William Pugh

A PRACTICAL ALGORITHM for Exact Array Dependence Analysis

A fundamental analysis step in an advanced optimizing compiler (as well as many other software tools) is data dependence analysis for arrays. This means deciding if two references to an array can refer to the same element and if so, under what conditions. This information is used to determine allowable program transformations and optimizations. For example, we can determine that in the following code fragment, no location of the array is both read and written. Once we also verify that no location is written more than once, we know that the writes can be done in any order.

```
for i = 1 to 100 do
  for j = i to 100 do
    A[i, j+1] = A[100,j]
```

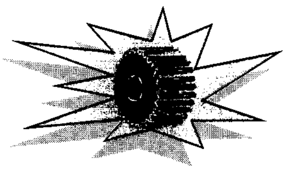
There has been extensive study of decision methods for array data dependences [1, 2, 5, 6, 8, 15, 18, 25]. Much of this work has focused on approximate methods that are guaranteed to be fast but only compute exact results in (commonly occurring) special cases. In other situations, approximate methods are conservative. They accurately report all actual dependences, but may also report spurious dependences.

Data dependency problems are equivalent to deciding whether there exists an integer solution to a set of linear equalities and inequalities, a form of integer programming. The problem as just shown would be formulated as an integer programming problem in the next example. In this example, i_w and j_w refer to the values of the loop variables at the time the write is performed and i_r and j_r refer to the values of the loop variables at the time the read is performed.

$$1 \le i_w \le j_w \le 100$$
$$1 \le i_r \le j_r \le 100$$
$$i_w = 100$$
$$j_w + 1 = j_r$$

Convention holds that integer programming techniques are far too expensive to be used for dependence analysis, except as a method of last resort for situations that cannot be decided by simpler, special-case methods. We present evidence that suggests this argument is wrong. We will describe the Omega test, which determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities. Our experiments that suggest that, for almost all programs, the average time required by the Omega test to determine the direction vectors for an array pair is less than 500 μsecs on a 12-MIPS workstation. We also found that the time required by the Omega test to analyze a problem is rarely more than twice the time required to scan the array subscripts and loop bounds. This would indicate that the Omega test is suitable for use in production compilers.

Conceptually, the Omega test combines new methods for eliminating equality constraints with an extension of Fourier-Motzkin variable elimination to integer programming. At a more detailed level, the Omega test also incorporates several implementation details



(described later in this article) that produce substantial speed improvements in practice.

Integer programming is a NP-Complete problem, and the Omega test has exponential worst-case time complexity. We will show that in many situations in which other (polynomial) methods are accurate, the Omega test has low-order polynomial worst-case time complexity.

Dependence analysis is often structured as a decision problem: tests simply answer yes or no. Compilers and other program restructuring tools need to know the data dependence direction vector [24] and data dependence distance vector [13, 19] that describe the relation between the iterations in which reads and writes of a particular array element occur. The data dependence distance vector describes the differences between the values of the common loop variables between the first and second access to the same array element. For example, in the following code fragment, the dependence distance of the flow dependence is (1,2):

```
for i := 1 to n do
  for j := 1 to m do
    A(i, j) := A(i−1, j−2)
```

Sometimes, dependence distance is not constant. In these cases, the dependence direction vector describes the possible combinations of signs of dependence distances.

Determining dependence direction vectors may require an exponential number call to a dependence testing algorithm that only returns yes/no. To be competitive, a dependence analysis method must be able to short-cut this enumeration process (e.g., [6, 8]). Later we will show how the Omega test can be modified to *project* integer programming problems onto a subset of the variables, rather than just *deciding* them. With this knowledge, we can efficiently produce a set of constraints that precisely and concisely describes all possible dependency distance vectors. This information can be used directly in deciding the validity of program transformations, or standard direction and distance vectors can be quickly computed from it. These techniques are described in the section on dependence direction and distance vectors.

The Omega Test

The Omega test determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities, referred to as a problem. The input to the Omega test is a set of linear equalities ($\Sigma_{1\le i\le n}\, a_i x_i = c$) and inequalities ($\Sigma_{1\le i\le n}\, a_i x_i \ge c$). To simplify our presentation (and our algorithms), we define $x_0 = 1$ and use $\Sigma_{0\le i\le n}\, a_i x_i = 0$ and $\Sigma_{0\le i\le n}\, a_i x_i \ge 0$ as our standard representations, and we use V to denote the set of indices of the variables being manipulated (i.e., $V = \{i | 0 \le i \le n\}$).

Normalizing (and Tightening) Constraints

Throughout this article, we assume that any constraint we are manipulating has been normalized. A normalized constraint is one in which all the coefficients are integers and the greatest common divisor of the coefficients (not including a_0) is 1.

If the initial constraints involve rational coefficients, they can be scaled to obtain integer coefficients (the algorithms described here do not produce any noninteger coefficients).

To normalize a constraint, we compute the greatest common divisor g of the coefficients $a_1, \ldots, a_n$. We then divide all the coefficients by g. If the constraint is an equality constraint and g does not evenly divide a_0, the constraint is unsatisfiable. If the constraint is an inequality constraint, we take the floor when dividing a_0 by g (i.e., we replace a_0 with $\lfloor a_0/g \rfloor$).

Taking floors in the constant term tightens the inequalities. If a problem P has rational but not integer solutions, tightening P may produce a problem without rational solutions, thus making it easier to determine that P has no integer solutions.

Equality Constraints

Given a problem involving equality and inequality constraints, we first eliminate all the equality constraints, producing a new problem of inequality constraints that has integer solutions if and only if the original problem had integer solutions. Of course, in the process we might decide that the problem has no integer solutions regardless of the inequality constraints.

Banerjee's Generalized Greatest Common Divisor (GCD) test [5] can be used to eliminate integer equality constraints. We found, however, the following approach better suited to our needs, since it is somewhat simpler and more appropriate for situations in which additional equalities may be added later.

To eliminate the equality $\Sigma_{i\in V}\, a_i x_i = 0$, we first check if there exists a $j \ne 0$ such that $|a_j| = 1$. If so, we eliminate the constraint by solving for x_j and substitute the result into all other constraints.

Otherwise, let k be the index of the variable with the coefficient that has the smallest absolute value ($k \ne 0$) and let $m = |a_k| + 1$. We define $\widehat{\text{mod}}$ as follows:

$$a \,\widehat{\text{mod}}\, b = a - b\lfloor a/b + 1/2 \rfloor$$

We create a new variable σ and produce the constraint:

$$m\sigma = \sum_{i\in V} (a_i \,\widehat{\text{mod}}\, m) x_i$$

Note that $a_k \,\widehat{\text{mod}}\, m = -\text{sign}(a_k)$. We then solve this constraint for x_k

$$x_k = -\text{sign}(a_k)m\sigma + \sum_{i\in V-\{k\}} \text{sign}(a_k)(a_i \,\widehat{\text{mod}}\, m)x_i$$

and substitute the result in all constraints. In the original constraint, this substitution produces:

$$-|a_k|m\sigma + \sum_{i\in V-\{k\}} (a_i \cdot + |a_k|(a_i \widehat{\text{mod}} m))x_i = 0$$

Since $|a_k| = m - 1$, this is equal to

$$-|a_k|m\sigma + \sum_{i\in V-\{k\}} ((a_i - (a_i \mathbin{\widehat{\bmod}} m)) + m(a_i \mathbin{\widehat{\bmod}} m))x_i = 0$$

Since all terms are now divisible by m, normalizing the constraint produces:

$$-|a_k|\sigma + \sum_{i\in V-\{k\}} \left(\left\lfloor a_i/m + \frac{1}{2}\right\rfloor + (a_i \mathbin{\widehat{\bmod}} m)\right)x_i = 0$$

In the original constraint, the absolute value of the coefficient of σ is the same as the absolute value of the original coefficient of x_k. For all other variables, the absolute value of coefficients are reduced to at most two-thirds of their previous value. Therefore, repeated applications of this rule will eventually force a unit coefficient to appear and allow us to eliminate the constraint. An application of these methods is shown in Figure 1.

Figure 1. Example of elimination of equality constraints

substitution	resulting constraints
Original problem	$7x + 12y + 31z = 17$ $3x + 5y + 14z = 7$ $1 \leq x \leq 40$ $-50 \leq y \leq 50$
$x = -8\sigma - 4y - z - 1$	$-7\sigma - 2y + 3z = 3$ $-24\sigma - 7y + 11z = 10$ $1 \leq -8\sigma - 4y - z - 1 \leq 40$ $-50 \leq y \leq 50$
$y = \sigma + 3\tau$	$-3\sigma - 2\tau + z = 1$ $-31\sigma - 21\tau + 11z = 10$ $1 \leq -1 - 12\sigma - 12\tau - z \leq 40$ $-50 \leq \sigma + 3\tau \leq 50$
$z = 3\sigma + 2\tau + 1$	$2\sigma + \tau + \; = -1$ $1 \leq -2 - 15\sigma - 14\tau \leq 40$ $-50 \leq \sigma + 3\tau \leq 50$
$\tau = -2\sigma - 1$	$1 \leq 12 + 13\sigma \leq 40$ $-50 \leq -3 - 5\sigma \leq 50$
after normalization	$0 \leq \sigma \leq 2$

Inequality Constraints

The following process is used once all equality constraints have been eliminated. We first check to see if any two inequality constraints directly contradict one another (e.g., the constraints $3x + 5y \geq 2$ and $3x + 5y \leq 0$). If we find a contradiction, we report that the problem has no solutions. We can deal with equality constraints more efficiently than inequality constraints. Therefore, if we find a pair of tight inequalities (such as $6 \leq 3x + 2y$ and $3x + 2y \leq 6$), we replace them with the appropriate equality constraint and revert to our methods for dealing with equality constraints. While checking for contradictory pairs of constraints, we also eliminate constraints that are made redundant by a single other constraint (e.g., $x + 2y \geq 0$ is made redundant by $x + 2y \geq 5$).

If the problem involves at most one variable and has passed the above tests, we report that it has integer solutions. Otherwise, we reduce the problem to one or more integer programming problems in fewer dimensions and repeat the above process, eventually getting to problems in one dimension.

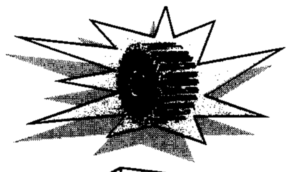
Detecting real solutions using Fourier-Motzkin variable elimination. Fourier-Motzkin variable elimination [7] eliminates a variable from a linear programming problem. Intuitively, Fourier-Motzkin variable elimination finds the $n - 1$ dimensional shadow cast by an n dimensional object.

Consider the dodecahedron in Figure 2a. We want to calculate the shadow of the dodecahedron when it is projected along the z dimension onto the xy plane (as shown). This dodecahedron and its shadow can each be specified by a set of 12 constraints (Figure 2b).

Consider two constraints on z: a lower bound $\beta \leq bz$ and an upper bound $az \leq \alpha$ (where a and b are positive integers). We can combine these constraints to get $a\beta \leq abz \leq b\alpha$. The constraint $a\beta \leq b\alpha$ is the shadow of the intersection of these two constraints (shown visually in Figure 2c). By combining the shadow of the intersection of each pair of upper and lower bounds on z (Figures 2d and 2e), we obtain a set of constraints that defines the shadow of the original object.

Since the shadow obtained this way describes real solutions, we refer to it as the *real shadow* of a set of constraints. If there are no integer points in the real shadow of a set of constraints, we know that there are no integer solutions to the set of constraints.

Note that the set of constraints we obtained includes many redundant constraints. Performing Fourier-Motzkin variable elimination can square the number of constraints and produce many redundant constraints. Few loop nests, however, have dodecahedrons for iteration spaces, and in practice the number of constraints does not grow substantially. Attempting to determine which constraints are redundant so as to remove them from consideration is usually not cost-effective.



Detecting integer solutions using Fourier-Motzkin variable elimination. There may be integer grid points in the shadow of an object, even if the object itself contains no integer points (consider the shadow of a very thin object). Ideally, we would like to calculate the *integer shadow* of an object: a shadow such that for every integer point in the shadow, there is at least one corresponding integer point in the object above it, and vise-versa. Unfortunately, we cannot always do this. The integer shadow of a convex region may not even be a convex region. We have, however, developed new methods that work well in practice. Our approach can be (informally) visualized as finding the shadow of a translucent object: thicker parts of the object cast a darker shadow. If we define *dark shadows* appropriately, we can guarantee that for every integer point in the dark shadow, there is an integer point in the object above it.

As an example, we reconsider our previous example of the dodecahedron, although we flatten the dodecahedron to illustrate the difficulty of finding integer shadows. The flattened dodecahedron is shown in Figure 3a, and the integer points in the dodecahedron and its shadow are shown in Figures 3b and 3c. There are integer points in the real shadow that have no integer point in the object above them. For every integer point in the dark shadow, however, there is an integer point in the object above it.

The shadow is clearly dark below any part of the object that is at least one unit thick. Since the coefficients of the constraints are integers, however, we can determine a

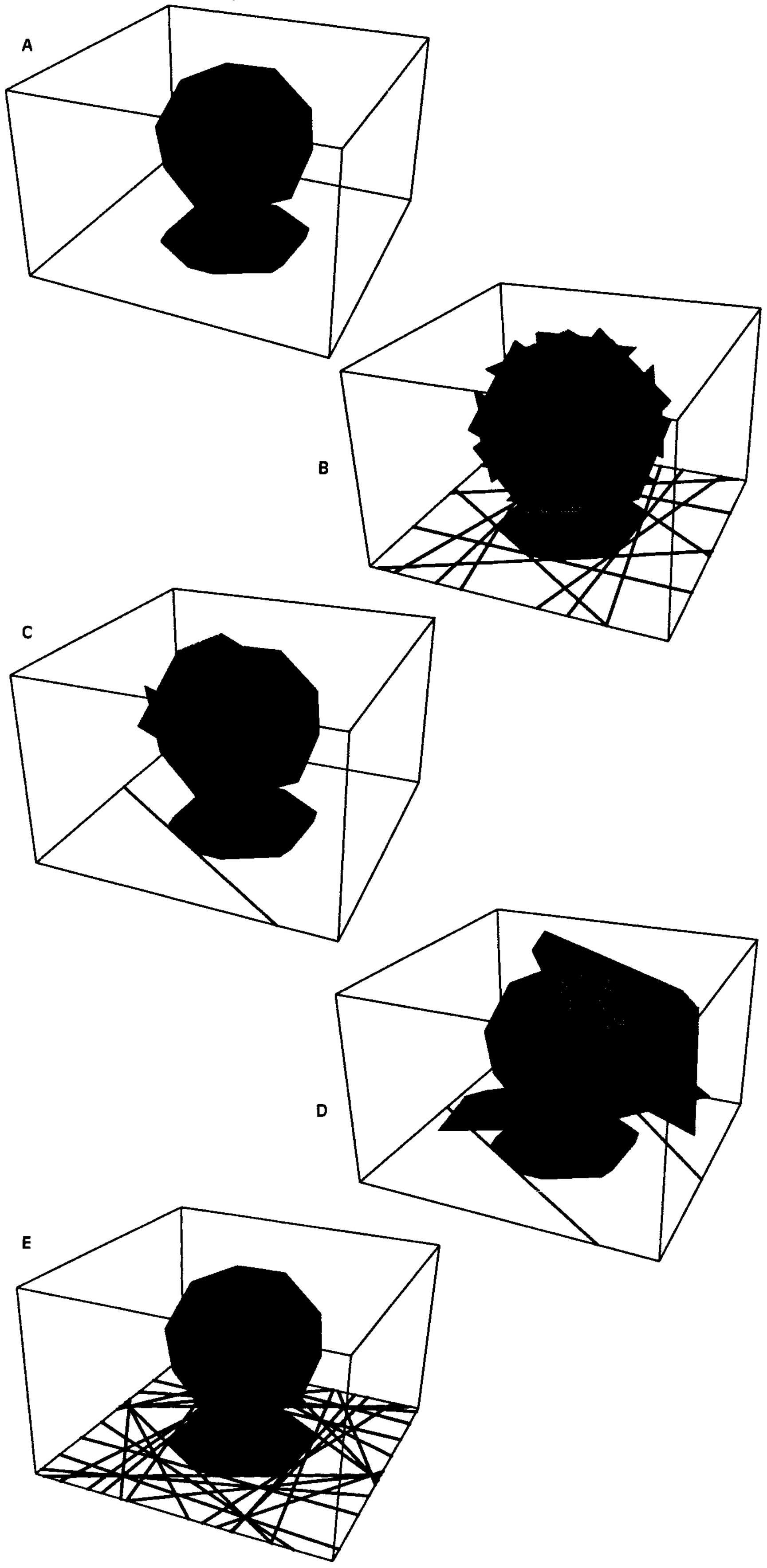


Figure 2. A visual depiction of Fourier-Motzkin variable elimination (a) A dodecahedron and its shadow (b) The constraints that specify a dodecahedron and its shadow (c) Finding the shadow of the intersection of two constraints (d) Finding the shadow of the intersection of two more constraints (e) Constraints resulting from the combination of all pairs of upper and lower bounds (most are redundant)

looser definition of dark that will still guarantee that any integer point in the dark shadow has an integer point above it.

To determine the dark shadow, consider the case in which there is an integer solution to $a\beta \le b\alpha$, but there is no integer solution to $a\beta \le abz \le b\alpha$ (i.e., there is no multiple of ab between $a\beta$ and $b\alpha$). Note that a and b are positive integers. In this case, let $i = \lfloor \beta/b \rfloor$. Then

$$ab\,i < a\beta \le b\alpha < ab\,(i + 1)$$

Since $ab(i + 1) - b\alpha \ge b$, $a\beta - abi \ge a$ and $ab(i + 1) - abi = ab$, $b\alpha - a\beta \le ab - a - b$. If $b\alpha - a\beta \ge ab - a - b + 1 = (a - 1)(b - 1)$, we know that there must be an *integer* solution to z. Therefore, the dark shadow of $\alpha \ge az$ and $bz \ge \beta$ is:

$$b\alpha - a\beta > (a - 1)(b - 1)$$

Note that if $a = 1$ or $b = 1$, the dark shadow and the real shadow are identical. If the dark and real shadow are identical, the projection is called an *exact* projection. This will happen, for example, if all of the coefficients of z in lower bounds on z are 1, or if all of the coefficients of z in upper bounds on z are 1. For the problems that arise in dependence analysis, we can almost always find an exact projection.

We now have a method for checking for the existence of integer solutions to a set of constraints:

1. We first decide which variable to eliminate. We choose this variable to perform an exact projection if possible, and to minimize the number of constraints that result from the combination of upper and

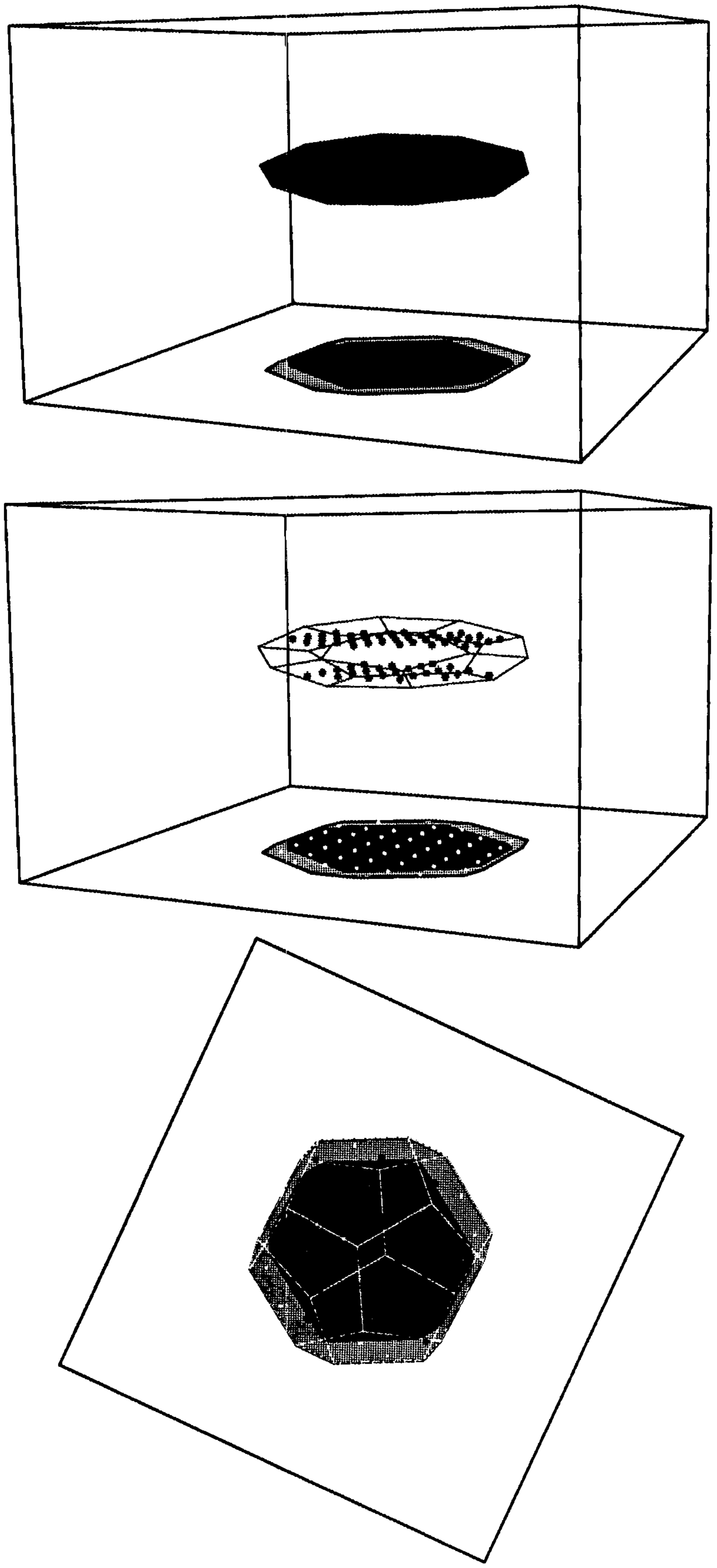
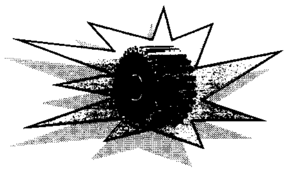


Figure 3. Checking for integer points in the dark shadow (a) The shadow cast by a translucent, flattened dodecahedron (b) View of the integer points inside a flattened dodecahedron, and inside its shadow (c) Overhead view of Figure 3b, showing that there are integer points within the flattened dodecahedron above every integer point in the dark shadow, but that there is not necessarily an integer point in the flattened dodecahedron above every integer point in the entire shadow (the real shadow).



lower bounds. If we are forced to perform nonexact reductions, we choose a variable with coefficients as close to zero as possible.

2. Calculate the real and dark shadows of the set of constraints along that dimension.

3. If the real and dark shadows are identical, there are integer solutions to the original set of constraints if there are integer solutions to the shadow.

4. Otherwise:

   (a) If there are no integers to the real shadow, we know there are no integer solutions to the original set of constraints.

   (b) If there are integer solutions to the dark shadow, we know there are integer solutions to the original constraints.

   (c) Otherwise, we know if an integer solution exists, it must be closely nestled between an upper bound and a lower bound. Therefore we consider a set of planes that are parallel to a lower bound and close to a lower bound. Any integer solution closely nestled between an upper bound and a lower bound must lie on one of these planes. Computationally, we analyze the problem as follows: We know that if there exists an integer solution to the original set of constraints, there must exist a pair of constraints $\alpha \geq az$ and $bz \geq \beta$ on z such that

   $$ab - a - b + a\beta \geq b\alpha \geq abz \geq a\beta$$

   We check this by determining the largest coefficient a of z in any upper bound on z, and, for each lower bound $bz \geq \beta$ on z, testing if there are integer solutions to the original problem combined with $bz = \beta + i$ for each i such that $(ab - a - b)/a \geq i \geq 0$. While these steps are expensive and complicated, they rarely, if ever, need to be used in practice.

An Omega test nightmare. To demonstrate (and show the limitations of) the techniques used, we illustrate the steps performed by the Omega test on an example designed to force the Omega test to work very hard for a small problem. Consider the inequalities P:

$$\begin{aligned} 27 \leq\ & 11x + 13y \leq 45 \\ -10 \leq\ & 7x - 9y \leq 4 \end{aligned}$$

There are no exact projections we can perform, and we would decide to eliminate x since the coefficients of x are (slightly) smaller. Figure 4a shows the constraints in the original problem, and the unnormalized constraints in the real and dark shadows. Since the real shadow has integer solutions but the dark shadow does not, we check if there are any integer solutions close to a lower bound. We do this by checking if the intersection of the original set of constraints and any one of the following constraints contains an integer point (this is shown graphically in Figure 4b). Since there are no such solutions, we know that no integer solutions exist.

$$7x = 9y - 10 + j \qquad 0 \leq j \leq \left\lfloor \frac{77 - 11 - 7}{11} \right\rfloor = 5$$

$$11x = 27 - 13y + j \qquad 0 \leq j \leq \left\lfloor \frac{121 - 11 - 11}{11} \right\rfloor = 9$$

The steps performed in this example appear complicated and expensive. This example, however, was *designed* to be expensive to resolve. We do not expect situations this difficult to arise frequently in practice. Also, although many steps are performed in this process, our implementation of the Omega test takes only 4.5 milliseconds on a 12-MIPS workstation to perform them all.

Worse nightmares are possible: on problems with only two variables and three constraints, the Omega test can take time proportional to the absolute value of the coefficients. While this is a frightening possibility, we do not expect these situations to arise frequently in practice.

A decision on better methods for dealing with Omega test nightmares will have to wait until more experience is gained about the type of nightmares that occur in practice.

Implementation Details

In implementing the Omega test we used several algorithmic ideas and tricks that substantially improved our running time. We report some of those ideas here.

Equalities and inequalities are represented as vectors of coefficients. The Omega test is crafted so that the algorithms only need to deal with integers; no rational number representation scheme needs to be used.

Once we have eliminated all the equality constraints from a problem, we check for any variables that have no lower bounds or have no upper bounds. We refer to such variables as unbounded variables. Performing Fourier-Motzkin elimination on an unbounded variable simply deletes all the constraints involving it. We delete all constraints involving unbounded variables. It is then checked to see if that has produced additional unbounded variables. We repeat this process until no unbounded variables remain.

Next, we normalize all the constraints and then assign hash keys and constraint keys to them. We only do this to constraints that have been modified since the last time they were normalized. The constraint key of a constraint is a unique tag based on the coefficients of the variables in the constraint; two constraints have equal constraint keys if and only if they differ only in their constant term. Constraint keys are both negative and positive, and the key of a constraint e_1 is the negation of the key of a constraint e_2 if and only if the coefficients of the variables in e_1 are the negation of the coefficients of the variables in e_2. We refer to this as opposing keys and opposing constraints. Constraint keys are assigned to constraints in constant expected time by recording, in a hash table, constraint keys previously assigned. We compute a hash key based on the coefficients of the constraint as an index into the hash table (hash keys are not guaranteed to be unique). Our method for computing hash keys is designed so that opposing constraints have opposing hash keys, which makes it easy to assign them opposing constraint keys. As constraints are normalized, we enter them into a table based on their constraint key. This allows us to check for redundant, contradictory or tight constraint pairs in constant time per constraint.

In the process of normalizing constraints, we check to see if any constraints involve more than one variable. After normalization, if we found no multivariable constraints, we know the system must have solutions, and we return immediately.

Next, we examine the variables to decide which variable to eliminate. If we can perform an exact projection, we perform the elimination in place (adding and deleting constraints from the current problem). Otherwise, we copy the constraints with zero coefficients for the eliminated variable into two new problem data structures (for the real shadow and for the dark) and then add the constraints produced for Fourier-Motzkin elimination. Since the constraints generated for the real and dark shadow differ only in their constant terms, we can share much of the work in adding these constraints.

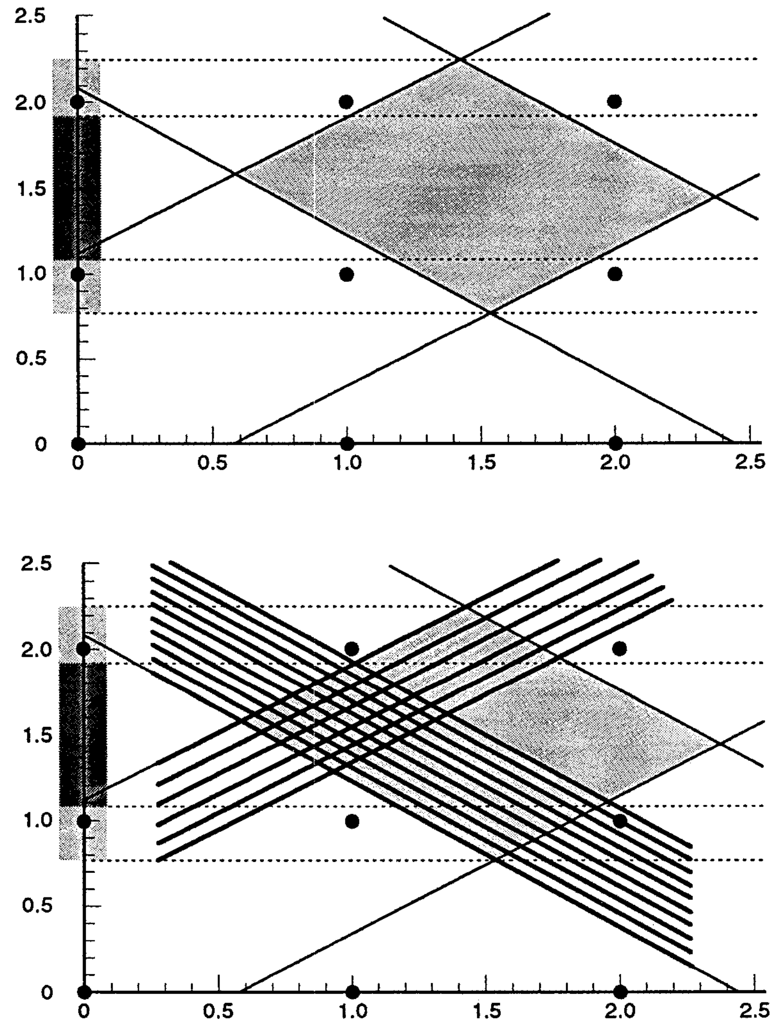


Figure 4. (a) Finding the real and dark shadow of an Omega test nightmare (b) Checking Figure 4a for solutions tightly nestled between an upper and lower bound

Nonlinear Subscripts

Integer programming dependence analysis methods allow us to properly handle symbolic constants [9, 16] and some types of min and max functions in loop bounds [27] and conditional assignments [17].

For example, even if we had no information about the value of n, we would like to be able to decide that there are no flow dependences in the following program:

```
for i = 1 to n do
   a[i+n] = a[i]
```

As previous authors have suggested, we can handle loop-invariant symbolic constants by adding them as additional variables to the integer programming problem. For example, the above problem would generate the following integer programming program (involving the variables *i_w*, *i_r* and *n*):

$$1 \le i\_w,\ i\_r \le n$$
$$i\_w + n = i\_r$$

We also can accommodate integer division and integer remainder operations, something that does not appear to have been previously recognized. Assume an expression e appears in a program that can be expressed as $e = \alpha$ div m where m is a positive integer. To handle this, we define a new variable σ, add the inequality constraints $0 \le \alpha - m\sigma \le m - 1$ and use σ as the value of e. Similarly, if $e = \alpha$ mod m we would add the same inequality constraint but use $\alpha - m\sigma$ as the value of e.

Projection of Integer Programming Problems

As described earlier, the Omega test simply *decides* if there is a solution to an integer programming problem. In this section, we describe how to adapt the Omega test to allow it to be used for *symbolic projection.* When used this way, the Omega test is given as input an integer programming program P and a designation of a set of protected variables $\hat{V} \subset V$. The Omega test projects P into one or more problems involving only variables in $\hat{V}$ that describe all the possible values of the variables in $\hat{V}$ such that there is an integer solution to P with those values. For example, projecting the integer programming problem $\{0 \leq a \leq 5;\ b \leq a \leq 5b\}$ onto a produces the problem $\{2 \leq a \leq 5\}$.

Actually, results of the projection process can be slightly more complicated than just described. The results may not be in terms of the variables in $\hat{V}$. Instead, the results are given in terms of a set $\hat{V}'$ of not more than $|\hat{V}|$ variables (possibly including new variables), along with methods for calculating the appropriate values for the values of $\hat{V}$ from the values of $\hat{V}'$. For example, if asked to project the integer programming problem $\{a = 10b + 25c;\ a \geq 13\}$ onto a, the Omega test will produce $\{\sigma \geq 3;\ a = 5\sigma\}$.

The projection process may produce multiple problems. For example, projecting the problem $\{5b \leq a \leq 6b\}$ onto a produces:

$$\{20 \leq a\}$$
$$\{0 \leq \sigma;\ a = 6\sigma\}$$
$$\{1 \leq \sigma;\ a = 6\sigma - 1\}$$
$$\{2 \leq \sigma;\ a = 6\sigma - 2\}$$
$$\{3 \leq \sigma;\ a = 6\sigma - 3\}$$

Changes to the Omega Test

Three of the changes required are simple, the other is not as simple. The quick changes are:

- If the current problem P involves only protected variables, check to see if there are integer solutions of P and if so, report P as one projection.
- When performing an inexact Fourier-Motzkin elimination, project the dark shadow and the intersection of the original constraints with all of the equality checks near the lower bounds. In other words, we must project all of the subproblems where we would look for an integer solution, not stopping when an integer solution is first verified. This might be expensive if projecting a system involved many inexact projections. We do not believe this will occur in practice for the problems arising from dependency analysis.
- We never perform Fourier-Motzkin variable elimination on a protected variable. This could require us to perform an inexact projection in a situation where we could have performed an exact projection if we were not protecting certain variables.

The not so simple change involves equalities. Given an equality constraint $\Sigma_{i \in V}\, a_i x_i = 0$, let g be the GCD of the coefficients of the nonprotected variables (we always assume that the constraint is normalized).

- If $g = 0$, the constraint involves only protected variables. We use our standard methods to eliminate the constraint. This will result in the elimination of a protected variable. All substitutions performed in this process are recorded in a substitution log. These substitutions involve only protected variables.
- If $g = 1$, we use our standard techniques (outlined in the section on equality constraints) to find a substitution involving only unprotected variables that simplifies or eliminates the constraint.
- If $g > 1$, we create a new protected variable σ, add the constraint:

$$g\sigma = \sum_{i \in V} (a_i \ \widehat{\bmod}\ g) x_i$$

Eliminating this new constraint will transform the original constraint so that the GCD of the nonprotected variables is 1 (after normalization).

When we report a projection, any substitutions involving protected variables are translated back into equality constraints.

Projection with Wildcards

As a modification of the approach just described, we could refuse to perform inexact reductions while performing projection. The advantage of this is that we only report one projected problem as our result. The disadvantage is that the projected problem has additional variables (that should be treated as wildcards).

In the applications we have found for projection, we have found projection with wildcards to be more useful than producing multiple results.

Using Projection

This projection technique can be used for several purposes. We describe some that have occurred to us.

Dependence direction and distance vectors

One problem with some dependence analysis methods is that they are only "yes/no" decision methods. In compilers and other program-structuring tools, we need to know the data dependence direction vector [24] and data dependence distance vector [13, 19] that describe the relation between the iterations in which the conflicting reads/writes occur. One way to determine dependence direction vectors is to make 3^L calls to a decision procedure (where L is the number of loops surrounding both references). In order to be competitive, a dependence analysis method must be able to short-cut this enumeration (e.g., [6, 8]).

In our method, we take the integer programming problem for determining if any dependence exists between two references, and introduce a new variable for the dependence distance in each shared loop (along with the appropriate equality constraints to define the value of the variable). We then project the problem onto the dependence distance variables. The projected system may be a better way to describe dependence conditions than dependence directions and distances; it accurately describes more information than is typically contained in dependence direction vectors (such as when a dependence distance is always greater than 5).

Alternatively, we can use the projected set of constraints to determine efficiently the dependence direction and distance vectors. We scan the dependences, and infer as much information as possible from constraints involving a single dependence distance variable. We next unprotect any dependence distance variable that is uncoupled or with a sign that is completely determined. If coupled variables were unprotected, we project the problem onto the protected variables and repeat this process. Otherwise, we choose one protected variable and generate the subproblems for two or three possible signs for the variable (negative, zero or positive), and recursively explore those.

For example, the dependence distances for the following array pair

```
for j = 0 to 20 do
for i = max(−j, −10) to 0 do
for k = max(−j, −10)−i to −1 do
for l = 0 to 5 do
a(l,i,j) = . . .
. . . = a(l,k,i+j)
```

simplify to:

$$
\begin{aligned}
&0 \le \Delta j \le 10\\
&\Delta i + \Delta j \le 10\\
&\Delta i + 2\Delta j \le 10\\
&3\Delta j + 2\Delta i + \Delta k \le 20\\
&2\Delta j + 2\Delta i + \Delta k \le 10\\
&1 \le \Delta j + \Delta i + k\\
&1 \le \Delta j + \Delta i\\
&\Delta l = 0
\end{aligned}
$$

We first unprotect Δl, and then consider sign(Δj) = 0 and sign(Δj) = 1. Considering sign(Δj) = 0 gives:

$$
\begin{aligned}
&1 \le \Delta i \le 10\\
&1 \le \Delta k + \Delta i\\
&\Delta k + 2\Delta i \le 10
\end{aligned}
$$

We would then unprotect Δi (since we know sign(Δi) = 1) and project the problem, obtaining $-8 \le \Delta k \le 8$, which gives a direction vector of (=,<,*,=).

Returning to consideration of sign(Δj) = 1 produces:

$$
\begin{aligned}
&-8 \le \Delta i \le 8\\
&-8 \le \Delta k \le 8\\
&-8 \le 2\Delta k + \Delta i\\
&-9 \le \Delta k + \Delta i\\
&\Delta k + 2\Delta i \le 8\\
&\Delta k - \Delta i \le 17
\end{aligned}
$$

Recursively analyzing the possibilities for the sign of Δi produces direction vectors of (<,>,*,=), (<,=,*,=) and (<,<,*,=). This example is the most difficult example seen in our testing, requiring 2,492 μsecs to analyze.

Run-time Checks and Compile-time Assertions

By projecting a problem onto the variables corresponding to symbolic constants that cannot be determined at compile-time, we can produce a predicate that will allow us to determine at run-time if a particular dependence or dependence direction exists (as described by [12]). Alternatively, at compile time we could ask the user if the predicate is true.

Summarizing Array References

In interprocedural analysis, we need to characterize the portions of an array that may be affected by a procedure call [4, 10, 11, 22]. We can use the Omega test to obtain an accurate summary of the locations of an array that might be affected by a single assignment statement. We do this by setting up an integer programming problem involving variables for each array index and all loop variables and symbolic constants, and adding appropriate constraints for the loop bounds, subscript expressions, and so on. Projecting this problem onto the variables for the array indexes and the symbolic constants gives an accurate summary of the locations of the array affected by the assignment statement. The summary is not limited to convex polyhedron. The projected problem will have solutions only for those locations that can actually be changed. Details such as strides are accurately represented.

The Omega test can easily be used to determine when two regions intersect. With more work, the Omega test can be used to check if one region is a subset of another. It is unclear how to use the Omega test to merge affected regions; however, the Omega test could be used to convert exact affected regions into approximate affected regions (such as described by [4, 10]) and then those regions could be merged.

Determining Loop Bounds

The Omega test can be used to determine appropriate loop bounds when interchanging nonrectangular loops. The use of integer programming and projection to perform this is described by [3].

Performance

We have implemented the Omega test in Wolfe's tiny tool [26]. We handle min and max expressions in loop bounds and symbolic constants, and compute exact sets of direction vectors (as opposed to the compressed direction vectors normally generated by tiny). We applied this tool to the programs 1, 3, 4, 5 and 7 of the NASA NAS benchmark suite and to all the tiny source files distributed with tiny, (which include Cholesky decomposition, LU decomposition, several versions of wavefront algorithms,

Table 1.
Execution times for programs in the NASA NAS benchmark suite

Program	Average Time	95%-Tile Time
#1: MXM	275 μsecs	316 μsecs
#3: CHOLSKY	504 μsecs	1024 μsecs
#4: BTRIX	250 μsecs	367 μsecs
#5: GMTRY	191 μsecs	534 μsecs
#7: VPENTA	129 μsecs	204 μsecs

and several more contrived examples), as well as several of our own test programs. Programs 2 and 6 of the NAS benchmark make extensive use of index arrays. Since we do not provide special treatment for index arrays, we decided that it would be misleading to include them. The analysis of array pairs that have different constant subscripts (e.g., a(4) and a(5)) are *not* included in the figures reported here; those cases are detected while scanning the subscripts (thus both avoiding the analysis time and the time required to scan the loop bounds). Standard optimizations such as induction variable recognition and forward substitution were performed by hand. We did not compute input dependences (an input dependence is a dependence between two reads of the same location of an array) or dependences between array pairs that did not share at least one common loop.

We timed the Omega test on a Decstation 3,100, a 12-MIPS workstation based on a MIPS R2000 CPU. Shown in Table 1 are our results on the time per array pair required to analyze programs in the NASA NAS benchmark:

The third program of the NAS benchmark (CHOLSKY) is substantially more complicated that almost all real-world Fortran code, involving loops nested four deep, triplely subscripted arrays and groups of 3 coupled-loop indices. We feel confident that it represents a good "worst-case example" for analyzing dusty deck Fortran code (excluding treatment of index arrays).

Our results on individual array pairs from all programs tested are shown in Figure 5. Each point is the timing result for a single array pair. To present the results in a somewhat machine independent fashion, the results are plotted on a log/log graph of analysis time vs. copying time (the time required just to copy the problem). All times were randomly perturbed by $\pm 1/2\mu$sec to spread out overlapping points. The diagonal lines are drawn at

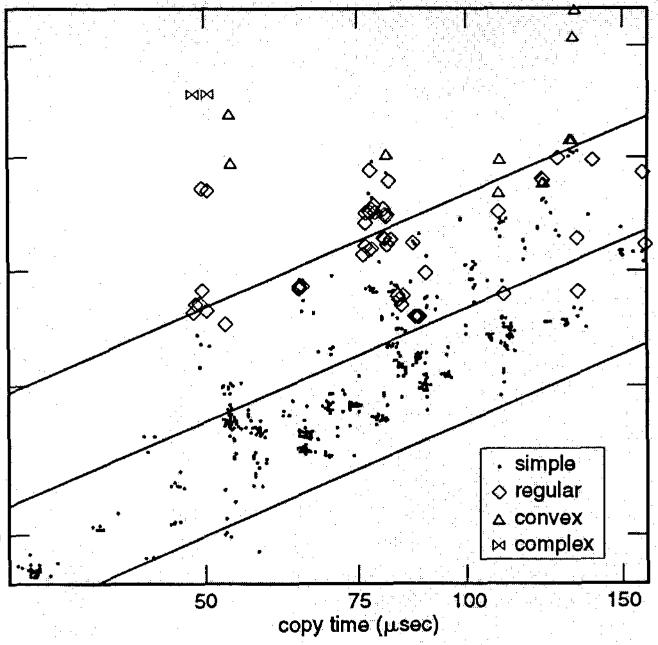


Figure 5. Omega Test Performance

analysis time = 8 × copying time, 4 × copying time and 2 × copying time.

The analysis time is the total time required to analyze the array pair, calculate the appropriate direction vectors and add the dependences to dependence graph. This is excluding the time required to scan the array subscripts and loop bounds and build the constraints that describe the dependence between the array pairs.

Across a range of test programs, we found the following break-down for how time was spent by the Omega test: about one-half the time was spent dealing with inequality constraints, about one-fourth of the time was spent on dealing with equality constraints, and one-fourth of the time was spent examining projected constraints to construct direction vectors. None of our test cases required inexact Fourier-Motzkin variable elimination.

To analyze our results, the set of constraints describing the dependence distances for each array pair were analyzed to remove any redundant constraints (this is not cost-effective normally). Based on the simplified constraints, each array pair was classified as follows:

- simple—Any case that does not involve coupled dependence distances.
- regular—A case where dependence distances are coupled, but all inequality constraints have unit coefficients (for example, $\{\Delta i \geq 0;\ \Delta i + \Delta j > 0\}$).
- convex—A case where the inequality constraints define a convex region but at least one constraint has a nonunit coefficient (for example, $\{0 \leq \Delta j \leq 10;\ \ 0 \leq \Delta i + \Delta j \leq 10;\ \Delta i + 2\Delta j \leq 10\}$ – the last constraint makes this nonregular).
- complex—A case where the inequality constraints define a nonconvex region. We only encountered two such cases, one shown here and another one identical except that the lower bound of the i loop is 2.

```
for i = 1 to 10 do
  for j = 0 to 4 do
    a(i-j) = a(j)
    endfor
endfor
```

The flow/anti dependence distances for the last example are all the distances that satisfy $\{-4 \leq \Delta j \leq 4;\ -7 \leq \Delta i - \Delta j, \Delta i + \Delta j \leq 10;\ \Delta i \leq 9\}$ except for $\{\Delta i = 9;\ \Delta j = 0\}$.

Maydan, Hennessy and Lam [18] use memoization to obtain better performance. Memoization could be added to the Omega test. The cost of computing a hash key and verifying a cache hit, however, would be about 2 to 4 times the copying cost for a problem, and therefore adding caching to the Omega test would not produce significant savings for typical, simple cases and may produce little or no overall speed improvement.

We found that the cost of scanning array subscripts and loop bounds to build a dependence problem was typically 2 to 4 times the copying cost for the problem. Thus, for many array pairs the cost of building the dependence problem was nearly as large or even larger than the time spent analyzing the resulting problem. We have not spent much effort trying to improve the performance of the code that builds dependence problems. It is difficult, however, to imagine building a dependence problem in much less than twice the time required to copy the problem. This suggests that for the majority of array pairs, using a dependence analysis algorithm significantly faster than the Omega test would not lead to significant overall speed improvements.

Polynomial Time Bounds

Described here are some general time bounds on parts of the Omega test, and then we describe polynomial time bounds for cases where other polynomial time algorithms are accurate. We will use m to denote the number of constraints and n to denote the number of variables.

The time taken by the methods in the section on equality constraints to eliminate one equality constraint is $O(mn \log |C|)$ worst-case time, where C is coefficient with the largest absolute value in the constraint. This cost arises because we might apply the perform $\log |C|$ substitutions before we can eliminate the constraint, and performing a substitution takes $O(nm)$ time.

Eliminating unbound variables takes $O(mnp)$ worst-case time, where p is the number of passes required to eliminate all the variables that become unbound. At least one variable is eliminated in each pass except the last.

Normalizing the constraints and checking for directly contradictory or redundant constraints requires $O(mn)$ expected time (the time bound is only expected, not worst-case, because hashing is used).

Producing the subproblems that result from Fourier-Motzkin variable elimination takes time proportional to the size of the subproblems produced.

Special Cases

During normalization, the Omega test checks to see if any variables are involved in constraints with other variables. If not, and if checking for contradictory constraint pairs has not produced a contradiction, we know the problem has solutions and we do not need to perform any additional computation. This applies if and only if the Single Variable Per Constraint (SVPC) test can be applied, which was found to be applicable in one-third of the unique cases found in the Perfect Club Benchmark (a higher percentage if duplicate cases were considered separately) [18].

The Acyclic Test can be applied in exactly those cases that the Omega test can resolve just by eliminating unbound variables and performing exact projections that do not increase the number of constraints, a process that takes $O(mn^2)$ worst-case time. They found that this test could be applied in over one fourth of the unique cases encountered [18].

The Loop Residue test [21] can be applied in just those cases where each constraint is of the form $x_i \geq x_j + c$, $x_i \geq c$, or $c \geq x_i$. In a set of constraints with this property, Fourier-Motzkin variable elimination is exact and preserves this property. On n variables, there can be at most $n^2 + n$ constraints of this form after eliminating redundant pairs. Thus, the Omega test will take $O(n^3)$ time to resolve a set of constraints that can be solved by the Loop Residue algorithm. Maydan, Hennessy and Lam [18] found that the Loop Residue algorithm could be applied in one-fourth of the unique cases encountered in their study of the Perfect Club benchmark.

Maydan, Hennessy and Lam found that 91% of the cases they encountered could be determined by constant tests and Banerjee's Generalized GCD tests. Of the remaining 9% of the cases, they found that their SVPC, Acyclic or Loop Residue tests could be applied in 86% of the unique cases.

The Delta test [8] works by searching for dependence distances that can be easily determined, and then propagating that information with the intent of making it possible to easily determine other dependence distances precisely. In the cases where their algorithm can determine a dependence distance without the use of Multiple Induction Variable (MIV) tests, the Omega test also will determine it efficiently (and in polynomial time) by a combination of solving equality constraints, tightening inequality constraints and converting tight inequality constraints into equality constraints. Since the Omega test treats the dependence analysis problem as a single integer programming problem, it automatically achieves the propagation effects of the Delta test. Therefore, any dependence analysis problem that can be solved by the Delta test without resorting to exponential algorithms or approximate methods (i.e., resorting to what they refer to as MIV tests) can be solved in polynomial time by the Omega test.

In their study of the RiCEPS, Perfect, SPEC benchmarks and LINPACK and EISPACK, they found that 97% percent of the cases could be solved without requiring the use of MIV tests.

The Omega test can solve, in effective polynomial time, any problem that can be solved by any combination of the Single Variable Per Constraint test, the Acyclic test, the Loop Residue test and the Delta test, effectively. Thus, we expect that it should be able to solve more problems exactly and efficiently than any one of them alone.

Related Work on Exact Dependence Analysis

The Constraint-Matrix test [23] makes use of the simplex algorithm modified for integer programming. The Constraint-Matrix test can fail to terminate and it is not clear how efficiently it works in practice.

Lu and Chen describe [17] an integer programming algorithm for dependence analysis. Their method, however, appears prohibitively expensive for use in a production compiler.

Triolet [22] used Fourier-Motzkin techniques for representing affected array regions in interprocedural analysis. Triolet found Fourier-Motzkin techniques to be expensive (22 to 28 times longer than using simpler methods for representing affected array regions).

Several implementations of Fourier-Motzkin variable elimination have been described for use in dependence analysis. The Power test described by Wolfe and Tseng [27] combines Banerjee's Generalized GCD test, constraint tightening, and Fourier-Motzkin variable elimination. They take no special action when performing an inexact projection except to flag the result as possibly being conservative. Fourier-Motzkin elimination is used [18] if none of the other methods they use give an exact answer. They use back substitution to determine a sample solution. If the sample solution is not integral, they suggest the use of branch and bound methods to verify or disprove the existence of integer solutions (they have not found the need to implement these methods as yet). It has been suggested that due to the expense of Fourier-Motzkin variable elimination, simpler tests should be used in situations where they are known to be accurate [18, 27].

Ancourt and Irigoin [3] describe the use of Fourier-Motzkin variable elimination to determine loop bounds for iterating over an iteration space described by a set of linear inequalities (using projection as described in the section on integer programming problems). Their work significantly overlaps with ours.

When performing what is apparently an inexact projection, they first perform a more elaborate process to check if the projection is inexact. They consider a concept similar to our `dark` shadow, except they force the difference between the upper and lower bounds to be at least $(a-1)b$, as opposed to $(a-1)(b-1)$. Since our definition is safe and makes the `dark` shadow larger, it is the preferred choice.

They do not actually generate the `dark` shadow as a separate problem. Rather, they check to see if the constraints in the `dark` shadow are redundant with respect to the `real` shadow. If they are, then the `dark` shadow and `real` shadow are identical, and the elimination is exact.

If the projection is not exact, then they add *pseudo-linear* constraints to the `real` shadow to obtain the `integer` shadow. These pseudo-linear constraints appear useful and appropriate for determining loop bounds. They are, however, difficult to use for determining the existence of integer solutions.

They do not provide any performance data for their algorithm.

A recent report [11] on the PIPS project mentions that Fourier-Motzkin variable elimination is used to analyze dependences (based on the work described in [3]). The methods used are not fully described, but the basic framework appears similar to that described in the section on dependence direction and distance vector. It is not clear how the pseudo-linear constraints of the latter are handled. They point out that in many simple cases, Fourier-Motzkin variable elimination is fast and efficient. They state that using integer programming techniques for dependence analysis incurs a very high cost (that is acceptable since PIPS is not a production system). They also state that in their implementation, dependence testing does not take a noticeable amount of time compared with the wholly parallelization process.

Source Code Availability

A C language implementation of the Omega test is freely available for anonymous `ftp` from `ftp.cs.umd.edu` in directly `pub/omega`. Files available include a stand-alone version of the Omega test and a version of Wolfe's `tiny` tool [26] extended to use the Omega test.

Conclusions

Conservative dependence analysis methods may be efficacious for the demands of vectorizing compilers. Transforming programs so as to make efficient use of massively parallel SIMD computers is a much more demanding task. Also, programs that have undergone transformations such as loop skewing and loop interchange present analysis problems substantially more difficult than encountered in typical dusty-deck Fortran.

Our studies have convinced us that the Omega test is a fast and practical method for performing data dependence analysis that is not only adequate for problems encountered in vectorizing Fortran code, but also for the demands of more sophisticated program transformation tools.

Performing projection of integer programming problems is an exciting concept. We have discussed how it can be used to determine efficiently information about dependence direction and distance vectors, as well for several other uses. It can make it much easier to describe and build program analysis and transformation tools. For example, it can be used for determining loop bounds after loop interchange, and we have made extensive use of it in work that considers loop transformations in a uniform manner [3, 20].

Acknowledgments
Thanks to everyone who gave me feedback on this work, especially Michael Wolfe and the anonymous referee who provided detailed comments, as well as to my research group (Dave Wonnacott, Udayan Borker and Wayne Kelly).

References

1. Allen, J.R. Dependence analysis for subscripted variables and its application to program transformations. PhD thesis, Rice University, Apr. 1983.
2. Allen, J.R. and Kennedy, K. Automatic translation of Fortran programs to vector form. *ACM Trans. Prog. Lang. Syst. 9*, 4 (Oct. 1987), 491–542.
3. Ancourt, C. and Irigoin, P. Scanning polyhedra with do loops. In *PPOPP '91*, 1991.
4. Balasundaram, V. and Kennedy, K. A technique for summarizing data access and its use in parallelism enhancing transformations. In *SIGPLAN Conference on Programming Language Design and Implementation, '89*, June, 1989.
5. Banerjee, U. Dependence analysis for supercomputing. Kluwer Academic Publishers, Boston, Mass., 1988.
6. Burke, M. and Cytron, R. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, Calif., July, 1986.
7. Dantzig, G.B. and Eaves, B.C. Fourier-Motzkin elimination and its dual. *J. Combin. Theo. A*, 14 (1973), 288–297.
8. Goff, G., Kennedy, K. and Tseng, C. Practical dependence testing. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.
9. Haghighat, M. and Polychronopoulos, C. Symbolic dependence analysis for high performance parallelizing compilers. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Aug. 1990.
10. Havlak, P. and Kennedy, K. Experience with interprocedural analysis of array side effects. In *Supercomputing '90*, 1990.
11. Irigoin, P., Jouvelot, P. and Triolet, R. Semantical interprocedural parallelization: An overview of the pips project. In *ICS '91*, 1991.
12. Klappholz, D. and Kong, X. Extending the Banerjee-Wolfe test to handle execution conditions. Tech. Rep. 9101, Dept. of EE/CS, Stevens Institute of Technology, 1991.
13. Kuck, D., Muraoka, Y. and Chen, S. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Comput.*, 1972.
14. Li, Z. and Yew, P. Some results on exact data dependence analysis. In D. Gelernter, A. Nicolau, and D. Padua, Eds., *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.
15. Li, Z., Yew, P. and Zhu, C. Data dependence analysis on multidimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*,
16. Lichnewsky, A. and Thomasset, F. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proceedings of the Second International Conference on Supercomputing* St. Malo, France, July 1988.
17. Lu, L. and Chen, M. Subdomain dependence test for massive parallelism. In *Proceedings of Supercomputing '90*, New York, Nov. 1990.
18. Maydan, D.E., Hennessy, J.L. and Lam, M.S. Efficient and exact data dependence analysis. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.
19. Muraoka, Y. Parallelism exposure and exploitation in programs. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Feb. 1971.
20. Pugh, W. Uniform methods for loop optimization. In *1991 International Conference on Supercomputing*, Cologne, Germany, June 1991.
21. Shostak, R. Deciding linear inequalities by computing loop residues. *J. ACM 28*, 4 (Oct. 1981), 769–779.
22. Triolet, R. Interprocedural analysis for program restructuring with Parafrase. CSRD Rep. 538, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Dec. 1985.
23. Wallace, D. Dependence of multidimensional array references. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.
24. Wolfe, M.J. Optimizing supercompilers for supercomputers. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1982.
25. Wolfe, M. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.
26. Wolfe, M. The tiny loop restructuring research tool. In *Proceedings of 1991 International Conference on Parallel Processing*, 1991.
27. Wolfe, M. and Tseng, C. The power test for data dependence. Tech. Rep. CS/E 90-015, Oregon Graduate Institute, Aug. 1990.

CR Category and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization;* F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs, Mechanical Verification; G.1.6 [**Numerical Analysis**]: Optimization—*integer programming*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Dependence Analysis, integer programming, linear Diophantine Equations, Omega test

About the Author:
WILLIAM PUGH is an assistant professor at the University of Maryland at College Park in the Department of Computer Science and Institute for Advanced Computer Studies. He has done research in the areas of incremental computation, randomized data structures, implementation of multiple inheritance, programming languages for hard real-time systems, and compilers for supercomputers. **Author's Present Address:** Dept. of Computer Science, University of Maryland, College Park, MD 20742, pugh@cs.umd.edu