**Exact Side Effects for
Interprocedural Dependence
Analysis**

Peiyi Tang

November 7, 1992

Exact Side Effects for Interprocedural Dependence Analysis*

Peiyi Tang

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

November 7, 1992

Abstract

Exact side effects of array references in subroutines are essential for exact interprocedural dependence analysis. To summarize the side effects 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 present a method to represent the exact side effects of multiple array references in the form of the projection of a single integer programming problem. Since the representation is collective, it dramatically reduces the number of pairs of dependences checking compared with other methods of exact interprocedural analysis.

The representation of the exact side effects proposed in this paper can be used by the Omega test to support the exact interprocedural dependence analysis in parallelizing compilers.

Keywords: exact side effects of array references, interprocedural dependence analysis, exact dependence tests, loop parallelization, call statements, parallelizing compilers.

*This work was done when the author was on leave in the Computer Science Laboratory at the Research School of the Physical Sciences and Engineering of the Australian National University and was supported in part by **the Australian Research Council** under Grant No. S6600132 and the donation of Fujitsu Laboratory, Ltd.

1 Introduction

One of the challenges to the parallelizing compiler technology for scientific programs 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 with 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 the call statements is needed. The major issue of the interprocedural dependence analysis is to determine the *side effects* of subroutine calls. The side effects of a subroutine call is the set of the elements of the actual array read or written by the subroutine.

There have been many works on approximate summary of side effects of subroutine calls. Triolet et al [1] use a convex set to summarize the set of array elements of multiple references in a subroutine. Callahan and Kennedy [2] use *regular sections* for summaries. Halvak and Kennedy [3] extend and sharpen the regular sections with *bounded regular sections*. Balasundaram and Kennedy [4] use *simple sections* for side effects summary. All these forms of summary are approximate. For instance, the exact write side effects on array B of the call statement in program LOOP1 in Figure 1(a) with $J = 5$ is shown in Figure 2(a). The approximate summaries of the side effects by convex sets, regular sections, and simple sections are illustrated in Figure 2 (b) (c) and (e), respectively. Approximate summaries make conservative estimations of the side effects. They may contain the array elements which are not accessed by the subroutine call and cause false data dependences which do not exist. False dependencies reduce the amount of parallelism detected and introduce unnecessary data synchronization or communication between parallel processors. A common belief behind these approximate summaries is that exact data dependence tests are too expensive to use in practice. The recent works by W. Pugh on the exact Omega dependence test [5,6,7] showed that by using the equalities elimination, the modified Fourier-Motzkin variable elimination, and other enhancement techniques such as redundant constraints elimination, the exact Omega dependence test is competitive with inexact analysis algorithms for real programs. The efficiency of the Omega test has brought the exact interprocedural dependence analysis back on the agenda.

To support the exact interprocedural dependence analysis, an exact representation of the side effects of subroutine calls is necessary. In this paper, we present a method to represent the exact side effects in the form of the projection of a single integer programming problem. This representation can be used by the Omega test for the interprocedural dependence analysis. The representation is exact in that it contains and only contains the array elements accessed. It is also collective in that a single integer programming projection contains the elements accessed by all the references in the subroutine.

Li and Yew [8,9] proposed to use *atom images* and *atoms* to summarize array side effects in subroutines. While the atom images and atoms are exact, they are not collective. Each reference in the subroutine has an atom image and becomes an atom after

```

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
  B(2*I, 1) = ...
  B(-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

propagated back to the calling subroutine. The number of array references including the atoms from the called subroutines is the same as if all the subroutines were in-line expanded. Therefore, the number of pairs of array references for dependence test could be huge if the call graph is deep. The number of array reference pairs for dependence test are important. According to [5,6], the time for dependence analysis of a reference pair is roughly the same as the time for scanning array subscripts and loop bounds, and forming and copying the integer programming problem.¹ The collective representation of the exact side effects proposed in this paper dramatically reduces the number of reference pairs for the dependence tests.

The exact side effects in this paper are obtained in two steps: (1) An exact image that merges all the images of the references of the formal array in the subroutine is formed and (2) the merged exact image is propagated back to the calling subroutine to form the exact side effects on the actual array. We first describe the program model and introduce notations in Section 2. The method of merging multiple images is presented in Section 3. The back propagation of the merged image is discussed in Section 4. In section 5,

¹It 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.

we present the method of using exact side effects for exact interprocedural dependence analysis. In Section 6, we discuss merging the exact side effects with array reference images in the calling subroutine. The concluding remarks and the acknowledgement in Sections 7 and 8 conclude the paper.

2 Preliminaries

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 both lower and upper loop 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

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

where $k_1, \dots, k_u$ are positive integers and each of $E_1, \dots, E_u$ is an affine function of the loop index variables of the enclosing loops. The constant term of the function may have 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

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

where $H_1, \dots, H_v$ 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 [10,11], which incorporates wide range of loop transformations such as loop interchange, loop skewing [12] and loop permutation [13].

Therefore, a statement enclosed in n loops with loop index variables $i_1, \dots, i_n$ 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. Each grid in the convex set represents a loop iteration. The convex set can be represented by

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

where $\vec{i}$ is the n -vector of index variables called *index vector*, B is an integer matrix with n columns called *bound matrix* and $\vec{b}$ is an n -vector called *bound vector*. The bound vector may have integer parameters. In this paper, we assume that each convex set defined by the nested loops is non-empty. That is, there is at least one instance for each loop body of the nested loops.

For instance, the bound matrix and vector of the loop in subroutine TOUCH in Figure 1(b) are

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

and we can assume that $K \geq 1$ and the convex set defined by the loop is non-empty.

A reference of an array, say A , enclosed in n loops is of the form

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

where $\phi_k, k = 1, \dots, d$, is an affine function of the index variables $i_1, \dots, i_n$ of the enclosing loops:

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

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

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

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

For instance, the coefficient matrices and the 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 An Array Reference) *The image of an array reference in nested loops in a subroutines is the set of the elements accessed by all instances of the reference in the nested loops:*

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

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

3 Merged Image of Multiple References

The image of an array reference defined in Definition 1 is, in fact, the projection of the integer programming problem to the subspace spanned by $(\phi_1, \dots, \phi_d)$. 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 , $A(\vec{\phi}_j)$, $j = 1, \dots, p$, in the subroutine. Let the coefficient matrix and displacement vector of the j -th ($1 \leq j \leq p$) reference be F_j and $\vec{c}_j$, respectively, and the bound matrix and vector be B_j and $\vec{b}_j$, respectively. The image of the j -th reference is $\mathcal{I}_j$. Clearly, the image of the multiple references is

$$\mathcal{M} = \mathcal{I}_1 \cup \dots \cup \mathcal{I}_p.$$

It is not too difficult to prove that $\mathcal{M}$ can be expressed as a disjunction of projections as follows:

$$\mathcal{M} = \{\vec{\phi} \in Z^d | \exists \vec{i}_1 \text{ s.t. } B_1 \vec{i}_1 \leq \vec{b}_1 \wedge \vec{\phi} = \vec{c}_1 + F_1 \vec{i}_1 \vee \dots \vee \exists \vec{i}_p \text{ s.t. } B_p \vec{i}_p \leq \vec{b}_p \wedge \vec{\phi} = \vec{c}_p + F_p \vec{i}_p\}$$

While this representation is exact, it is not collective. After it is propagated back the calling subroutine, it will be intersected with other reference sets for the dependence tests. Because of the disjunction form of the representation, the intersection with another reference set will also be in the disjunction form. This is equivalent to exporting multiple images like atom images [8,9] to the calling subroutine. What we need is an exact representation in the form of the projection of a single integer programming problem. The challenge is to merge multiple projections into a single projection.

Ancourt and Irigoien [14] proposed a method to merge the images of the references with constant dependence distances between them. In our terms, their method only merges the images of the references with the same coefficient matrix F_j , i.e., $F_1 = \dots = F_p$. Our method is for the general subscripts functions with different F_j .

Before introducing the integer programming projection for the merged image of the references, we need to clarify the notations for the loop index vectors. If two array refer-

ences 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 $\vec{i}_{j_1}, \dots, \vec{i}_{j_r}$ 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 of $\vec{i}_j$ ($j = 1, \dots, p$) is one of $\vec{i}_{j_1}, \dots, \vec{i}_{j_r}$. For instance, there is only one convex set defined by the loop in subroutine **TOUCH**, $1 \leq I \leq K$, and the two references belong to the same convex set. Therefore, $\vec{i}_1$ and $\vec{i}_2$ are actually two notations of the same vector: $\vec{i}_{j_1}$. We also use $(\vec{i}_{j_1}, \dots, \vec{i}_{j_r})$ to denote the concatenation of vectors $\vec{i}_{j_1}, \dots, \vec{i}_{j_r}$.

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

$$\vec{\phi}^{\min} \leq F_j \vec{i}_j + \vec{c}_j \leq \vec{\phi}^{\max}, \text{ for all } \vec{i}_j \text{ such that } B_j \vec{i}_j \leq \vec{b}_j \quad (1)$$

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

Definition 2 (Merged Image of Multiple References) *The merged image of p array references is the set:*

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

where $\vec{x} = (x_1, \dots, x_p)$ is an auxiliary integer vector, B_j and $\vec{b}_j$ are the bound matrices and vectors, F_j and $\vec{c}_j$ are the coefficient matrices and displacement vectors, $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ are a pair of common bound vectors of the references.

The idea behind Definition 2 for the merged image is to introduce p integer variables, $x_1, \dots, x_p$, between 0 and 1 [15]. Since the sum of them is $p - 1$, only one variable can be 0 and the remaining variables must be 1. The following theorem shows that if all the convex sets of nested loops are non-empty, the merged image is the exact set of the elements accessed by all the references.

Theorem 1 *If each convex set defined by the nested loops enclosing a reference is non-empty, the merged image defined in Definition 2 is the exact set of the elements accessed by all the references. That is*

$$\mathcal{S} = \mathcal{I}_1 \cup \dots \cup \mathcal{I}_p.$$

Proof: ($\Rightarrow$) Let $\vec{\phi}$ be an element of $\mathcal{S}$. There exists a $(\vec{i}_{j_1}, \dots, \vec{i}_{j_r}, \vec{x}) \in Z^{n_{j_1} + \dots, n_{j_r} + p}$, and hence, $\vec{i}_1, \dots, \vec{i}_p$, such that all the constraints in the definition of $\mathcal{S}$ are satisfied. In particular, $\sum_{j=1}^p x_j = p - 1$ and $0 \leq x_j \leq 1 (j = 1, \dots, p)$. Only one of x_j is 0 and all the others are 1. Let $x_k = 0 (1 \leq k \leq p)$, and $x_j = 1 (j \neq k)$. Thus, $B_k \vec{i}_k \leq \vec{b}_k$ and $\vec{\phi} = F_k \vec{i}_k + \vec{c}_k$. Since $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ are the upper and lower bounds of $\vec{\phi}$, we have $\vec{\phi}^{\min} \leq \vec{\phi} \leq \vec{\phi}^{\max}$. For other index vectors $\vec{i}_j, (j \neq k)$, $\vec{\phi}^{\min} \leq \vec{\phi} \leq \vec{\phi}^{\max}$ and $B_j \vec{i}_j \leq \vec{b}_j$ imply $\vec{\phi} - F_j \vec{i}_j - \vec{c}_j \leq \vec{\phi}^{\max} - \vec{\phi}^{\min}$ and $\vec{\phi} - F_j \vec{i}_j - \vec{c}_j \geq \vec{\phi}^{\min} - \vec{\phi}^{\max}$. Therefore, these constraints for $\vec{\phi}$ by $\vec{i}_j$ are redundant. The only non-redundant constraints involving $\vec{\phi}$ are $\vec{\phi} = F_k \vec{i}_k + \vec{c}_k$. Hence, $\vec{\phi} \in \mathcal{I}_k$ and we have $\mathcal{S} \subseteq \mathcal{I}_1 \cup \dots \cup \mathcal{I}_p$.

($\Leftarrow$) Let $\vec{\phi}$ be an element of $\mathcal{I}_1 \cup \dots \cup \mathcal{I}_p$. Without loss of generality, we assume that $\vec{\phi} \in \mathcal{I}_k (1 \leq k \leq p)$. Thus, there exists an $\vec{i}_k$ such that $B_k \vec{i}_k \leq \vec{b}_k$ and $\vec{\phi} = F_k \vec{i}_k + \vec{c}_k$. This implies $\vec{\phi}^{\min} \leq \vec{\phi} \leq \vec{\phi}^{\max}$. Since all the convex sets defined by the nested loops are non-empty, there exists at least one independent vector in each of the other convex sets. Therefore, there exist index vectors $\vec{i}_{j_1}, \dots, \vec{i}_{j_r}$ (including this $\vec{i}_k$) such that $B_j \vec{i}_j \leq \vec{b}_j (j = 1, \dots, p)$. For the j -th reference such that $j \neq k$, $B_j \vec{i}_j \leq \vec{b}_j$ implies $\vec{\phi}^{\min} \leq F_j \vec{i}_j + \vec{c}_j \leq \vec{\phi}^{\max}$. Then, $\vec{\phi}^{\min} \leq \vec{\phi} \leq \vec{\phi}^{\max}$ and $\vec{\phi}^{\min} \leq F_j \vec{i}_j + \vec{c}_j \leq \vec{\phi}^{\max}$ imply $\vec{\phi} - F_j \vec{i}_j - \vec{c}_j \leq \vec{\phi}^{\max} - \vec{\phi}^{\min}$ and $\vec{\phi} - F_j \vec{i}_j - \vec{c}_j \geq \vec{\phi}^{\min} - \vec{\phi}^{\max}$. We can take $x_k = 0$ and $x_j = 1$ for all $j \neq k$ and conclude that $\vec{\phi} \in \mathcal{S}$ and $\mathcal{I}_1 \cup \dots \cup \mathcal{I}_p \subseteq \mathcal{S}$. $\square$

Notice that we did not specify the contents of $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ in definition 2. Theorem 1 tells us that we can use any pair of vectors that satisfy (1) to replace $\vec{\phi}^{\max}$ and $\vec{\phi}^{\min}$ in the definition and the resulting sets are all the same: $\mathcal{I}_1 \cup \dots \cup \mathcal{I}_p$. One convenient choice is the array limits declared in the subroutine. Let the array be declared as $A(l_1 : u_1, \dots, l_d : u_d)$ in the subroutine. Each of the limits, l_k or $u_k (k = 1, \dots, d)$, can be an affine function of integer parameters². The vectors

$$\vec{l} = \begin{pmatrix} l_1 \\ \vdots \\ l_d \end{pmatrix} \quad \vec{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_d \end{pmatrix}$$

are called *lower and upper limit vectors* of the array. We can assume that sequential programs at which parallelizing compilers are targeted are always correct and well-behaved. In particular, there should be no array references outside of the array limits, and, therefore, we can use $\vec{l}$ and $\vec{u}$ for $\vec{\phi}^{\min}$ and $\vec{\phi}^{\max}$, respectively, in the definition 2.

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

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

²The upper limit of the last dimension, u_d , can be an *assumed size* denoted by a $*$ [16]. The assumed upper limit can be regarded as a special parameter, which will be replaced by the corresponding upper limit of the actual array.

and the merged image of the two references in the subroutine is

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

When the subroutine is called with $N = 100$ and $K = 5$, the above projection gives the exact side effects shown in Figure 2(a).

4 Back Propagation of Merged Image

The merged image in Definition 2 only defines the elements of the formal array accessed by the subroutine. It is also symbolic because the integer programming problem may have integer formal parameters. To obtain the exact side effects 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 images to the exact side effects on the actual array. 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 the subroutine can be divided into three groups:

- Array names.
- 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 $\vec{u}$.
- 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 the following two steps:

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 effects on the actual array according to the actual for the array name.

4.1 The Template of the Merged Image

Let $\vec{z} = (z_1, \dots, z_w)$ be the integer parameters used in the constraints of the merged image. We assume that the actual for each integer parameter is an affine function of variables or integer parameters in the calling subroutine. Let these variables or parameters be $\vec{v} = (v_1, \dots, v_t)$. These affine functions 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 a 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 $\vec{z} = (z_1, \dots, z_w)$ can only appear in $\vec{b}_j$, $\vec{c}_j$, $\vec{l}$ and $\vec{u}$. In this paper, we assume that each of $\vec{b}_j$, $\vec{c}_j$, $\vec{l}$ and $\vec{u}$ is an affine function of these parameters with constant coefficients. By substituting $F_z \vec{v} + \vec{g}_z$ for the parameters, $\vec{b}_j$, $\vec{c}_j$, $\vec{l}$ and $\vec{u}$ become the vectors which are affine functions of $\vec{v}$ with constant coefficients. 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 $\vec{u}'$ and $\vec{l}'$ define the size of the formal array. Let $\vec{\phi}'$ 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 3 (The Template of the Merged Image) *The template of the merged image is the set defined by*

$$\begin{aligned} \mathcal{S}' = \{ \vec{\phi}' \in Z^d \mid & \exists (\vec{i}_{j_1}, \dots, \vec{i}_{j_r}, \vec{x}) \in Z^{n_{j_1} + \dots + n_{j_r} + p + w} \text{ s.t.} \\ & x_1 + \dots + x_p = p - 1 \wedge \vec{0} \leq \vec{x} \leq \vec{1} \wedge \\ & \wedge_{j=1}^p B_j \vec{i}_j \leq \vec{b}_j' \wedge \\ & \wedge_{j=1}^p \vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \leq (\vec{u}' - \vec{l}') x_j \wedge \\ & \wedge_{j=1}^p \vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \geq (\vec{l}' - \vec{u}') x_j \} \end{aligned}$$

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

4.2 Subscript Translation

To find the exact side effects 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. 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 assumptions regarding the shape of the actual array.

Assumption 1 (Shape of the Actual Array) *Given actual array, B,*

1. *The number of dimensions of B is equal or larger than that of the formal array A.*
2. *The sizes of the first $d - 1$ dimensions of the actual array, B, is the same as those of the template formal array A. The size of the d -th dimension of the actual array is no less than that of the template formal array.*

These assumptions should be satisfied in most real FORTRAN programs. Let the actual array be declared as $B(\tilde{l}_1 : \tilde{u}_1, \dots, \tilde{l}_d : \tilde{u}_d, \dots)$ in the calling subroutine. Let the actual for the array name be $B(s_1, \dots, s_d, s_{d+1} \dots)$. The situation is illustrated in Figure 3(a).

According to Assumption 1, we have

$$\tilde{u}_k - \tilde{l}_k + 1 = u'_k - l'_k + 1 = m_k, k = 1, \dots, d - 1$$

Let m_0 be 1 and

$$M_k = m_0 \dots, m_{k-1}.$$

Given an array element in the template formal array, $A(\phi'_1, \dots, \phi'_d)$, its location in the memory map is

$$L_t(\phi'_1, \dots, \phi'_d) = (\phi'_1 - l'_1)M_1 + \dots + (\phi'_d - l'_d)M_d + 1.$$

Let $\tilde{\phi} = (\tilde{\phi}_1, \dots, \tilde{\phi}_d)$ be the first d subscripts of the actual array. Due to the conditions in Assumption 1, the location of the element $B(\tilde{\phi}_1, \dots, \tilde{\phi}_d, s_{d+1}, \dots)$ in the memory map of the actual array is

$$qM + L_a(\tilde{\phi}_1, \dots, \tilde{\phi}_d)$$

where q is an integer determined by the subscripts in the array name actual for the dimensions higher than the d -th, M is the size of the subarray by the first d dimensions and

$$L_a(\tilde{\phi}_1, \dots, \tilde{\phi}_d) = (\tilde{\phi}_1 - \tilde{l}_1)M_1 + \dots + (\tilde{\phi}_d - \tilde{l}_d)M_d + 1.$$

The binding between the formal array name and its actual, $B(s_1, \dots, s_d, \dots)$, makes the first element of the the template formal array be located at location $qM + L_a(s_1, \dots, s_d)$ in the memory map of the actual array. Let $\vec{s}$ be $(s_1, \dots, s_d)$. To relate $\vec{\phi}'$ to $\tilde{\phi}$, we only need to note that $L_t(\vec{\phi}')$ and $qM + L_a(\tilde{\phi})$ are in the same location of the memory as shown in Figure 3(b) and we have

$$qM + L_a(\tilde{\phi}) = qM + L_a(\vec{s}) + L_t(\vec{\phi}') - 1 \quad (2)$$

Hence,

$$\sum_{k=1}^d (\tilde{\phi}_k - s_k)M_k = \sum_{k=1}^d (\vec{\phi}'_k - l'_k)M_k$$

Thus,

$$(\tilde{\phi}_k - s_k) = (\vec{\phi}'_k - l'_k), k = 1, \dots, d$$

or

$$\tilde{\phi} = \vec{\phi}' + \vec{s} - \vec{l}' \quad (3)$$

Let $\vec{\delta} = \vec{s} - \vec{l}'$. The above observation can be summarized 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.

In our example, the assumptions about the actual array are 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 effects 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 effects 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 $(s_{d+1}, \dots)$ and the first d subscripts $\tilde{\phi} = (\tilde{\phi}_1, \dots, \tilde{\phi}_d)$ are specified by the following projection:*

$$\begin{aligned} \tilde{\mathcal{S}} = \{ \tilde{\phi} \in Z^d \mid & \exists (\vec{i}_{j_1}, \dots, \vec{i}_{j_r}, \vec{x}) \in Z^{n_{j_1} + \dots + n_{j_r} + p} s.t. \\ & x_1 + \dots + x_p = p - 1 \wedge \vec{0} \leq \vec{x} \leq \vec{1} \wedge \\ & \wedge_{j=1}^p B_j \vec{i}_j \leq \vec{b}_j' \wedge \\ & \wedge_{j=1}^p \tilde{\phi} - \vec{\delta} - F_j \vec{i}_j - \vec{c}_j' \leq (\tilde{u} - \tilde{l}) x_j \wedge \\ & \wedge_{j=1}^p \tilde{\phi} - \vec{\delta} - F_j \vec{i}_j - \vec{c}_j' \geq (\tilde{l} - \tilde{u}) x_j \} \end{aligned}$$

where $\vec{\delta}$ is the same as described in Lemma 1, $\vec{b}_j'$ and $\vec{c}_j'$ are the same as described in Definition 3, and $\tilde{l}$ and $\tilde{u}$ are the low and upper limit vectors for the first d dimensions for the first d dimensions of the actual array declared in the calling subroutine.

Proof: The template in Definition 3 defines the exact set of the elements of the template formal array accessed by the subroutine. The constraints involving $\vec{\phi}'$ are $\vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \leq (\vec{u}' - \vec{l}') x_j$ and $\vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \geq (\vec{l}' - \vec{u}') x_j$. According to Lemma 1, $\tilde{\phi} = \vec{\phi}' + \vec{\delta}$. Since there are no array references outside of the array limits, we have $\tilde{l} \leq \tilde{\phi} \leq \tilde{u}$. Hence, $\tilde{l} - \vec{\delta} \leq \vec{\phi}' \leq \tilde{u} - \vec{\delta}$. According to Theorem 1, we can replace $\vec{l}'$ and $\vec{u}'$ with $\tilde{l} - \vec{\delta}$ and $\tilde{u} - \vec{\delta}$, respectively, without changing the elements defined by the template. The constraints become $\vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \leq (\tilde{u} - \tilde{l}) x_j$ and $\vec{\phi}' - F_j \vec{i}_j - \vec{c}_j' \geq (\tilde{l} - \tilde{u}) x_j$. Further replacing the $\vec{\phi}'$ with $\tilde{\phi} - \vec{\delta}$ yields the projection described in the theorem. $\square$

For instance, the exact side effects of the instance of the call statement with a particular value of J in program LOOP3 is

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

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

5 Exact Interprocedural Analysis

The exact side effects on the actual array can be used for the exact interprocedural dependence analysis. The Omega test is a practical exact dependence test based on integer programming projection [5,6,7]. The representation of the exact side effects in this paper can be used in the Omega test. 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 multiple instances of it. The exact side effects specified by Theorem 2 is for a single instance of the call statement. Recall that the actuals for the integer parameters of the subroutine, $\vec{z} = (z_1, \dots, z_w)$, are affine functions of variables or integer parameters of the calling subroutine, $\vec{v} = (v_1, \dots, v_t)$. Let the first y elements of $\vec{v}$, $\vec{v}_y = (v_1, \dots, v_y)$ are variables in the calling subroutine and the rest are parameters. The variables are usually the index variables of the enclosing loops, or can be converted as such (*e.g.*, induction variables). Let us use $\tilde{\mathcal{S}}(\vec{v}_y)$ to denote the exact side effects of the instance of the call statement for a particular $\vec{v}_y$. We also assume that bound matrix and vector of the enclosing loops are B_y and $\vec{b}_y$, respectively. Hence, the values of the variables can be represented by the following convex set.

$$C = \{\vec{v}_y \in Z^y \mid B_y \vec{v}_y \leq \vec{b}_y\}.$$

Let us concentrate on the dependence test between a call statement and an array reference. The dependence test between two call statements can be worked out similarly. Suppose that the subscript function for the first d -dimensions of an array reference is $\tilde{\phi}^1 = F_x^1 \vec{v}_x + \vec{c}_x^1$ and the function for the remaining dimensions is $\tilde{\phi}^2 = F_x^2 \vec{v}_x + \vec{c}_x^2$. Let the bound matrix and vector of the loops enclosing 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 a write access³. Formally, we can have the following definition:

Definition 4 (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*

$$\begin{aligned} B_x \vec{v}_x &\leq \vec{b}_x \wedge B_y \vec{v}_y \leq \vec{b}_y \wedge \\ \tilde{\phi} &\in \tilde{\mathcal{S}}(\vec{v}_y) \wedge \\ \tilde{\phi} &= F_x^1 \vec{v}_x + \vec{c}_x^1 \wedge \\ \vec{s}^2 &= F_x^2 \vec{v}_x + \vec{c}_x^2 \end{aligned}$$

where $\vec{s}^2$ is the subscripts of the dimensions higher than the d -th of the actual for the

³To simplify the discussion, we do not address the issue of dependence killing and covering [7] in this paper.

array name, $(s_{d+1}, \dots)$, in the call statement, and everything else are as described in the above discussion, and

2. Either the reference is a write reference or the side effects $\tilde{S}$ is the write side effects.

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:

$$\begin{aligned} \{\tilde{\phi} \in Z^d \mid & \exists \vec{v}_x, \vec{v}_y, \vec{i}_{j_1}, \dots, \vec{i}_{j_r}, \vec{x} s.t. \\ & B_y \vec{v}_y \leq \vec{b}_y \wedge \\ & x_1 + \dots + x_p = p - 1 \wedge \vec{0} \leq \vec{x} \leq \vec{1} \wedge \\ & \wedge_{j=1}^p B_j \vec{i}_j \leq \vec{b}_j' \wedge \\ & \wedge_{j=1}^p \tilde{\phi} - \vec{\delta} - F_j \vec{i}_j - \vec{c}_j' \leq (\tilde{u} - \tilde{l})x_j \wedge \\ & \wedge_{j=1}^p \tilde{\phi} - \vec{\delta} - F_j \vec{i}_j - \vec{c}_j' \geq (\tilde{l} - \tilde{u})x_j \wedge \\ & B_x \vec{v}_x \leq \vec{b}_x \wedge \\ & \tilde{\phi} = F_x^1 \vec{v}_x + \vec{c}_x^1 \wedge \\ & \vec{s}^2 = F_x^2 \vec{v}_x + \vec{c}_x^2\} \end{aligned}$$

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 as described in Definition 4 and everything else are as described in Theorem 2.

2. Either the reference is a write reference or the side effects $\tilde{S}$ is the write side effects.

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

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

(a) Elements accessed by loop 1

$$\mathcal{D} = \{ (\Delta) \mid \exists(\bar{\phi}_1, \bar{\phi}_2, I, x_1, x_2, J, J_1, I_1) \in \mathbb{Z}^8 s.t. \\ 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge x_1 + x_2 = 1 \wedge \\ 1 \leq I \leq J \wedge 2 \leq J \leq 100 \wedge \\ -199x_1 \leq \bar{\phi}_1 - 2I \leq 199x_1 \wedge \\ -99x_1 \leq \bar{\phi}_2 - 1 \leq 99x_1 \wedge \\ -199x_2 \leq \bar{\phi}_1 + 2I - 2J - 1 \leq 199x_2 \wedge \\ -99x_2 \leq \bar{\phi}_2 - I \leq 99x_2 \wedge \\ 2 \leq J_1 \leq 100 \wedge 2 \leq I_1 \leq J_1 \wedge \\ \bar{\phi}_1 = 2I_1 \wedge \bar{\phi}_2 = I_1 \wedge \\ \Delta = J_1 - J \}$$

(b) Omega test for loop 1

(c) Elements accessed by loop 2

$$\mathcal{D} = \{ (\Delta) \mid \exists(\bar{\phi}_1, \bar{\phi}_2, I, x_1, x_2, J, J_1, I_1) \in \mathbb{Z}^8 s.t. \\ 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge x_1 + x_2 = 1 \wedge \\ 1 \leq I \leq J \wedge 2 \leq J \leq 100 \wedge \\ -199x_1 \leq \bar{\phi}_1 - 2I \leq 199x_1 \wedge \\ -99x_1 \leq \bar{\phi}_2 - 1 \leq 99x_1 \wedge \\ -199x_2 \leq \bar{\phi}_1 + 2I - 2J - 1 \leq 199x_2 \wedge \\ -99x_2 \leq \bar{\phi}_2 - I \leq 99x_2 \wedge \\ 2 \leq J_1 \leq 100 \wedge 1 \leq I_1 \leq J_1 - 1 \wedge \\ \bar{\phi}_1 = -2I_1 + 2J_1 - 1 \wedge \bar{\phi}_2 = I_1 \wedge \\ \Delta = J_1 - J \}$$

(d) Omega test for loop 2

(e) Elements accessed by loop 3

$$\mathcal{D} = \{ (\Delta) \mid \exists(\bar{\phi}_1, \bar{\phi}_2, I, x_1, x_2, J, J_1, I_1) \in \mathbb{Z}^8 s.t. \\ 0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge x_1 + x_2 = 1 \wedge \\ 1 \leq I \leq J \wedge 2 \leq J \leq 100 \wedge \\ -199x_1 \leq \bar{\phi}_1 - 2I \leq 199x_1 \wedge \\ -100x_1 \leq \bar{\phi}_2 + 1 - 1 \leq 100x_1 \wedge \\ -199x_2 \leq \bar{\phi}_1 + 2I - 2J - 1 \leq 199x_2 \wedge \\ -100x_2 \leq \bar{\phi}_2 + 1 - I \leq 100x_2 \wedge \\ 2 \leq J_1 \leq 100 \wedge 1 \leq I_1 \leq J_1 - 1 \wedge \\ \bar{\phi}_1 = -2I_1 + 2J_1 - 1 \wedge \bar{\phi}_2 = I_1 \wedge \\ \Delta = J_1 - J \}$$

(f) Omega test for loop 3

Figure 4. Interprocedural Omega Tests

To find distance vectors of the dependences, we only need to project the integer programming problem on the differences of the index variables of the common enclosing loops. The details of the technique can be found in [5,6]. 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 projections for the distance vectors of the dependences in programs LOOP1, LOOP2 and LOOP3 are shown in Figure 4 (b), (d) and (f), respectively. We have run the exact Omega tests on these projections and the results are:

1. There are no dependences from the call statement to the assignment in program LOOP1. Therefore, loop J can be executed as a DOALL loop⁴.
2. There are data dependences from the call statement to the assignment statement in program LOOP2. The distance vector is $(\Delta) = (1)$. Loop J has to remain serial.
3. There are data dependences from the call statement to the assignment statement in program LOOP3. The distance vector is $(\Delta) = (0)$. Since the dependences do not cross iterations, loop J 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 4 (a), (c) and (e) show the elements of array B accessed in programs LOOP1, LOOP2 and LOOP3, respectively, when $J = 5$. The shaded small squares show the side effects of the subroutine call and the round dark dots show the elements read by the assignment statement. It can be seen from these figures that all the three interprocedural dependence tests are accurate. Notice that the accuracy of these tests cannot be achieved if the approximate summaries of side effects shown in Figure 2 were used.

6 Merging Side Effects and Reference Images

We have shown how to form the exact side effects of a subroutine call and use them for interprocedural dependence analysis. The calling subroutine can itself be called by another subroutine. Figure 5 shows two subroutines and one main program: program MAIN calls subroutine LOOP which, in turn, calls subroutine TOUCH. To determine the side effects on array C of subroutine call in program MAIN, both the write reference of array B and the call statement in subroutine LOOP need to be considered. We can merge the side effects of the subroutine call of TOUCH with the image of the write reference. The merged image then can be propagated back further to program MAIN.

The method of merging side effects of call statements with images of array references is similar to the method of merging array reference images described in Section 3. Instead

⁴A 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.

<pre> 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 </pre> <p style="text-align: center;">(a) main program</p>	<pre> 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 </pre> <p style="text-align: center;">(b) Subroutine LOOP</p>	<pre> SUBROUTINE TOUCH(A,N,K) REAL A(2*N,N) DO 10 I=1, K B(2*I, 1) = ... B(-2*I+2*K+1,I) = .. 10 CONTINUE END </pre> <p style="text-align: center;">(c) Subroutine TOUCH</p>
--	--	--

Figure 5. Serial subroutine calls

of going through the theorem with formal notations, we are going to use the example in Figure 5 to show the method.

According to Theorem 2, the exact side effects of the subroutine call of TOUCH is

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

Here, J is a variable and the convex set for its values is defined by $1 \leq J \leq L$. The index vector for the enclosing loops of the write reference is (J_1, I_1) ⁵. The convex set for the write reference is $\{1 \leq J_1 \leq L, 1 \leq I_1 \leq J_1\}$. To merge the side effects of all instances of the call statement with the image of the reference, we introduce y_1 and y_2 such that $y_1 + y_2 = 1$ and $0 \leq y_1, y_2 \leq 1$. If y_1 is used for the reference and y_2 for the side effects, the merged image is:

$$\begin{aligned}
\tilde{S} = \{(\tilde{\phi}_1, \tilde{\phi}_2) \mid & \exists(I, x_1, x_2, J_1, I_1, y_1, y_2) \in Z^7 s.t. \\
& 0 \leq y_1 \leq 1 \wedge 0 \leq y_2 \leq 1 \wedge \\
& y_1 + y_2 = 1 \wedge \\
& 1 \leq J_1 \leq L \wedge 1 \leq I_1 \leq J_1 \wedge \\
& (1 - 2M)y_1 \leq \tilde{\phi}_1 - 2J_1 + 1 \leq (2M - 1)y_1 \wedge \\
& (1 - M)y_1 \leq \tilde{\phi}_2 - I_1 \leq (M - 1)y_1 \wedge
\end{aligned}$$

⁵Note that the call statement and the write reference belong to the different loop bodies. Therefore the index vectors for them are different.

$$\begin{aligned}
&1 \leq J \leq L \wedge \\
&0 \leq x_1 \leq 1 \wedge 0 \leq x_2 \leq 1 \wedge \\
&x_1 + x_2 = 1 \wedge 1 \leq I \leq J \wedge \\
&(1 - 2M)(x_1 + y_2) \leq \tilde{\phi}_1 - 2I \leq (2M - 1)(x_1 + y_2) \wedge \\
&(1 - M)(x_1 + y_2) \leq \tilde{\phi}_2 - 1 \leq (M - 1)(x_1 + y_2) \wedge \\
&(1 - 2M)(x_2 + y_2) \leq \tilde{\phi}_1 + 2I - 2J - 1 \leq (2M - 1)(x_2 + y_2) \wedge \\
&(1 - M)(x_2 + y_2) \leq \tilde{\phi}_2 - I \leq (M - 1)(x_2 + y_2) \}
\end{aligned}$$

The merged image of the reference and the side effects of subroutine call of **TOUCH** can be propagated back further to program **MAIN** following the same algorithm described in Section 4. As we can see, the side effects can be merged with all the references or other side effects in the back propagations all the way to the entry of the call graph.

We can also choose to merge array references only and let the side effects be propagated back alone. For instance, the side effects of subroutine call of **TOUCH** can be further propagated back alone using the following projection:

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

Note the constraint $1 \leq J \leq L$ in the above projection. This means that the above projection represent the side effects of *all* instances of the call statement in subroutine **LOOP**.

In essence, merging all references including all side effects of subroutine calls and not merging at all using atom images [8,9] are the two extremes on the spectrum of the degree of merging array side effects for dependence analysis. To achieve the best performance, the balance between the number of reference pairs for dependence test and the size of the projections for the side effects need to be determined. We are currently incorporating the Omega test into our system and implementing the merging algorithms of this paper. The experiments on real FORTRAN programs to find the cost of merging and the best balance will be carried out in due course.

7 Concluding Remarks

We have presented a method to merge the images of multiple references of arrays in subroutines. The merged image is represented in the form a single integer programming projection and is exact. The idea is to introduce integer variables with values 0 or 1 and

use the array limits as the common bounds for the references.

We have presented the algorithms to propagate the merged image back to the calling subroutine and form the projection for the exact side effects on the actual arrays.

We have showed how the projection for the exact side effects can be incorporated into the Omega tests for the exact interprocedural dependence analysis.

If the calling subroutine is itself called by other subroutines, the exact side effects can also be merged with the array references in the calling subroutine or other side effects using the extended method based on the same idea.

We are incorporating the Omega test into our system to evaluate the performance for the proposed exact side effects for interprocedural analysis on real FORTRAN programs. The degree of merging side effects will also be investigated to find the best balance between the number of reference pairs and the size of integer programming projections.

8 Acknowledgement

The Omega tests in this paper are run on the Omega System of W. Pugh from the University of Maryland. I would like to thank W. Pugh for making the Omega System freely 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, Oregon, 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] 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.
- [9] Z. Li and P-C. Yew, “Program Parallelization with Interprocedural Analysis,” *The Journal of Supercomputing*, vol. 2, no. , pp. 225–244, 1988.
- [10] 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.
- [11] 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.
- [12] M. Wolfe, *Optimizing Compilers for Supercomputers*. Cambridge, MA, MIT Press, 1989.
- [13] U. Banerjee, “A Theory of Loop Permutations,” in *Proceedings of the 2nd Workshop on Languages and Compilers for Parallel Computing*, , August 1989, pp. 54–75.
- [14] 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*, Williamsburg, Virginia, April 21-24, 1991, pp. 39–50.
- [15] H. M. Salkin, *Integer Programming*. Addison-Wesley Publishing Company, 1975.
- [16] D. M. Morno, *A Crash Course in FORTRAN 77*. London, Edward Arnold, Publishing Division of Hodder and Stoughton Limited, 1989.