Data Dependence Analysis For Supercompilers: The λ Test Revisited

Dirk Grunwald

CU-CS-452-89 November 1989

University of Colorado at Boulder

Technical Report CU-CS-452-89
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Data Dependence Analysis For Supercompilers: The λ Test Revisited

Dirk Grunwald

November 1989

Abstract

In [LYZ89b, LYZ89a], Li, Yew and Zhu describe the λ-test for analysis of data dependence between array references. Li *et al* measured the efficacy of the λ-test, and found that it determines non-dependence in many cases where another common test, the Wolfe-Banerjee subscript-by-subscript test, indicated dependence. In this report, we show their algorithm is a simple, yet effective, generalization of the subscript-by-subscript test. A simplified implementation for the algorithm is given, and several examples are provided.

1 Introduction

In [LYZ89b, LYZ89a], Li, Yew and Zhu describe the λ-test for analysis of data dependence between array references. Li *et al* measured the efficacy of the λ-test, and found that it determines non-dependence in many cases where another common test, the Banerjee-Wolfe subscript-by-subscript test, indicated dependence. In sample scientific libraries, Li *et al* found the λ-test increased the average execution time for dependence analysis by approximately 50% to 70%.

At first reading, [LYZ89b, LYZ89a] does not clearly define the mechanics of an actual implementation. In this report, we present their algorithm in a clearer framework. A simplified implementation for the algorithm is given, and several examples are provided. These examples show that the λ-test is simple to understand and efficient to implement. Moreover, we show that the λ-test subsumes the Banerjee-Wolfe subscript-by-subscript test and subscript linearization. We also illustrate the reason for the increased execution time reported by Li [LYZ89b, LYZ89a].

2 Problem Statement

We are concerned with potential data dependence between multiply subscripted array references within multiple nested loops. In general, we're concerned with arrays references with s subscripts nested in l loops,

```
        DO I1 = L1,U1,N1
            DO I2 = L2,U2,N2
                ...
                    DO Il = Ll,Ul,Nl
S1:                     A(f1(I1,I2, ...,Il), f2(I1,I2, ...,Il), ..., fs(I1,I2, ...,Il))
                        ...
S2:                     A(g1(I1,I2, ...,Il), g2(I1,I2, ...,Il), ..., gs(I1,I2, ...,Il))
                    END
            END
        END
```

We assume each function, f_i or g_i, is a linear function of the iteration variables. We let $\vec{i}$ and $\vec{j}$ be vectors of particular instances of the iteration variables, $\mathrm{I}_1, \mathrm{I}_2, \ldots, \mathrm{I}_l$ for some execution of the loops. Dependence between S1 and S2 can only exist when

$$\exists \vec{i}, \vec{j} \quad \text{s.t.} \quad \forall_{1 \leq k \leq s} \quad f_k(\vec{i}) = g_k(\vec{j}) .$$

In short, there must exist instances of the iteration variables such that S1 and S2 explicitly refer to the same array location within the same iteration. If there exist any k such that $f_k(\vec{i}) \neq g_k(\vec{j})$ for all instances of the iteration variables $\vec{i}$ and $\vec{j}$, we can dismiss any possibility of dependence between S1 and S2, assuming that all array subscripts match the array dimensions.

We can restate our constraints as a series of s equalities in $2l$ variables: array references S1 and S2 are dependent *iff* we can show that, for some $\vec{i}, \vec{j}$,

$$\begin{aligned} f_1(\vec{i}) - g_1(\vec{j}) &= 0 \\ \text{and } f_2(\vec{i}) - g_2(\vec{j}) &= 0 \\ &\vdots \\ \text{and } f_s(\vec{i}) - g_s(\vec{j}) &= 0 . \end{aligned} \tag{1}$$

In other words, all subscripts must be equal for the iteration.

A *coupled* array reference involves multiple iteration variables in one or more array subscripts. In an empirical study of array subscripts, Shen, Li and Yew [SLY89] showed that coupled subscripts constitute 30% of two dimensional array references. Li *et al* [LYZ89b, LYZ89a] show that the Banerjee-Wolfe subscript-by-subscript test can falsely indicate dependence in array references involving coupled subscripts. For example, in the following program

```
              REAL A(100,100)
              DO I=1,100
                  DO J=2,100
      S1:             X(I, J) = ...
      S2:             ... = X(J, I)
                  ENDDO
              ENDDO
```

the Banerjee-Wolfe subscript-by-subscript [BCKT79, Wol82, Ban88, Wol89] test indicates dependence with all direction vectors. By comparison the λ-test demonstrates dependence with specific direction vectors: S1 $\delta_{(<,>)}$ S2, S1 $\delta_{(=,=)}$ S2, S1 $\delta_{(>,<)}$ S2. This allows the inner loop to be parallelized, because we know there is no dependence with direction S1 $\delta_{(<,=)}$ S2.

3 The λ-test As Originally Formulated

In the λ-test, we solve a less exact problem: can we find multiplicative constants $\lambda_1, \lambda_2, \ldots, \lambda_s$ such that a linear combination of our equalities are zero:

$$\lambda_1(f_1(\vec{i}) - g_1(\vec{j})) + \lambda_2(f_2(\vec{i}) - g_2(\vec{j})) + \ldots + \lambda_s(f_s(\vec{i}) - g_s(\vec{j})) = 0 \tag{2}$$

We call any particular combination $(\lambda_1, \lambda_2, \ldots, \lambda_s)$ a λ-tuple. In the special case of double subscripted loops, or $s = 2$, we call (λ_1, λ_2) a λ-pair.

Clearly, (2) is a less precise test than (1). There are instances where the individual terms of (2) are non-zero, indicating independence, while their sum is zero, indicating dependence. The choice of the λ-tuple is arbitrary because (2) is an approximation (1). If all terms of (1) are zero, indicating dependence, then any λ-tuple will also produce zero, indicating dependence. Thus, (2) is a conservative approximation of our actual goal.

Li *et al* [LYZ89b, LYZ89a] show that each linear equation of (1) defines a hyperplane π in $\Re^{2l}$ space. The intersection of these hyperplanes, called S, corresponds to the common solutions of the linear equations. Hyperplanes defining the possible iteration space delimit a bounded convex set V in $\Re^{2l}$. If S is empty, or if S does not intersect V, there can be no dependence between the two array references.

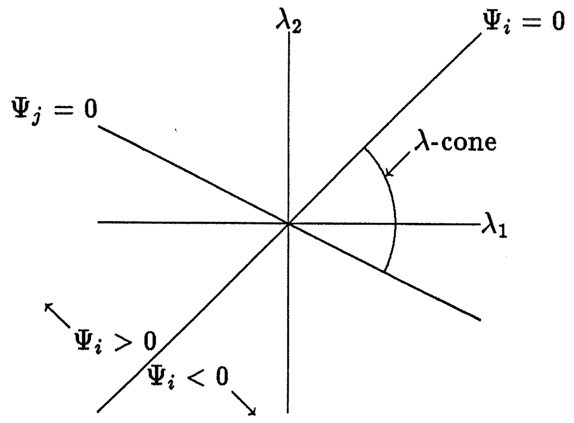

Figure 1: Two Possible Ψ-lines in $\Re^2$

Li *et al* show that S and V can be shown to intersect if there exists a hyperplane π corresponding to a linear combination of inequalities in (1). Li rewrites (2) using a vector of iteration variables, $\vec{v}$:

$$\left\langle \sum_{1 \leq i \leq s} \lambda_i \vec{u_i},\ \vec{v} \right\rangle + \sum_{1 \leq i \leq s} \lambda_i c_i = 0 \tag{3}$$

where $< \vec{u_i},\ \vec{v} >$ denotes the inner product of vector $\vec{u_i}$ and $\vec{v}$, and u_i is a vector of constants particular to the function $f_i - g_i$ in (1).

We can rewrite the equality using a λ-tuple to define (3) as a hyperplane in $\Re^{2l}$. The λ-tuple can be any tuple in $\Re^s$. For example, consider a double subscript array reference, where (3) becomes

$$\begin{aligned} \lambda_1(u_{1,1}v_1 + u_{1,2}v_2 + c_1) + \lambda_2(u_{2,1}v_1 + u_{2,2}v_2 + c_1) &= 0 \\ v_1(\lambda_1 u_{1,1} + \lambda_2 u_{2,1}) + v_2(\lambda_1 u_{2,1} + \lambda_2 u_{2,2}) &= -(\lambda_1 c_1 + \lambda_2 c_2) \end{aligned}$$

The coefficients of v_1 and v_2 define so-called Ψ-lines in $\Re^s$; see Figure 1. Each Ψ-line divides $\Re^s$ into two sections, $\Psi_i > 0$ and $\Psi_i < 0$. The intersection of two Ψ-lines defines a λ-cone. Within a λ-cone, no function defining a Ψ-line can change its sign. The λ-test as defined by Li *et al* defines v_{min} and v_{max} for each λ-cone; these vectors are substituted in (3) to produce minimal and maximal values of (3) within the λ-cone. When considering dependence direction vectors, additional lines, called Φ-lines, are used to subdivide $\Re^s$ into additional λ-cones. The definition of Ψ-lines and Φ-lines are somewhat involved, as is the computation of v_{min} and v_{max}. The interested reader is referred to [LYZ89b, LYZ89a].

4 The λ-test Revisited

We begin our explanation using (2). The λ-test, like the Banerjee-Wolfe inequalities, computes lower and upper bounds for the left hand side of (2). The equality is conservatively recast as an inequality; dependence is assumed if the lower bound, LB, and upper bound, UB, bracket zero. Both LB and UB are computed using the loop limits of individual loops, L_i and U_i, as well as the loop increment, N_i. We compute a unique LB and UB for each λ-tuple chosen. Our goal is to demonstrate independence between two array references or to determine dependence with a particular dependence direction, Ψ. We do so by selecting a λ-tuple, computing UB and LB and proclaiming dependence if

$$LB \leq 0 \leq UB\,.$$

In practice, we collect all constants in each term of (2) into a constant C, unique for each λ-tuple and then proclaim dependence if

$$LB' \leq C \leq UB'\,,$$

where LB' and UB' are computed from (2) with the constants collected on the right hand side. The lower and upper bounds are actually computed from lower and upper bounds for each element of the iteration vector. Each bound, LB_i or UB_i, corresponds to the minimum or maximum for the i^{th} element of the iteration vectors; for example, the i^{th} lower bound is

$$LB_i = v_{min,i} \sum_{1 \leq k \leq s} \lambda_k a_{i,k}\,,$$

and the upper bound is similar. Thus, our basic test is that we declare dependence if, for a particular λ-tuple we can show

$$\sum_{1 \leq i \leq 2l} LB_i \leq -(\sum_{1 \leq p \leq s} \lambda_p c_p) \leq \sum_{1 \leq i \leq 2l} UB_i\,,$$

where LB_i and UB_i are the lower and upper bounds for a particular λ-tuple. Our task, then, is to determine what λ-tuples should be selected, and, given those λ-tuples, what are appropriate values for LB_i and UB_i.

Presented this way, we see the relationship between the λ-test and subscript-by-subscript tests using Banerjee-Wolfe inequalities [BCKT79, Wol82, Ban88, Wol89]. Consider the λ-test with the λ-tuple $(1, 0, \ldots, 0)$. In effect, this causes the λ-test to only examine the equations involving the first subscript. With this λ-tuple, manipulation of (2) shows that the λ-test subsumes the subscript-by-subscript test. The latter is a special case using a particular set of λ-tuples:

$$\begin{aligned}\Lambda &= \{(1,0,\ldots,0,0),\ (0,1,\ldots,0,0),\\ &\quad \ldots,\ (0,0,\ldots,1,0),\ (0,0,\ldots,0,1)\}.\end{aligned}$$

For example, in the following code fragment

```
          REAL A(100,100)
          DO I=1,100
             DO J=2,100
S1:             X(I, J) = ...
S2:             ... = X(J, I)
             ENDDO
          ENDDO
```

the dependence equation (2) becomes

$$\lambda_1(I - J') + \lambda_2(J - I') = 0\,,$$

where I' and J' refer to variables I and J in another iteration. If we select $(\lambda_1, \lambda_2) = (1, 0)$, this equality becomes $I = J'$, while $(\lambda_1, \lambda_2) = (0, 1)$ produces $J = I'$. These are precisely the dependence relations used in testing the first and second subscripts in the Banerjee-Wolfe subscript-by-subscript test. The additional computation for the λ-test reported by Li [LYZ89b, LYZ89a] arises from a larger set Λ, and the concomitant examination of the additional inequalities. In the subscript-by-subscript test, the set Λ has s elements, one for each subscript. As presented in [LYZ89b, LYZ89a], the λ-test typically uses a set with $2s$ elements, or $3s$ elements when considering dependence direction vectors. The λ-test need not always take longer, because early detection of independence terminates the test.

A suitable λ-tuple also shows that the λ-test subsumes the linearized subscript test, another common data dependence test. Recall that each λ_i is associated with subscript i. We can restate (2) as

$$\lambda_1 f_1(\vec{i}) + \lambda_2 f_2(\vec{i}) + \ldots + \lambda_s f_s(\vec{ic}) = \lambda_1 g_1(\vec{i}) + \lambda_2 g_2(\vec{i}) + \ldots + \lambda_s g_s(\vec{i}) \tag{4}$$

Assume our array, A, has span S_i in dimension i and that we are using a language such as FORTRAN, with column major ordering. If we let $\lambda_1 = 1$ and

$$\lambda_i = \prod_{1 \le k \le i} S_k$$

for $i > 1$, each term in (4) refers to the linearized address of the array members identified by the references, if arrays originate at zero. If, as in FORTRAN, arrays originate at one, the array references can be shifted to zero. For example, consider the following code fragment:

```
          REAL A(100,100)
          DO I=1,100
             DO J=2,100
S1:             X(I, J) = ...
S2:             ... = X(J, I)
             ENDDO
          ENDDO
```

Because the array has span 100, a linearized reference for `A(I,J)` is `A(I + J*100 - 100)`, and `A(J,I)` is linearized as `A(J+I*100 - 100)`. The λ-pair $(1, 100)$ yields the following equality from (4).

$$\begin{aligned} 1 * f_1(i,j)) + 100 * f_2((i-1),(j-1)) &= 1 * g_1(i',j')) + 100 * g_2((i'-1),(j'-1)) \\ i + 100 * (j-1) &= j' + 100 * (i'-1) \\ i + 100 * j - 100 &= j' + 100 * i' - 100 \end{aligned}$$

This is precisely the equality used in the linearized array test.

We have shown that the λ-test subsumes two common inexact dependence tests. In the remainder of this paper, we consider implementations for the case $s = 2$, or doubly subscripted array references, and follow with the more general solution for arbitrary array references.

5 Double Subscripted Array References

For the case $s = 2$, or array references with double subscripts, our canonical example is

```
        DO I1 = L1,U1,N1
            DO I2 = L2,U2,N2
                ...
                    DO Il = Ll,Ul,Nl
S1:                     A(f1(I1,I2, ...,Il), f2(I1,I2, ...,Il))
                        ...
S2:                     A(g1(I1,I2, ...,Il), g2(I1,I2, ...,Il)
                    END
            END
        END
```

Our test for dependence is thus

$$\begin{aligned} f_1(\vec{i}) - g_1(\vec{j}) &= 0 \\ f_2(\vec{i}) - g_2(\vec{j}) &= 0 \end{aligned}$$

or, restated as a linear combination of the two equations,

$$\lambda_1(f_1(\vec{i}) - g_1(\vec{j})) + \lambda_2(f_2(\vec{i}) - g_2(\vec{j})) = 0 \tag{5}$$

We define a vector,

$$\vec{v} = (i_1, i_2, \ldots, i_l, j_1, j_2, \ldots, j_l)$$

combining both $\vec{i}$ and $\vec{j}$. We enumerate the terms of f_1 and g_1

$$\begin{aligned} f_1 &= a_1v_1 + a_2v_2 + \ldots + a_lv_l + c_1' \\ g_1 &= -a_{l+1}v_{l+1} + -a_{l+2}v_{l+2} + \ldots + -a_{l+l}v_{l+l} - c_{l+1}'' , \end{aligned}$$

where the vector of constants $\vec{a}$ is determined from the subscript expressions in the program. This allows us to recast the equality $f_1(\vec{i}) - g_1(\vec{j}) = 0$ as $\vec{a}v^T + c_1 = 0$ where $c_1 = c_1' + c_1''$ and v^T is the transpose of vector v. We define a similar constant vector $\vec{b}$ for the second equality, and recast (5) as

$$\begin{aligned} \lambda_1(\vec{a}v^T + c_1) + \lambda_2(\vec{b}v^T + c_2) &= 0 \\ \lambda_1\vec{a}v^T + \lambda_2\vec{b}v^T &= -(\lambda_1c_1 + \lambda_2c_2) \\ (\lambda_1\vec{a} + \lambda_2\vec{b})v^T &= -(\lambda_1c_1 + \lambda_2c_2) . \end{aligned}$$

This equality is isomorphic to (3) for double-subscripted arrays; we have simply used a more common notation for the dot-product. Later, we will discard explicit determination of v_{min} and v_{max}, but we initially follow the same goal as Li *et al*, and try to find constant vectors v_{min} and v_{max} such that

$$\forall_{v \in [L_1 \ldots U_1] \times [L_2 \ldots U_2] \times \ldots \times [L_l \ldots U_l]}, \quad (\lambda_1 \vec{a} + \lambda_2 \vec{b}) v_{min}^T \le (\lambda_1 \vec{a} + \lambda_2 \vec{b}) v^T \le (\lambda_1 \vec{a} + \lambda_2 \vec{b}) v_{max}^T .$$

We will assume a possible dependence exists if

$$(\lambda_1 \vec{a} + \lambda_2 \vec{b}) v_{min}^T \le -(\lambda_1 c_1 + \lambda_2 c_2) \le (\lambda_1 \vec{a} + \lambda_2 \vec{b}) v_{max}^T . \tag{6}$$

At first blush, it would not appear that we have reduced the complexity of our original problem; instead of our original two equations in $2l$ variables, we now have one equation in $2l+s$ variables. However, we can select values of λ_1 and λ_2 that simplify the equation, and allow us to compute v_{min} and v_{max}. If we can show independence for *any* λ-pair, we can show independence for the original problem.

Consider the expanded version of (6):

$$(\vec{a}\lambda_1 + \vec{b}\lambda_2) v^T = (a_1\lambda_1 + b_1\lambda_2) v_1 + (a_2\lambda_2 + b_2\lambda_2) v_2 + \ldots + (a_{2l}\lambda_{2l} + b_{2l}\lambda_{2l}) v_{2l} . \tag{7}$$

We can select arbitrary λ-pairs. However, in the original λ-test, Li *et al* select λ-pairs such that each pair reduces a single term in (7) to zero. In their terminology, this corresponds to selecting a λ-pair defining a Ψ-line in $\Re^s$. For example, the first term, $(a_1\lambda_1 + b_1\lambda_2)v_1$ can be zeroed when

$$(\lambda_1, \lambda_2) = \begin{cases} (1,0) & \text{if } a_1 = 0 \\ (0,1) & \text{if } b_1 = 0 \\ (b_1, -a_1) & \text{otherwise} . \end{cases}$$

This reduces the problem to $2l - 1$ variables because the first term, $\lambda_1 a_1 + \lambda_2 b_1$, becomes zero. The specific value of v_1 can be ignored because the product $(a_1\lambda_1 + b_1\lambda_2)v_1$ will always be zero. Selection of λ_1 and λ_2 also determines the sign of each constant term, $(a_i\lambda_1 + b_i\lambda_2)$. This, coupled with bounds for $\vec{v}$ allow us to determine v_{min} and v_{max}. Figure 2 shows our first algorithm for the double-subscripted version of the λ-test; this algorithm will be refined later. Determination of v_{min} and v_{max} differs with the dependence direction vector. In the

```
    DEPENDENCE? = True;
L1: For i = 1, 21 do
        determine $\lambda_1$, $\lambda_2$
        determine $v_{min}$, $v_{max}$
        if      $(\lambda_1\vec{a} + \lambda_2\vec{b})v_{min}^T > -(\lambda_1 c_1 + \lambda_2 c_2)$
            or      $(\lambda_1\vec{a} + \lambda_2\vec{b})v_{max}^T < -(\lambda_1 c_1 + \lambda_2 c_2)$
            then DEPENDENCE? = False;
            Exit L1;
```

Figure 2: First Algorithm for λ-test

next section, we motivate the solution using examples drawn from [LYZ89b]. Following that, we consider the problem of dependence with a particular dependence direction. In that section, we show that the explicit values of v_{min} and v_{max} are not needed.

5.1 Arbitrary Dependence

In this section, we examine the question of *arbitrary* dependence, or dependence with direction $\Psi = *$. To determine if S1 $\delta_{(*,*)}$ S2, we must find appropriate v_{min} and v_{max} such that

$$(\lambda_1\vec{a} + \lambda_2\vec{b})v_{min}^T \le (\lambda_1\vec{a} + \lambda_2\vec{b})v^T \le (\lambda_1\vec{a} + \lambda_2\vec{b})v_{max}^T$$

Because our function

$$(\lambda_1\vec{a} - \lambda_2\vec{b})v$$

is linear, we need only find v_i that minimize or maximize $(a_i\lambda_1 - b_i\lambda_2)v_i$ for all v_i.

We know $L_i \le v_i \le U_i$, where L_i and U_i are determined by the loop limits. If $(a_i\lambda_1 + b_i\lambda_2) \ge 0$, $v_i = L_i$ minimizes $(a_i\lambda_1 + b_i\lambda_2)v_i$, giving us the least positive value. If $(a_i\lambda_1 + b_i\lambda_2) < 0$, $v_i = U_i$ minimizes $(a_i\lambda_1 + b_i\lambda_2)v_i$, giving us the most negative value. The maximum values are the converse of this. If $(a_i\lambda_1 + b_i\lambda_2) \ge 0$, $v_i = U_i$ maximizes $(a_i\lambda_1 + b_i\lambda_2)v_i$, giving us the most positive value. If $(a_i\lambda_1 + b_i\lambda_2) < 0$, $v_i = L_i$ minimizes $(a_i\lambda_1 + b_i\lambda_2)v_i$, giving us the least negative value. Using the notation of Wolfe [Wol89], we can express this more succinctly as

$$v_{min,i} = (a_i\lambda_1 + b_i\lambda_2)^+ L_i + (a_i\lambda_1 + b_i\lambda_2)^- U_i \quad (8)$$

$$v_{max,i} = (a_i\lambda_1 + b_i\lambda_2)^+ U_i + (a_i\lambda_1 + b_i\lambda_2)^- L_i \quad (9)$$

where

$$r^+ = \begin{cases} 0 & r < 0 \\ r & r \geq 0 \end{cases} \qquad r^- = \begin{cases} r & r \leq 0 \\ 0 & r > 0 \end{cases}$$

As mentioned, when $(a_i\lambda_1 + b_i\lambda_2) = 0$, we do not care what value is chosen for v_i, because the product $(a_i\lambda_1 + b_i\lambda_2)v_i = 0$ will be zero.

Example 1 : Consider S1 and S2:

```
          DO I=1,50
             DO J=2,50
S1:             X(2I + 3J + 50, 3I + J + 49) = ...
S2:             ... = X(I - J + 51, 2I - J + 48)
             ENDDO
          ENDDO
```

Examination yields the constants

$$\vec{a} = (2, 3, -1, 1) \qquad c_1 = -1$$

$$\vec{b} = (3, 1, -2, 1) \qquad c_2 = 1\,.$$

Recall that $\vec{a}$ corresponds to the constants in the first subscript and $\vec{b}$ to the constants in the second subscript. From this, we derive four pairs of λ constants:

$$\Lambda = \{(3, -2), (1, -3), (-2, 1), (1, -1)\}$$

In the terminology of Li *et al*, these constants would determine four Ψ-lines in $\Re^s$. We initiate the test using the first pair, $\lambda_1 = 3, \lambda_2 = -2$. We compute the constant terms in our expression with this λ_1 and λ_2:

$$\begin{aligned} (\lambda_1\vec{a} + \lambda_2\vec{b})v^T &= (2*3 + 3*-2)v_1 + (3*3 + 1*-2)v_2 \\ &\quad +(-1*3 + -2*-2)v_3 + (1*3 + 1*-2)v_4 \\ &= (0, 7, 1, 1)v^T\,. \end{aligned}$$

Using the signs of the constant terms, we apply (8) and (9) to get

$$\begin{aligned} v_{min} &= (x, L_2, L_3, L_4) \\ &= (x, 2, 1, 2) \\ v_{max} &= (x, U_2, U_3, U_4) \\ &= (x, 50, 50, 50) \end{aligned}$$

where x represents a "don't care" condition. We now apply our dependence test. There is a possible dependence if:

$$\begin{array}{rcl} (\lambda_1\vec{a} + \lambda_2\vec{b})v_{min}^T \leq & -(\lambda_1 c_1 + \lambda_2 c_2) & \leq (\lambda_1\vec{a} + \lambda_2\vec{b})v_{max}^T \\ (0,7,1,1)(x,2,1,2)^T \leq & -(3 * -1 + -2 * 1) & \leq (0,7,1,1)(x,50,50,50)^T \\ 17 \leq & 5 & \leq 450 \end{array}$$

Because $17 > 5$, there can be no dependence in this loop, and we need not check further. By comparison, the Banerjee-Wolfe subscript-by-subscript test would indicate a dependence, requiring further evaluation.

Example 2: Consider a slight modification to the previous example.

```
        DO I=1,50
           DO J=2,50
S1:           X(2I - 3J + 50)(3I - J + 49) = ...
S2:           ... = X(I + J + 51)(2I +J + 48)
           ENDDO
        ENDDO
```

The vectors of constants are

$$\begin{array}{ll} \vec{a} = (2,-3,-1,-1) & c_1 = -1 \\ \vec{b} = (3,-1,-2,-1) & c_2 = 1 \end{array}$$

The λ-pairs are thus

$$\Lambda = \{(3,-2),(-1,3),(-2,1),(-1,1)\}.$$

When we test with the first pair $\lambda_1 = 3, \lambda_2 = -2$, we get

$$\begin{aligned} (\lambda_1\vec{a} + \lambda_2\vec{b})v^T &= (2 * 3 + 3 * -2)v_1 + (-3 * 3 + -1 * -2)v_2 \\ &\quad +(-1 * 3 + -2 * -2)v_3 + (-1 * 3 + -1 * -2)v_4 \\ &= (0,-7,1,-1)v^T \end{aligned}$$

Again, using the signs of the constant terms, we apply (8) and (9) to get

$$\begin{aligned} v_{min} &= (x, U_2, L_3, U_4) \\ &= (x, 50, 1, 50) \\ v_{max} &= (x, L_2, U_3, L_4) \\ &= (x, 2, 50, 2) \end{aligned}$$

Testing the inequality, we have

$$\begin{array}{ccc} (0,-7,1,-1)(x,50,1,50)^T \leq & -(3*-1+-2*1) & \leq (0,-7,1,-1)(x,2,50,2)^T \\ -399 \leq & 5 & \leq 34\,, \end{array}$$

Thus, there is a possible dependence. Subsequent λ-pairs give us the following bounds:

(λ_1, λ_2)	$(\lambda_1\vec{a} + \lambda_2\vec{b})$	$(\lambda_1\vec{a} + \lambda_2\vec{b})v_{min}^T$	$-(\lambda_1 c_1 + \lambda_2 c_2)$	$(\lambda_1\vec{a} + \lambda_2\vec{b})v_{max}^T$	Dependence?
$(3,-2)$	$(0,-7,1,-1)$	-399	5	34	Maybe
$(-1,3)$	$(7,0,-5,-2)$	-343	-4	341	Maybe
$(-2,1)$	$(-1,5,0,1)$	-38	-3	299	Maybe
$(-1,1)$	$(1,2,-1,0)$	-45	-2	149	Maybe

Because independence is not shown by any λ-pair, we must assume a true dependence exists for these array references. Using this example, we illustrate the mechanism for testing data dependence for a particular dependence direction vector.

5.2 Dependence With Direction Vectors

Although there may be a possible data dependence between two statements, the combination of the data dependence and execution direction vectors may indicate no actual dependence. To check if this is the case, we generate the data dependence direction vector for the two statements; this is done experimentally by testing for dependence in each of three dependence directions.

Dependence direction vectors establish constraints on related instances of iteration variables. For example, in the notation of the previous section, the dependence direction S1 $\delta_{(<,=)}$ S2 states that execution of S1 must precede execution of S2 when $v_1 < v_3$ and $v_2 = v_4$. We translate these constraints into modifications of the computation of v_{min} and v_{max}.

In [LYZ89b, LYZ89a], the corresponding dependence direction and the sign of the term $(a_i\lambda_1 + b_i\lambda_2)$ are used to compute $v_{min,i}$ and $v_{max,i}$, as was done in the previous section for arbitrary dependence. This algorithm is involved, and contributes to the apparent complexity of the λ-test.

Can we improve on this method? Happily, the answer is 'yes'. Rather than initially computing individual terms of $v_{min,i}$ and $v_{max,i}$, consider the actual goal of these computations. To determine the lower bound for the dependence test, we want to compute

$$\sum_{1\leq i\leq 2l} (a_i\lambda_1 + b_i\lambda_2)v_{min,i}\,.$$

Dependence directions for loop i involve terms i and $l+i$; rather than solving for v_i and v_{l+i} with a particular dependence relation and summing

$$(a_i\lambda_1 + b_i\lambda_2)v_{min,i} + (a_{l+i}\lambda_1 + b_{l+i}\lambda_2)v_{min,l+i}\,,$$

we can solve for both terms simultaneously to find a lower bound for the sum. For l nested loops, we compute l pairs of bounds, $LB_1,\ldots,LB_l$ and $UB_1,\ldots,UB_l$ where

$$\begin{aligned} LB_i &= (a_i\lambda_1 + b_i\lambda_2)v_{min,i} + (a_{l+i}\lambda_1 + b_{l+i}\lambda_2)v_{min,l+i} \\ UB_i &= (a_i\lambda_1 + b_i\lambda_2)v_{max,i} + (a_{l+i}\lambda_1 + b_{l+i}\lambda_2)v_{max,l+i} \end{aligned}$$

This transforms our dependence test (6) into

$$\sum_{1\leq i\leq l} LB_i \leq -(\lambda_1 c_1 + \lambda_2 c_2) \leq \sum_{1\leq i\leq l} UB_i\,.$$

Computing LB_i and UB_i is an established procedure; [Wol89] gives solutions to the inequalities

$$LB \leq (Ai - Bj) \leq UB$$

where A and B are constants and i and j are instances of iteration variables subject to dependence relations. These equations are shown in Table 1. Both i and j refer to specific iteration instances from the same loop; thus, both i and j have the same increment and upper and lower limits,

$$L \leq i,j \leq U\,.$$

$$\begin{array}{llcl}
\Psi \text{ is } * : & LB^{=}(A,B) & = & (A^{-}-B^{+})(U-L)+(A-B)L \\
& UB^{=}(A,B) & = & (A^{+}-B^{-})(U-L)+(A-B)L
\end{array}$$

$$\begin{array}{llcl}
\Psi \text{ is } < : & LB^{<}(A,B) & = & (A^{-}-B)^{-}(U-L-N)+(A-B)L-BN \\
& UB^{<}(A,B) & = & (A^{+}-B)^{+}(U-L-N)+(A-B)L-BN
\end{array}$$

$$\begin{array}{llcl}
\Psi \text{ is } = : & LB^{=}(A,B) & = & (A-B)^{-}(U-L)+(A-B)L \\
& UB^{=}(A,B) & = & (A-B)^{+}(U-L)+(A-B)L
\end{array}$$

$$\begin{array}{llcl}
\Psi \text{ is } > : & LB^{>}(A,B) & = & (A-B^{+})^{-}(U-L-N)+(A-B)L-AN \\
& UB^{>}(A,B) & = & (A-B^{-})^{+}(U-L-N)+(A-B)L-AN
\end{array}$$

Table 1: Banerjee-Wolfe Inequalities for Direction Ψ

The Banerjee-Wolfe inequalities compute

$$LB \leq (Ai - Bj) \leq UB$$

while we are interested in

$$LB_i \leq (a_i\lambda_1 + b_i\lambda_2)v_i + (a_{l+i}\lambda_1 + b_{l+i}\lambda_2)v_{l+i} \leq UB_i\,.$$

Thus, to compute the lower and upper bounds for a given direction, we substitute the appropriate quantities and negate the second term. For example, to compute bounds for the $<$ direction, we use

$$\begin{array}{lcl}
LB_i^{<} & = & LB^{<}(a_i\lambda_1 + b_i\lambda_2,\ -(a_j\lambda_1 + b_{l+i}\lambda_2)) \\
UB_i^{<} & = & UB^{<}(a_i\lambda_1 + b_i\lambda_2,\ -(a_j\lambda_1 + b_{l+i}\lambda_2))
\end{array}$$

where $LB^{<}$ and $UB^{<}$ are given in Table 1. Expansion of these equations followed by case-by-case comparison to the equations presented in [LYZ89b, LYZ89a] demonstrates the equivalence of this method and the λ-test. Because LB_i and UB_i determine bounds for both v_i and v_{i+l}, we expect that our formulation of the λ-test would be more efficient than that of Li *et al.* Figure 3 generalizes the algorithm of Figure 2 to include direction vector constraints for double subscripted loops.

The choice of λ-pairs is, as previously stated, arbitrary. Li *et al* choose λ-pairs that draw $(\vec{a}\lambda_1 + \vec{b}\lambda_2)v^T$ closer to zero, represented by Ψ-lines in their formulation. Li *et al* also use

```
         DEPENDENCE? = True;
         Let dep[1..l] = dependence direction for loops
         Let $\Lambda$ be the set of trial $\lambda_1$, $\lambda_2$ pairs
    L1:  For t = 1, $|\Lambda|$ do
             Let $\lambda_1$, $\lambda_2$ = $\Lambda[t]$
             LB = 0; UB = 0;
             For i = 1, l do
                 LB = LB + LB$^{\texttt{dep[i]}}(a_i\lambda_1 + b_i\lambda_2,\ -(a_{l+i}\lambda_1 + b_{l+i}\lambda_2))$
                 UB =UB + UB$^{\texttt{dep[i]}}(a_i\lambda_1 + b_i\lambda_2,\ -(a_{l+i}\lambda_1 + b_{l+i}\lambda_2))$
             End
             Let target = $-(\lambda_1 c_1 + \lambda_2 c_2)$
             If LB > target or UB < target
                 then DEPENDENCE? = FALSE; Exit L1;
         End
```

Figure 3: Second Algorithm for λ-test

λ-pairs defined by Φ-lines in cases involving dependence direction vectors. Φ-lines correspond to λ-pairs that zero multiple terms under $=$ dependence. Recall that loop i involves the i^{th} and $(i+l)^{th}$ iteration variables. For $=$ dependence in loop i, we can zero two terms using

$$(\lambda_1, \lambda_2) = (b_i + b_{l+i}, -(a_i + a_{l+i}))\,.$$

The selection of this λ-pair is, as mentioned, arbitrary.

Example 3: We now have effective procedures for selecting λ-pairs and computing lower and upper bounds. We conclude this section by computing the dependence direction vectors for S1 and S2 of Example Two. We use the Burke-Cytron [BC86] dependence hierarchy, shown in Figure 4, to limit the number of tests needed.

If independence can be shown for the root of a tree or subtree, all direction vectors in the subtree are also independent. Example Two has shown that some form of dependence exists between S1 and S2; therefore, we attempt to refine the direction for the outer loop, first in the $<$ direction, followed by $=$ and $>$.

Recall that we had the vectors of constants

$$\vec{a} = (2, -3, -1, -1) \qquad c_1 = -1$$

$$\vec{b} = (3, -1, -2, -1) \qquad c_2 = 1$$

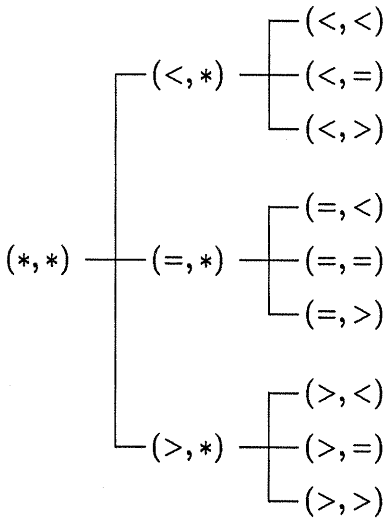

Figure 4: Burke-Cytron Dependence Hierarchy

and had selected the following λ-pairs:

$$(\lambda_1, \lambda_2) = \{(3,-2), (1,3), (-2,1), (-1,1)\}.$$

In cases where the dependence direction, Ψ, is $=$, we may wish to test additional λ-pairs. An $=$ dependence direction in the first loop implies that $v_1 = v_3$, giving us

$$(a_1\lambda_1 + b_1\lambda_2)v_1 + (a_3\lambda_1 + b_3\lambda_2)v_3 = ((a_1 + a_3)\lambda_1 + (b_1 + b_3)\lambda_2)v_1 .$$

This suggests the λ-pair $(1, -1)$ for the $=$ direction in the first loop, and by similar argument, $(-2, 4)$ in the second loop. These λ-pairs corresponds to the Φ-lines of Li *et al.* We can eliminate linear combinations of λ-pairs, disposing of $(1, -1)$, because it duplicates an existing entry in Λ.

The first branch of the dependence hierarchy indicates a test for S1 $\delta_{(<,*)}$ S2. Our test is then

$$LB_1^< + LB_2^* \le -(\lambda_1 c_1 + \lambda_2 c_2) \le UB_1^< + UB_2^* .$$

Enumerating the bounds for each λ-pair gives us the following tests:

(λ_1, λ_2)	$LB_1^< + LB_2^*$	$-(\lambda_1 c_1 + \lambda_2 c_2)$	$UB_1^< + UB_2^*$	Dependence?
$(3,-2)$	$2 - 400$	5	$50 - 16$	Maybe
$(-1,3)$	$-243 - 100$	-4	$93 - 4$	Maybe
$(-2,1)$	$-49 + 12$	-3	$-1 + 300$	Maybe
$(-1,1)$	$-49 + 4$	-2	$-1 + 100$	Maybe

Because we did not find a λ-pair indicating independence, we assume dependence with S1 $\delta_{(<,*)}$ S2. Similar computations determine that S1 $\delta_{(<,<)}$ S2, S1 $\delta_{(<,=)}$ S2 and S1 $\delta_{(<,>)}$ S2 are possible dependence direction vectors.

Similarly, for = in the first loop, we find:

(λ_1, λ_2)	$LB_1^= + LB_2^*$	$-(\lambda_1 c_1 + \lambda_2 c_2)$	$UB_1^= + UB_2^*$	Dependence?
$(3,-2)$	$-400 + 50$	5	$50 - 16$	Maybe
$(-1,3)$	$4 - 500$	-2	$200 - 20$	Maybe
$(-2,1)$	$-50 + 12$	-3	$-1 + 300$	Maybe
$(-1,1)$	$0 + 4$	-2	$0 + 100$	No
$(-2,4)$	$2 -96$	-6	$100 + 96$	Maybe

The λ-pair (-1,1) indicates there is independence between S1 and S2 with an = direction in the first loop; thus, we need not examine the subtree below $(=,*)$. The same λ-pair also indicates there is no dependence in the > direction; thus, the final dependence direction vector computed by the λ-test is

$$\begin{array}{lll} \text{S1} & \delta_{(<,<)} & \text{S2} \\ \text{S1} & \delta_{(<,=)} & \text{S2} \\ \text{S1} & \delta_{(<,>)} & \text{S2} \end{array}$$

By comparison, the Banerjee-Wolfe subscript-by-subscript test, or, equivalently, the λ-test with $\Lambda = \{(1,0),(0,1)\}$, shows the following dependencies

$$\begin{array}{lll} \text{S1} & \delta_{(<,<)} & \text{S2} \\ \text{S1} & \delta_{(<,=)} & \text{S2} \\ \text{S1} & \delta_{(<,>)} & \text{S2} \\ \text{S1} & \delta_{(=,<)} & \text{S2} \\ \text{S1} & \delta_{(=,=)} & \text{S2} \\ \text{S1} & \delta_{(=,>)} & \text{S2} \\ \text{S1} & \delta_{(>,<)} & \text{S2} \\ \text{S1} & \delta_{(>,=)} & \text{S2} \\ \text{S1} & \delta_{(>,>)} & \text{S2} \end{array}$$

In other words, the Banerjee-Wolfe test finds dependence in every direction in every loop.

6 Multiple Subscript Array References

With the notation presented in §5.2, we see that array references with multiple subscripts can be treated in a similar manner. However, rather than selecting λ-pairs, we select a λ-tuple with

s members. As before, selection of the λ-tuple is arbitrary because the λ-test is conservative. Consider the example with l loops and s subscripts:

```
        DO I1 = L1,U1,N1
            DO I2 = L2,U2,N2
                ...
                    DO Il = Ll,Ul,Nl
  S1:                   A(f1(I1,I2, ...,Il), f2(I1,I2, ...,Il), ..., fs(I1,I2, ...,Il))
                        ...
  S2:                   A(g1(I1,I2, ...,Il), g2(I1,I2, ...,Il), ..., gs(I1,I2, ...,Il))
                    END
            END
        END
```

In general, the dependence test to be solved is

$$\lambda_1(f_1(\vec{i}) - g_1(\vec{j})) + \lambda_2(f_2(\vec{i}) - g_2(\vec{j})) + \ldots + \lambda_s(f_s(\vec{i}) - g_s(\vec{j})) = 0\,.$$

To treat the general case, we introduce new notation for the terms of each function, f_i or g_i. We let u_i be the vector of elements corresponding to the function pair multiplied by λ_i. That is, we recast the previous equation as

$$\begin{aligned}
0 &= \lambda_1(f_1(\vec{i}) - g_1(\vec{j})) + \lambda_2(f_2(\vec{i}) - g_2(\vec{j})) + \ldots + \lambda_s(f_s(\vec{i}) - g_s(\vec{j})) \\
0 &= \lambda_1(\vec{u}_1 v^T + c_1) + \lambda_2(\vec{u}_2 v^T + c_2) + \ldots + \lambda_s(\vec{u}_s v^T + c_s) \\
0 &= \lambda_1(u_{1,1}v_1 + u_{1,2}v_2 + \ldots + u_{1,2l}v_{2l} + c_1) \\
&\quad +\lambda_1(u_{2,1}v_1 + u_{2,2}v_2 + \ldots + u_{2,2l}v_{2l} + c_2) \\
&\quad +\ldots \\
&\quad +\lambda_s(u_{s,1}v_1 + u_{s,2}v_2 + \ldots + u_{s,2l}v_{sl} + c_s) \\
-(\sum_{1\le p\le s} \lambda_p c_p) &= \sum_{1\le m\le 2l} \left(\sum_{1\le n\le s} \lambda_n u_{m,n} \right) v_m\,.
\end{aligned}$$

Recall that our actual goal is to determine bounds on the sum $-(\sum_{1\le p\le s} \lambda_p c_p)$ based on loop directions. Let D_i be the l-vector of loop directions. The general form for our dependence test is then

$$\sum_{1\le i\le l} LB_i^{D_i} \le -(\sum_{1\le p\le s} \lambda_p c_p) \le \sum_{1\le i\le l} UB_i^{D_i}\,.$$

where

$$
\begin{aligned}
LB_i^{D_i} &= LB^{D_i}\left(\sum_{1\le k\le s}\lambda_k u_{i,k}\ ,\ -(\sum_{1\le k\le s}\lambda_k u_{l+i,k})\right)\\
UB_i^{D_i} &= UB^{D_i}\left(\sum_{1\le k\le s}\lambda_k u_{i,k}\ ,\ -(\sum_{1\le k\le s}\lambda_k u_{l+i,k})\right)
\end{aligned}
$$

This test is repeated for each λ-tuple. Selection of the λ-tuple is problematic; unlike the case of $s = 2$, there is no simple method for selecting a λ-tuple that matches the criteria of [LYZ89b, LYZ89a], where a particular term is set to zero. As mentioned previously, the Banerjee-Wolfe subscript-by-subscript test selects the λ-tuples

$$
\begin{aligned}
\Lambda &= \{(1,0,\ldots,0,0),\ (0,1,\ldots,0,0),\\
&\quad \ldots,\ (0,0,\ldots,1,0),\ (0,0,\ldots,0,1)\}.
\end{aligned}
$$

Obviously, additional λ-tuples with single elements are pointless; they are linear variants of the subscript-by-subscript test. We can compute pair-wise λ-tuples that zero out specific terms; for example a tuple $(3,-2,0)$ would zero out a term such as $(2v_1 + 3v_2 + 49v_3)$. However, it is not clear that the practice of zeroing terms leads to a more exact test. In the light of this formulation of the λ-test, the criteria for zeroing a term appears rather arbitrary. It may simply be a hueristic to limit the number of λ-tuples tested; further study on this aspect of the λ-test is needed. In a production compiler, we may simply wish to define an upper limit on the number of trials or the time taken for dependence testing, and generate λ-tuples until the limit is reached.

Example 4 :

Again, we consider a slight variation on Example One.

```
          DO I=1,50
             DO J=2,50
                DO K=3,50,2
      S1:          X(2I + 3J + K + 50)(3I + J + 2K + 49)(K-1) = ...
      S2:          ... = X(I - J + 51)(2I - J -K + 48)(K+1)
             ENDDO
          ENDDO
```

We have the following constants:

$$\vec{u}_1 = (2,3,1,-1,1,0) \qquad c_1 = 50 - 51 = -1$$

$$\vec{u}_2 = (3,1,2,-2,1,1) \qquad c_2 = 49 - 48 = 1$$

$$\vec{u}_3 = (0,0,1,0,0,-1) \qquad c_3 = -1 - 0 = -2$$

We select an arbitrary set of λ-tuples. Our set includes the tuples used in the subscript-by-subscript test, as well as the set of tuples used in Example One with an additional zero term.

$$\begin{aligned}\Lambda \;=\; &\{(1,0,0),\ (0,1,0),\ (0,0,1)\\ &(3,-2,0),\ (1,3,0),\ (1,3,0)\\ &(-2,1,0),\ (-1,1,0),\ (-2,4,0)\}\end{aligned}$$

$(\lambda_1, \lambda_2, \lambda_3)$	$LB_1^* + LB_2^* + LB_3^*$	$-(\lambda_1 c_1 + \lambda_2 c_2 + \lambda_3 c_3)$	$UB_1^* + UB_2^* + UB_3^*$	Dependence?
$(1,0,0)$	$-48+8+3$	1	$99 + 200 + 50$	Maybe
$(0,1,0)$	$-97+4+9$	-1	$148+100+150$	Maybe
$(0,0,1)$	$0+0+-47$	1	$0+0+47$	Maybe
$(3,-2,0)$	$1+16+-150$	5	$50+400+-9$	Maybe
$(1,3,0)$	$-339+20+30$	-2	$543+500+500$	Maybe
$(-2,1,0)$	$-50+-300+3$	-3	$-1+-12+50$	Maybe
$(-1,1,0)$	$-49+-100+6$	-2	$49+-4+100$	Maybe
$(-2,4,0)$	$-292+-96+30$	-6	$394+96+500$	Maybe

Again, because no λ-tuple shows independence, we must assume dependence. Refining the Burke-Cytron dependence hierarchy, we eventually reach the test for $(<,<,<)$ dependence.

$(\lambda_1, \lambda_2, \lambda_3)$	$LB_1^< + LB_2^< + LB_3^<$	$-(\lambda_1 c_1 + \lambda_2 c_2 + \lambda_3 c_3)$	$UB_1^< + UB_2^< + UB_3^<$	Dependence?
$(1,0,0)$	$-48+9+3$	1	$48+197+49$	Maybe
$(0,1,0)$	$-97+5+9$	1	$47+99+147$	Maybe
$(0,0,1)$	$0+0+-47$	2	$0+0+-1$	No
$(3,-2,0)$	$2+17+-149$	5	$50+393-11$	Maybe
$(1,3,0)$	$-399+24+33$	-2	$189+494+493$	Maybe
$(-2,1,0)$	$-49-395+4$	-3	$-1+-13+50$	Maybe
$(-1,1,0)$	$-49-98+7$	-2	$-1-4+99$	Maybe
$(-2,4,0)$	$-292+3+34$	-6	$92+96+494$	Maybe

Note that independence is indicated by the λ-tuples defined by the subscript-by-subscript test. The remaining λ-tuples are ineffective because they do not include the third subscript. This indicates that the λ-test is a not a panacea, and that proper selection of λ-tuples is important. A suitable algorithm is needed, and is the subject of further research.

7 Summary

We have shown that the λ-test is a straight-forward, yet effective, extension of the classical Banerjee-Wolfe subscript-by-subscript test, and that it also subsumes the linearized array test. We have shown that the additional overhead attributed to the test arises from repeated application of the Banerjee-Wolfe inequalities, and that recasting the test can produce a more efficient implementation.

In light of our formulation of the λ-test, an appropriate method of selecting of the λ-tuples is open to conjecture. While a particular method for selecting λ-tuples, such as the one presented in [LYZ89b, LYZ89a], may generate suitable combinations, further study is needed to determine if this is an actual algorithm or simply a useful hueristic.

The computations for the Banerjee-Wolfe inequalities include many redundant terms. It will be interesting to see if our formulation of the λ-test leads to faster execution of the λ-test in the context of dependence analysis compilers such as Parafrase. Furthermore, it may be possible to use the λ-test in the presence of unknown quantities, as done for the subscript-by-subscript test in [Wol89].

Acknowledgements: I gratefully acknowledge the technical and editorial comments of Michael Wolfe, and the time and patience of the students of CSci7000.

References

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwar Academic Publishers, 1988.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proc. of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[BCKT79] Utpal Banerjee, Steve C. Chen, David J. Kuck, and R. A. Towle. Time and parallel processor bounds for fortran-like loops. *IEEE Transactions on Computers*, C-28(9):660–670, September 1979.

[LYZ89a] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. Data dependence analysis on multi-dimensional array references. In *Conference Proceedings of the 3rd International Conf. on Supercomputing*, pages 215–224, June 1989.

[LYZ89b] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. An efficient data dependence analaysis for parallelizing compilers. Technical Report 852, Center for Supercomputing Research and Development, University of Illinois, 104 S. Wright Street, Urbana, Illinois 61801, May 1989.

[SLY89] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empricial study on array subscripts and data dependence. Technical Report 842, Center for Supercomputing Research and Development, University of Illinois, 104 S. Wright Street, Urbana, Illinois 61801, May 1989.

[Wol82] Michael Wolfe. Optimizing supercompilers for supercomputers. Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 W. Springfield, Urbana, Il, October 1982.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

A An Implementation of the λ-test

The following program is an implementation of λ-test in the Scheme programming language. Sample executions of the program, using the examples from the report, are also shown.

```
;;;;;;; file: lambda.scm
;;;;;;;
;;;;;;;
;;;;;;;  The Lambda-Test Revisited -- A simple implementation of the lambda
;;;;;;; implemented in Scheme.
;;;;;;;
;;
;;  Definitions for the Banerjee-Wolfe Inequality Tests
;;
(define (r+ x)
  "Positive portion of a number"
  (if (> x 0) x 0))
(define (r- x)
  "Negative portion of a number"
  (if (< x 0) x 0))
;;
;;  Banerjee-Wolfe bounds equations for
;;
;;    LB <= (Ai - Bj) <= UB
;;
;;    where L <= i,j <= U and loop stride is N.
;;    Each name is followed by the dependence direction.
;;
(define (LB* A B U L N)
  (+ (* (- (r- A) (r+ B)) (- U L )) (* (- A B) L)))
(define (UB* A B U L N)
  (+ (* (- (r+ A) (r- B)) (- U L)) (* (- A B) L )))
(define (LB= A B U L N)
  (+ (* (r- (- A B)) (- U L )) (* (- A B) L)))
(define (UB= A B U L N)
  (+ (* (r+ (- A B)) (- U L )) (* (- A B) L)))
(define (LB< A B U L N)
  (- (+ (* (r- (- (r- A) B)) (- U L N) ) (* (- A B) L)) (* B N)))
(define (UB< A B U L N)
  (- (+ (* (r+ (- (r+ A) B)) (- U L N) ) (* (- A B) L)) (* B N)))
(define (LB> A B U L N)
  (+ (* (r- (- A (r+ B))) (- U L N) ) (* (- A B) L) (* A N)))
(define (UB> A B U L N)
  (+ (* (r+ (- A (r- B))) (- U L N) ) (* (- A B) L) (* A N)))

;;
;;  Helper functions
;;
(define (null? x)
  "Is a list null?"
  (if x nil t))
(define (el index array)
  "Reference an array element, indexed from 1"
  (nth (- index 1) array))

;;
;; Actual computation functions..
;;
(define (LambdaSum i Lvec ConstVecList)
  "Compute Sum a_i lambda_1 + b_i lambda_2 + c_i lambda_3 + ...."
  (if (null? Lvec)
      0
      (+
       (* (el i (car ConstVecList)) (car Lvec))
       (LambdaSum i (cdr Lvec) (cdr ConstVecList)))))

(define (LB op i j LambdaList ConstVecList U L N)
  "Compute the lower bound based on the direction kind (single)"
  (let (
        (ai (LambdaSum i LambdaList ConstVecList))
        (aj (- 0 (LambdaSum j LambdaList ConstVecList)))
        (li (el i L))
        (ui (el i U))
        (ni (el i N)))
    (cond
     ((eq? op '*) (LB* ai aj ui li ni))
     ((eq? op '<) (LB< ai aj ui li ni))
     ((eq? op '=) (LB= ai aj ui li ni))
     ((eq? op '>) (LB> ai aj ui li ni))
     (t (error "oops")))))

(define (UB op i j LambdaList ConstVecList U L N)
  "Compute the upper bound based on the direction kind (single)"
  (let (
        (ai (LambdaSum i LambdaList ConstVecList))
        (aj (- 0 (LambdaSum j LambdaList ConstVecList)))
        (li (el i L))
        (ui (el i U))
        (ni (el i N)))
    (cond
     ((eq? op '*) (UB* ai aj ui li ni))
     ((eq? op '<) (UB< ai aj ui li ni))
     ((eq? op '=) (UB= ai aj ui li ni))
     ((eq? op '>) (UB> ai aj ui li ni))
     (t (error "oops")))))

(define (dot-product A B)
  "Compute A[1] * B[1] + A[2] * B[2] + ....+ A[n] B[n]"
  (if (null? A) 0
      (+ (* (car A) (car B))
         (dot-product (cdr A) (cdr B)))))

(define (bound-list op DirList i lth LambdaList ConstVecList U L N)
  "Build a list of the lower bound values for a single loop. Peel of the
```

```
test direction vector for this loop and identify elements to be
test (i and i+number-of-loops). "

  (if (null? DirList) nil
      (cons
       (apply op
              (list (car DirList) i (+ i lth) LambdaList ConstVecList U L N))
       (bound-list op (cdr DirList) (1+ i) lth LambdaList ConstVecList U L N))))


(define (single-test DIR LambdaList ConstVecList TargetList U L N)
  "Returns a results of a single test as a list.
First element is the lambda-tuple used.

Second element is a list of triples, Sum_LB C Sum_UB. Dependence is possible
if LB <= C <= UB.

The third element is a list of components used in calculating the lower bound.
The fourth element is a list of components used in calculating the upper bound.
"
  (letrec
      (
       (lth (length DIR))
       (lb-list (bound-list LB DIR 1 lth LambdaList ConstVecList U L N))
       (lb (apply + lb-list))
       (target (- 0 (dot-product TargetList LambdaList)))
       (ub-list (bound-list UB DIR 1 lth LambdaList ConstVecList U L N))
       (ub (apply + ub-list))
       )
    (list (list
           LambdaList
           (if (or (< target lb) (> target ub)) 'Independent 'Dependent)
           lb target ub)
          lb-list ub-list)))

(define (multiple-tests DirList LambdaListList ConstVecList TargetList U L N)
  "Run tests with a single direction vector for multiple lambda-tuples"
  (let ((l nil))
    (if (not (= (length DirList) (length (car LambdaListList))))
        (print "You must specify a different direction vector"))
    (while (not (null? LambdaListList))
           (let
               ((LambdaList (car LambdaListList)))
             (set! LambdaListList (cdr LambdaListList))
             (set! l (append l (list (single-test DirList LambdaList
                                                  ConstVecList TargetList U L N))))))
    l))

(define (print-first-level l)
  "Print a double level list with one line per inner term. Used
to display information about a dependence test"
  (while (not (null? l)) (print (car l)) (set! l (cdr l))))
  )

;;;;
;;;;  Examples from the report.
;;;;


;;
;;
;;Example 1
;;
(define (example-1 DirList)
  "First example"
  (let
      ((ConstVecList
        (list
         '( 2 3 -1 1)
         '( 3 1 -2 1)))
       (TargetList '( -1 1))
       (L '( 1 2 1 2))
       (U '( 50 50 50 50))
       (N '( 1 1 1 1 ))
       (LambdaListList
        (list '(3 -2) '(1 -3) '(-2 1) '(-1 -1) '(1 0) '(0 1) )))
    (print-first-level
     (multiple-tests DirList LambdaListList ConstVecList TargetList U L N))))


(define (example-2 DirList)
  (letrec
      (
       (ConstVecList
        (list '( 2 -3 -1 -1)
              '( 3 -1 -2 -1)))

       (TargetList
        (list -1 1))

       (L '( 1 2 1 2))
       (U '( 50 50 50 50))
       (N '( 1 1 1 1 ))
       (LambdaListList
        (list
         '(3 -2) '(-1 3) '(-2 1) '(-1 1)
         '(-2 4) '(1 0) '(0 1) )))
    (print-first-level
     (multiple-tests DirList LambdaListList ConstVecList TargetList U L N))))

(define (example-4 DirList)
  (letrec
      (
       (ConstVecList
  (list '( 2 3 1 -1 1 0)
        '( 3 1 2 -2 1 1)
        '( 0 0 1 0 0 -1) ))
 (TargetList (list -1 1 -1))
 (L '( 1 2 3 1 2 3))
 (U '( 50 50 50 50 50 50))
 (N '( 1 1 1 1 1 1 ))
 (LambdaListList
    (list
     '(1 0 0) '(0 1 1) '(0 0 1) '(1 1 1)
     '(3 -2 0) '(1 3 0) '(-2 1 0) '(-1 1 0) '(-2 4 0))))
(print-first-level
 (multiple-tests DirList LambdaListList
                 ConstVecList TargetList U L N))))
```

The following is a sample execution of for specific dependence directions. Each test prints a list. The first element is a list of the λ-tuple used, dependence status, $\sum_{1 \leq i \leq l} LB_i$ and $\sum_{1 \leq i \leq l} UB_i$. The two following lists contain individual elements of LB_i and UB_i.

```
fools' lisp 1.0 Wed Oct 18 12:51:45 MDT 1989
> (load 'lambda.scm)
#t
> (example-1 '( * * ))
(((3 -2) Independent 17 5 450)
  (1 16)
  (50 400))
(((1 -3) Dependent -445 4 239)
  (-345 -100)
  (243 -4))
(((-2 1) Independent -350 -3 -13)
  (-50 -300)
  (-1 -12))
(((-1 -1) Dependent -547 0 133)
  (-247 -300)
  (145 -12))
(((1 0) Dependent -40 1 299)
  (-48 8)
  (99 200))
(((0 1) Dependent -93 -1 248)
  (-97 4)
  (148 100))
nil
> (example-2 '(* *))
(((3 -2) Dependent -399 5 34)
  (1 -400)
  (50 -16))
(((-1 3) Dependent -343 -4 341)
  (-243 -100)
  (345 -4))
(((-2 1) Dependent -38 -3 299)
  (-50 12)
  (-1 300))
(((-1 1) Dependent -45 -2 149)
  (-49 4)
  (49 100))
(((-2 4) Dependent -388 -6 490)
  (-292 -96)
  (394 96))
(((1 0) Dependent -248 1 91)
  (-48 -200)
  (99 -8))
(((0 1) Dependent -197 -1 144)
  (-97 -100)
  (148 -4))
nil
> (example-2 '(< *))
(((3 -2) Dependent -398 5 34)
  (2 -400)
  (50 -16))
(((-1 3) Dependent -343 -4 89)
  (-243 -100)
  (93 -4))
(((-2 1) Dependent -37 -3 299)
  (-49 12)
  (-1 300))
(((-1 1) Dependent -45 -2 99)
  (-49 4)
  (-1 100))
(((-2 4) Dependent -388 -6 188)
  (-292 -96)
  (92 96))
(((1 0) Dependent -248 1 40)
  (-48 -200)
  (48 -8))
(((0 1) Dependent -197 -1 43)
  (-97 -100)
  (47 -4))
nil
> (example-2 '(= *))
(((3 -2) Dependent -399 5 34)
  (1 -400)
  (50 -16))
(((-1 3) Dependent -98 -4 96)
  (2 -100)
  (100 -4))
(((-2 1) Dependent -38 -3 299)
  (-50 12)
  (-1 300))
(((-1 1) Independent 4 -2 100)
  (0 4)
  (0 100))
(((-2 4) Dependent -94 -6 196)
  (2 -96)
  (100 96))
(((1 0) Dependent -199 1 42)
  (1 -200)
  (50 -8))
(((0 1) Dependent -99 -1 46)
  (1 -100)
  (50 -4))
nil
> (example-4 '(* * *))
(((1 0 0) Dependent -37 1 349)
  (-48 8 3)
  (99 200 50))
(((0 1 1) Dependent -84 0 398)
  (-97 4 9)
  (148 100 150))
(((0 0 1) Dependent -47 1 47)
  (0 0 -47)
  (0 0 47))
(((1 1 1) Dependent -121 1 747)
  (-145 12 12)
  (247 300 200))
(((3 -2 0) Dependent -133 5 441)
  (1 16 -150)
  (50 400 -9))
(((1 3 0) Dependent -289 -2 1543)
  (-339 20 30)
  (543 500 500))
(((-2 1 0) Dependent -347 -3 37)
  (-50 -300 3)
  (-1 -12 50))
(((-1 1 0) Dependent -143 -2 145)
  (-49 -100 6)
  (49 -4 100))
(((-2 4 0) Dependent -358 -6 990)
  (-292 -96 30)
  (394 96 500))
nil
> (example-4 '(< < < ))
(((1 0 0) Dependent -36 1 294)
  (-48 9 3)
  (48 197 49))
(((0 1 1) Dependent -83 0 293)
  (-97 5 9)
  (47 99 147))
(((0 0 1) Independent -47 1 -1)
  (0 0 -47)
  (0 0 -1))
(((1 1 1) Dependent -119 1 587)
  (-145 14 12)
  (95 296 196))
(((3 -2 0) Dependent -130 5 432)
  (2 17 -149)
  (50 393 -11))
(((1 3 0) Dependent -282 -2 1176)
  (-339 24 33)
  (189 494 493))
(((-2 1 0) Dependent -340 -3 36)
  (-49 -295 4)
  (-1 -13 50))
(((-1 1 0) Dependent -140 -2 94)
  (-49 -98 7)
  (-1 -4 99))
(((-2 4 0) Dependent -256 -6 682)
  (-292 2 34)
  (92 96 494))
nil
> (exit)
```