UMIACS-TR-93-132 December, 1992
CS-TR-3191

Going beyond Integer Programming with the Omega Test to Eliminate False Data Dependences

William Pugh
pugh@cs.umd.edu
Institute for Advanced Computer Studies
Dept. of Computer Science

David Wonnacott
davew@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

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.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship. This is a revised version of a paper that originally appeared at the ACM SIGPLAN PLDI'92 conference

1 Introduction

Recent studies [HKK$^+$93, CP91] suggest that array data dependence testing 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, May92]. 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).

For example, there is a flow dependence from an array access $A(\mathcal{I})$ to an array access $B(\mathcal{I}')$ iff

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

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 larger subclass of Presburger formulas [KK67, Coo72]. Presburger formulas are those that can be built by applying the first order logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$ and $\exists$) to equality and inequality constraints on sums of integer variables and integer constants. We can use these primitives to handle multiplication by integer constants (e.g. we treat $3n \leq 2m$ as $n + n + n \leq m + m$) or subtraction (treating $a-b = c$ as $a = c+b$). Presburger formulas are decidable, but the fastest known decision procedures that handle the full class take worst-case $2^{2^{2^{O(n)}}}$ time [Coo72, Opp78].

Our original work on the Omega test [Pug92] 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, questions that allow us to obtain more precise information about data dependences: Section 4 gives techniques to distinguish dependences that are merely artifacts of the re-use of memory from those that describe to the flow of values within the program. In Section 5, we show how to produce information about the effects of the values of symbolic constants on dependences. 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.

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

Figure 1: Code demonstrating disagreement about dependence distance definition

2 Dependence Abstractions: Dependence Differences/Distances/Directions

We generally need to know more information about data dependence than just its existence. Things we need to know might include:

- Is the dependence carried by a loop?
  - If so, which one?
- Does the dependence prevent loop interchange?
  - If not, does loop interchange change which loop carries the dependence?

One way to characterize a data dependence is by the difference between the values of the shared loop variables. We call this the dependence difference. The term dependence distance is generally used by other authors to refer to the same thing, but the term dependence distance does not have a widely accepted definition for unnormalized loops. The different definitions and the issues behind them are discussed in [Pug93]. For example, depending on who you ask, you will be told that the dependence distance for code in Figure 1 is $(1,-2),(1,-1),(1,\frac{-2}{3})$, or you might even be told (incorrectly) that no dependence exists.

Often, there will not be a single unique dependence difference for an array pair. In this case, we need to appropriately summarize the dependence differences. One way to do this is by summarizing the possible signs of the dependence differences. For example, the dependence differences $\{(\Delta i,\Delta j) \mid 0 \leq \Delta j < \Delta i\}$ can be summarized as $\{(+,0+)\}$ and the dependence differences $\{(\Delta i,\Delta j) \mid 0 \leq \Delta i = \Delta j\}$ can be summarized as $\{(0,0),(+,+)\}$ (summarizing the dependence differences for this example as $\{(0+,0+)\}$ would falsely suggest that the signs $\{(0,+),(+,0)\}$ were possible). A summary of the signs of the dependence difference is closely related to direction vectors (although for the direction vector, the signs are normalized according to the sign of the step in each loop).

The techniques in this paper are intented for dependence testing of sequential programs (to facilitate parallelization). Thus, we can ensure that a data dependence points forward in time by requiring that the outermost nonzero dependence difference have the same sign as the step of the loop. This creates a minor problem: since this constraint cannot be formulated as the conjunction of a set of linear constraints on the dependence differences, we can't create a single set of constraints that have solutions just for dependence differences that point forward in time.

The methods described in [Pug92] produce a summary of the possible dependence differences for a dependence, not taking into account the requirement that dependences point forward in time. This summary information is then filtered so as to only describe dependences that point forward in time. This summary information is in the form of an exact dependence difference (when constant), or the possible signs of the dependence difference (when not constant). The filtering and summarization may split a dependence. For example, a dependence with dependence differences $\{(\Delta i,\Delta j) \mid 0 \leq \Delta i = \Delta j\}$ would be summarized as dependence differences $(0,0)$ and $(+,+)$. From this point onward, these would be considered two, distinct data dependences between these two array references (this holds throughout this paper). These dependence

```
for i := 0 to n do          for i := 0 to n do          for i := 0 to n do
  A[i] := ...                 A[i] := ...                 A[i] := ...
  ...  := A[1]                ...  := A[n-i]              ...  := A[p]
```

Figure 2: Different flow dependences with identical dependence differences

difference summaries are used to constrain the dependence differences appropriately when creating linear constraints describing these dependences.

A consensus is starting to build that dependence difference/distance is not a completely adequate data dependence abstraction [Wol91a]. In particular, the flow dependences in Figure 2 all have (0+), dependence differences, yet completely different techniques are needed to parallelize the loops. Also, dependence difference/distance is not sufficient to check for the legality of loop fusion or interchange of imperfectly nested loops. Although some researchers have suggested more exact dependence abstractions that describe exactly which pairs of iterations are involved in a dependence [Pug91, Fea91, MAL92], we limit our discussion in this paper to dependence difference abstractions.

3 Extending the Omega test

The Omega test [Pug92] 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 difference in each common loop, we can project that set of constraints onto the variables for the dependence difference. The projected system can be efficiently used to determine the dependence differences.

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 $\bigcup_{i=0}^{p} S_i = \pi_x(S) \subseteq T$. 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.

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 [Pug92], 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 [Pug92]). 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 [Pug92], that are almost never needed in practice.

3.2 Determining the validity of certain 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 [Pug92].

Is $p \Rightarrow q$ a tautology? This could not be efficiently answered using the techniques described in [Pug92], 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:

$\forall x, \exists y$ 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 ((gist p given q) $\wedge q) = (p \wedge q)$). Note that (gist p given q) = True $\Leftrightarrow q = (p \wedge q) \Leftrightarrow (q \Rightarrow p)$.

If q is satisfiable, we could compute gist p given q as follows:

gist p given q =
    if p = True then return True
    else let c be a constraint in p

if $p^c_{\neg c} \wedge q$ is satisfiable,
then return $c \wedge$ (gist p^c_{True} given $(q \wedge c)$)
else return gist p^c_{True} given q

where p^{oldc}_{newc} is p with the constraint $oldc$ replaced by $newc$. If q is not satisfiable, gist p given q = True.

Unfortunately, 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.
- If an equation e in p is implied by any two other constraints in p and/or q, e is redundant and not in the gist.

Note: if an equation is determined to be redundant and not in the gist, that equation may not be used to infer that other equations are redundant.

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 implication 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 $\pi_{\neg(y,z)}(p \wedge q)$ given $\pi_{\neg z}(q)$. We could perform this computation by performing the projections independently, and then computing the gists. However, 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 equations as we go. During this projection, black equations are considered both black and red, while red equations are just red. Thus, combining a red and black inequality produces a red inequality. When eliminating a red equality via substitution, we only perform the substitution for red equations. If any black equations involve the variable being eliminated, we must also convert the equality into a pair of inequalities (e.g., convert $x = 1$ into $1 \leq x \leq 1$). 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]. The work of [Ble75, Sho77] cannot handle nested, alternating quantifiers. The work described in [JM87] can only handle constraints of the form $v \leq v' + c$ (for variables v and v' and constant c). These limitations prevent this use of these techniques for the types of dependence analysis problems we need to analyze.

4 Handling array kills

We say there is a "memory-based" dependence between two accesses (executions of reads or writes) of a variable if these accesses refer to the same storage location. Traditional array data dependence tests use this definition of dependence. We say there is a "value-based" dependence if they refer to the same location and if there are no writes to that memory location between the two accesses (i.e, the value that is in that location after the first access reaches the second access). Traditional scalar data dependence tests use this definition of dependence. If there is no value-based dependence between two accesses to a location but there is a memory-based dependence, we say the dependence is "dead" (i.e., "killed" by the intervening write). We refer to value-based dependences as "live" dependences, as they are not dead.

In this section, we give techniques for computing value-based dependences for arrays. Though most research in this area has focused on value-based flow dependences, our techniques can be applied to output and anti dependences as well as flow dependences.

Note that the transitive closure of all dependences is unaffected by our choice of memory-based or value-based dependence testing. Any value-based dependence is also a memory-based dependence. Any memory-based dependence is equivalent to a chain of one or more value-based dependences. Either the dependence is live, or there is a value-based dependence from the first access to the first of the intervening writes, value-based output dependences between consecutive intervening writes, and a value-based dependence from the last of these writes to the second access.

Program transformations that simply re-arrange the order of execution of array accesses must preserve the transitive closure of the flow, output, and anti dependences. Memory-based dependence tests are therefore sufficient to check for the legality of such transformations.

However, we need information about the value-based flow dependences to test for the legality of storage dependence breaking transformations such as variable expansion, privatization, and renaming. Value-based flow dependences represent the flow of information in the program, and must be preserved by any transformation that is to preserve the program semantics. On the other hand, output and anti dependences, and dead flow dependences, occur because a memory location holds several different values during the execution of the program. We can eliminate these dependences by mapping the different values to distinct memory locations using variable renaming, expansion, or privatization, as long as we preserve the flow of information in the program (i.e., the value-based flow dependences).

Our techniques also eliminate dead anti and output dependences, which has little semantic importance but can be useful in interactive environments (to reduce the amount of useless information displayed to the user).

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 a write B 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 A overwrites the elements of the array that will be accessed by B, before B accesses them. In this case, any dependence to B from an access that precedes A is killed by A.

Terminating A write B terminates a read or write A iff 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 B.

Refinement Given a dependence from a write A to a read or write B, it is possible that some executions of A kill the dependences from earlier executions of A. Similarly, given a dependence from a read or write A to a write B, some executions of B may kill dependences to later executions of B. We say

$A, B, \ldots$	Refers to a specific array reference in a program
$\mathcal{I}, \mathcal{I}', \mathcal{I}'', \ldots$	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
$A(\mathcal{I})$	The iteration of reference A when the loop variables have the values specified by $\mathcal{I}$
$A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}')$	The references A and B refer to the same array and the subscripts of $A(\mathcal{I})$ and $B(\mathcal{I}')$ are equal.
$A(\mathcal{I}) \ll B(\mathcal{I}')$	$A(\mathcal{I})$ is executed before $B(\mathcal{I}')$
`Sym`	The set of symbolic constants (e.g., loop-invariant scalar variables)

Figure 3: Notation used in this paper

```
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)
  for L2 = 1 to n do
    a(L2+1) := ...
```

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

the dependence can be refined to a subset D of its dependence differences iff any dependence with a difference not in D is killed by a dependence with difference in D.

The following subsections give the formulae we use to perform the above types of analysis. Our notation (adapted from [ZC91]) is shown in Figure 3.

These formulae need to enforce the constraint that one access precedes another (e.g. $A(\mathcal{I}) \ll C(\mathcal{I}'')$). Since this may not be a convex set of constraints, we perform each test once per dependence difference summary obtained using conventional data dependence analysis.

4.1 Killing dependences

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

$$\begin{aligned} &\forall\, \mathcal{I}, \mathcal{I}'', \mathtt{Sym},\ \mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll C(\mathcal{I}'') \wedge A(\mathcal{I}) \stackrel{sub}{=} C(\mathcal{I}'') \Rightarrow \\ &\quad \exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \ll C(\mathcal{I}'') \wedge B(\mathcal{I}') \stackrel{sub}{=} C(\mathcal{I}'') \end{aligned}$$

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

$$\mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll C(\mathcal{I}'') \wedge A(\mathcal{I}) \stackrel{sub}{=} C(\mathcal{I}'') \quad \equiv \quad \mathcal{I}''_1 = \mathtt{n}$$

$$\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \ll C(\mathcal{I}'') \wedge B(\mathcal{I}') \stackrel{sub}{=} C(\mathcal{I}'') \quad \equiv \quad \mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 10$$

$$\mathcal{I}''_1 = \mathtt{n} \quad \Rightarrow \quad \mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 10$$

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

$$\mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll C(\mathcal{I}'') \wedge A(\mathcal{I}) \stackrel{sub}{=} C(\mathcal{I}'') \quad \equiv \quad \mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 20 \wedge \mathcal{I}''_1 = \mathtt{m}$$

$$\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \ll C(\mathcal{I}'') \wedge B(\mathcal{I}') \stackrel{sub}{=} C(\mathcal{I}'') \quad \equiv \quad \mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 10$$

$$\mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 20 \wedge \mathcal{I}''_1 = \mathtt{m} \quad \not\Rightarrow \quad \mathtt{n} \leq \mathcal{I}''_1 \leq \mathtt{n} + 10$$

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

If there are multiple dependence difference summaries between A and C, we test each one independently to see if it can be killed. If there are multiple dependence summaries from A to B, or from B to C, we perform this test once for each combination of one A to B summary with one B to C summary. Our test will therefore not detect cases in which a some dependence summary from A to C is not killed by a combination of several A to B or several B to C summaries. We are working on generalizing our tests to handle this case and kills by a comb (a group of writes B_1, B_2, ..., B_n, which together kill the dependence). See [PW93] for more details.

4.2 Covering dependences

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

A covers B iff:

$$\forall\ \mathcal{I}', \mathtt{Sym},\ \mathcal{I}' \in [B] \Rightarrow \exists \mathcal{I} \text{ s.t. } \mathcal{I} \in [A] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \wedge A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}')$$

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

$$\mathcal{I}' \in [B] \quad \equiv \quad 1 \leq \mathcal{I}'_1 \leq 100 \wedge 2 \leq \mathcal{I}'_2 \leq \mathtt{n} - 1$$

$$\exists \mathcal{I} \text{ s.t. } \mathcal{I} \in [A] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \wedge A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}') \quad \equiv \quad 1 \leq \mathcal{I}'_1 \leq 100 \wedge 0 \leq \mathcal{I}'_2 \leq \mathtt{n} - 1$$

$$1 \leq \mathcal{I}'_1 \leq 100 \wedge 2 \leq \mathcal{I}'_2 \leq \mathtt{n} - 1 \quad \Rightarrow \quad 1 \leq \mathcal{I}'_1 \leq 100 \wedge 0 \leq \mathcal{I}'_2 \leq \mathtt{n} - 1$$

The level at which the dependence is carried determines which other array accesses must be killed by the cover. In Example 2, the dependence from the write `a(L2-1)` to the read `a(L2)` is loop independent, so it must kill the dependence from the write `a(L1)` to the read. If the dependence from the cover had been carried by the `L1` loop, some writes to `a(L1)` could come after the covering writes. Note that traditional dependence tests would not be able to determine that the dependence from `a(L2-1)` is loop independent.

4.3 Terminating dependences

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

B terminates A iff:

$$\forall\ \mathcal{I}, \texttt{Sym},\ \mathcal{I} \in [A] \Rightarrow \exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \wedge A(\mathcal{I}) \overset{sub}{=} B(\mathcal{I}')$$

In Example 2, the read of `a(L2)` is terminated by the write to `a(L2+1)`.

4.4 Refining dependence differences/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 execution of A with dependence difference $\leq D$, we say the dependence can be refined at its source to D. Example 3 shows a loop with a flow dependence that can be refined (at the source) from (0+,1) to (0,1). Due to space limitations, we are unable to discuss the equations used to analyze refinement; they are given in [PW92].

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 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 sum of the dependence difference from A to B and B to C to equal the dependence difference from A to C. Similarly, for there to be any possibility of refining the dependence from A to B, the write at the end we are refining must have a self-output dependence with a non-zero dependence difference in the loop level being refined.

If there exists a loop l such that the dependence difference from A to B cannot be 0, the dependence from A to B cannot cover B in the first iteration of loop l, so we do not check for coverage. Note that A might 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. If the dependence from B is closer than some subset of the dependences differences from A, we can kill that subset of the dependence differences from A.

4.6 Testing Order

We order our investigation of dependences by the number of loops containing both accesses, examining pairs with the greatest shared loop depth first. As soon as we find a dependence, we test it to determine if it covers its destination or terminates its source, and if it does, we perform a special form of refinement to determine which loop carries the dependence (i.e. we refine leading 0+ to 0 if possible). This ordering lets us avoid doing dependence testing for some access pairs. We can skip the test for dependence between 2 accesses that share l or fewer loops if either access is both covered and terminated by dependences with differences of 0 in the outer $l+1$ loops, or if both accesses are covered by such dependences, or both are terminated by such dependences.

In Example 2, the read of `a(L2)` is covered at level 1 by the write to `a(L2-1)`, and terminated at level 1 by the write to `a(L2+1)`. Our testing order ensures that we will find the cover and terminator before we test for dependence between the read and the write to `a(m)`. We can skip this test entirely, since any flow dependence from the write must be killed by the cover, and any antidependence to the write must be killed by the terminator. Note that since there are no loops enclosing Example 2, there can be no antidependence from the read to the write `a(m)`, so simply knowing about the cover would be sufficient grounds to skip the dependence test.

Once we have completed the basic dependence testing, cover and termination testing, we try to eliminate dependences with the quick kill tests described in 4.5. We apply the complete kill test only to those dependences that could not be killed with the quick test, and finally refine the remaining dependences.

4.7 Related Work

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

- Extend the pair-wise methods typically used for array data dependence to recognize array kills [Bra88, Rib90, Fea91, MAL92, MAL93].
- Extend scalar dataflow methods by recording which array sections are killed and/or defined [GS90, Ros90, Li92].

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

4.7.1 Extending pair-wise methods

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 differences 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 $\mathcal{I} \in [A]$ and $\mathcal{I}' \in [B]$, there is a flow dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ iff $\mathcal{I}' - \mathcal{I} = d$. The definition we use is that a dependence from A to B has constant distance d iff for all iteration vectors $\mathcal{I} \in [A]$ and $\mathcal{I}' \in [B]$, a flow dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ implies $\mathcal{I}' - \mathcal{I} = d$. While Ribas's definition is useful in the context of deriving VLSI designs, our definition is more appropriate for standard compiler optimizations.

Paul Feautrier has described a more detailed form of analysis for array references [Fea91]. His methods are designed to produce exact information: for each read of an array element, he determines the precise statement and iteration which wrote the value. His methods are much more expensive than ours (about 100$\times$ more expensive) and work only for programs with a special static control structure (defined in [Fea91]).

Maydan, Amarasinghe, and Lam ([MAL92, MAL93]) provide an efficient way of generating the information produced by Feautrier's technique under specific conditions. They also present evidence that and show that these specific conditions are frequently satisfied in real programs. Voevodin and Voevodin ([Voe92a], [Voe92b]) have also done work that is similar to Feautrier's.

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

Example 4

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

Example 5

```
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 6

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

Example 7

```
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 8

4.7.2 Extending scalar data-flow methods

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

Zhiyuan Li [Li92] presents a technique for determining whether or not an array is privatizable. His technique, like Rosene's, is based on computing approximations of the sets of array elements defined and used in the body of a loop. He does not calculate a dependence difference, and thus would also be unable to handle our Example 6.

5 Symbolic dependence analysis

A data dependence may only exist if certain variables take on particular values. In Example 4, there is a flow dependence carried by the inner loop iff

$$\exists L1, L2, L1', L2' \text{ s.t. } \mathtt{x} \le L1 = L1' \le \mathtt{n} \wedge 1 \le L2 < L2' \le \mathtt{m} \wedge L1 = L1' - \mathtt{x} \wedge L2 = \mathtt{y}$$

Using the techniques described in [Pug92] we can determine that this is equivalent to: $\mathtt{x} = 0 \wedge 1 \le \mathtt{y} < \mathtt{m} \wedge 0 \le \mathtt{n}$. We could then allow the user to add assertions that would disprove the dependence. Assertions can easily be incorporated into the dependence tests.

Unfortunately, this approach doesn't work so well in practice. For most dependences, the conditions that are produced are very boring:

- conditions that are false only when the loop that carries the dependence either executes only one iteration or doesn't execute at all (for the example above, $0 \le \mathtt{n}$ is false only when the outer loop has zero iterations).

- conditions that can be inferred from other assertions in the program.

We can use the methods described in section 3 to determine the *interesting* conditions under which a dependence exist. More technically, let p be the conditions on the symbolic variables that must be true in order for the dependence to be interesting. At a minimum, this would include:

- anything that can be inferred from analysis of the program,
- user assertions, and
- the fact that the loop that carries the dependence has multiple iterations.

Additional things could be included, such as the fact that both the source and destination of the dependence execute, or whatever could be inferred from the fact that all array references are in bounds. Let q be the conditions on the symbolic variables that must be true in order for the dependence to exist. We then compute gist q given p as the interesting conditions that must be true in order for the dependence to exist.

Almost all dependences in the linear algebra routines and the NAS CHOLSKY routine exist conditionally (since the loops have symbolic bounds, and don't execute under certain conditions). However, the only dependences that can be deleted by adding interesting assertions are the loop independent output and anti-dependences from line 6 to line 7 which exist only when $\mathtt{n} \geq 2$.

We can guide the user's attempts to eliminate dependences symbolically by testing all dependences in advance, to determine whether or not symbolic elimination is possible. We compute the conditions under which the dependence must exist - if this condition is not simply "True", we mark the dependence to show the user that there are conditions under which the dependence can be eliminated.

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 5, we first check for an output dependence, assuming nothing about Q. This leads to an output dependence with dependence difference summary of $(+)$. We next take the set of constraints for determining if there is a dependence and constraints that enforce the dependence difference, 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}). We set-up p and q as:

$$p = \left\{ \begin{array}{c} 1 \leq i_1 < j_1 \leq \mathtt{n} \\ s = i_1 \wedge s' = j_1 \end{array} \right\}$$

$$q = \left\{ \; Q_s = Q_{s'} \; \right\}$$

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. $1 \leq a < b \leq n, Q[a] \neq Q[b]$ 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, 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 6). We handle non-linear terms (such as `i*j` in Example 7) 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 8 (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 stand-alone implementation of the Omega test, papers describing the Omega test, and an implementation of Michael Wolfe's `tiny` tool [Wol91b] 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

This work is supported by NSF grant CCR-9157384 and a Packard Fellowship. 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*, pages 41–53, 1989.

[Ble75] W. W. 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*, pages 407–417, July 1988.

[Coo72] D. C. Cooper. Theorem proving in arithmetic with multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. American Elsevier, New York, 1972.

[CP91] D. Y. Cheng and D. M. Pase. An evaluation of automatic and interactive parallel programming tools. In *Supercomputing '91*, pages 412–423, 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.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[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+93] M. W. Hall, T. Karvey, K. Kennedy, N. McIntosh, K.S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth. Experiences using the parascope editor: an interactive parallel programming tool. In *Principles and Practice of Parallel Programming*, April 1993.

[HP91] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high-performance parallelizing compilers. In *Advances In Languages And Compilers for Parallel Processing*, August 1991.

[JM87] Farnam Jahanian and Aloysius Ka-Lau Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, August 1987.

[KK67] G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic*. North-Holland Pub. Co., 1967.

[KPK90] David Klappholz, Kleanthis Psarris, and Xiangyun Kong. On the perfect accuracy of an approximate subscript analysis test. In *Proc. of the 1990 International Conference on Supercomputing*, pages 201–212, 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.

[Li92] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of the 1992 International Conference on Supercomputing*, pages 313–322, July 1992.

[MAL92] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Data dependence and data-flow analysis of arrays. In *5th Workshop on Languages and Compilers for Parallel Computing (Yale University tech. report YALEU/DCS/RR-915)*, pages 283–292, August 1992.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM '93 Conf. on Principles of Programming Languages*, January 1993.

[May92] Dror Eliezer Maydan. *Accurate Analysis of Array References.* PhD thesis, Computer Systems Laboratory, Stanford U., September 1992.

[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.

[Opp78] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, July 1978.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[Pug93] William Pugh. Definitions of dependence distance. *Letters on Programming Languages and Systems*, September 1993.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 140–151, San Francisco, California, June 1992.

[PW93] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 1993. accepted for publication.

[Rib90] Hudson Ribas. Obtaining dependence vectors for nested-loop computations. In *Proc of 1990 International Conference on Parallel Processing*, pages II–212 – II–219, 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.

[Voe92a] Valentin V. Voevodin. *Mathematical Foundations of Parallel Computing.* World Scientific Publishers, 1992. World Scientific Series in Computer Science, vol. 33.

[Voe92b] Vladimir V. Voevodin. Theory and practice of parallelism detection in sequential programs. *Programming and Computer Software (Programmirovaniye)*, 18(3), May 1992.

[Wol91a] Michael Wolfe. Experiences with data dependence abstractions. In *Proc. of the 1991 International Conference on Supercomputing*, pages 321–329, June 1991.

[Wol91b] Michael Wolfe. The tiny loop restructuring research tool. In *Proc of 1991 International Conference on Parallel Processing*, pages II–46 – II–53, 1991.

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