Eliminating False Data Dependences using the Omega Test *

William Pugh

Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742
pugh@cs.umd.edu, (301)-405-2705

David Wonnacott

Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742
davew@cs.umd.edu, (301)-405-2726

Abstract

Array data dependence analysis methods currently in use generate false dependences that can prevent useful program transformations. These false dependences arise because the questions asked are conservative approximations to the questions we really should be asking. Unfortunately, the questions we really should be asking go beyond integer programming and require decision procedures for a subclass of Presburger formulas. In this paper, we describe how to extend the Omega test so that it can answer these queries and allow us to eliminate these false data dependences. We have implemented the techniques described here and believe they are suitable for use in production compilers.

1 Introduction

Recent studies [HKK+91, CP91] suggest that array data dependence analysis methods currently in use generate false dependences that can prevent useful program transformations. For the most part, these false dependences are *not* generated by the conservative nature of algorithms such as Banerjee's inequalities [SLY89, KPK90]. These false dependences arise because the *questions* we ask of dependence analysis algorithms are conservative approximations to the questions we really should be asking (methods currently in use are unable to address the more complicated questions we should be asking).

*This work is supported by NSF grant CCR-9157384 and a Packard Fellowship.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0140...$1.50

For example, there is a flow dependence from an array access $A(\mathbf{i})$ to an array access $B(\mathbf{j})$ iff

- A is executed with iteration vector $\mathbf{i}$,
- B is executed with iteration vector $\mathbf{j}$,
- $A(\mathbf{i})$ writes to the same location as is read by $B(\mathbf{j})$,
- $A(\mathbf{i})$ is executed before $B(\mathbf{j})$, and
- there is no write to the location read by $B(\mathbf{j})$ between the execution of $A(\mathbf{i})$ and $B(\mathbf{j})$.

However, most array data dependence algorithms ignore the last criterion (either explicitly or implicitly). While ignoring this criterion does not change the total order imposed by the dependences, it does cause flow dependences to become contaminated with output dependences (storage dependences). There are techniques (such as privatization, renaming, and array expansion) that can eliminate storage-related dependences. However, these methods cannot be applied if they appear to affect the flow dependences of a program. Also, flow dependences represent more than ordering constraints: they also represent the flow of information. In order to make effective use of caches or distributed memories, a compiler must have accurate information about the flow of information in a program.

Similarly, many dependence testing algorithms do not handle assertions about relationships among non-induction variables or array references that appear in subscripts or loop bounds. To be useful, a system must not only be able to incorporate assertions about these relationships, but also be able to generate a useful dialog with the user about which relationships hold.

Unfortunately, the questions we really should be asking go beyond integer programming and require decision procedures for a subclass of Presburger formulas.

Presburger formulas are those that can be built up from integer constants, integer variables, addition, $>$, $=$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$ and $\exists$. We can use these primitives

to allow us to also handle subtraction, multiplication by integer constants, and the other arithmetic relations ($\geq$, $<$, $\leq$). Presburger formulas are decidable, but the only known decision procedures that handle the full class take at least doubly-exponential worst-cast time.

Our original work on the Omega test [Pug91] described efficient ways to answer the usual questions asked for dependence analysis. In this paper, we show how the Omega test can be extended so that it can be used to answer questions in a subclass of Presburger arithmetic. We then show how to phrase within that subclass the questions we need to ask. We also describe experiences with an implementation of the methods described here that convince us that these techniques are suitable for use in production compilers.

2 Notation

The notation in Table 1 is adapted from [ZC91]. Below, we briefly discuss dependence distance, direction and restraint vectors.

2.1 Dependence Distance

A data dependence has a dependence distance in each loop, which is is the difference between the values of the loop variables for some pair of iterations involved in the dependence.[1] The dependence distance vector for a dependence is a vector of the dependence distances for the loops common to both statements involved in the dependence.

Often, a dependence distance is not constant, and the dependence distance in one loop can be coupled to the dependence distance in another loop. For example a dependence might have dependence distances $\{(\Delta i, \Delta j) \mid 0 \leq \Delta i \leq 5 \wedge 1 \leq \Delta i + \Delta j \leq 10\}$. Since dependences represent a constraint from one statement execution to a later statement execution, dependence distances must be lexicographically positive[2], with a zero dependence distance in all loops possible only if the dependence is syntactically forward.

2.1.1 Direction vectors

A direction vector summarizes, for each loop, the possible signs of the dependence distance in that loop. For example a dependence with dependence distances $(\Delta i, \Delta j)$ such that $0 \leq \Delta i \leq 5 \wedge 0 \leq \Delta j \leq 10$ would be represented by the direction vectors $\{(0+, 0+)\}$. We also can give a specific distance or range of distances in a direction vector (e.g., $\{(0+, 1)\}$ or $\{(0:1, 1)\}$).

It is not always possible to summarize accurately the possible signs of the dependence distance with a single direction vector. The signs of the dependence distances $\{(\Delta i, \Delta j) \mid \Delta i = \Delta j\}$ can be accurately represented by the direction vectors $\{(+,+), (0,0), (-,-)\}$. After filtering for lexicographically forward directions, this dependence is represented by $\{(+,+), (0,0)\}$. If these two are compressed into a single direction vector $(0+, 0+)$, it falsely suggests the signs $(0,+)$ and $(+,0)$ are possible.

Given a set of constraints that describe the possible forward and backward dependence distances, we analyze the constraints to produce a set of partially compressed direction vectors (as described in [Pug91]). These direction vectors are filtered to produce a set of forward direction vectors.

2.1.2 Restraint vectors

While checking for dependence killing, covering and refinement (Section 4), we deal with dependences as integer programming problems. We need to force the dependence directions to be lexicographically positive (e.g., force $\Delta i > 0 \vee \Delta i = 0 \wedge \Delta j \geq 0$). Unfortunately, this is not a conjunction of linear constraints. However, we can often find a conjunction of linear constraints that will force the dependence distance to be lexicographically forward. For example, we can force the dependence distances $\Delta i = \Delta j$ to be lexicographically positive by adding the constraint $\Delta i \geq 0$.

We represent constraints that force the dependence directions to be lexicographically positive in the same form as direction vectors (specifying the possible signs for each dependence distance). However, these "restraint" vectors only have to filter out lexicographically negative directions which are legal solutions to the integer programming problem that describes all forward and backward dependence distances. We can use the set of direction vectors as a set of restraint vectors, but using restraint vectors usually requires that fewer constraints be added to the problem.

Our algorithms in Sections 4 and 5 are unable to deal directly with dependences that cannot be represented

[1] There is some disagreement within the research community as to whether dependence distance reflects the difference in the values of the loop variables ([ZC91]), or the difference in the loop trip counts ([Wol89]). This disagreement is usually submerged by the fact that many researchers only discuss normalized loops with an increment of 1. One problem with using the difference in loop trip counts is that it is not always well defined. For example, in the following code segment, the dependence distance is $(1, -2)$ if the difference in the loop variables is used; however, the dependence distance doesn't seem to be defined if loop trip counts are used and the loop is not normalized:

```
for i := 0 to 10
    for j := i to 10 by 3
        A[i,j] := A[i-1,j+2]
```

The problem with using the difference in the values of loop variables as dependence distance is that negative steps complicate our discussion of forward dependence distances (usually described as lexicographically positive distances) and the description of the conditions under which loop interchange and circulation are allowed. In this paper, we use the difference in the loop variables as the dependence distance, and ignore the side issue of complicating the definition of forward dependence distances.

[2] Assuming loop increments are positive, see footnote 1 for details.

$A, B, \ldots$	Refers to a specific array reference in a program
$\mathbf{i}, \mathbf{j}, \mathbf{k}, \ldots$	An iteration vector that represents a specific set of values of the loop variables for a loop nest.
$[A]$	The set of iteration vectors for which A is executed
$A(\mathbf{i})$	The instance of array reference A when the loop variables have the values specified by $\mathbf{i}$
$A(\mathbf{i}) \stackrel{sub}{=} A'(\mathbf{i}')$	The references A and A' refer to the same array and the subscripts of $A(\mathbf{i})$ and $A'(\mathbf{i}')$ are equal.
$A(\mathbf{i}) \ll A'(\mathbf{i}')$	Execution of $A(\mathbf{i})$ occurs previous to $A'(\mathbf{i}')$
$A(\mathbf{i}) \ll_D A'(\mathbf{i}')$	The dependence distance from $A(\mathbf{i})$ to $A'(\mathbf{i}')$ is described by the direction/distance vector D
Sym	The set of symbolic constants (e.g., loop-invariant scalar variables)

Figure 1: Notation used in this paper

by a single restraint vector. Therefore, such dependences are split into several dependences, one for each restraint vector.

3 Extending the Omega test

The Omega test [Pug91] is an integer programming algorithm based on Fourier-Motzkin variable elimination. The basic operation supported by the Omega test is projection. Intuitively, the projection of a set of constraints is the shadow of a set of constraints. More formally, given a set of linear equalities and inequalities on a set of variables V, projecting the constraints onto the variables $\widehat{V}$ (where $\widehat{V} \subset V$) produces a set of constraints on variables $\widehat{V}$ that has the same integer solutions for $\widehat{V}$ as the original problem. For example, projecting $\{0 \leq a \leq 5; b < a \leq 5b\}$ onto a gives $\{2 \leq a \leq 5\}$. We use the notation $\pi_{x_1,\ldots,x_n}(S)$ to represent the projection of the problem S onto the set of variables $x_1, \ldots, x_n$ and the notation $\pi_{\neg x}(S)$ to represent the projection of the problem S onto all variables other than x.

The Omega test determines if a set of constraints has integer solutions by using projection to eliminate variables until the constraints involve a single variable, at which point is it easy to check for integer solutions.

There are many other applications of projection. For example, if we define a set of constraints for an array pair that includes variables for the possible dependence distances, we can project that set of constraints onto the variables for the dependence distance. The projected system can be efficiently used to determine the dependence directions and distances.

Because the Omega test checks for integer solutions, not real solutions, it is sometimes unable to produce a single set of constraints when computing $\pi_x(S)$. Instead, the Omega test is forced to produce a set of problems $S_0, S_1, \ldots, S_p$ and a problem T such that $\pi_x(S) = \bigcup_{i=0}^{p} S_i \subseteq T$. This is called *splintering*, and we call S_0 the Dark Shadow of $\pi_x(S)$ and call T the Real Shadow of $\pi_x(S)$ (the Real Shadow may include solutions for x that only have real but not integer solutions for the variables that have been eliminated).

In practice, projection rarely splinters and when it does, S_0 contains almost all of the points of $\pi_x(S)$, T doesn't contain many more points than $\pi_x(S)$, and p is small. If we are checking to see if S has solutions, we first check if $S_0 \neq \emptyset$ or $T = \emptyset$. Only if both tests fail are we required to examine $S_1, S_2, \ldots, S_p$. Also, when checking for integer solutions, we choose which variable to eliminate to avoid splintering when possible.

Although integer programming is an NP-Complete problem, the Omega test is efficient in practice [Pug91].

3.1 How the Omega test works

Fourier-Motzkin variable elimination [DE73] 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 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 shadow of this pair of constraints is $a\beta \leq b\alpha$. Fourier-Motzkin variable elimination calculates the shadow of a set of constraints by combining all constraints that do not involve the variable being eliminated with the result from each combination of a lower and upper bound on the variable being eliminated. The real shadow is a conservative approximation to the integer shadow of the set of constraints.

In [Pug91], we extended Fourier-Motzkin variable elimination to be an integer programming method. Even if $a\beta \leq b\alpha$, there may be no integer solution to z such that $a\beta \leq abz \leq b\alpha$. However, if $a\beta + (a-1)(b-1) \leq b\alpha$, we know that an integer solution to z must exist. This is the dark shadow of this pair of constraints (described in [Pug91]). The dark shadow is a pessimistic approximation to the integer shadow of the set of constraints. Note that if $a = 1$ or $b = 1$, the dark shadow and the real shadow are identical, and therefore also identical to the integer shadow.

There are cases when the real shadow contains integer points but the dark shadow does not. In this case, determining the existence of integer solutions to the original set of constraints requires the use of special case techniques, described in [Pug91], that are almost never needed in practice.

3.2 Determining the validity of Presburger formulas

Assume that p and q are propositions that can each be represented as a conjunction of linear equalities and inequalities. We can determine the truthfulness of the following predicates:

Is p a tautology? Trivial to check when p is a conjunction.

Is p satisfiable? We can check this using techniques described in Section 3.1 and in [Pug91].

Is $p \Rightarrow q$ a tautology? This could not be efficiently answered using the techniques described in [Pug91], but can be efficiently answered in practice using techniques described in Section 3.3.

The projection transformation offered by the Omega test allows us to handle embeded existential qualifiers: $\pi_{\neg x}(p) = (\exists x \text{ s.t. } p)$. We can combine these abilities, as well as any standard transformation of predicate calculus, to determine the validity of certain Presburger formulas. We have not attempted to formally capture the subclass of Presburger formulas we can answer efficiently. The following are examples of some Presburger formulas we can answer efficiently:

$\forall x, \exists y \text{ s.t. } p$: True iff $\pi_{\neg y}(p)$ is a tautology.

$\forall x, (\exists y \text{ s.t. } p) \Rightarrow (\exists z \text{ s.t. } q)$: True iff $\pi_{\neg y}(p) \Rightarrow \pi_{\neg z}(q)$ is a tautology. We can easily determine this if $\pi_{\neg z}(q)$ does not splinter.

$\forall x, \neg p \vee q \vee \neg r$: True iff $p \wedge r \Rightarrow q$ is a tautology.

where p, q and r are conjunctions of linear equalities and inequalities

3.3 Computing Gists and Checking when $p \Rightarrow q$ is a tautology

Intuitively, we define (gist p given q) as the new information contained in p, given that we already know q. More formally, (gist p given q) is a conjunction containing a minimal subset of the constraints of p such that $((\text{gist } p \text{ given } q) \wedge q) = (p \wedge q))$. Note that $(\text{gist } p \text{ given } q) = \text{True} \equiv q = (p \wedge q) \equiv (q \Rightarrow p)$.

When computing (gist p given q), we first convert any equalities in p into a matched pair of inequalities (e.g., convert $x = 1$ into $1 \leq x \leq 1$). The following is a naive method for computing gists. The algorithm treats a conjunction as a list of individual constraints:

$$\begin{aligned}
&\text{gist } [\,]\ q = [\,] \\
&\text{gist } (e : p)\ q = e : (\text{gist } p\ (e : q)), \\
&\qquad\qquad \text{if } (\neg e) \wedge p \wedge q \text{ is satisfiable} \\
&\text{gist } (e : p)\ q = \text{gist } p\ q, \text{ otherwise}
\end{aligned}$$

This algorithm requires many satisfiability tests, each of which takes a non-trivial amount of time. We handle this problem by checking for a number of special cases (listed in order of increasing difficulty to check):

- For each equation e in p, we check to see if e is implied by any single constraint in p or q. If so, e is redundant and not in the gist.
- We check to see if there is any variable that has an upper bound in p but not in q. If so, we know that at least one of the upper bounds from p must be in the gist. A similar check is made for lower bounds.
- If there does not exist some constraint e' in p or q such that the inner product of the normals of e and e' is positive, then e must be in the gist.
- For each equation e in p, we check to see if e is implied by any two constraints in p and/or q (other than e).

These fast checks often completely determine a gist. When they do not, they usually greatly simplify the problem before we utilize the naive algorithm.

3.3.1 Checking implications

As noted earlier, we determine if $q \Rightarrow p$ is a tautology by checking if (gist p given q) = True. When performing subset tests using the above algorithms, we short-circuit the computation of the gist as soon as we are sure that the gist is not "True".

3.3.2 Combining Projection and Gist computation

If is often the case that we need to compute problems of the form gist $\pi_{\neg y}(p)$ given $\pi_{\neg z}(q)$. We could perform this computation by performing the projections independently, and then computing the gists. However, if z is free in p and y is free in q, there is a more efficient solution. We can combine p and q into a single set of constraints, tagging the equations from p red and the equations from q black. We then project away the variables y and z and eliminate any obviously redundant red equations as we go. Once we have projected away y and z, we then compute the gist of the red equations with respect to the black equations.

3.4 Related Work

Several authors have explored methods for using integer programming methods to decide subclasses of Presburger formulas [Ble75, Sho77, JM87]. Previous approaches have not had any way to compute projections (and thereby handle embeded existential quantifiers) nor have they had special efficient methods for checking when an implication is a tautology. We have not yet looked at wider applications of our work or done any direct comparison of our implementation against implementations of other approaches. We believe our approach and implementation will compare favorably with previous ones.

```
a(n) := ...
for L1 := n to n+10 do
  a(L1) := ...

for L1 := n to n+20 do
  ... := a(L1)
```

Example 1: Killed flow dep

```
a(m) := ...
for L1 := 1 to 100
  a(L1) := ...
  for L2 := 1 to n do
    a(L2) := ...
    a(L2-1) := ...
  for L2 := 2 to n-1 do
     ... := a(L2)
```

Example 2: Covering and Killed dep

```
for L1 := 1 to n do
  for L2 := 2 to m do
    a(L2) := a(L2-1)
```

Unrefined flow dependence: (0+,1)
Refined flow dependence: (0,1)

Example 3: Refinement

```
for L1 := 1 to n do
  for L2 := n+2-L1 to m do
    a(L2) := a(L2-1)
```

Unrefined flow dependence: (0+,1)
Refined flow dependence: (0,1)

Example 4: Trapezoidal Refinement

```
for L1 := 1 to n do
  for L2 := L1 to m do
    a(L2) := a(L2-1)
```

Unrefined flow dependence: (0+,1)
Refined flow dependence: (0:1,1)

Example 5: Partial Refinement

```
for L1 := 1 to n do
  for L2 := 2 to m do
    a(L1-L2) := a(L1-L2)
```

Unrefined flow dependence: $(\alpha, \alpha), \alpha \geq 1$
Refined flow dependence: (1,1)

Example 6: Coupled Refinement

4 Handling array kills

In this section, we discuss the elimination of false dependences in which an intervening write eliminates an apparent dependency. These techniques are most useful when applied to flow dependences. However, we can also apply them to output and anti-dependences.

There are four kinds of analysis we perform:

Killing A dependence from a read or write A to a read or write C is killed by the dependence from a write B to C iff all array elements accessed by A are overwritten by B before C can access them.

Covering A write A covers a read or write B iff it writes to all the elements of the array that will be accessed by B. In this case, any dependence to B from an access that precedes A is killed by the dependence from A.

Terminating A dependence from a read or write A to a write B terminates A if B overwrites all elements that were accessed in A. In this case, any dependences from A to a read or write after B is killed by the dependence from A to B.

Refinement It may be possible to refine the dependence distances for a dependence from a write A to a read or write B to a subset D of these dependence distances. This is possible iff any dependence with a distance not in D is killed by a dependence in D.

We first compute all output dependences (output dependences give us fast checks for when to test for killing and refinement). Then, for each array read reference B, we start computing flow/anti dependences to/from B. For each apparent flow dependence from a write A, we attempt to refine the dependence distance, and then check if it is covering. If it is covering, we can rule out a flow dependence from any write that occurs completely before A. If there appear to be multiple flow dependences to B, we check them pairwise for killing.

4.1 Killing dependences

A dependence from a read or write A to a read or write C is killed by the dependence from a write B to C iff all elements accessed by A are overwritten by B before C can access them. This is the case if:

$$\begin{aligned}\forall\, \mathbf{i}, \mathbf{k}, \mathtt{Sym},\ \mathbf{i} \in [A] \wedge \mathbf{k} \in [C] \\ \wedge A(\mathbf{i}) \ll C(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} C(\mathbf{k}) \Rightarrow \\ \exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [B] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \ll C(\mathbf{k}) \\ \wedge B(\mathbf{j}) \stackrel{sub}{=} C(\mathbf{k})\end{aligned}$$

In Example 1, the write to `a(L1)` kills the flow from the write of `a(n)` to the read of `a(L1)`:

$$\begin{aligned}&\mathbf{i} \in [A] \wedge \mathbf{k} \in [C] \wedge A(\mathbf{i}) \ll C(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} C(\mathbf{k}) \\ &\quad \equiv \mathbf{k}_1 = \mathbf{n}\end{aligned}$$

$$\begin{aligned}&\exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [B] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \ll C(\mathbf{k}) \wedge B(\mathbf{j}) \stackrel{sub}{=} C(\mathbf{k}) \\ &\quad \equiv \mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 10\end{aligned}$$

$$\mathbf{k}_1 = \mathbf{n} \Rightarrow \mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 10$$

If the first write were to `a(m)`, we would not be able to verify the kill:

$$\begin{aligned}&\mathbf{i} \in [A] \wedge \mathbf{k} \in [C] \wedge A(\mathbf{i}) \ll C(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} C(\mathbf{k}) \\ &\quad \equiv \mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 20 \wedge \mathbf{k}_1 = \mathbf{m}\end{aligned}$$

$$\begin{aligned}&\exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [B] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \ll C(\mathbf{k}) \wedge B(\mathbf{j}) \stackrel{sub}{=} C(\mathbf{k}) \\ &\quad \equiv \mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 10\end{aligned}$$

$$\mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 20 \wedge \mathbf{k}_1 = \mathbf{m} \not\Rightarrow \mathbf{n} \leq \mathbf{k}_1 \leq \mathbf{n} + 10$$

If $\mathtt{n} \leq \mathtt{m} \leq \mathtt{n}+10$ had been asserted by the user, we would be able to verify the kill.

4.2 Covering dependences

A dependence from a write A to a read or write B is a covering dependence iff every location accessed by B is previously written to by A. If the dependence from A to B is a covering dependence, we need not examine any dependences to B from any accesses that would precede the writes of A (since the dependence from A would kill such a dependence).

A dependence from a write A covers B iff:

$$\begin{aligned}&\forall\, \mathbf{j}, \mathtt{Sym},\ \mathbf{j} \in [B]\\ &\quad \Rightarrow \exists \mathbf{i} \text{ s.t. } \mathbf{i} \in [A] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{j})\end{aligned}$$

In Example 2, the read of `a(L2)` is covered by the write to `a(L2-1)`:

$$\begin{aligned}&\mathbf{j} \in [B]\\ &\quad \equiv 1 \leq \mathbf{j}_1 \leq 100 \wedge 2 \leq \mathbf{j}_2 \leq \mathtt{n}-1\end{aligned}$$

$$\begin{aligned}&\exists \mathbf{i} \text{ s.t. } \mathbf{i} \in [A] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{j})\\ &\quad \equiv 1 \leq \mathbf{j}_1 \leq 100 \wedge 0 \leq \mathbf{j}_2 \leq \mathtt{n}-1\end{aligned}$$

$$\begin{aligned}&1 \leq \mathbf{j}_1 \leq 100 \wedge 2 \leq \mathbf{j}_2 \leq \mathtt{n}-1\\ &\quad \Rightarrow 1 \leq \mathbf{j}_1 \leq 100 \wedge 0 \leq \mathbf{j}_2 \leq \mathtt{n}-1\end{aligned}$$

Since we have determined that this dependence is a covering dependence, there is no need to check for dependencies with writes that must precede `a(L2-1)` (such as the write to `a(m)`). However, for writes that may be executed after some executions of the cover (such as the write to`a(L2)`), we must check separately for a kill. In general, we may have to determine which loop carries the cover to know which write accesses must precede it. The cover in this example is loop independent, so we know that all executions of the write `a(L1)` must precede the cover. Note that we only find out that the cover is loop independent when we refine its dependence vector from (0+) to (0). For this reason, we perform refinement before coverage analysis.

4.3 Terminating dependences

A dependence from a read or write A to a write B is a terminating dependence iff every location accessed by A is subsequently overwritten by B. If the dependence from A to B is a terminating dependence, we need not examine any dependences from A to any accesses that would follow the writes of B.

A dependence from A to a write B terminates A iff:

$$\begin{aligned}&\forall\, \mathbf{i}, \mathtt{Sym},\ \mathbf{i} \in [A] \Rightarrow\\ &\quad \exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [B] \wedge A(\mathbf{i}) \ll B(\mathbf{j}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{j})\end{aligned}$$

4.4 Refining dependence distances/directions

If all iterations of a read or write B that receive a dependence from a write A also receive a dependence from a more recent iteration of A within distance D, the dependence distances can be refined to D:

$$\begin{aligned}&\forall\, \mathbf{i}, \mathbf{k}, \mathtt{Sym},\ \mathbf{i} \in [A] \wedge \mathbf{k} \in [B]\\ &\qquad \wedge A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{k})\\ &\quad \Rightarrow A(\mathbf{i}) \ll_D B(\mathbf{k})\\ &\qquad \vee \exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [A]\\ &\qquad\qquad \wedge A(\mathbf{i}) \ll A(\mathbf{j}) \ll_D B(\mathbf{k})\\ &\qquad\qquad \wedge A(\mathbf{j}) \stackrel{sub}{=} B(\mathbf{k})\end{aligned}$$

When we attempt to refine dependence vectors, we do so in a way that ensures that the refined dependence contains the most recent executions of A, or in other words:

$$\begin{aligned}&A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \not\ll_D B(\mathbf{k}) \wedge A(\mathbf{j}) \ll_D B(\mathbf{k})\\ &\quad \Rightarrow A(\mathbf{i}) \ll A(\mathbf{j})\end{aligned}$$

In this case, we can simplify the condition under which we can perform refinement to:

$$\begin{aligned}&\forall\, \mathbf{i}, \mathbf{k}, \mathtt{Sym},\ \mathbf{i} \in [A] \wedge \mathbf{k} \in [B]\\ &\qquad \wedge A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{k})\\ &\quad \Rightarrow \exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [A] \wedge A(\mathbf{j}) \ll_D B(\mathbf{k})\\ &\qquad\qquad \wedge A(\mathbf{j}) \stackrel{sub}{=} B(\mathbf{k})\end{aligned}$$

which can be further transformed to

$$\begin{aligned}&\forall\, \mathbf{k}, \mathtt{Sym},\\ &\quad (\exists \mathbf{i} \text{ s.t. } i \in [A] \wedge \mathbf{k} \in [B] \wedge A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{k}))\\ &\quad \Rightarrow (\exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [A] \wedge A(\mathbf{j}) \ll_D B(\mathbf{k}) \wedge A(\mathbf{j}) \stackrel{sub}{=} B(\mathbf{k}))\end{aligned}$$

Example 3 shows a loop with a flow dependence that can be refined from (0+,1) to (0,1):

$$\begin{aligned}&\exists \mathbf{i} \text{ s.t. } \mathbf{k} \in [B] \wedge i \in [A] \wedge A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \stackrel{sub}{=} B(\mathbf{k})\\ &\quad \equiv 1 \leq \mathbf{k}_1 \leq \mathtt{n} \wedge 3 \leq \mathbf{k}_2 \leq \mathtt{m}\end{aligned}$$

$$\begin{aligned}&\exists \mathbf{j} \text{ s.t. } \mathbf{j} \in [A] \wedge A(\mathbf{j}) \ll_D B(\mathbf{k}) \wedge A(\mathbf{j}) \stackrel{sub}{=} B(\mathbf{k})\\ &\quad \equiv 1 \leq \mathbf{k}_1 \leq \mathtt{n} \wedge 3 \leq \mathbf{k}_2 \leq \mathtt{m}\end{aligned}$$

$$\begin{aligned}&1 \leq \mathbf{k}_1 \leq \mathtt{n} \wedge 3 \leq \mathbf{k}_2 \leq \mathtt{m}\\ &\quad \Rightarrow 1 \leq \mathbf{k}_1 \leq \mathtt{n} \wedge 3 \leq \mathbf{k}_2 \leq \mathtt{m}\end{aligned}$$

Example 4 is similar to Example 3, but includes a triangular loop. Example 5 is similar to Example 4, but here the distance can only be refined to (0:1,1) because iterations such that $1 < \mathtt{L1} = \mathtt{L2}$ receive a flow from iteration $(\mathtt{L1}-1, \mathtt{L2}-1)$. Example 6 shows a case where the dependence distances are coupled. Neither of the two approaches similar to ours ([Bra88, Ros90]) would handle Examples 4, 5 or 6.

The above methods allow us to check to see if any specific D is a correct refinement of a direction/distance vector. We generate the D's by attempting to fix the dependence distance, starting with the

outermost loop. For each loop, we attempt to set the dependence distance to be the minimum possible distance in that loop (which is easily extracted from the set of constraints describing the possible (unrefined) dependence distances). If we succeed, we move on to the next loop. If we fail, we stop refining that dependence (attempts to further refine the dependence distance for inner loops would not satisfy $A(\mathbf{i}) \ll B(\mathbf{k}) \wedge A(\mathbf{i}) \not\ll_D B(\mathbf{k}) \wedge A(\mathbf{j}) \ll_D B(\mathbf{k}) \Rightarrow A(\mathbf{i}) \ll A(\mathbf{j})$).

This method for generating D's to test will not automatically find the partial refinement in Example 5.

4.5 Quick tests for when to check for the above

We can often avoid performing the general tests described above by doing some quick tests for special cases. For example, for the dependence from B to C to kill the dependence between A and C, there must be an output dependence between A and B, and it must be possible for the dependence distance from A to C to equal the total distance from A to B and B to C. Similarly, for there to be any possibility of refining the dependence distance in that loop from A to C, A must have a self-output dependence with a non-zero distance in a loop in order.

If a dependence from A to B does not include the distance 0 in some loop l, it can not cover the execution of B the first time through l, so we do not test it for coverage. Note that A may actually cover B if B is not executed the first time through l - we would fail to detect this cover, and be forced to kill the covered dependencies with the A to B dependence later.

Finally, if we are trying to kill a dependence from A to C with a covering dependence from B to C, and the dependence from B is always closer than the dependence from A, then we know the dependence from A to C is killed without having to perform the general test.

4.6 Related Work

In analyzing false array flow data dependences (caused by output dependences), there are two basic approaches:

- Extend scalar dataflow methods by recording which array sections are killed and/or defined [GS90, Ros90].
- Extend the pair-wise methods typically used for array data dependence to recognize array kills [Bra88, Rib90].

Both approaches have merits. Our work is an example of the second approach, and we believe it corrects several limitations and flaws in previous work on that approach.

Brandes [Bra88] describes methods factoring out transitive dependences to determine "direct" dependences, and his work is similar to our computations for refinement, killing and covering. However, his methods do not apply if the dependence distances are coupled or the loop is non-rectangular.

Ribas describes [Rib90] techniques to refine dependence distances. However, Ribas only discusses perfectly nested loops, and there are some problems with his Theorem 1:

> Given two references $M_v x + m$ and $U_{v,r} y + u$, the refined dependence distance from x to y is constant iff $M_v = U_{v,r}$.

In our Example 5, we have $M_v = U_{v,r}$ (using Ribas's terminology), but the dependence distance is not constant. The error is that (6) in [Rib90] should include $(y - \delta^i_{v,r}(y)) \in Int(A, b)$ and (7) in [Rib90] should include $(x + \delta^i_{v,r}(x)) \in Int(A, b)$. Ribas's Theorem holds only for iterations not near the beginning or end of any loop.

Ribas uses a slightly different definition of "constant dependence distance" than we do. His definition states that a dependence from A to B has constant distance d iff for all iteration vectors $\mathbf{i} \in [A]$ and $\mathbf{j} \in [B]$, there is a flow dependence from $A(\mathbf{i})$ to $B(\mathbf{j})$ iff $\mathbf{j} - \mathbf{i} = d$. The definition we use is that a dependence from A to B has constant distance d iff for all iteration vectors $\mathbf{i} \in [A]$ and $\mathbf{j} \in [B]$, a flow dependence from $A(\mathbf{i})$ to $B(\mathbf{j})$ implies $\mathbf{j} - \mathbf{i} = d$. While Ribas's definition is useful in the context of deriving VLSI designs, our definition is more appropriate for standard compiler optimizations.

Rosene [Ros90] extended standard scalar data flow analysis techniques by using Data Access Descriptors [BK89] to keep track of an approximation of the set of array elements that are defined, modified and/or killed by each statement. Rosene only determines which levels carry a dependence, and doesn't calculate the dependence distance or direction. Thus, his approach would be unable to handle our Example 6. His use of Data Access Descriptors means that his techniques are approximate in situations in which our methods are exact. It should be possible to modify his tests to use integer programming constraints to define sets of array elements, but that would involve significant work beyond that described in [Ros90] (the Omega test could be used to represent array regions, but the Omega test cannot directly form the union of two sets of constraints). Rosene's techniques have not been fully implemented.

Thomas Gross and Peter Steenkiste describe [GS90] methods similar to that of Rosene. Gross and Steenkiste's work is not as thorough as that of Rosene's.

```
      SUBROUTINE CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDB, B)
C
C     CHOLESKY DECOMPOSITION/SUBSTITUTION SUBROUTINE.
C
C     11/28/84  D H BAILEY  MODIFIED FOR NAS KERNEL TEST
C      1/28/92  W W PUGH    PERFORMED FORWARD SUB. AND
C                           NORMALIZED LOOP THAT HAD STEP OF -1
C
      REAL A(0:IDA, -M:0, 0:N), B(0:NRHS, 0:IDB, 0:N), EPSS(0:256)
      DATA EPS/1E-13/
C
C     CHOLESKY DECOMPOSITION
C
      DO 1 J = 0, N
C
C     OFF DIAGONAL ELEMENTS
C
        DO 2 I = MAX(-M,-J), -1
          DO 3 JJ = MAX(-M,-J) - I, -1
            DO 3 L = 0, NMAT
3             A(L,I,J) = A(L,I,J) - A(L,JJ,I+J) * A(L,I+JJ,J)
          DO 2 L = 0, NMAT
2           A(L,I,J) = A(L,I,J) * A(L,0,I+J)
C
C     STORE INVERSE OF DIAGONAL ELEMENTS
C
        DO 4 L = 0, NMAT
4         EPSS(L) = EPS * A(L,0,J)
        DO 5 JJ = MAX(-M,-J), -1
          DO 5 L = 0, NMAT
5           A(L,0,J) = A(L,0,J) - A(L,JJ,J) ** 2
        DO 1 L = 0, NMAT
1         A(L,0,J) = 1. / SQRT ( ABS (EPSS(L) + A(L,0,J)) )
C
C     SOLUTION
C
      DO 6 I = 0, NRHS
        DO 7 K = 0, N
          DO 8 L = 0, NMAT
8           B(I,L,K) = B(I,L,K) * A(L,0,K)
          DO 7 JJ = 1, MIN (M, N-K)
            DO 7 L = 0, NMAT
7             B(I,L,K+JJ) = B(I,L,K+JJ) -  A(L,-JJ,K+JJ) * B(I,L,K)
C
        DO 6 K = 0, N
          DO 9 L = 0, NMAT
9           B(I,L,N-K) = B(I,L,N-K) * A(L,0,N-K)
          DO 6 JJ = 1, MIN (M, N-K)
            DO 6 L = 0, NMAT
6             B(I,L,N-K-JJ) = B(I,L,N-K-JJ) - A(L,-JJ,N-K) * B(I,L,N-K)
C
      RETURN
      END
```

Figure 2: Source code for CHOLSKY, one of the original NASA NAS benchmark kernals

However, they have implemented their approach, and obtained some experience with it.

4.7 Experimental Results

We have implemented the extensions to the Omega test described in Section 3, and have added tests from section 4 to an augmented version of Wolfe's `tiny` tool [Wol91]. Our efforts to date have focused on testing flow dependences, so our changes have no effect on the output or anti dependences computed, and we do not test for terminating or covering output dependences.

We have performed an analysis of the time taken by the Omega test to analyze dependencies. These timing figures were measured on on Sun Sparc IPX, and are inclusive: they include the time required to scan the loop bounds and subscriptions, the time required to build and analyze the array pair, and the cost of overhead routines such as malloc and free. We ran our tests on the program CHOLSKY from the original NASA NAS benchmark kernals, the source files that were originally distributed all the `tiny` source files distributed with `tiny`, (which include Cholesky decomposition, LU decomposition, several versions of wavefront algorithms, and several more contrived examples), as well as several of our own test programs. Unfortunately, FORTRAN

FROM	TO	dir/dist	status
3: A(L,I,J)	3: A(L,I,J)	(0,0,1,0)	[r]
3: A(L,I,J)	2: A(L,I,J)	(0,0)	
2: A(L,I,J)	3: A(L,I+JJ,J)	(0,+)	
2: A(L,I,J)	3: A(L,JJ,I+J)	(+,*)	
2: A(L,I,J)	5: A(L,JJ,J)	(0)	[C]
2: A(L,I,J)	7: A(L,-JJ,K+JJ)		[C]
2: A(L,I,J)	6: A(L,-JJ,N-K)		[C]
4: EPSS(L)	1: EPSS(L)	(0)	[Cr]
5: A(L,0,J)	5: A(L,0,J)	(0,1,0)	[r]
5: A(L,0,J)	1: A(L,0,J)	(0)	
1: A(L,0,J)	2: A(L,0,I+J)	(+)	
1: A(L,0,J)	8: A(L,0,K)		[C]
1: A(L,0,J)	9: A(L,0,N-K)		[C]
8: B(I,L,K)	7: B(I,L,K)	(0,0)	[C]
8: B(I,L,K)	9: B(I,L,N-K)	(0)	[C]
8: B(I,L,K)	6: B(I,L,N-K-JJ)	(0)	[C]
7: B(I,L,K+JJ)	8: B(I,L,K)	(0,1)	[r]
7: B(I,L,K+JJ)	7: B(I,L,K+JJ)	(0,1,-1,0)	[r]
9: B(I,L,N-K)	6: B(I,L,N-K)	(0,0)	[C]
6: B(I,L,N-K-JJ)	9: B(I,L,N-K)	(0,1)	[r]
6: B(I,L,N-K-JJ)	6: B(I,L,N-K-JJ)	(0,1,-1,0)	[r]

Figure 3: Live flow dependencies for CHOLSKY

FROM	TO	dir/dist	status
3: A(L,I,J)	3: A(L,I+JJ,J)	(0,+,*,0)	[k]
3: A(L,I,J)	3: A(L,JJ,I+J)	(+,*,*,0)	[k]
3: A(L,I,J)	5: A(L,JJ,J)	(0)	[k]
3: A(L,I,J)	7: A(L,-JJ,K+JJ)		[k]
3: A(L,I,J)	6: A(L,-JJ,N-K)		[k]
5: A(L,0,J)	2: A(L,0,I+J)	(+)	[k]
5: A(L,0,J)	8: A(L,0,K)		[k]
5: A(L,0,J)	9: A(L,0,N-K)		[k]
8: B(I,L,K)	6: B(I,L,N-K)	(0)	[c]
7: B(I,L,K+JJ)	7: B(I,L,K)	(0,1,*,0)	[kr]
7: B(I,L,K+JJ)	9: B(I,L,N-K)	(0)	[k]
7: B(I,L,K+JJ)	6: B(I,L,N-K)	(0)	[c]
7: B(I,L,K+JJ)	6: B(I,L,N-K-JJ)	(0)	[k]
6: B(I,L,N-K-JJ)	6: B(I,L,N-K)	(0,1,*,0)	[kr]

Figure 4: Dead flow dependencies for CHOLSKY

programs must be translated by hand to restricted language before `tiny` can analyze them, and so we have been unable to to try our analysis methods on large benchmarks. However, we feel the data presented here gives a good feel for the range of analysis times, even if it cannot be used to predicate an average analysis time.

4.7.1 Effects on flow dependences

We have found that, in many cases, the techniques described here significantly simplify the set of flow dependences. Figure 2 shows our version of the NAS kernel test that performs the Cholesky decomposition of a set of banded matrices. We have modified the original test by forward-substituting the expression $MAX(-M,-J)$, which was originally computed at the top of the J loop, and by normalizing the second K loop (which had a negative step).

Figure 3 lists all of the flow dependencies that are the Omega test has determined are live (i.e., not dead). The dependences in Figure 3 that have been refined are marked with the suffix $[r]$. Those that cover their read have been marked with $[C]$ (note that this tag does not affect the dependence itself, but shows that it can be used to eliminate other dependences).

The flow dependences in Figure 4 are dead: because of an intermediate write, no actual data flows from the source to the destination. These dependencies have been eliminated by being either covered (marked with $[c]$) or killed ($[k]$). Almost all other dependence analysis algorithms would report these as true flow dependencies.

4.7.2 Efficiency

Figure 6 shows timing results for 417 write/read access pairs in CHOLSKY and a variety of other test programs. The graph on the left compares the time needed to perform an extended analysis of an array pair that includes checking for refinement and coverage against the cost of standard analysis (no checks for refinement or covering). Both times were obtained using the Omega test to perform the analysis. The solid lines show $y = x$, $y = 2x$, and $y = 4x$. In 264 cases, the extended capabilities of the Omega test were not needed (i.e., we could determine that refinement and coverage were not possible without needing to consult the Omega test). The 81 $\star$'s show the cases in which we performed the general test for either covering or refinement on one flow dependence vector. These generally took 2 or 3 times the amount of time needed to generate the dependence. The 72 $\diamond$'s show the cases in which we performed the general test for covering or refinement and in which we were forced to split the dependence into several dependence vectors.

The graph on the right of Figure 6 contains one point for each pair of dependences to the same read access (that is, one point for each possible kill). It compares the time needed to test for a kill (horizontal axis) to the time needed to generate and perform coverage and refinement testing of the dependence being killed (vertical axis). The solid line shows $y = x$. Note that one point on the left graph may correspond to zero, one, or many points on the right. The 284 points with kill time below 0.3 msecs correspond to cases in which we did not have to consult the Omega test to perform a kill test; there were 54 cases in which the Omega test was consulted, leading to kill test times of longer than 0.3 msecs.

In Figure 7, we show the time required to perform standard and extended analysis of each array pair (extended analysis includes checks for refinement, covering and killing). The timing results were sorted according to the times required for extended analysis.

5 Symbolic dependence analysis

A data dependence may only exist if certain variables take on particular values. In Example 7, there is a flow dependence iff $2\mathtt{x} \leq \mathtt{n} \wedge 1 \leq \mathtt{y} \leq \mathtt{m} \wedge (\mathtt{x} > 0 \vee \mathtt{x} = 0 \wedge \mathtt{y} < \mathtt{m})$. We can determine the set of constraints under which a dependence exists by setting up the integer programming problem describing when there is a dependence, and projecting the problem onto the set of loop-invariant scalar variables. If we are interested in the set of constraints under which a flow dependence exists, we first determine a set of restraint vectors (Section 2.1.2) for the flow dependence, assuming nothing about the symbolic variables. We then add the restraint vectors to the integer programming problem before projecting onto the symbolic variables.

Alternatively, we can add any user assertions about the relations between variables to any integer programming problem involving those variables.

What if we have some information about relations between variables, but not enough to rule out a dependence? We use our ability to compute gists (Section 3.3) to determine the appropriate concise queries to make of the user, given what we already know. We consider the analysis of Example 7, in the circumstance that that the user has asserted that all array references are in bounds, $50 \leq \mathtt{n} \leq 100$ and no additional information is available about n. There are two apparent restraint vectors for this dependence: $(+,*)$ and $(0,+)$.

For the first restraint vector (which corresponds to a dependence carried by the outer loop), we define p as the things that are known plus the fact that there the outer loop has multiple iterations (which must be true in order for a dependence carried by that loop to be interesting). We define q to be the additional things we known when there actually is a dependence. The constraints that define p and q for this example are shown in Figure 5. The conditions on variables (other than n) under which this dependence actually exists,

```
real A[1:n,1:m], C[1:n,1:m]
...
for L1 := x to n do
  for L2 := 1 to m do
    A[L1,L2] := A[L1-x,y] + C[L1,L2];
```

Example 7

```
int Q[1:n]
...
for L1 := 1 to n do
  A[Q[L1]] := A[Q[L1+1]-1] + C[L1];
```

Example 8

```
for b := 1 to maxB
  for i := B[b] to B[b+1]-1
    for j := B[b] to B[b+1]-1
      A[i,j] := ...
```

Example 9

```
for i := 1 to n do
  for j := 1 to n do
    A[i*j] := ...
```

Example 10

```
for i := 1 to n do
  k := i*(i-1)/2+i
  for j := i to n do
    a[k] := a[k] + bb[i,j]
    k := k+j
```

Example 11

$$p = \left\{ \begin{array}{ll} \begin{pmatrix} \mathbf{x} \le \mathbf{i}_1 < \mathbf{j}_1 \le \mathbf{n} \\ 1 \le \mathbf{i}_2, \mathbf{j}_2 \le \mathbf{m} \end{pmatrix} & \text{loop bounds and restraint vector} \\ \begin{pmatrix} 1 \le \mathbf{i}_1, \mathbf{j}_1 - \mathbf{x} \le \mathbf{n} \\ 1 \le \mathbf{i}_2, \mathbf{y} \le \mathbf{m} \end{pmatrix} & \text{array references in bounds} \\ 50 \le \mathbf{n} \le 100 & \text{user assertion} \end{array} \right\}$$

$$q = \left\{ \begin{pmatrix} \mathbf{i}_1 = \mathbf{j}_1 - \mathbf{x} \\ \mathbf{i}_2 = \mathbf{y} \end{pmatrix} \quad \text{dependence exists} \right\}$$

Figure 5: Set-up of equations for symbolic analysis of outer loop carried dependence Example 7

given that we know p, are given by (gist $\pi_{\mathbf{x},\mathbf{y},\mathbf{m}}(p \wedge q)$ given $\pi_{\mathbf{x},\mathbf{y},\mathbf{m}}(p)$) $= \{1 \le \mathbf{x} \le 50\}$. For the restraint vector $(0,+)$, we similarly compute that it exists only if $\{\mathbf{x} = 0 \wedge \mathbf{y} < \mathbf{m}\}$. We can then ask the user whether or not this condition must always hold.

What about expressions other than scalar loop-invariant variables (such as `i*j` or `P[i]`) that appear in a subscript or loop bound? In this case, we add a different symbolic variable for each appearance of the expression. If the expression is parameterized by a set of other symbolic variables, we also introduce additional symbolic variables for those parameters. We can now use the methods described above to ask the user queries about the relations between these symbolic variables.

In Example 8, we first check for an output dependence, assuming nothing about Q. This leads to an output dependence with direction/restraint vector $(+)$. We next take the set of constraints for determining if there is a dependence and, constraints that enforce the restraint vector $(+)$, and add variables for the index array subscripts (s_1 and s_2) and the index array values (Q_{s_1} and Q_{s_2}). Given that all array references are in bounds and the dependence has a restraint vector $(+)$, we set-up p and q as:

$$p = \left\{ \begin{array}{l} 1 \le i_1 < j_1 \le \mathbf{n} \\ 1 \le Q_s, Q_{s'}, s, s' \le \mathbf{n} \\ s = i_1 \wedge s' = j_1 \end{array} \right\}$$

$$q = \{\ Q_s = Q_{s'}\ \}$$

We then determine that:

$$(\text{gist } \pi_{s,s',Q_s,Q_{s'},n}(p \wedge q) \text{ given } \pi_{s,s',Q_s,Q_{s'},n}(p)) \equiv Q_s = Q_{s'}$$

This would prompt us to ask the user the following:

```
Is it the case that for all a & b such that
1 <= a < b <= n, the following never happens?

        Q[a] = Q[b]
```

If the user answers yes, we rule out an output dependence and add $\forall a \& b$ s.t. $1 \le a < b \le n, Q[a] \ne Q[b]$ as an assertion.

Checking for a flow dependence would produce the query:

```
Is it the case that for all a & b such that
1 <= a < b-1 <= n-1, the following never happens?

        Q[a] = Q[b]-1
```

Instead of answering such a question directly, the user may choose to tell us more specifically what properties the array has. For example, the user might tell us that the array is strictly increasing, or is a permutation array. This has the advantage of being more natural to the user, and possibly supplying more information than a yes/no answer would.

By applying these techniques, we can handle a wide range of situations. These techniques apply directly to situations where array values appear in loop bounds

(such as Example 9). We handle non-linear terms (such as `i*j` in Example 10) as an array indexed by all the non-constant variables. In other words, a term `i*j` would be treated as an array $Q[i, j]$, with the actual term substituted whenever conducting a dialogue with the user. By adding additional algorithms that perform non-linear induction variable recognition and recognize summations and by knowning appropriate linear constraints on summations, these techniques allow us to handle Example 11 (from program `s141` of [LCD91]), which could not be handled by any compiler tested by [LCD91].

5.1 Related Work

Methods for incorporating assertions about invariant scalar variables into dependence analysis algorithms and producing queries to ask the user have been part of the compiler folklore for some time (see [HP91] for a recent discussion). However, previous work has not addressed how to ask concise questions given that some information is already known.

Kathryn McKinley [McK90] describes how to handle index arrays in dependence analysis. Her work enumerates many typical cases and discusses how each can be handled. It is not a general purpose method and cannot handle cases such as array values in loop bounds or complicated subscripts of index arrays. Special purpose methods may prove useful from an efficiency viewpoint for dealing with typical, common cases. Our goal here is to describe as general a method as possible to fall back on.

6 Availability

An implementation of the Omega test is freely available for anonymous ftp from `ftp.cs.umd.edu` in the directory `pub/omega`. The directory contains a standalone implementation of the Omega test, papers describing the Omega test, and an implementation of Michael Wolfe's `tiny` tool [Wol91] augmented to use the Omega test as described in this paper.

7 Conclusions

We have shown how the Omega test can be extended and utilized to answer a wide range of questions that previous analysis methods could not address. The primary questions we considered are

- array kills,
- handling assertions and generating a dialog about the values of scalar variables, and
- handling assertions and generating a dialog about array values and non-linear expressions.

While previous methods could handle special cases of the problems considered here, our work describes much more general methods.

Previous approaches to these problems have not been widely implemented. By taking advantage of the power of the Omega test, we have been able to add these advanced data dependence analysis capabilities with relatively modest implementation investment. We hope that our approach will lead to a more widespread incorporation of these capabilities in compilers and interactive analysis tools.

8 Acknowledgements

Thanks to Udayan Borkar and Wayne Kelly for their help in obtaining the experimental results and their comments on the paper.

Also, special thanks to Michael Wolfe for making his `tiny` program freely available.

References

[BK89] Vasanth Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, 1989.

[Ble75] W.W̃. Bledsoe. A new method for proving certain presburger formulas. In *Advance Papers, 4th Int. Joint Conference on Artif. Intell.*, Tibilisi, Georgia, U.S.S.R, 1975.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelism. In *Proc of 1988 International Conference on Supercomputing*, July 1988.

[CP91] D. Y. Cheng and D. M. Pase. An evaluation of automatic and interactive parallel programming tools. In *Supercomputing '91*, November 1991.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[GS90] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software – Practice and Experience*, 20:133–155, February 1990.

[HKK+91] M. W. Hall, T. Karvey, K. Kennedy, N. McIntosh, K.S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences from the parascope editor workshop. Technical Report RICE COMP TR91-173, Dept. of Computer Science, Rice University, September 1991.

[HP91] Mohammand Reza Haghighat and Costantine D. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances in Languages and Compilers for Parallel Processing*. The MIT Press, 1991.

[JM87] Farnam Jahanian and Aloysius Ka-Lau Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, 1987.

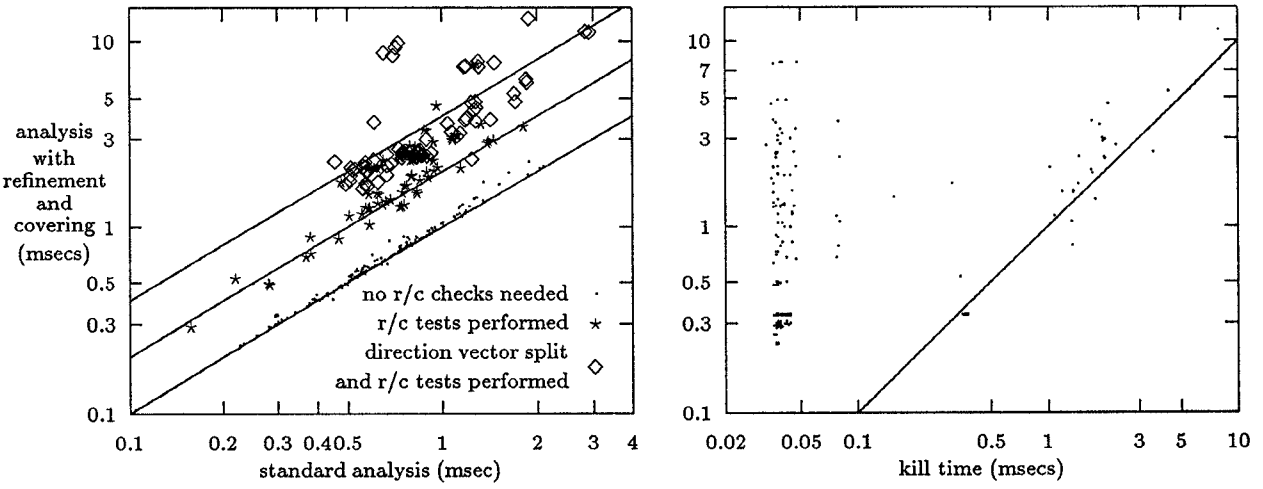

Figure 6: Relative times of standard analysis, analysis time including computing refinement and covering infomation, and time to compute killing information

[KPK90] David Klappholz, Kleanthis Psarris, and Xiangyun Kong. On the perfect accuracy of an approximate subscript analysis test. In *Supercomputing '90*, November 1990.

[LCD91] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. Technical Report MCS-P218-0391, Argonne National Laboratory, April 1991.

[McK90] Kathryn S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report RICE COMP TR91-162, Dept. of Computer Science, Rice University, December 1990.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, Nov 1991. To appear in Communications of the ACM.

[Rib90] Hudson Ribas. Obtaining dependence vectors for nested-loop computations. In *Proc of 1990 International Conference on Parallel Processing*, August 1990.

[Ros90] Carl Rosene. *Incremental Dependence Analysis.* PhD thesis, Dept. of Computer Science, Rice University, March 1990.

[Sho77] Robert E. Shostak. On the sup-inf method for proving presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[SLY89] Z. Shen, Z. Li, and P. Yew. An emperical student of array subscripts and data dependences. In *Proc of 1989 International Conference on Parallel Processing*, August 1989.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.

[Wol91] Michael Wolfe. The tiny loop restructuring research tool. In *Proc of 1991 International Conference on Parallel Processing*, 1991.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.

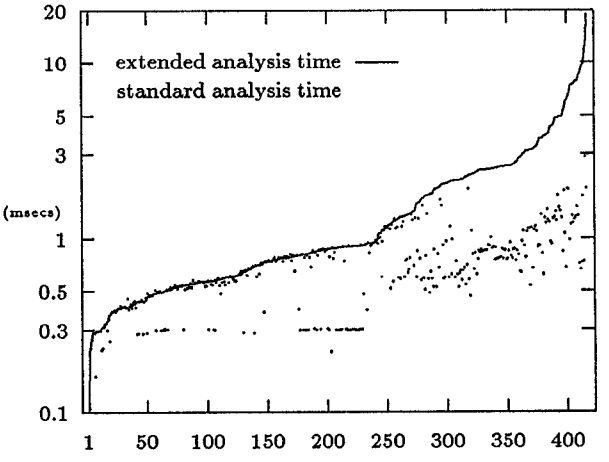

Figure 7: Graph of analysis time per array pair, with and without extended analysis, sorted in order of extended analysis time