)$ for $\kappa > 0$ and $\lambda > 0$ and that concludes our proof in this case.

Case 2: $\kappa < 0$ and $\lambda < 0$

In this case both bounds functions $L + \kappa X$ and $U + \lambda X$ are decreasing, so the integer intervals $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q$, have the following form in increasing order of their bounds:

$$[L + \kappa Q, U + \lambda Q], \ldots, [L + \kappa(X_i + 1), U + \lambda(X_i + 1)], [L + \kappa X_i, U + \lambda X_i], \ldots, [L + \kappa P, U + \lambda P]$$

Since the union of all integer intervals constitutes an integer interval then the union of every two consecutive integer intervals $[L + \kappa(X_i + 1), U + \lambda(X_i + 1)]$ and $[L + \kappa X_i, U + \lambda X_i]$, where $P \leq X_i \leq Q - 1$, must also constitute an integer interval. For the same reason as in the previous case we can derive:

$$L + \kappa X_i \leq U + \lambda(X_i + 1) + 1$$
$$\Leftrightarrow \quad U - L + (\lambda - \kappa)X_i + 1 \geq -\lambda$$

Since the above inequality holds for all values of X_i between P and $Q - 1$, then it must hold for the minimum value of the expression on the left hand side. Therefore:

$$U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^-(Q - 1) + 1 \geq -\lambda$$
$$\Leftrightarrow \quad U - L + (\lambda - \kappa)^+ P - (\lambda - \kappa)^- Q + 1 \geq -\lambda - (\lambda - \kappa)^-$$

The expression in the right hand side of the above inequality is equal to $\min(|\kappa|, |\lambda|)$ for $\kappa < 0$ and $\lambda < 0$ and that concludes our proof in this case too.

∎

Proof of Theorem 3-3

According to the hypothesis $\kappa\lambda \leq 0$, or $\kappa\lambda > 0$ and $\min(U(\boldsymbol{x}) - L(\boldsymbol{x}) + (\lambda - \kappa)^+ P(\boldsymbol{x}) - (\lambda - \kappa)^- Q(\boldsymbol{x}) + 1)$ $\geq \min(|\kappa|, |\lambda|)$. Since the minimum value of the expression in the left hand side is greater or equal than the constant value on the right hand side we can derive that $\kappa\lambda \leq 0$, or $\kappa\lambda > 0$ and $U(\boldsymbol{x}_i) - L(\boldsymbol{x}_i) + (\lambda - \kappa)^+ P(\boldsymbol{x}_i) - (\lambda - \kappa)^- Q(\boldsymbol{x}_i) + 1 \geq \min(|\kappa|, |\lambda|)$, for all $\boldsymbol{x}_i \in \mathfrak{R}$ and $P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)$.

According to Theorem 3-2, each variable integer interval $[L(\boldsymbol{x}_i) + \kappa X, U(\boldsymbol{x}_i) + \lambda X]$, where $P(\boldsymbol{x}_i) \leq X \leq Q(\boldsymbol{x}_i)$, is equal to the integer interval $[L(\boldsymbol{x}_i) + \kappa^+ P(\boldsymbol{x}_i) - \kappa^- Q(\boldsymbol{x}_i), U(\boldsymbol{x}_i) + \lambda^+ Q(\boldsymbol{x}_i) - \lambda^- P(\boldsymbol{x}_i)]$. Now according to Definition 3-2:

$$\bigcup_{P(\boldsymbol{x}_i) \leq X_j \leq Q(\boldsymbol{x}_i)} [L(\boldsymbol{x}_i) + \kappa X_j, U(\boldsymbol{x}_i) + \lambda X_j] = [L(\boldsymbol{x}_i) + \kappa^+ P(\boldsymbol{x}_i) - \kappa^- Q(\boldsymbol{x}_i), U(\boldsymbol{x}_i) + \lambda^+ Q(\boldsymbol{x}_i) - \lambda^- P(\boldsymbol{x}_i)],$$

for all $\boldsymbol{x}_i \in \mathfrak{R}$ and $P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)$. Therefore:

$$\bigcup_{\substack{\boldsymbol{x}_i \in \mathfrak{R} \\ P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)}} \bigcup_{P(\boldsymbol{x}_i) \leq X_j \leq Q(\boldsymbol{x}_i)} [L(\boldsymbol{x}_i) + \kappa X_j, U(\boldsymbol{x}_i) + \lambda X_j] = \bigcup_{\substack{\boldsymbol{x}_i \in \mathfrak{R} \\ P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)}} [L(\boldsymbol{x}_i) + \kappa^+ P(\boldsymbol{x}_i) - \kappa^- Q(\boldsymbol{x}_i), U(\boldsymbol{x}_i) + \lambda^+ Q(\boldsymbol{x}_i) - \lambda^- P(\boldsymbol{x}_i)]$$

By Definition 3-2 the double union of the integer intervals on the left hand side of the above equation is equal to the variable integer interval $[L(\boldsymbol{x}) + \kappa X, U(\boldsymbol{x}) + \lambda X]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$. The union of the integer intervals on the right hand side of the equation is equal to the variable integer interval $[L(\boldsymbol{x}) + \kappa^+ P(\boldsymbol{x}) - \kappa^- Q(\boldsymbol{x}), U(\boldsymbol{x}) + \lambda^+ Q(\boldsymbol{x}) - \lambda^- P(\boldsymbol{x})]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$. We conclude that the variable integer interval $[L(\boldsymbol{x}) + \kappa X, U(\boldsymbol{x}) + \lambda X]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$ is equal to the variable integer interval $[L(\boldsymbol{x}) + \kappa^+ P(\boldsymbol{x}) - \kappa^- Q(\boldsymbol{x}), U(\boldsymbol{x}) + \lambda^+ Q(\boldsymbol{x}) - \lambda^- P(\boldsymbol{x})]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$.

∎

Proof of Theorem 3-6

Assume that the following variable interval equation is integer solvable in a region $\mathfrak{R} \subseteq Z^n$:

$$a_1X_1 + \ldots + a_nX_n = [L + b_1X_1 + \ldots + b_nX_n, U + c_1X_1 + \ldots + c_nX_n]$$

According to Definition 3-3 there exists $(x_1, \ldots, x_n) \in \Re$ such that:

$$L + b_1x_1 + \ldots + b_nx_n \leq a_1x_1 + \ldots + a_nx_n \leq U + c_1x_1 + \ldots + c_nx_n$$

We can write the above constraints equivalently as

$$L + (b_1 - v_1)x_1 + \ldots + (b_n - v_n)x_n \leq (a_1 - v_1)x_1 + \ldots + (a_n - v_n)x_n \leq$$
$$U + (c_1 - v_1)x_1 + \ldots + (c_n - v_n)x_n$$

The greatest common divisor d of $(a_i - b_i)$, $(a_i - c_i)$, for $1 \leq i \leq n$, also divides $(b_i - c_i)$ and $(c_i - b_i)$. Since v_i is either b_i or c_i, then d divides all the coefficients of the linear expressions in the above constraints, namely $(a_i - v_i)$, $(b_i - v_i)$ and $(c_i - v_i)$, for $1 \leq i \leq n$. In the integer domain the above constraints are equivalent to:

$$\lceil L/d \rceil + (b_1 - v_1)/dx_1 + \ldots + (b_n - v_n)/dx_n \leq (a_1 - v_1)/dx_1 + \ldots + (a_n - v_n)/dx_n$$
$$\leq \lfloor U/d \rfloor + (c_1 - v_1)/dx_1 + \ldots + (c_n - v_n)/dx_n$$

This implies that the variable interval equation

$$(a_1 - v_1)/dX_1 + \ldots + (a_n - v_n)/dX_n = [\lceil L/d \rceil + (b_1 - v_1)/dX_1 + \ldots + (b_n - v_n)/dX_n,$$
$$\lfloor U/d \rfloor + (c_1 - v_1)/dX_1 + \ldots + (c_n - v_n)/dX_n]$$

is integer solvable in $\Re$.

If the second variable interval equation is integer solvable in $\Re$, then the first variable interval equation is also integer solvable in $\Re$ because of the equivalence of the above constraints.

∎

Lemma 0-1:

Consider a sequence of non-empty integer intervals of the form $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$.

a. If L and U are increasing and $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then the union of all integer intervals $[L(X_i), U(X_i)]$ is equal to the integer interval $[L(P), U(Q)]$.

b. If L and U are decreasing and $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then the union of all integer intervals $[L(X_i), U(X_i)]$ is equal to the integer interval $[L(Q), U(P)]$.

Proof

Case a:

Since both L and U are increasing, the integer intervals $[L(X_i), U(X_i)]$ have the following form in increasing order of their bounds:

$$[L(P), U(P)], \ldots, [L(X_i), U(X_i)], [L(X_i + 1), U(X_i + 1)], \ldots, [L(Q), U(Q)]$$

We can rewrite $U(X_i) - L(X_i + 1) + 1 \geq 0$ as $L(X_i + 1) \leq U(X_i) + 1$. The above condition guarantees that the union of two consecutive integer intervals $[L(X_i), U(X_i)]$ and $[L(X_i + 1), U(X_i + 1)]$ is the integer interval $[L(X_i), U(X_i + 1)]$. By induction we can prove that the union of all integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L(P), U(Q)]$.

Case b:

Since both L and U are decreasing, the integer intervals $[L(X_i), U(X_i)]$ have the following form in increasing order of their bounds:

$$[L(Q), U(Q)], \ldots, [L(X_i + 1), U(X_i + 1)], [L(X_i), U(X_i)], \ldots, [L(P), U(P)]$$

We can rewrite $U(X_i + 1) - L(X_i) + 1 \geq 0$ as $L(X_i) \leq U(X_i + 1) + 1$. The above condition guarantees that the union of two consecutive integer intervals $[L(X_i + 1), U(X_i + 1)]$ and $[L(X_i), U(X_i)]$ is the integer interval $[L(X_i + 1), U(X_i)]$. By induction we can prove that the union of all integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L(Q), U(P)]$.

∎

Lemma 0-2:

Consider a sequence of integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, such that $[L(X_i), U(X_i)]$ is empty for all X_i, $P \leq X_i \leq Q - 1$.

a. If L and U are increasing and $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $L(X_i + 1) = L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

b. If L and U are decreasing and $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $U(X_i + 1) = U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

Proof

Case a:

Since integer intervals $[L(X_i), U(X_i)]$ are empty for all X_i, $P \leq X_i \leq Q - 1$, it follows that $U(X_i) < L(X_i)$ or $U(X_i) - L(X_i) + 1 \leq 0$ for all X_i, $P \leq X_i \leq Q - 1$. We know from the hypothesis that $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$. Combining the two inequalities we derive $L(X_i + 1) \leq L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$. Since L is increasing, $L(X_i + 1) \geq L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$. From the last two inequalities we conclude that $L(X_i + 1) = L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

Case b:

Similarly with case a, we derive that $U(X_i) - L(X_i) + 1 \leq 0$ for all X_i, $P \leq X_i \leq Q - 1$. From the hypothesis we have $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$. Combining the two inequalities we derive $U(X_i) \leq U(X_i + 1)$ for all X_i, $P \leq X_i \leq Q - 1$. Since U is decreasing, $U(X_i) \geq U(X_i + 1)$ for all X_i, $P \leq X_i \leq Q - 1$. From the last two inequalities we conclude that $U(X_i + 1) = U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

∎

Lemma 0-3:

Consider a sequence of integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, such that $[L(X_i), U(X_i)]$ is empty for all X_i, $P + 1 \leq X_i \leq Q$.

a. If L and U are increasing and $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $U(X_i + 1) = U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

b. If L and U are decreasing and $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$, then $L(X_i + 1) = L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

Proof

Case a:

Since integer intervals $[L(X_i), U(X_i)]$ are empty for all X_i, $P + 1 \leq X_i \leq Q$, it follows that $U(X_i) < L(X_i)$ or $U(X_i) - L(X_i) + 1 \leq 0$ for all X_i, $P + 1 \leq X_i \leq Q$. This can be written equivalently as $U(X_i + 1) - L(X_i + 1) + 1 \leq 0$ for all X_i, $P \leq X_i \leq Q - 1$. We know from the hypothesis that $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$. Combining the two inequalities we derive $U(X_i + 1) \leq U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$. Since U is increasing, $U(X_i + 1) \geq U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$. From the last two inequalities we conclude that $U(X_i + 1) = U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

Case b:

Similarly with case a, we derive that $U(X_i + 1) - L(X_i + 1) + 1 \leq 0$ for all X_i, $P \leq X_i \leq Q - 1$. From the hypothesis we have $U(X_i + 1) - L(X_i) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$. Combining the two

inequalities we derive $L(X_i) \leq L(X_i + 1)$ for all X_i, $P \leq X_i \leq Q - 1$. Since L is decreasing, $L(X_i) \geq L(X_i + 1)$ for all X_i, $P \leq X_i \leq Q - 1$. From the last two inequalities we conclude that $L(X_i + 1) = L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$.

∎

Proof of Theorem 4-1

According to Definition 3-2:

$$[L(X), U(X)] = \bigcup_{i=1}^{n} [L(X_i), U(X_i)], \text{ where } X_1 = P, X_2 = P + 1, \ldots, X_n = Q.$$

Case a: L is decreasing and U is increasing

Since L is decreasing and U is increasing, $L(X_i + 1) \leq L(X_i)$ and $U(X_i) \leq U(X_i + 1)$ for all X_i, $P \leq X_i \leq Q - 1$. Therefore, $[L(X_i), U(X_i)] \subseteq [L(X_i + 1), U(X_i + 1)]$ for all X_i, $P \leq X_i \leq Q - 1$. It follows that $[L(P), U(P)] \subseteq [L(P + 1), U(P + 1)] \subseteq \ldots \subseteq [L(Q), U(Q)]$ and therefore the union of all integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, is equal to the integer interval $[L(Q), U(Q)]$. We conclude that the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, is equal to the integer interval $[L(Q), U(Q)]$.

Case b: L is increasing and U is decreasing

Since L is increasing and U is decreasing, $L(X_i) \leq L(X_i + 1)$ and $U(X_i + 1) \leq U(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$. In this case $[L(P), U(P)] \supseteq [L(P + 1), U(P + 1)] \supseteq \ldots \supseteq [L(Q), U(Q)]$ and therefore the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, is equal to the integer interval $[L(P), U(P)]$.

Case c: L and U are increasing and $U(X_i) - L(X_i + 1) + 1 \geq 0$ for all X_i, $P \leq X_i \leq Q - 1$

Case c.1: In case all integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, are non-empty, according to Lemma 1.a, their union is equal to $[L(P), U(Q)]$. Therefore, the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, is equal to the integer interval $[L(P), U(Q)]$.

Case c.2: In case all intervals integer intervals $[L(X_i), U(X_i)]$, where $P \leq X_i \leq Q$, are empty, their union is empty and therefore the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, is also empty. According to Lemma *0*-2.a, $L(X_i + 1) = L(X_i)$ for all X_i, $P \leq X_i \leq Q - 1$, which implies that $L(P) = L(Q)$. Therefore, the integer interval $[L(P), U(Q)]$ is equal to the integer interval $[L(Q), U(Q)]$, which is empty. In conclusion the variable integer interval $[L(X), U(X)]$, where $P \leq X \leq Q$, is equal to the integer interval $[L(P), U(Q)]$.

Case c.3: In case there exist both empty and non-empty integer intervals, consider two sequences P_k, Q_k, for $k = 1, \ldots, m$, where $P \leq P_1 \leq Q_1 < \ldots < P_k \leq Q_k < P_{k+1} \leq Q_{k+1} < \ldots < P_m \leq Q_m \leq Q$. Each pair P_k, Q_k defines the maximum range of consecutive values X_i, such that all integer intervals $[L(X_i), U(X_i)]$,

where $P_k \le X_i \le Q_k$, are non-empty. In this case the integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le Q$, have the following form in increasing order of their bounds:

$[L(P), U(P)]$, …, $[L(P_1), U(P_1)]$, …, $[L(Q_1), U(Q_1)]$, …, $[L(P_k), U(P_k)]$, …, $[L(Q_k), U(Q_k)]$,

$[L(Q_k + 1), U(Q_k + 1)]$, …, $[L(P_{k+1} - 1), U(P_{k+1} - 1)]$, $[L(P_{k+1}), U(P_{k+1})]$, …, $[L(Q_{k+1}), U(Q_{k+1})]$, …,

$[L(P_m), U(P_m)]$, …, $[L(Q_m), U(Q_m)]$, …, $[L(Q), U(Q)]$

The integer intervals $[L(X_i), U(X_i)]$, where $P_k \le X_i \le Q_k$, for $k = 1, 2, …, m$ are the only non-empty integer intervals in the union . The integer intervals $[L(X_i), U(X_i)]$, where $Q_k + 1 \le X_i \le P_{k+1} - 1$, for $k = 1, 2, …, m - 1$ are empty. Also if $P < P_1$ the integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le P_1 - 1$, are empty and if $Q_m < Q$ the integer intervals $[L(X_i), U(X_i)]$, where $Q_m + 1 \le X_i \le Q$, are empty.

From Lemma *0*-1.a we derive that the union of the non-empty integer intervals $[L(X_i), U(X_i)]$, $P_k \le X_i \le Q_k$ is equal to the integer interval $[L(P_k), U(Q_k)]$ for $k = 1, 2, …, m$. From Lemma *0*-2.a, since all integer intervals $[L(X_i), U(X_i)]$, where $Q_k + 1 \le X_i \le P_{k+1} - 1$ are empty, we derive that $L(X_i + 1) = L(X_i)$ for all X_i, $Q_k + 1 \le X_i \le P_{k+1} - 1$, and therefore $L(P_{k+1}) = L(Q_k + 1)$ for $k = 1, 2, …, m - 1$. From the hypothesis we also know that $U(Q_k) - L(Q_k + 1) + 1 \ge 0$ and by the last equality we can derive that $U(Q_k) - L(P_{k+1}) + 1 \ge 0$. The previous inequality can be written as $L(P_{k+1}) \le U(Q_k) + 1$, which guarantees that the union of the non-empty integer intervals $[L(P_k), U(Q_k)]$ and $[L(P_{k+1}), U(Q_{k+1})]$ constitutes the non-empty integer interval $[L(P_k), U(Q_{k+1})]$ for $k = 1, 2, …, m - 1$. By induction, similarly to Lemma *0*-1, we can prove that the union of all $[L(P_k), U(Q_k)]$ for $k = 1, …, m$ is equal to $[L(P_1), U(Q_m)]$. The union of all integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le Q$, is equal to the union of the non-empty integer intervals $[L(P_k), U(Q_k)]$ for $k = 1, …, m$. Therefore the union of all integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le Q$ is equal to the integer interval to $[L(P_1), U(Q_m)]$.

If $P_1 = P$, i.e. no empty integer intervals exist between P and P_1, then $L(P_1) = L(P)$. If $P < P_1$, i.e. there exist empty integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le P_1 - 1$, then by Lemma *0*-2.a we also derive that $L(P_1) = L(P)$. Similarly, if $Q_m = Q$, then $U(Q_m) = U(Q)$ and if $Q_m < Q$, i.e. there exist empty integer intervals $[L(X_i), U(X_i)]$, where $Q_m + 1 \le X_i \le Q$, then by Lemma *0*-3.a we also derive that $U(Q_m) = U(Q)$. Therefore, the union of all integer intervals $[L(X_i), U(X_i)]$, where $P \le X_i \le Q$ is equal to the integer interval to $[L(P), U(Q)]$.

In conclusion the variable integer interval $[L(X), U(X)]$, where $P \le X \le Q$, is equal to the integer interval $[L(P), U(Q)]$.

Case d: L and U are decreasing and $U(X_i + 1) - L(X_i) + 1 \ge 0$ for all X_i, $P \le X_i \le Q - 1$

Similarly with case c, using Lemmas *0*-1.b, *0*-2.b and *0*-3.b we can prove that the variable integer interval $[L(X), U(X)]$, where $P \le X \le Q$, is equal to $[L(Q), U(P)]$

■

Proof of Theorem 4-2

Case a: L is decreasing and U is increasing for variable X

Consider the single variable integer intervals $[L(x_i, X), U(x_i, X)]$, where $P(x_i) \leq X \leq Q(x_i)$, for all $x_i \in \Re$ such that $P(x_i) \leq Q(x_i)$. Because L is decreasing and U is increasing for variable X we derive by Theorem 4-1.a that each variable integer interval $[L(x_i, X), U(x_i, X)]$, where $P(x_i) \leq X \leq Q(x_i)$ is equal to the integer interval $[L(x_i, Q(x_i)), U(x_i, Q(x_i))]$ for all $x_i \in \Re$ such that $P(x_i) \leq Q(x_i)$. Now according to Definition 3-2 of the variable integer interval $[L(x_i, X), U(x_i, X)]$, where $P(x_i) \leq X \leq Q(x_i)$ it follows that:

$$\bigcup_{P(x_i) \leq X_j \leq Q(x_i)} [L(x_i, X_j), U(x_i, X_j)] = [L(x_i, Q(x_i)), U(x_i, Q(x_i))]$$

for all $x_i \in \Re$ such that $P(x_i) \leq Q(x_i)$. Therefore,

$$\bigcup_{\substack{x_i \in \Re \\ P(x_i) \leq Q(x_i)}} \bigcup_{P(x_i) \leq X_j \leq Q(x_i)} [L(x_i, X_j), U(x_i, X_j)] = \bigcup_{\substack{x_i \in \Re \\ P(x_i) \leq Q(x_i)}} [L(x_i, Q(x_i)), U(x_i, Q(x_i))]$$

By Definition 3-2 the double union of the integer intervals on the left hand side of the above equation is equal to the variable integer interval $[L(x, X), U(x, X)]$, where x in $\Re$ and $P(x) \leq X \leq Q(x)$. Also, the union of the integer intervals on the right hand side of the equation is equal to the variable integer interval $[L(x, Q(x)), U(x, Q(x))]$, where x in $\Re$ and $P(x) \leq Q(x)$. We conclude that the variable integer interval $[L(x, X), U(x, X)]$, where x in $\Re$ and $P(x) \leq X \leq Q(x)$ is equal to the variable integer interval $[L(x, Q(x)), U(x, Q(x))]$, where x in $\Re$ and $P(x) \leq Q(x)$.

Case b: L is increasing and U is decreasing for variable X

Similarly with case a, we derive by Theorem 4-1.b and Definition 3-2 that the variable integer interval $[L(x, X), U(x, X)]$, where x in $\Re$ and $P(x) \leq X \leq Q(x)$ is equal to the variable integer interval $[L(x, P(x)), U(x, P(x))]$, where x in $\Re$ and $P(x) \leq Q(x)$.

Case c: L and U are increasing for variable X and $\min(U(x, X) - L(x, X + 1) + 1) \geq 0$

Since $\min(U(x, X) - L(x, X + 1) + 1) \geq 0$, where x in $\Re$ and $P(x) \leq X \leq Q(x)$, we can derive that $\min(U(x_i, X) - L(x_i, X + 1) + 1) \geq 0$, where $P(x_i) \leq X \leq Q(x_i)$, for all $x_i \in \Re$ such that $P(x_i) \leq Q(x_i)$. For each single variable integer interval $[L(x_i, X), U(x_i, X)]$, where $P(x_i) \leq X \leq Q(x_i)$, we can derive that $U(x_i, X_j) - L(x_i, X_j + 1) + 1 \geq 0$, for all X_j, $P(x_i) \leq X_j \leq Q(x_i) - 1$. In addition because L and U are

increasing for variable X we derive by Theorem 4-1.c that each variable integer interval $[L(\boldsymbol{x}_i, X), U(\boldsymbol{x}_i, X)]$, $P(\boldsymbol{x}_i) \leq X \leq Q(\boldsymbol{x}_i)$ is equal to the integer interval $[L(\boldsymbol{x}_i, P(\boldsymbol{x}_i)), U(\boldsymbol{x}_i, Q(\boldsymbol{x}_i))]$ for all $\boldsymbol{x}_i \in \mathfrak{R}$ such that $P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)$. Now according to Definition 3-2 of the variable integer interval $[L(\boldsymbol{x}_i, X), U(\boldsymbol{x}_i, X)]$, where $P(\boldsymbol{x}_i) \leq X \leq Q(\boldsymbol{x}_i)$ it follows that:

$$\bigcup_{P(\boldsymbol{x}_i) \leq X_j \leq Q(\boldsymbol{x}_i)} [L(\boldsymbol{x}_i, X_j), U(\boldsymbol{x}_i, X_j)] = [L(\boldsymbol{x}_i, P(\boldsymbol{x}_i)), U(\boldsymbol{x}_i, Q(\boldsymbol{x}_i))]$$

for all $\boldsymbol{x}_i \in \mathfrak{R}$ such that $P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)$. Therefore,

$$\bigcup_{\substack{\boldsymbol{x}_i \in \mathfrak{R} \\ P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)}} \bigcup_{P(\boldsymbol{x}_i) \leq X_j \leq Q(\boldsymbol{x}_i)} [L(\boldsymbol{x}_i, X_j), U(\boldsymbol{x}_i, X_j)] = \bigcup_{\substack{\boldsymbol{x}_i \in \mathfrak{R} \\ P(\boldsymbol{x}_i) \leq Q(\boldsymbol{x}_i)}} [L(\boldsymbol{x}_i, P(\boldsymbol{x}_i)), U(\boldsymbol{x}_i, Q(\boldsymbol{x}_i))]$$

Similarly with case a, by Definition 3-2, we conclude that the variable integer interval $[L(\boldsymbol{x}, X), U(\boldsymbol{x}, X)]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$ is equal to the variable integer interval $[L(\boldsymbol{x}, P(\boldsymbol{x})), U(\boldsymbol{x}, Q(\boldsymbol{x}))]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$.

Case d: L and U are decreasing for variable X and $\min(U(\boldsymbol{x}, X + 1) - L(\boldsymbol{x}, X) + 1) \geq 0$

Similarly with case c, we derive by Theorem 4-1.d and Definition 3-2 that the variable integer interval $[L(\boldsymbol{x}, X), U(\boldsymbol{x}, X)]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq X \leq Q(\boldsymbol{x})$ is equal to the variable integer interval $[L(\boldsymbol{x}, Q(\boldsymbol{x})), U(\boldsymbol{x}, P(\boldsymbol{x}))]$, where $\boldsymbol{x}$ in $\mathfrak{R}$ and $P(\boldsymbol{x}) \leq Q(\boldsymbol{x})$.

∎

Proof of Theorem 4-4

Assume that the following variable interval equation is integer solvable in a region $\mathfrak{R} \subseteq Z^n$:

$$F(\boldsymbol{x}) = [L(\boldsymbol{x}) + L_0, U(\boldsymbol{x}) + U_0]$$

According to Definition 3-3 there exists $\boldsymbol{x}_0 \in \mathfrak{R}$ such that:

$$L(\boldsymbol{x}_0) + L_0 \leq F(\boldsymbol{x}_0) \leq U(\boldsymbol{x}_0) + U_0$$

Since d is the greatest common divisor of the coefficients of all the terms in expressions $F(\boldsymbol{x})$, $L(\boldsymbol{x})$, $U(\boldsymbol{x})$, in the integer domain the above constraints are equivalent to:

$$L(\boldsymbol{x}_0)/d + \lceil L_0/d \rceil \leq F(\boldsymbol{x}_0)/d \leq U(\boldsymbol{x}_0)/d + \lfloor U_0/d \rfloor$$

This implies that the variable interval equation

$$F(\boldsymbol{x})/d = [L(\boldsymbol{x})/d + \lceil L_0/d \rceil, U(\boldsymbol{x})/d + \lfloor U_0/d \rfloor]$$

110

is integer solvable in $\mathfrak{R}$.

If the second variable interval equation is integer solvable in $\mathfrak{R}$ then the first interval equation is also integer solvable in $\mathfrak{R}$ because of the equivalence between the above constraints.

∎

BIBLIOGRAPHY

[1] K. Atkinson and W. Han, *Elementary Numerical Analysis*, Wiley, Hoboken, NJ, 2003.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[3] David F. Bacon, Susan L. Graham and Oliver J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing. Surveys*, Vol. 26, No. 4, December 1994.

[4] U. Banerjee, *Dependence Analysis*, Kluwer Academic Publishers, Norwell, MA, 1997.

[5] J. Birtch, R. A. van Engelen, K. A. Gallivan and Y. Shou, "An Empirical Evaluation of Chains of Eecurrences for Array Dependence Testing," *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques*, Seattle, WA, September 2006.

[6] W. Blume and R. Eigenmann, "Nonlinear and Symbolic Data Dependence Testing," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 12, December 1998.

[7] W. Blume and R. Eigenmann, "Symbolic Range Propagation," *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[8] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. A. Padua, Y. Paek, W. M. Pottenger, L. Rauchwerger and P. Tu, "Parallel Programming with Polaris," *IEEE Computer*, Vol. 29, No. 12, December 1996.

[9] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *Proceedings of SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[10] D. Callahan, "Interprocedural Constant Propagation", *Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto Cal, June 1986.

[11] G. Dantzig and B. Eaves, "Fourier-Motzkin Elimination and its Dual," *Journal of Combinatorial Theory (A)*, Vol. 14, 1973.

[12] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the Automatic Parallelization of the Perfect benchmarks"*, IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 1, January 1998.

[13] R. A. van Engelen, J. Birtch, Y. Shou, B. Walsh and K. A. Gallivan, "A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis," *Proceedings of the ACM International Conference on Supercomputing*, Saint-Malo, France, June 2004.

[14] G. Golf, K. Kennedy, and C. Tseng, "Practical Dependence Testing," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[15] M. R. Haghighat, "Symbolic Analysis for Parallelizing Compilers," *Kluwer Academic Publishers,* Norwell, MA, 1995.

[16] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. S. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," *Proceedings of the ACM International Conference on Supercomputing*, San Diego, CA, December, 1995.

[17] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, Vol. 33, No. 7, July 2000

[18] D. Knuth, *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Addison-Wesley, 1981.

[19] X. Kong, D. Klappholz, and K. Psarris, "The I-Test: An Improved Dependence Test for Automatic Parallelization and Vectorization," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991.

[20] D. Kuck, Y. Muraoka, S. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup", *IEEE Transactions on Computers*, Vol. C-21, No. 12, December 1972.

[21] D. E. Knuth, *The art of computer programming, Vol. 2, Seminumerical Algorithms*, Addison-Wesley 1981.

[22] Z. Li, P. Yew, and C. Zhu, "An Efficient Data Dependence Analysis for Parallelizing Compilers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January 1990.

[23] T. J. Marlowe, "Pointer-Induced Aliasing: A Clarification", *Sigplan Notices*, Vol. 28, No. 9, September 1993.

[24] D. Maydan, J. Hennesy, and M. Lam, "Efficient and Exact Data Dependence Analysis for Parallelizing Compilers," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[25] D. Niedzielski and K. Psarris, "An Analytical Comparison of the I-Test and Omega Test," *Proceedings of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[26] Y Paek, J. Hoelflinger, D. Padua, "Efficient and Precise Array Access Analysis," *ACM Transactions on Programming Languages and Systems*, Vol. 24, No. 1, January 2002.

[27] P. Petersen and D. Padua, "Static and Dynamic Evaluation of Dependence Analysis Techniques". *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 11, November 1996.

[28] K. Psarris, "The Banerjee-Wolfe and GCD Tests on Exact Data Dependence Information," *Journal of Parallel and Distributed Computing*, Vol. 32, No. 2, February 1996.

[29] K. Psarris, D. Klappholz, and X. Kong, "On the Accuracy of the Banerjee Test," *Journal of Parallel and Distributed Computing,* Vol. 12, No. 2, June 1991.

[30] K. Psarris, X. Kong, and D. Klappholz, "The Direction Vector I Test," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 11, November 1993.

[31] W. Pugh, "A Practical Algorithm for Exact Array Dependence Analysis," *Communications of the ACM*, Vol. 35, No. 8, August 1992.

[32] W. Pugh and D. Wonnacott, "Eliminating False Data Dependences using the Omega Test," *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[33] W. Pugh and D. Wonnacott, "Constraint-based array dependence analysis," *ACM Transactions on Programming Languages and Systems*, Vol 20, No 3, May 1998.

[34] S. Rus, D. Zhang, L. Rauchwerger, "The Value Evolution Graph and its Use in Memory Reference Analysis," *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, Antibes Juan-les-Pins, France, October 2004.

[35] R. Seater and D. Wonnacott, "Efficient Manipulation of Disqualities During Dependence Analysis," 15th Workshop on Languages and Compilers for Parallel Computing, College Park, MD, July 2002.

[36] Z. Shen, Z. Li, and P. C. Yew, "An Empirical Study of Fortran Programs for Parallelizing Compilers" *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, July 1992..

[37] M. E. Wolfe, *Optimizing Compilers for Supercomputers*, Pitman, London, and MIT Press, Cambridge, MA, 1989.

[38] M. E. Wolfe, "The Tiny Loop Restructuring Research Tool," *Proceedings of the 1991 International Conference on Parallel Processing*, St Charles, IL, August 1991.

[39] M. E. Wolfe and C. Tseng, "The Power Test for Data Dependence," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, September 1992.

[40] M. E. Wolf, "Beyond Induction Variables", *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 162-174, San Francisco Cal, June 1992.

[41] P. Wu, A. Cohen, J. Hoeflinger and D. Padua, "Monotonic evolution: an alternative to induction variable substitution for dependence analysis," *Proceedings of the ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.

VITA

Konstantinos Kyriakopoulos was born in November 2nd 1974 in Kalamata of Messinia in Greece. He received a BS degree in computer science from the University of Piraeus, Greece, in 1997. He earned an MS degree in computer science from the University of Texas at San Antonio in 1999. From 2000 until 2002, he worked as a software engineer for Advanced Micro Devices in Austin, Texas. From May of 2002 until his graduation in May of 2007 he has be working on the PLATO research project.