Exact Side Effects for Interprocedural Dependence Analysis*

Peiyi Tang

Department of Computer Science
The Australian National University
Canberra ACT 2601 Australia

Abstract:
Exact side effects of subroutine calls are essential for exact interprocedural dependence analysis. To summarize the side effect of multiple array references, a collective representation of all the array elements accessed is needed. So far all existing forms of collective summary of side effects of multiple array references are approximate.

In this paper, we propose an approach for exact interprocedural dependence analysis based on the Omega test. In particular, we provide a method of representing the exact image of multiple array references in the form of integer programming projection and a method of back-propagation to form the exact side effect on the actual array.

The representation of the exact side effect proposed in this paper can be used by the Omega test to support the exact interprocedural dependence analysis in parallelizing compilers or semi-automatic parallelization tools.

1 Introduction

One of the challenges to the parallelizing compiler technology is the accurate interprocedural dependence analysis. Subroutines or procedures, as the product of structural and modular programming style, appear frequently in scientific FORTRAN programs. Since almost all scientific programs operate on large arrays in loops and subroutines are usually used to operate on subarrays, subroutine calls nested in loops are very common. To parallelize loops with subroutine calls, the interprocedural dependence analysis between all statements including call statements is needed. The major issue of interprocedural dependence analysis is to determine the *side effects* of subroutine calls. The side effect of a subroutine call is the set of the elements of the actual array read or written by the subroutine.

There has been considerable work on approximate summaries of side effects of subroutine calls. Triolet et al [1] use convex sets to summarize the side effects of subroutine calls. Callahan and Kennedy [2] use *regular sections* for the same purpose. Halvak and Kennedy [3] extend and sharpen the regular sections with *bounded regular sections*. Balasundaram and Kennedy [4] use *simple sections* for the side effects summary. All these forms of summaries are approximate. Approximate summaries make conservative estimation of the side effects. They may contain the array elements which are not accessed by the subroutine and cause false data dependences which do not exist, thus reducing the amount of parallelism detected. A common belief behind these approximate summaries is that exact data dependence tests are too expensive in practice. The recent work on the exact dependence tests [5,6,7,8] showed that by using equality elimination, redundant constraints elimination and modified Fourier-Motzkin variable elimination, exact dependence tests can be competitive with inexact analysis algorithms for real programs. The efficiency of exact test algorithms has brought the exact interprocedural dependence analysis back on the agenda.

In this paper, we propose an approach for exact interprocedural dependence analysis based on the exact Omega dependence test [5,6]. To support the exact interprocedural dependence analysis, an exact representation of side effects of subroutine calls is needed. The exact side effects in this paper are obtained in two steps: (1) An exact image is obtained by merging the multiple

*This work was supported in part by **the Australian Research Council** under Grant No. S6600132 and **Fujitsu Laboratories Ltd.**, Japan.

ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0137...$1.50

```
      PROGRAM LOOP1
      REAL B(200,100)
      DO 10 J=2, 100
        CALL TOUCH(B,100,J)
        DO 20 I=2,J
          ...=B(2*I, I)
20      CONTINUE
10    CONTINUE
      END
```

(a) Loop 1

```
      SUBROUTINE TOUCH(A,N,K)
      REAL A(2*N,N)
      DO 10 I=1, K
        A(2*I, 1) = ...
        A(-2*I+2*K+1,I) = ...
10    CONTINUE
      END
```

(b) Subroutine

```
      PROGRAM LOOP2
      REAL B(200,100)
      DO 10 J=2, 100
        CALL TOUCH(B,100,J)
        DO 20 I=1,J-1
          ...=B(-2*I+2*J-1, I)
20      CONTINUE
10    CONTINUE
      END
```

(c) Loop 2

```
      PROGRAM LOOP3
      REAL B(1:200,0:100)
      DO 10 J=2, 100
        CALL TOUCH(B(1,0),100,J)
        DO 20 I=1,J-1
          ...=B(-2*I+2*J-1, I)
20      CONTINUE
10    CONTINUE
      END
```

(d) Loop 3

Figure 1. Example of Subroutine Calls in Loops

images of the references of the formal array in the subroutine and (2) the merged image is propagated back to the calling subroutine to form the exact side effect on the actual array. Both the merged image and the exact side effect are represented in the form of a single integer programming projection.

The organization of the paper is as follows. Preliminaries about program model and integer programming projection are given in Section 2. The method of merging multiple images is presented in Section 3. The back-propagation of the merged image to form the exact side effect is discussed in Section 4. The exact interprocedural dependence analysis based on the Omega test is presented in Section 5. The method of merging the exact side effect with the images of other references in the calling subroutine is presented in Section 6. In Section 7, we discuss the degree of merging and its impact on performance. The acknowledgement in Sections 8 concludes the paper.

2 Preliminaries

2.1 Program Model

The program in a subroutine can be regarded as a single nested loop, because the statements not enclosed in any loop can always be put in a dummy outermost loop with the both lower and upper bounds equal to 1. We can also assume that the loop stride of each loop is 1 because any loop can be normalized as such. The lower bound of each loop is of the form

$$\max(\lceil\frac{E_1}{k_1}\rceil,\cdots,\lceil\frac{E_u}{k_u}\rceil)$$

where $k_1,\cdots,k_u$ are positive integers and each of $E_1,\cdots,E_u$ is an affine function of the loop index variables of the enclosing loops. The constant term of the affine function may contain integer formal parameters of the subroutine, but the coefficients of the index variables are integer constants. Similarly, the upper bound of each loop is of the form

$$\min(\lfloor\frac{H_1}{k_1}\rfloor,\cdots,\lfloor\frac{H_v}{k_v}\rfloor)$$

where $H_1,\cdots,H_v$ are affine functions of the index variable of the enclosing loops. Although the loop bounds of such form are rare in real programs, they are common after the loops are transformed by unimodular transformation [9,10].

A statement enclosed in n loops has as many instances as the integer grids in the n-dimensional convex set defined by the lower and upper bounds of the enclosed loops. Let $\vec{i} = (i_1,\cdots,i_n)$[1] be the *index vector* of the index variables of the enclosing loops. The convex set can be represented by

$$\{\vec{i} \in Z^n | B\vec{i} \leq \vec{b}\}$$

[1] All vectors in this paper are column vectors, although we use tulples to denote them in the text.

where B is an integer matrix with n columns called *bound matrix* and $\vec{b}$ is an n-vector called *bound vector*. Both B and $\vec{b}$ are extracted from the upper and lower bounds of the enclosing loops. The bound vector may contain integer parameters. In this paper, we assume that each convex set defined by the nested loops is non-empty.

For instance, the index vector of the loop in subroutine TOUCH in Figure 1(b) is $\vec{i} = (I)$ and the convex set for the loop body is $\{(I) \in Z^1 | 1 \leq I \leq K\}$. The bound matrix and vector of the convex set are

$$B = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \ \vec{b} = \begin{pmatrix} K \\ -1 \end{pmatrix}$$

A reference of a d-dimensional array, say A, enclosed in n loops is of the form

$$A(\phi_1, \cdots, \phi_d)$$

where $\phi_k (k = 1, \cdots, d)$ is an affine function of the index variables of the enclosing loops:

$$\phi_k = c_k + \sum_{j=1}^{n} f_{kj} i_j$$

The constant term, c_k, may contain integer parameters. The vector $\vec{\phi} = (\phi_1, \cdots, \phi_d)$ is called *subscripts* of the reference and can be expressed in the matrix form

$$\vec{\phi} = F\vec{i} + \vec{c}$$

where F is a $d \times n$ integer matrix $F = (f_{kj})$, called *coefficient matrix*, and $\vec{c}$ is a d-vector $\vec{c} = (c_1, \cdots, c_d)$, called *displacement vector*.

For instance, the coefficient matrices and displacement vectors for the two references in subroutine TOUCH are:

$$F_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \vec{c_1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$F_2 = \begin{pmatrix} -2 \\ 1 \end{pmatrix}, \vec{c_2} = \begin{pmatrix} 2K+1 \\ 0 \end{pmatrix}$$

Now we can define the *image* of an array reference in nested loops as follows:

Definition 1 (Image of Array Reference) *The image of an array reference in a nested loop is the set of the array elements accessed by all instances of the reference:*

$$\mathcal{I} = \{\vec{\phi} \in Z^d | \exists \vec{i} \in Z^n s.t. \vec{\phi} = \vec{c} + F\vec{i} \wedge B\vec{i} \leq \vec{b}\}$$

where $\vec{i}$ is the index vector of the enclosing loops, B and $\vec{b}$ are the bound matrix and vector, and F and $\vec{c}$ are the coefficient matrix and displacement vector of the reference.

2.2 Integer Programming Projection

The exact Omega dependence test is based on an efficient integer programming projection algorithm. The form of integer programming problem in data dependence analysis is the set of linear equations and inequalities. Suppose $\vec{x} = (x_1, \cdots, x_n)$ is the vector of integer variables. The set of equations and inequalities can be represented by $G\vec{x} = \vec{g}$ and $H\vec{x} \leq \vec{h}$, where G and H are integer matrices with n columns and $\vec{g}$ and $\vec{h}$ are integer n-vectors which may contain integer parameters. The set of feasible solutions of the integer programming problem is $\{\vec{x} \in Z^n | G\vec{x} = \vec{g} \wedge H\vec{x} \leq \vec{h}\}$. The projection of the problem onto a subset of the variables, say $(x_1, \cdots, x_k)$ $(k < n)$, is the set

$$\begin{aligned} \{(x_1, \cdots, x_k) \ \in \ & Z^k | \exists (x_{k+1}, \cdots, x_n) \in Z^{n-k} s.t. \\ & G\vec{x} = \vec{g} \wedge H\vec{x} \leq \vec{h}\} \end{aligned}$$

The basic problem of dependence test is to determine whether the set of feasible solutions of an integer programming problem is non-empty. The basic technique of the Omega test is to project the problem onto one variable by eliminating all the other variables and then determine whether the projection is non-empty.

The image of an array reference defined in Definition 1 is, in fact, the projection of the integer programming problem in the space spanned by $(\phi_1, \cdots, \phi_d, i_1, \cdots, i_n)$ onto the subspace spanned by $(\phi_1, \cdots, \phi_d)$.

Since our method of interprocedural dependence analysis is based on the Omega test, the integer programming projection of the form above will be used throughout this paper.

3 Merged Image of Multiple References

If there are multiple references of an array in the subroutine, we need to find a collective representation for the union of the images of all the references.

Let us concentrate on write references and assume that there are p write references of formal array A, $A(\vec{\phi_j})(j = 1, \cdots, p)$, in the subroutine. Let the coefficient matrix and displacement vector of the j-th ($1 \leq j \leq p$) reference be F_j and $\vec{c_j}$, and its bound matrix and vector be B_j and $\vec{b_j}$. The image of the j-th reference is denoted by $\mathcal{I}_j$. Clearly, the exact image of the multiple references is

$$\mathcal{M} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_p$$

$\mathcal{M}$ can be expressed as a disjunction of projections as follows:

$$\begin{aligned} \mathcal{M} \ = \ & \{\vec{\phi} \in Z^d | \\ & \exists \vec{i_1} s.t. B_1 \vec{i_1} \leq \vec{b_1} \wedge \vec{\phi} = \vec{c_1} + F_1 \vec{i_1} \end{aligned}$$

$$\vee \cdots \vee$$
$$\exists \vec{i_p} s.t. B_p \vec{i_p} \leq \vec{b_p} \wedge \vec{\phi} = \vec{c_p} + F_p \vec{i_p}\}$$

where $\vec{i_j}$ is the index vector of the j-th reference.

While this representation is exact, it is not collective. Ancourt and Irigoin [11] proposed a method to merge the images of the references with constant dependence distances between them. In our terminology, their method can only merge the images of the references with the same coefficient matrix F_j, i.e. $F_1 = \cdots = F_p$. Our method is for general subscript functions with different F_j.

Before going further, we need to clarify the notations for the loop index vectors. If two array references belong to the same loop body, they share all their enclosing loops and, therefore, belong to the same convex set. Let there be $r(r \leq p)$ different convex sets defined by the nested loops of the subroutine. We use $\vec{i}_{j_1}, \cdots, \vec{i}_{j_r}$ to denote the index vectors of the corresponding convex sets. These vectors are independent although some of them may share some outer common loops. In the following discussion, we also use $\vec{i_j}$ for the index vector for the j-th reference and each $\vec{i_j}$ $(j = 1, \cdots, p)$ is one of $\vec{i}_{j_1}, \cdots, \vec{i}_{j_r}$. For instance the two references in subroutine `TOUCH` belong to the same loop body and there is only one convex set defined by the loop. Therefore, $\vec{i_1}$ and $\vec{i_2}$ are actually two notations of the same vector: $\vec{i}_{j_1}$. We use $n_{j_k} (k = 1, \cdots, r)$ to denote the number of elements in vector $\vec{i}_{j_k}$. We also use $(\vec{i}_{j_1}, \cdots, \vec{i}_{j_r})$ to denote the concatenation of vectors $\vec{i}_{j_1}, \cdots, \vec{i}_{j_r}$.

Let $\vec{\phi}^{\min}$ and $\vec{\phi}^{\max}$ be a pair of vectors such that the following is true for every reference (i.e. $j = 1, \cdots, p$),

$$\vec{\phi}^{\min} \leq F_j \vec{i_j} + \vec{c_j} \leq \vec{\phi}^{\max}, \forall \vec{i_j} s.t.\ B_j \vec{i_j} \leq \vec{b_j} \qquad (1)$$

We call $\vec{\phi}^{\min}$ and $\vec{\phi}^{\max}$ the *common bound vectors* of the references.

The images of multiple references can be merged and represented by a single integer programming projection as shown in the following theorem.

Theorem 1 *Given p images of array references, $\mathcal{I}_1, \cdots, \mathcal{I}_p$, defined in Definition 1, if each convex set of enclosing loops is non-empty, the union of the images, $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_p$, can be represented by the following integer programming projection:*

$$\begin{aligned}
\mathcal{S} = \{ & \vec{\phi} \in Z^d | \\
& \exists (\vec{i}_{j_1}, \cdots, \vec{i}_{j_r}, \vec{x}) \in Z^{n_{j_1} + \cdots + n_{j_r} + p} s.t. \\
& x_1 + \cdots + x_p = p - 1 \wedge \vec{0} \leq \vec{x} \leq \vec{1} \wedge \\
& \wedge_{j=1}^{p} B_j \vec{i_j} \leq \vec{b_j} \wedge \\
& \wedge_{j=1}^{p} \vec{\phi} - F_j \vec{i_j} - \vec{c_j} \leq (\vec{\phi}^{\max} - \vec{\phi}^{\min}) x_j \wedge \\
& \wedge_{j=1}^{p} \vec{\phi} - F_j \vec{i_j} - \vec{c_j} \geq (\vec{\phi}^{\min} - \vec{\phi}^{\max}) x_j \}
\end{aligned}$$

where $\vec{x} = (x_1, \cdots, x_p)$ is an auxilliary integer vector, B_j and $\vec{b_j}$ are the bound matrix and vector of the j-th reference, F_j and $\vec{c_j}$ are the coefficient matrix and displacement vector of the j-th reference, $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ are a pair of common bound vectors of the references.

The idea is to introduce p integer variables , $x_1, \cdots, x_p$, between 0 and 1 [12]. In any feasible solution of the projection, only one of $x_1, \cdots, x_p$ is 0 and the remaining are 1. This fact leads to $\mathcal{S} \subseteq \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_p$. $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_p \subseteq \mathcal{S}$ can be established from the condition that each convex set of the nested loops is non-empty. The details of the proof of the theorem can be found in [13].

Notice that we did not specify the contents of $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ in Theorem 1. This means that we can use any pair of vectors that satisfy (1) to replace $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$. One convenient choice is the array limits declared in the subroutine. Let the array be declared as $A(l_1 : u_1, \cdots, l_d : u_d)$ in the subroutine. Each of the limits, l_k or u_k $(k = 1, \cdots, d)$, can be an affine function of integer parameters[2]. The vectors $\vec{l} = (l_1, \cdots, l_d)$ and $\vec{u} = (u_1, \cdots, u_d)$ are called *lower and upper limit vectors* of the array. We can assume that FORTRAN programs at which parallelizing compilers are targeted are always correct and well-behaved. In particular, there should be no array references outside the array limits, and, therefore, we can use $\vec{l}$ and $\vec{u}$ for $\vec{\phi}^{\min}$ and $\vec{\phi}^{\max}$, respectively.

For instance, the lower and upper limit vectors of array `A` in subroutine `TOUCH` are

$$\vec{l} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \ \vec{u} = \begin{pmatrix} 2N \\ N \end{pmatrix}$$

and, according to Theorem 1, the merged image of the two references in the subroutine is represented by

$$\begin{aligned}
\mathcal{S} = \{ & (\phi_1, \phi_2) | \exists (I, x_1, x_2) \in Z^3 s.t. \\
& 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge \\
& x_1 + x_2 = 1 \wedge 1 \leq I \leq K \wedge \\
& (1 - 2N) x_1 \leq \phi_1 - 2I \leq (2N - 1) x_1 \wedge \\
& (1 - N) x_1 \leq \phi_2 - 1 \leq (N - 1) x_1 \wedge \\
& (1 - 2N) x_2 \leq \phi_1 + 2I - 2K - 1 \wedge \\
& \phi_1 + 2I - 2K - 1 \leq (2N - 1) x_2 \wedge \\
& (1 - N) x_2 \leq \phi_2 - I \leq (N - 1) x_2 \}
\end{aligned}$$

Note that there is only one convex set, $1 \leq I \leq K$, in the constraints, because both references belong to the same loop body. In other words, we have $\vec{i_1} = \vec{i_2} = (I)$, $B_1 = B_2$ and $\vec{b_1} = \vec{b_2}$ in this example.

[2]The upper limit of the last dimension, u_d, can be an *assumed size* denoted by a * in FORTRAN [14]. The assumed upper limit can be regarded as a special parameter, which will be replaced by the corresponding upper limit of the actual array.

4 Back-Propagation of Merged Image

The merged image obtained from Theorem 1 only defines the elements of the formal array accessed by the subroutine. It is also symbolic because the integer programming problem may contain formal parameters. To obtain the exact side effect on the actual array, the merged image needs to be *propagated back* to the calling subroutine according to the *execution context* of the call statement. Back-propagation is the process of mapping the exact merged image to the exact side effect on the actual array in the calling subroutine. The execution context is the environment of the call statement in the calling subroutine and is defined by the values of the actuals in the call statement. The formal parameters of a subroutine can be divided into three categories:

Category 1: Array names.

Category 2: Integer parameters used in the constraints of the merged image. These parameters may appear in vectors $\vec{b_j}$, $\vec{c_j}$, $\vec{l}$ and $\vec{u}$.

Category 3: Other parameters. We do not consider them here, because they do not affect the back-propagation of the image.

The back-propagation of the image consists of two steps as follows:

1. Integer Parameter Substitution. In this step, the integer parameters in the constraints are replaced by the actuals and the merged image becomes a *template* of the image.
2. Subscript Translation. The template is translated to the exact side effect on the actual array according to the actual for the array name.

4.1 The Template of the Merged Image

Let $\vec{z} = (z_1, \cdots, z_w)$ be the integer parameters used in the constraints of the merged image (Category 2). The actual for each integer parameter can be an integer constant or an affine function of variables and parameters in the calling subroutine. Let these variables and parameters in the calling subroutine be $\vec{v} = (v_1, \cdots, v_t)$. Then, the actuals in the call statement can be represented by

$$\vec{z} = F_z\vec{v} + \vec{g_z}$$

where F_z is a $w \times t$ constant integer matrix and $\vec{g_z}$ is a constant integer w-vector. For the parameters whose actuals are constants, the corresponding rows of F_z are simply zero vectors. For instance, the integer parameters of subroutine `TOUCH` are $\vec{z} = (K, N)$. In the call statements in all the three programs (`LOOP1`, `LOOP2` and `LOOP3`) in Figure 1(a)(c)(d), the actual for N is a constant and the actual for K is variable J. Thus, we have

$$\vec{z} = \begin{pmatrix} K \\ N \end{pmatrix}, \vec{v} = \begin{pmatrix} J \end{pmatrix}$$

$$\vec{F_z} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \vec{g_z} = \begin{pmatrix} 0 \\ 100 \end{pmatrix}$$

In the constraints of the merged image, the integer parameters $\vec{z} = (z_1, \cdots, z_w)$ can only appear in $\vec{b_j}$, $\vec{c_j}$, $\vec{l}$ and $\vec{u}$. In this paper, we assume that each element of $\vec{b_j}$, $\vec{c_j}$, $\vec{l}$ and $\vec{u}$ is an affine function of these parameters. By substituting $F_z\vec{v} + \vec{g_z}$ for parameters $\vec{z}$, vectors $\vec{b_j}$, $\vec{c_j}$, $\vec{l}$ and $\vec{u}$ become vectors of affine functions of $\vec{v}$. Let them be denoted by $\vec{b_j}'$, $\vec{c_j}'$, $\vec{l}'$ and $\vec{u}'$, respectively.

For instance, these vectors in our example are as follows:

$$\vec{b_1}' = \vec{b_2}' = \begin{pmatrix} J \\ -1 \end{pmatrix}, \vec{c_1}' = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$\vec{c_2}' = \begin{pmatrix} 2J+1 \\ 0 \end{pmatrix}, \vec{l}' = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \vec{u}' = \begin{pmatrix} 200 \\ 100 \end{pmatrix}$$

The formal array now is confined by the lower and upper limit vectors $\vec{l}'$ and $\vec{u}'$. We use $\vec{\phi}'$ to denote the subscripts of the formal array within these limits. Then, the exact set of the elements of the formal array accessed for a particular set of integer actuals can be represented by the following *template* of the image.

Definition 2 (Template) *The template of the merged image is the set defined by*

$$\begin{aligned} \mathcal{S}' = \{ \; & \vec{\phi}' \in Z^d \mid \\ & \exists(\vec{i}_{j_1}, \cdots, \vec{i}_{j_r}, \vec{x}) \in Z^{n_{j_1}+\cdots+n_{j_r}+p+w} \; s.t. \\ & x_1 + \cdots + x_p = p - 1 \wedge \vec{0} \le \vec{x} \le \vec{1} \wedge \\ & \wedge_{j=1}^{p} B_j\vec{i_j} \le \vec{b_j}' \wedge \\ & \wedge_{j=1}^{p} \vec{\phi}' - F_j\vec{i_j} - \vec{c_j}' \le (\vec{u}' - \vec{l}')x_j \wedge \\ & \wedge_{j=1}^{p} \vec{\phi}' - F_j\vec{i_j} - \vec{c_j}' \ge (\vec{l}' - \vec{u}')x_j \} \end{aligned}$$

The formal array with the limit vectors $\vec{l}'$ and $\vec{u}'$ is called the *template formal array*.

4.2 Subscript Translation

To find the exact side effect on the actual array, we need to translate the subscripts of the template formal array to those of the actual array according to the actual for the array name (Category 1). In general, the shapes of the formal and the actual arrays are allowed to be different in FORTRAN and the subscript translation between them could be complicated. However, in most real FORTRAN programs, formal arrays are subarrays of the actual arrays. We have the following assumption with regard to the shape of the actual array.

Assumption 1 (Shape of Actual Array) *Given the actual array* B *for the template formal array* A,

1. *the number of dimensions of* B *is equal to or greater than that of* A, *and*
2. *the sizes of the first* $d-1$ *dimensions of* B *are the same as those of* A. *The size of the d-th dimension of* B *is no less than that of* A.

This assumption should be satisfied in most real FORTRAN programs. Let the actual array be declared as $B(\tilde{l_1}:\tilde{u_1},\cdots,\tilde{l_d}:\tilde{u_d},\cdots)$ in the calling subroutine. Let the actual for the array name be $B(s_1,\cdots,s_d,s_{d+1},\cdots)$. Then $\vec{s}=(s_1,\cdots,s_d)$ are the first d subscripts of the actual. We use $\tilde{\phi}=(\tilde{\phi}_1,\cdots,\tilde{\phi}_d)$ to denote the first d subscripts of the actual array. If Assumption 1 is satisfied, the subscript translation between the formal and actual arrays is very simple as shown in the following lemma.

Lemma 1 *If the conditions in Assumption 1 are satisfied, the subscripts of the template,* $\vec{\phi}'$, *can be translated to the first d subscripts of the actual array,* $\tilde{\phi}$, *by*

$$\tilde{\phi}=\vec{\phi}'+\vec{\delta}$$

where $\vec{\delta}=\vec{s}-\vec{l}'$, *and* $\vec{l}'$ *is the lower limit vector of the template formal array and* $\vec{s}$ *is the first d subscripts of the actual for the array name.*

The proof of the lemma can be found in [13].

In our example, Assumption 1 is satisfied in all the three programs in Figure 1. For the subroutine calls in programs LOOP1 and LOOP2, $\vec{\delta}=\vec{0}$, but for program LOOP3, it is

$$\vec{\delta}=\begin{pmatrix}0\\-1\end{pmatrix}$$

Based on the template of the merged image and the subscript translation scheme, the exact side effect of a subroutine call can be represented by an integer programming projection shown in the following theorem.

Theorem 2 *If the conditions of Assumption 1 are satisfied, the side effect on the actual array of an instance of the call statement is the set of the elements of the actual array, whose subscripts for the dimensions higher than the d-th are* $(s_{d+1},\cdots)$ *and whose first d subscripts,* $\tilde{\phi}=(\tilde{\phi}_1,\cdots,\tilde{\phi}_d)$, *are specified by the following projection:*

$$\begin{aligned}\widetilde{\mathcal{S}}=\ \{\ & \tilde{\phi}\in Z^d|\\ & \exists(\vec{i}_{j_1},\cdots,\vec{i}_{j_r},\vec{x})\in Z^{n_{j_1}+\cdots+n_{j_r}+p}\,s.t.\\ & x_1+\cdots+x_p=p-1\wedge\vec{0}\le\vec{x}\le\vec{1}\wedge\\ & \wedge_{j=1}^{p}B_j\vec{i_j}\le\vec{b_j}'\wedge\\ & \wedge_{j=1}^{p}\tilde{\phi}-\vec{\delta}-F_j\vec{i_j}-\vec{c_j}'\le(\tilde{u}-\tilde{l})x_j\wedge\\ & \wedge_{j=1}^{p}\tilde{\phi}-\vec{\delta}-F_j\vec{i_j}-\vec{c_j}'\ge(\tilde{l}-\tilde{u})x_j\}\end{aligned}$$

where $\vec{\delta}$ *is the same as in Lemma 1,* $\vec{b_j}'$ *and* $\vec{c_j}'$ *are the same as in Definition 2, and* $\tilde{l}$ *and* $\tilde{u}$ *are the lower and upper limit vectors for the first d dimensions of the actual array.*

Note that the common bound vectors, $\vec{u}'$ and $\vec{l}'$, in the constraints of the template are replaced by $\tilde{u}$ and $\tilde{l}$, the upper and lower limit vectors for the first d dimensions of the actual array. The replacement can be justified by the assumption that there are no references outside the limits of the actual array. The details of the proof of Theorem 2 can be found in [13].

For instance, in program LOOP3 the exact side effect on the actual array B of the call statement for a particular J is

$$\begin{aligned}\widetilde{\mathcal{S}}=\ \{\ & (\tilde{\phi}_1,\tilde{\phi}_2)|\exists(I,x_1,x_2)\in Z^3\,s.t.\\ & 0\le x_1\le 1\wedge 0\le x_2\le 1\wedge\\ & x_1+x_2=1\wedge 1\le I\le J\wedge\\ & -199x_1\le\tilde{\phi}_1-2I\le 199x_1\wedge\\ & -100x_1\le\tilde{\phi}_2+1-1\le 100x_1\wedge\\ & -199x_2\le\tilde{\phi}_1+2I-2J-1\le 199x_2\wedge\\ & -100x_2\le\tilde{\phi}_2+1-I\le 100x_2\}\end{aligned}$$

Note that limit vectors of the actual array are $\tilde{l}=(1,0)$ and $\tilde{u}=(200,100)$.

5 Exact Interprocedural Analysis

The purpose of representing exact side effects in the form of integer programming projection is to use the Omega test for exact interprocedural dependence analysis. The exact Omega dependence tests between array references are covered in [5,6,7]. In this section, we discuss the exact Omega dependence tests between call statements and array references.

If a call statement is enclosed in loops, there could be many instances of it. The exact side effect specified by Theorem 2 is only for one instance of the call statement. Recall that the actuals for the integer parameters of the subroutine, $\vec{z}=(z_1,\cdots,z_w)$, are affine functions of variables and parameters of the calling subroutine, $\vec{v}=(v_1,\cdots,v_t)$. Let the first $s(s\le t)$ elements of $\vec{v}$, $\vec{v_y}=(v_1,\cdots,v_s)$, be the variables and the rest the parameters. The variables are usually the index variables of the enclosing loops. Let us use $\widetilde{\mathcal{S}}(\vec{v_y})$ to denote the exact side effect of the instance of the call statment for a particular $\vec{v_y}$. We also assume that the bound matrix and vector of the loops enclosing the call statement are B_y and $\vec{b_y}$, respectively. Hence, all the instances of the call statement can be represented by the following convex set:

$$C=\{\vec{v_y}\in Z^s|B_y\vec{v_y}\le\vec{b_y}\}$$

We concentrate on the dependence test between a call statement and an array reference in the calling subroutine. The dependence test between two call statements can be worked out similarly. Suppose that the subscripts of the array reference for the first d dimensions are $\tilde{\phi}^1 = F_x^1\vec{v_x} + \vec{c_x^1}$ and the subscripts for the remaining dimensions are $\tilde{\phi}^2 = F_x^2\vec{v_x} + \vec{c_x^2}$. Let the bound matrix and vector of the enclosing loops of the reference be B_x and $\vec{b_x}$, respectively.

There is a data dependence (flow, anti, or output) between the call statement and the array reference if and only if there is an element accessed by both and at least one access is write access[3]. Formally, we can have the following definition:

Definition 3 (Data Dependence) *There is a data dependence between a call statement and an array reference if and only if*

1. *there exist* $\tilde{\phi}$, $\vec{v_x}$ *and* $\vec{v_y}$ *such that*

$$\begin{aligned}
&B_x\vec{v_x} \le \vec{b_x} \wedge B_y\vec{v_y} \le \vec{b_y} \wedge \\
&\tilde{\phi} \in \tilde{\mathcal{S}}(\vec{v_y}) \wedge \\
&\tilde{\phi} = F_x^1\vec{v_x} + \vec{c_x^1} \wedge \\
&\vec{s^2} = F_x^2\vec{v_x} + \vec{c_x^2}
\end{aligned}$$

   where $\vec{s^2} = (s_{d+1}, \cdots)$ *are the subscripts from the* $(d+1)$*-th dimension in the array name actual of the call statement, and everything else is as described in the above discussion, and*

2. *either the reference is a write reference or the side effect* $\tilde{\mathcal{S}}$ *is the write side effect.*

The following theorem shows how to form an integer programming projection to test the existence of dependences.

Theorem 3 *There is a data dependence between a call statement and an array reference if and only if*

1. *the set defined by the following projection is non-empty:*

$$\begin{aligned}
\{\ \ &\tilde{\phi} \in Z^d | \exists(\vec{v_x}, \vec{v_y}, \vec{i_{j_1}}, \cdots, \vec{i_{j_r}}, \vec{x})\ s.t. \\
&B_y\vec{v_y} \le \vec{b_y} \wedge \\
&x_1 + \cdots + x_p = p - 1 \wedge \vec{0} \le \vec{x} \le \vec{1} \wedge \\
&\wedge_{j=1}^{p} B_j\vec{i_j} \le \vec{b_j}' \wedge \\
&\wedge_{j=1}^{p} \tilde{\phi} - \vec{\delta} - F_j\vec{i_j} - \vec{c_j}' \le (\tilde{u} - \tilde{l})x_j \wedge \\
&\wedge_{j=1}^{p} \tilde{\phi} - \vec{\delta} - F_j\vec{i_j} - \vec{c_j}' \ge (\tilde{l} - \tilde{u})x_j \wedge \\
&B_x\vec{v_x} \le \vec{b_x} \wedge \\
&\tilde{\phi} = F_x^1\vec{v_x} + \vec{c_x^1} \wedge \\
&\vec{s^2} = F_x^2\vec{v_x} + \vec{c_x^2}\}
\end{aligned}$$

   where B_y, $\vec{b_y}$, B_x, $\vec{b_x}$, F_x^1, $\vec{c_x^1}$, F_x^2, $\vec{c_x^2}$, *and* $\vec{s^2}$ *are the same as in Definition 3 and everything else is the same as in Theorem 2, and*

2. *either the reference is a write reference or the side effect* $\tilde{\mathcal{S}}$ *is the write side effect.*

For instance, to find out whether there are dependences between the call statement and the read reference in program LOOP3, we can test whether the following projection is non-empty:

$$\begin{aligned}
\{\ \ &(\tilde{\phi}_1, \tilde{\phi}_2) | \exists(I, x_1, x_2, J, J_1, I_1) \in Z^6 s.t. \\
&0 \le x_1 \le 1 \wedge 0 \le x_2 \le 1 \wedge x_1 + x_2 = 1 \wedge \\
&1 \le I \le J \wedge 2 \le J \le 100 \wedge \\
&-199x_1 \le \tilde{\phi}_1 - 2I \le 199x_1 \wedge \\
&-100x_1 \le \tilde{\phi}_2 + 1 - 1 \le 100x_1 \wedge \\
&-199x_2 \le \tilde{\phi}_1 + 2I - 2J - 1 \le 199x_2 \wedge \\
&-100x_2 \le \tilde{\phi}_2 + 1 - I \le 100x_2 \wedge \\
&2 \le J_1 \le 100 \wedge 1 \le I_1 \le J_1 - 1 \wedge \\
&\tilde{\phi}_1 = -2I_1 + 2J_1 - 1 \wedge \tilde{\phi}_2 = I_1\}
\end{aligned}$$

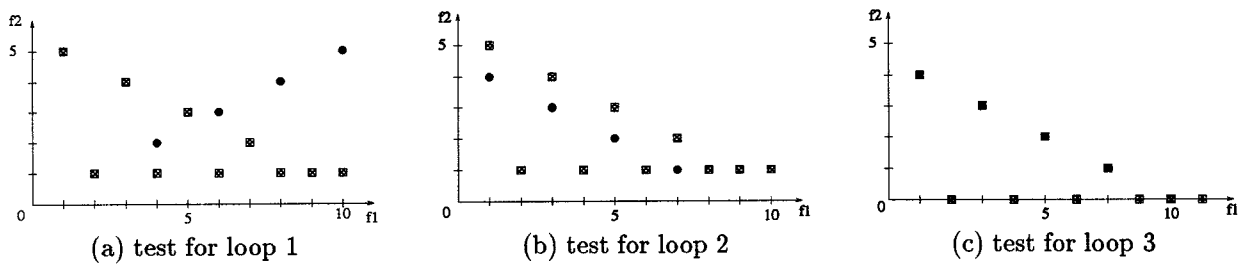
To find distance vectors of the dependences, we only need to project the integer programming problem onto the differences of the index variables of the common enclosing loops. In the above example, the distance vector is defined as (Δ), where $\Delta = J_1 - J$, because the call statement and the assignment statement share only loop J. The integer programming projection for the distance vector is

$$\begin{aligned}
\mathcal{D} = \{\ \ &(\Delta)\ |\ \exists(\tilde{\phi}_1, \tilde{\phi}_2, I, x_1, x_2, J, J_1, I_1) \in Z^8 s.t. \\
&0 \le x_1 \le 1 \wedge 0 \le x_2 \le 1 \wedge x_1 + x_2 = 1 \wedge \\
&1 \le I \le J \wedge 2 \le J \le 100 \wedge \\
&-199x_1 \le \tilde{\phi}_1 - 2I \le 199x_1 \wedge \\
&-100x_1 \le \tilde{\phi}_2 + 1 - 1 \le 100x_1 \wedge \\
&-199x_2 \le \tilde{\phi}_1 + 2I - 2J - 1 \le 199x_2 \wedge \\
&-100x_2 \le \tilde{\phi}_2 + 1 - I \le 100x_2 \wedge \\
&2 \le J_1 \le 100 \wedge 1 \le I_1 \le J_1 - 1 \wedge \\
&\tilde{\phi}_1 = -2I_1 + 2J_1 - 1 \wedge \tilde{\phi}_2 = I_1 \wedge \\
&\Delta = J_1 - J\}
\end{aligned}$$

The integer programming projections for distance vectors for programs LOOP1 and LOOP2 can be formulated similarly. We have run the exact Omega tests for these projections and the results are:

1. There are no dependences between the call statement and the assignment statement in program

[3] To simplify the discussion, we do not distinguish between direct and indirect data dependences [7] in this paper.



(a) test for loop 1 (b) test for loop 2 (c) test for loop 3

Figure 2. Interprocedural Omega Tests

LOOP1. Therefore, loop J can be executed as a DOALL loop[4].

2. There are data dependences between the call statement and the assignment statment in program LOOP2. The distance vector is $(\Delta) = (1)$.

3. There are data dependences between the call statement and the assignment statment in program LOOP3. The distance vector is $(\Delta) = (0)$. Since the dependences do not cross iterations, loop J still can be transformed to a DOALL loop.

These Omega tests were run on a Sparc IPC Workstation based on the 15.8 MIPS Sun Sparc CPU. The times for the three tests are 68.164 msecs, 80.830 msecs and 93.029 msecs, respectively.

Figure 2 (a), (b) and (c) show the elements of array B accessed in programs LOOP1, LOOP2 and LOOP3, respectively, for $J = 5$. The shaded small squares show the side effects of the subroutine call for $J = 5$ and the dark dots show the elements read by the assignment statement for $J = 5$ (some squares and dots are overlapped in Figure 2(c)). It can be seen from these figures that these interprocedural dependence tests are accurate. Notice that the accuracy of these tests cannot be achieved if any of the approximate summaries of side effects in [1,2,3,4] was used.

6 Multi-Level Subroutine Calls

We have shown how to merge the images of multiple array references in a subroutine (say, AA) and propagate them back to the calling subroutine (say, BB) for data dependence tests. If subroutine BB is called by another subroutine (say, CC), there is the question of how to process the side effect of subroutine call for AA for the data dependence tests in subroutine CC. Figure 3 shows such a situation: program MAIN calls subroutine LOOP which, in turn, calls subroutine TOUCH.

Basically, we have two choices:

- We can further propagate the exact side effect back to the subroutine that calls this subroutine. In the example of Figure 3, the side effect of the call statement for TOUCH in (b) can be further propagated back to the main program in (a).
- We can merge the side effect with the images of array references or other side effects of call statements before the further back-propagation. In the same example, the side effect of the call statement can be merged with the image of the write reference of B in (b) and, then, propagated back to the main program in (a).

According to the discussions in Sections 3 and 4, the exact side effect on array B of an instance of the call statement for TOUCH is

$$\begin{aligned}\widetilde{\mathcal{S}} &= \{(\tilde{\phi}_1, \tilde{\phi}_2) | \exists (I, x_1, x_2) \in Z^3 s.t. \\ &\quad 0 \le x_1 \le 1 \wedge 0 \le x_2 \le 1 \wedge \\ &\quad x_1 + x_2 = 1 \wedge 1 \le I \le J \wedge \\ &\quad (1-2M)x_1 \le \tilde{\phi}_1 - 2I \le (2M-1)x_1 \wedge \\ &\quad (1-M)x_1 \le \tilde{\phi}_2 - 1 \le (M-1)x_1 \wedge \\ &\quad (1-2M)x_2 \le \tilde{\phi}_1 + 2I - 2J - 1 \\ &\quad \le (2M-1)x_2 \wedge \\ &\quad (1-M)x_2 \le \tilde{\phi}_2 - I \le (M-1)x_2\}\end{aligned}$$

In this integer programming projection, J is a parameter and has different values for different instances of the call statement. It is also a variable in the calling subroutine LOOP and the convex set for its values is $1 \le J \le L$. To obtain the side effect of the *all* instances of the call statement, all we need to do is to add $1 \le J \le L$ into the constraints of the projection and treat J as a variable as follows:

$$\begin{aligned}\widetilde{\mathcal{S}} &= \{(\tilde{\phi}_1, \tilde{\phi}_2) | \exists (I, J, x_1, x_2) \in Z^4 s.t. \\ &\quad 0 \le x_1 \le 1 \wedge 0 \le x_2 \le 1 \wedge\end{aligned}$$

[4]A DOALL loop is a parallel loop whose iterations can be executed in any order without cross-iteration synchronization or communication. If a loop does not have dependences across its iterations, it can be transformed to a DOALL loop.

```
      PROGRAM MAIN
      REAL C(200,100)
      DO 10 I=2, 100
        CALL LOOP(C,100,I)
        DO 20 J=2, I
          DO 30 K=2, I
            ...=C(2*J, K)
30        CONTINUE
20      CONTINUE
10    CONTINUE
      END
```

(a) main program

```
      SUBROUTINE LOOP(B,M,L)
      REAL B(2*M,M)
      DO 10 J=1, L
        CALL TOUCH(B, M, J)
        DO 20 I=1, J
          B(2*J-1,I) = ...
20      CONTINUE
10    CONTINUE
      END
```

(b) Subroutine LOOP

```
      SUBROUTINE TOUCH(A,N,K)
      REAL A(2*N,N)
      DO 10 I=1, K
        A(2*I, 1) = ...
        A(-2*I+2*K+1,I) = ...
10    CONTINUE
      END
```

(c) Subroutine TOUCH

Figure 3. Multi-level Subroutine Calls

$$
\begin{aligned}
& x_1 + x_2 = 1 \wedge 1 \le I \le J \wedge 1 \le J \le L \\
& (1-2M)x_1 \le \tilde{\phi}_1 - 2I \le (2M-1)x_1 \wedge \\
& (1-M)x_1 \le \tilde{\phi}_2 - 1 \le (M-1)x_1 \wedge \\
& (1-2M)x_2 \le \tilde{\phi}_1 + 2I - 2J - 1 \\
& \le (2M-1)x_2 \wedge \\
& (1-M)x_2 \le \tilde{\phi}_2 - I \le (M-1)x_2\}
\end{aligned}
$$

We can propagate this side effect of all instances of the call statement back to program MAIN using the same technique described in Section 4.

Alternatively, the above side effect can be merged with the image of the reference of array B to form a single projection before the back-propagation. What we need is to introduce y_1 and y_2 such that $y_1 + y_2 = 1$ and $0 \le y_1, y_2 \le 1$. We use y_1 for the reference and y_2 for the call statement. The merged image is

$$
\begin{aligned}
\widetilde{\mathcal{S}} \; = & \{(\tilde{\phi}_1, \tilde{\phi}_2) | \\
& \exists (I, J, x_1, x_2, J_1, I_1, y_1, y_2) \in Z^8 \, s.t. \\
& 0 \le y_1 \le 1 \wedge 0 \le y_2 \le 1 \wedge \\
& y_1 + y_2 = 1 \wedge \\
& 1 \le J_1 \le L \wedge 1 \le I_1 \le J_1 \wedge \\
& (1-2M)y_1 \le \tilde{\phi}_1 - 2J_1 + 1 \\
& \le (2M-1)y_1 \wedge \\
& (1-M)y_1 \le \tilde{\phi}_2 - I_1 \le (M-1)y_1 \wedge \\
& 1 \le J \le L \wedge \\
& 0 \le x_1 \le 1 \wedge 0 \le x_2 \le 1 \wedge \\
& x_1 + x_2 = 1 \wedge 1 \le I \le J \wedge \\
& (1-2M)(x_1 + y_2) \le \tilde{\phi}_1 - 2I \\
& \le (2M-1)(x_1 + y_2) \wedge \\
& (1-M)(x_1 + y_2) \le \tilde{\phi}_2 - 1 \\
& \le (M-1)(x_1 + y_2) \wedge \\
& (1-2M)(x_2 + y_2) \le \tilde{\phi}_1 + 2I - 2J - 1 \\
& \le (2M-1)(x_2 + y_2) \wedge \\
& (1-M)(x_2 + y_2) \le \tilde{\phi}_2 - I \\
& \le (M-1)(x_2 + y_2)\}
\end{aligned}
$$

Notice that the index vector for the array reference is (J_1, I_1).

7 Discussion and Related Work

We have presented a method of exact interprocedural dependence analysis based on the Omega test. We described how to represent the image of an array reference in the form of integer programming projection and how to propagate the image back to the calling subroutine for the exact side effect on the actual array. To represent the images of multiple references collectively, we offered a method to merge multiple images to a single integer programming projection. The advantage is that the number of pairs of array references (including call statements) for dependence tests can be reduced, because the exact side effect of a call statement is represented collectively as in most approximate side effects summaries. We believe that the number of pairs of references for dependence tests is important. According to [5,6], the time of running the integer programming projection for a dependence test is roughly the same as the time for scanning array subscripts and loop bounds, and forming and copying the integer programming problem.[5] A potential problem is the size of the integer programming problem, because we introduce extra variables when merging the multiple images. The degree of merging can vary. Merging images of all references including side effects of call statements and not merging at all are the two extremes on the spectrum. For the best performance, the balance between the num-

[5] It is mentioned in [5,6] that the dependence test time is about 2-8 times the copying time. The time of scanning array subscripts and loop bounds to build the dependence problem is typically 2-4 times the copying cost. For many pairs, the time of building the problem is larger than the time of analysis.

ber of pairs of dependence tests and the size of integer programming projection needs to be determined. While the performance of the Omega tests for the examples in this paper is acceptable, we need more experiments on real programs. We are currently incorporating the Omega test into our system and implementing the interprocedural dependence test of this paper. It is hoped that the experiments on real FORTRAN programs will show us the best degree of merging in the near future.

In [15,16], Li and Yew proposed to use *atom images* and *atoms* to carry the information about array references and loop bounds for exact interprocedural dependence analysis. Their work is equivalent to the "no merging" extreme on the spectrum of degree of merging. Our work is also in the framework of the exact Omega dependence test.

8 Acknowledgement

The Omega tests in this paper are run on the Omega System ported from University of Maryland. We would like to thank W. Pugh for making the Omega System available for anonymous `ftp`.

References

[1] R. Triolet, F. Irigoin and P. Feautrier, "Direct Parallelization of Call Statements," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 21, no. 7, pp. 176–185, July 1986.

[2] D. Callahan and K. Kennedy, "Analysis of Interprocedural Side Effects in a Parallel Programming Environment," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 517–550, 1988.

[3] P. Havlak and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, July 1991.

[4] V. Balasundaram and K. Kennedy, "A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations," in *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, Orgen, June 21-23, 1989, pp. 41–53.

[5] W. Pugh, "The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis," in *Proceedings of Supercomputing '91*, November 1991, pp. 4–13.

[6] ———, "A Practical Algorithm for Exact Array Dependence Analysis," *Communications of the ACM*, vol. 35, no. 8, pp. 102–114, August 1992.

[7] W. Pugh and D. Wonnacott, "Eliminating False Data Dependences Using the Omega Test," in *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Francisco, California, June 17-19, 1992, pp. 140–151.

[8] C. Eisenbeis and J-C. Sogno, "A General Algorithm for Data Dependence Analysis," in *Proceedings of 1992 ACM International Conference on Supercomputing*, Washington, D.C., USA, July, 1992, pp. 292–302.

[9] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, October 1991.

[10] U. Banerjee, "Unimodular Transformations of Double Loops," in *Advances in Languages and Compilers for Parallel Processing: Proceedings of the Third (1990) Workshop on Languages and Compilers for Parallel Computing*, A. Nicolau, D. Gelernter, T. Gross and D. Padua, Eds. London, UK: Pitman, pp. 192–219, 1991.

[11] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, Wiliamsburg, Virginia, April 21-24, 1991, pp. 39–50.

[12] H. M. Salkin, *Integer Programming*. Addison-Wesley Publishing Company, 1975.

[13] P. Tang, "Exact Side Effects for Interprocedural Dependence Analysis," Department of Computer Science, The Australian National University, Technical Report TR-CS-92-15, November 1992.

[14] D. M. Morno, *A Crash Course in FORTRAN 77*. London, Edward Arnold, Publishing Division of Hodder and Stoughton Limited, 1989.

[15] Z. Li and P-C. Yew, "Interprocedural Analysis for Parallel Computing," *Proceedings of the 1988 International Conference on Parallel Processing*, vol. II, pp. 221–228, August 1988.

[16] Z. Li and P-C. Yew, "Program Parallelization with Interprocedural Analysis," *The Journal of Supercomputing*, vol. 2, no. , pp. 225–244, 1988.