UMIACS-TR-94-123 November, 1994
CS-TR-3372

Nonlinear Array Dependence Analysis

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

Standard array data dependence techniques can only reason about linear constraints. There has also been work on analyzing some dependences involving polynomial constraints. Analyzing array data dependences in real-world programs requires handling many "unanalyzable" terms: subscript arrays, run-time tests, function calls.

The standard approach to analyzing such programs has been to omit and ignore any constraints that cannot be reasoned about. This is unsound when reasoning about value-based dependences and whether privatization is legal. Also, this prevents us from determining the conditions that must be true to disprove the dependence. These conditions could be checked by a run-time test or verified by a programmer or aggressive, demand-driven interprocedural analysis.

We describe a solution to these problems. Our solution makes our system sound and more accurate for analyzing value-based dependences and derives conditions that can be used to disprove dependences. We also give some preliminary results from applying our techniques to programs from the Perfect benchmark suite.

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

extended abstract

Nonlinear Array Dependence Analysis

William Pugh — pugh@cs.umd.edu, (301) 405-2705

David Wonnacott — davew@cs.umd.edu, (301) 405-2726

Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742

January 6, 1995

Abstract

Standard array data dependence techniques can only reason about linear constraints. There has also been work on analyzing some dependences involving polynomial constraints. Analyzing array data dependences in real-world programs requires handling many "unanalyzable" terms: subscript arrays, run-time tests, function calls.

The standard approach to analyzing such programs has been to omit and ignore any constraints that cannot be reasoned about. This is unsound when reasoning about value-based dependences and whether privatization is legal. Also, this prevents us from determining the conditions that must be true to disprove the dependence. These conditions could be checked by a run-time test or verified by a programmer or aggressive, demand-driven interprocedural analysis.

We describe a solution to these problems. Our solution makes our system sound and more accurate for analyzing value-based dependences and derives conditions that can be used to disprove dependences. We also give some preliminary results from applying our techniques to programs from the Perfect benchmark suite.

1 Introduction

Standard algorithms for determining if two array references are aliased (i.e., might refer to the same memory location) are posed in terms of checking to see if a set of linear constraints has an integer solution. This problem is NP-Complete [GJ79]. Both approximate and exact algorithms have been proposed for solving this problem.

Unfortunately, many array dependence problems cannot be exactly translated into linear constraints, such as Example 1. The lhs and rhs of the assignment statement might be aliased if and only if

$$\exists a, b, c, p, n \text{ s.t. } 1 \leq a, b, c \leq n \wedge p > 2 \wedge a^p + b^p = c^p$$

Allowing arbitrary constraints makes checking for solutions undecidable (not to mention difficult). Disproving the above conditions is the same as proving Fermat's last theorem.

The standard approach in cases such as this is to simply omit any non-linear constraints when building the set of constraints to be tested. For the above example, we would simply check the constraints:

$$\exists a, b, c, p, n \text{ s.t. } 1 \leq a, b, c \leq n \wedge p > 2$$

Omitting the non-linear constraints gives us an upper bound on the conditions under which the array references are aliased. Checking this upper bound for solutions will give us a conservative solution, which is what we need if we are to prove the safety of a program transformation that requires independence.

```
if p > 2 then
  for a := 1 to n
    for b := 1 to n
      for c := 1 to n
        a[a^p+b^p] = a[c^p]
```

Example 1

```
for i := 1 to n do
  for j := 1 to p do
    A[i,j] := A[i-x,j] + C[j]
```

Example 2

```
for i := 1 to 100 do
  A[P[i]] := C[i]
```

Example 3

However, there are a number of situations in which this approach is unsatisfactory. It is inadequate when we want to know the conditions under which a solution exists, rather than just testing for the potential existence of a dependence, or when we wish to produce dependence information that corresponds to the flow of values in the program (rather than memory aliasing).

1.1 Symbolic dependence analysis

We represent data dependences as relations between tuples of integer variables. The input tuple represents the values of the loop index variables at the source of the dependence, and the output tuple the values of the loop index variables at the sink. We use a Presburger formula (see Section 2) to constrain these tuples to the iterations that are connected by a dependence. Note that our techniques should not be blindly applied to all scalar dependences in a program, as other techniques can produce the same results more efficiently ([SW94]).

Consider Example 2: we can describe the flow dependence carried by the outer loop as a relation from source iteration $\mathrm{i,j}$ to destination iteration $\mathrm{i',j'}$:

$$\{[\mathrm{i,j'}] \rightarrow [\mathrm{i',j'}] \mid 1 \le \mathrm{i} < \mathrm{i'} \le \mathtt{n} \wedge 1 \le \mathrm{j,j'} \le \mathtt{p} \wedge \mathrm{i} = \mathrm{i'} - \mathtt{x} \wedge \mathrm{j} = \mathrm{j'}\}$$

Since all the terms are affine, we can use techniques described in [PW92] to compute the conditions on symbolic constants ($\mathtt{x}, \mathtt{n}$ and $\mathtt{p}$) that must be true in order for a flow dependence to exist:

$$1 \le \mathtt{x} < \mathtt{n} \wedge 1 \le \mathtt{p}$$

Once we compute these conditions, we might use more powerful analysis techniques to see if they can be disproved, allow the user the assert that they are false, and/or check at run-time to see if they are false. As described in [PW92], we can eliminate the test $1 \le \mathtt{p}$ as uninteresting.

If a program contains non-linear expressions and we simply omit the corresponding non-linear constraints from the dependence problem, we will be unable to accurately compute necessary and sufficient conditions for the dependence to exist. For example, if we omit the non-linear terms in

$$\{[\mathrm{i}] \rightarrow [\mathrm{i'}] \mid 1 \le \mathrm{i} < \mathrm{i'} \le 100 \wedge \mathtt{P}[\mathrm{i}] = \mathtt{P}[\mathrm{i'}]\}$$

(the relation describing the output dependence in Example 3), we would conclude that a dependence is inevitable. However, if we include the non-linear constraints, we can determine that there is a dependence iff `P[1:100]` contains repeated elements.

1.2 Computing value-based dependences

Standard array data dependence tests only determine if two array references touch the same memory location; they are oblivious to intervening writes. For a number of program optimizations and transformations, it is desirable to also compute "value-based" dependences [Fea88, PW92, PW93, PW94, Mas94], in which there are no intervening writes. In our approach [PW93], we start with a set of constraints describing the iterations that are aliased and then subtract out the pairs clobbered by an intervening write. For example, the memory-based flow dependence in Example 4 is

$$\{[\mathrm{i,j}] \rightarrow [\mathrm{i',j'}] \mid 1 \le \mathrm{i} \le \mathrm{i'} \le \mathtt{n} \wedge 1 \le \mathrm{j} = \mathrm{j'} \le \mathtt{m}\}.$$

When we subtract out the pairs clobbered by an intervening write:

$$\{[\mathrm{i,j}] \rightarrow [\mathrm{i',j'}] \mid 1 \le \mathrm{i} < \mathrm{i'} \le \mathtt{n} \wedge 1 \le \mathrm{j} = \mathrm{j'} \le \mathtt{m}\}$$

We find that values are only communicated within iterations of the `i` loop:

$$\{[\mathrm{i,j}] \rightarrow [\mathrm{i',j'}] \mid 1 \le \mathrm{i} = \mathrm{i'} \le \mathtt{n} \wedge 1 \le \mathrm{j} = \mathrm{j'} \le \mathtt{m}\}$$

This approach cannot be applied with conservative approximations of the dependences: subtracting an upper bound from an upper bound gives something that is neither an upper bound nor a lower bound. So we can't just blindly omit non-linear constraints if we wish to compute value-based dependences, even if we are willing to settle for a conservative approximation. If we tried to do so in Example 5, we would conclude that there cannot be a loop-carried flow dependence, even though there can. In Section 5.1, we will see that our techniques produce a correct, though approximate, result for this example. In some cases, we can produce exact value-based dependence information despite the presence of non-linear terms (Section 5's Example 11 shows one such case).

```
for i := 1 to n do
  for j := 1 to m do
    work[j] := ...
  for j := 1 to m do
    ... := work[j]
```

Example 4

```
for i := 1 to n do
  for j := 1 to L[i] do
    work[j] := ...
  for j := 1 to m do
    ... := work[j]
```

Example 5

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

Example 6

1.3 Representing control-flow information

There have been two approaches to the calculation of array dataflow information. Some methods, like ours, are based on extensions of array data dependence techniques. Others are based on extensions of scalar dataflow equations to deal with array sections. In general, the former are better at dealing with complicated subscripts, and the latter handle more complicated control flow. As we will see in Section 3.1, a sufficiently rich constraint language will let us handle control flow constructs other than **for** loops and structured **if**'s.

When we want to analyze value-based dependences or find the conditions under which a dependence exists, we must have some way to avoid making approximations whenever we are faced with a non-linear term. This paper describes our use of constraints containing *uninterpreted function symbols* to represent non-linear expressions from the program, and thus avoid the problems described above. In Section 2, we describe the class of constraints that we can manipulate (Appendix A gives some details of our implementation). In Section 3, we show how to use uninterpreted function symbols for program analysis. Sections 4 and 5 show that these more powerful constraints let us perform accurate symbolic and value-based dependence analysis in the presence of non-linear terms. We present our conclusions in Section 7.

2 Presburger Formulas with Uninterpreted Function Symbols

Presburger formulas ([KK67]) are those formulas that can be constructed by combining affine constraints on integer variables with the logical operations $\wedge$, $\vee$, and $\neg$, and the quantifiers $\forall$ and $\exists$. For example, the formulas we constructed for the analysis of Examples 2 and 4 in Section 1 are Presburger formulas. There are a number of algorithms for testing the satisfiability of arbitrary Presburger formulas ([KK67, Coo72, PW93]). This problem appears to have worst-case complexity of $2^{2^{2^{O(n)}}}$ [Opp78].

We formulate value-based array data dependence in terms of (simple) Presburger formulas, with only two nested alternating quantifiers. Memory based array data dependence can be formulated in terms of even simpler Presburger formulas, with only a single quantifier. Fortunately, the formulas we generate for array dependence analysis of real programs can be solved quite efficiently [PW93].

Presburger arithmetic can be extended to allow *uninterpreted function symbols*: terms representing the application of a function to a list of argument terms. The functions are termed "uninterpreted" because the only thing we know about them is that they are functions: two applications of a function to the same arguments will produce the same value. The formula we used in the analysis of Example 3 is in this class (since the array `P` is not modified, it is equivalent to a function). Downey ([Dow72]) proved that full Presburger arithmetic with uninterpreted function symbols is undecidable. We will therefore need to restrict our attention to a subclass of the general problem, and produce approximations whenever a formula is outside of the subclass.

Shostak ([Sho79]) developed a procedure for testing the validity of quantifier-free Presburger formulas with uninterpreted function symbols, based on the following observation: Consider a formula F that contains references $f(i)$ and $f(j)$, where i and j are free in F. Let F_0 be F with f_i and f_j substituted for $f(i)$ and $f(j)$. F is satisfiable iff $F' = (((i = j) \Rightarrow (f_i = f_j)) \wedge F_0)$ is satisfiable. Shostak provides several ways of improving over the naive replacement of F with F'.

We are currently restricting our attention to formulas in which all function symbols are free, and functions are only applied to an affine mapping of the input or output tuple of the relation containing the formula. For example, in a relation from $[\mathrm{i},\mathrm{j},\mathrm{k}]$ to $[\mathrm{i}',\mathrm{j}']$, a binary function f could be applied $f(\mathrm{i},\mathrm{j})$, $f(\mathrm{i}',\mathrm{j}')$, or $f(\mathrm{k}+2,\mathrm{i}+\mathrm{j})$. In many cases (and in our current implementation), we only need considering applying a function to a prefix of the input or

```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      if p[i,j,k] < 0 then
        goto L1 // 2 level break
      A[i] := ...
    endfor
  endfor
L1:
endfor
```

Example 7

```
for i := 1 to n
  for j := 1 to n*i do
    a[j] := ...
  endfor
  for j := n*i+1 to m do
    a[j] := ...
  endfor
  for k := 1 to m do
    ... := a[k]
  endfor
endfor
```

Example 8

```
for i := 1 to n
  for j := 1 to m
    for k := 1 to p[i] do
      a[q*i+k] := ...
    endfor
    ...
  endfor
endfor
```

Example 9

output tuple (e.g., the first two applications of f). Note that the formula we used for Example 3 is in this class.

Our techniques are based on adapting the observation used by Shostak to our problem domain. The details of this adaption and its integration with our previous work are covered in Appendix A.

3 Using Function Symbols in Dependence Analysis

Consider a program that contains a nonlinear term, such as Example 6. For each non-linear expression e that is nested within n loops, we use an n-ary function $f(\mathcal{I})$ to represent the value of e in iteration $\mathcal{I}$. For example, the flow dependence in this example is

$$\{[\mathrm{i}] \rightarrow [\mathrm{i}'] \mid 1 \leq \mathrm{i} \leq \mathrm{i}' \leq \mathtt{n} \wedge f_{\mathtt{m*i}}(\mathrm{i}) = \mathrm{i}'\}$$

Of course, we do not know the value of $f_{\mathtt{m*i}}$ at compile time, but we can use it in the constraints that describe a dependence (after all, we're using `n`, and we don't know what it is).

This basic technique works for non-linear expressions in subscript expressions, loop bounds and the guard of conditional branches that do not break out of loops. This technique doesn't work well for breaks (i.e., jumps out of a loop). The problem is that to describe the conditions under which iteration i of a loop executes, we must express the fact that the break was not taken in any previous iteration. We therefore use a slightly different approach for breaks: we create a function symbol who's value is the iteration in which the break is performed. In Example 7, there may be different `break`'s for each iteration of `i`. So we create function symbols $b_j(i)$ and $b_k(i)$ to describe when the break occurs. The constraints describing the iterations of the assignment statement that are executed are:

$$\{[i,j,k] : 1 \leq i,j,k \leq n \wedge (j < b_j(i) \vee j = b_j(i) \wedge k < b_k(i))\}$$

We treat **while** loops as **for** loops with infinite upper bounds (i.e., no upper bound) and a **break** statement.

3.1 More Sophisticated Selection of Functions

The previous representation is somewhat crude: the only thing it captures is that a value computed at some iteration of a statement has some fixed but unknown value. We can do substantially better by recognizing when values computed at different points must be the same. For example, if we use the same function to represent the two occurrences of `n*i` in Example 8, we can show that there are no upwards exposed reads. Also, an expression may be partially loop invariant. In Example 9, the expression `q * i` is independent of the `j` loop, and would be represented as a function of `i`. If `p` is constant, `p[i]` is also independent of `j`.

To summarize, we determine the level at which an expression is loop variant, and create a function symbol with as few arguments as possible. We identify syntactically distinct expressions that should be represented with the same function symbol by an adaptation of global value numbering [AWZ88, RWZ88]. More details will be provided in the full paper.

4 Symbolic Dependence Analysis

In previous work ([PW92]), we discussed a general method for finding conditions that prove independence in programs without non-linear terms, and described some ad-hoc methods for applying our techniques in the presence of such terms. Our use of uninterpreted function symbols gives us a general framework for performing this analysis in the presence of non-linear terms.

4.1 Symbolic dependence analysis without non-linear terms

We begin with a review of our previous work. If a relation contains no function symbols, we first replace the variables representing the iteration space with existentially quantified variables, giving the conditions on the symbolic constants under which a dependence exists somewhere in the iteration space. For example, the recall the flow dependence in Example 2: $\{[\mathrm{i},\mathrm{j}'] \rightarrow [\mathrm{i}',\mathrm{j}'] \mid 1 \leq \mathrm{i} < \mathrm{i}' \leq \mathtt{n} \wedge 1 \leq \mathrm{j},\mathrm{j}' \leq \mathtt{p} \wedge \mathrm{i} = \mathrm{i}' - \mathtt{x} \wedge \mathrm{j} = \mathrm{j}'\}$. This dependence exists iff $1 \leq \mathtt{x} < \mathtt{n} \wedge 1 \leq \mathtt{p}$.

These conditions often contain conditions that are only false only in situations in which we don't care if the loop is parallel (in this case, $1 \leq \mathtt{p}$). We wish to avoid asking the user about such constraints or generating run-time tests for them. We avoid this problem by collecting another set of constraints that give the conditions that must be true for the dependence to be interesting: each surrounding loop executes at least one iteration, the loop that carries the dependence executes more than one, plus any facts about the program provided by user assertions or other analysis.

We then consider only the *gist* of the dependence conditions given these uninteresting conditions ([PW92]). Informally, gist `p` given `q` this is the conditions that are interesting in `p`given that `q`holds. More formally, it is a minimal set of constraints such that (gist `p` given `q`) $\wedge\, \mathtt{q} = \mathtt{p} \wedge \mathtt{q}$.

If we find that there are interesting conditions under which the dependence will not exist, we might choose to compile a run-time test or query the user about whether the conditions can actually occur (for example, the programmer might have knowledge about `x` in Example 2 that is not available to the compiler).

4.2 Symbolic dependence analysis in the presence of non-linear terms

When a Presburger formula contains no function symbols, we can eliminate any arbitrary set of variables. This ability derives from the fact that our methods are based on a variation of Fourier's method of variable elimination, which relies on classifying individual constraints as either upper or lower bounds on the variable to be eliminated. However, a constraint like $p(\mathrm{i}) > 0$ constrains both p and i, but it is neither an upper nor a lower bound on i. Therefore, we cannot eliminate i exactly if $p(\mathrm{i})$ remains in the formula.

This fact limits our ability to eliminate arbitrary variables from a formula. For example, we cannot simply eliminate i and i$'$ from the relation describing the output dependence in Example 3:

$$\{[\mathrm{i}] \rightarrow [\mathrm{i}'] \mid 1 \leq \mathrm{i} < \mathrm{i}' \leq 100 \wedge p(\mathrm{i}) = p(\mathrm{i}')\}$$

We therefore leave in the relation any iteration space variables that are used as function arguments. In this example, we cannot eliminate any variables, and the relation is not changed in the first step of our symbolic analysis. Fortunately, the uninteresting conditions on these variables are eliminated in the second step: the gist of the above relation given $1 \leq \mathrm{i} \leq \mathrm{i}' \leq 100$ is

$$\{[\mathrm{i}] \rightarrow [\mathrm{i}'] \mid p(\mathrm{i}) = p(\mathrm{i}')\}$$

Note that we may wish to include the "uninteresting" conditions in any run-time tests we compile (to check for repeated elements only in `p[1:100]`) or dialogue with the programmer ("`Is it the case that p[i] != p[i']` `whenever` $1 \leq \mathrm{i} < \mathrm{i}' \leq 100$").

4.3 Inductive Simplification

With our methods, we frequently derive dependence breaking conditions of the form:

$$C \equiv \forall \mathrm{i},\mathrm{i}', \mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \Rightarrow Q(\mathrm{i},\mathrm{i}')$$

Verifying this condition with a run-time test typically requires $O(n^2)$ work, since we must check all ordered i,i$'$ pairs. We can often simplify this. We can derive C_1, the subset of C for when $\mathrm{i}' = \mathrm{i} + 1$:

$$C_1 \equiv \forall \mathrm{i} P(\mathrm{i},\mathrm{i}+1) \Rightarrow Q(\mathrm{i},\mathrm{i}+1)$$

Clearly, proving C_1 is a necessary condition to proving C. We can then attempt an automatic inductive proof that $C_1 \Rightarrow C$. We do this by checking if

$$(\mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \Rightarrow Q(\mathrm{i},\mathrm{i}')) \wedge (P(\mathrm{i}',\mathrm{i}'+1) \Rightarrow Q(\mathrm{i}',\mathrm{i}'+1)) \Rightarrow (P(\mathrm{i},\mathrm{i}'+1) \Rightarrow Q(\mathrm{i},\mathrm{i}'+1))$$

is a tautology. We have generally found that it is sufficient to prove a stronger claim, that

$$\mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \wedge Q(\mathrm{i},\mathrm{i}') \wedge P(\mathrm{i}',\mathrm{i}'+1) \wedge Q(\mathrm{i}',\mathrm{i}'+1) \wedge P(\mathrm{i},\mathrm{i}'+1) \Rightarrow Q(\mathrm{i},\mathrm{i}'+1))$$

is a tautology. If so, we know that checking C_1 is sufficient. If not, we can sometimes derive additional information that, along with C_1, is sufficient to prove C (see the DYFESM example in section 6) for an example of this).

Sometimes, we have breaking conditions of the form:

$$C \equiv \forall \mathrm{i},\mathrm{i}', \mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \Rightarrow (Q'(\mathrm{i},\mathrm{i}') \vee Q''(\mathrm{i},\mathrm{i}'))$$

and find that $C_1 \not\Rightarrow C$. However, if we define

$$C' \equiv \forall \mathrm{i},\mathrm{i}', \mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \Rightarrow Q'(\mathrm{i},\mathrm{i}')$$

$$C'' \equiv \forall \mathrm{i},\mathrm{i}', \mathrm{i} < \mathrm{i}' \wedge P(\mathrm{i},\mathrm{i}') \Rightarrow Q''(\mathrm{i},\mathrm{i}')$$

we may find that $C_1' \Rightarrow C' \Rightarrow C$ and $C_1'' \Rightarrow C'' \Rightarrow C$. This occurs when we can show that all the elements of an array are distinct by showing that they are strictly increasing or strictly decreasing, and in cases such as those shown in the TRFD INTGRL and DYFESM HOP examples described in Section 6.

Once such a condition is derived to disprove one dependence, it can often be used to disprove many others.

5 Value-Based Dependence Analysis

Our use of function symbols to represent non-linear expressions increases the accuracy of our value-based dependence analysis. We can compute value-based dependence information from access A to B by subtracting, from the relation describing the pairs where memory aliasing occurs, all the pairs in which the memory location is over-written by some write B. This set of pairs to be subtracted is the union of all compositions of a dependence B to C with one from A to B. This composition operation may produce results that are outside the class of relations we can handle, and therefore produce approximate results.

For example, we cannot produce exact information about the loop-carried flow of values in Example 10. To do so, we would have to calculate the composition of the loop-carried flow dependence with the loop-carried output dependence, which produces the relation:

$$\{\ [\mathrm{i},\mathrm{j}] \rightarrow [\mathrm{i}'] \mid \exists [\mathrm{i}'',\mathrm{j}''] \text{ s.t. } 1 \leq \mathrm{i} < \mathrm{i}'' < \mathrm{i}' \leq \mathtt{n} \wedge p(\mathrm{i}) > 0 \wedge \mathbf{p}(\mathrm{i}'') > 0 \wedge 1 \leq \mathrm{j}'' = \mathrm{j}' \leq \mathtt{n} \wedge \mathrm{j} = \mathrm{j}'' = \mathrm{i}'\ \}$$

Since this relation contains an application of a function to a quantified variable, it is outside the class of formulas we can handle. It is possible to apply the techniques described in [Sho79] to the formula in this relation, but these techniques simply test for satisfiability; we need to subtract this relation from the relation giving memory-based dependences.

Note that we can compose the loop-independent flow dependence with the loop-carried output dependence. Since $\mathrm{i}'' = \mathrm{i}'$, we can replace $p(\mathrm{i}'')$ with $p(\mathrm{i}')$:

$$\{\ [\mathrm{i},\mathrm{j}] \rightarrow [\mathrm{i}'] \mid \exists [\mathrm{i}'',\mathrm{j}''] \text{ s.t. } 1 \leq \mathrm{i} < \mathrm{i}'' = \mathrm{i}' \leq \mathtt{n} \wedge p(\mathrm{i}) > 0 \wedge \underbrace{p(\mathrm{i}'')} > 0 \wedge 1 \leq \mathrm{j}'' = \mathrm{j}' \leq \mathtt{n} \wedge \mathrm{j} = \mathrm{j}'' = \mathrm{i}'\ \}$$

$$\equiv$$

$$\{\ [\mathrm{i},\mathrm{j}] \rightarrow [\mathrm{i}'] \mid \exists [\mathrm{i}'',\mathrm{j}''] \text{ s.t. } 1 \leq \mathrm{i} < \mathrm{i}'' = \mathrm{i}' \leq \mathtt{n} \wedge p(\mathrm{i}) > 0 \wedge \overbrace{p(\mathrm{i}')} > 0 \wedge 1 \leq \mathrm{j}'' = \mathrm{j}' \leq \mathtt{n} \wedge \mathrm{j} = \mathrm{j}'' = \mathrm{i}'\ \}$$

When we subtract this from the relation that describes the memory-based dependence, we eliminate any loop-carried flow dependences to iterations in which $p(\mathrm{i}) > 0$.

We can produce exact value-based dependences for Example 11. There are two compositions of output dependences and flow dependences that describe potential kills. We can represent each of these exactly, and when we subtract them both from the loop-carried memory-based dependence, nothing is left.

```
for i := 1 to n do
  if p(i) > 0 then
    for j := 1 to n do
      a[j] := ...
    endfor
  endif

  ... := a[i]
endfor
```

Example 10

```
for i := 1 to n do
  if p[i] > 0 then
    for j := 1 to n do
      a[j] := ...
    endfor
  else
    for j := 1 to n do
      a[j] := ...
    endfor
  endif

  ... := a[i]
endfor
```

Example 11

5.1 Symbolic array dataflow analysis

We can apply the techniques of Section 4 to value-based dependence relations, even if they are inexact. For example, we cannot produce an exact description of the data flow in Example 5. If we choose to be conservative, we can show that the value-based loop-carried flow dependences are a subset of

$$\{\ [\mathrm{i},\mathrm{j}] \to [\mathrm{i}',\mathrm{j}'] \mid 1 \leq \mathrm{i} < \mathrm{i}' \leq \mathtt{n} \wedge 1, l(\mathrm{i}') + 1 \leq \mathrm{j} = \mathrm{j}' \leq \mathtt{m}, l(\mathrm{i})\ \}$$

We eliminate j and j′, producing

$$\{\ [\mathrm{i}] \to [\mathrm{i}'] \mid 1 \leq \mathrm{i} < \mathrm{i}' \leq \mathtt{n} \wedge 1, l(\mathrm{i}') + 1 \leq \mathtt{m}, l(\mathrm{i})\ \}$$

When we take the gist of this relation given our usual set of uninteresting conditions, we get:

$$\{[\mathrm{i}] \to [\mathrm{i}'] \mid l(\mathrm{i}') < l(\mathrm{i}), \mathtt{m}\}$$

Thus, we can disprove a dependence by asserting that `L` is nondecreasing, or that all elements of `L` are greater than `m` (in which case the read is covered). Therefore, we could privatize the `work` array and run this loop in parallel under either of these conditions. Similarly, we could conclude that Example 10 could be run in parallel if `p` is always greater than 0. However, the inexactness of our result keeps us from showing that this example can also be parallelized if `p` is nondecreasing.

5.2 Related Work

Paul Feautrier and Jean-Francois Collard ([CF94]) have extended their array dataflow analysis technique to handle non-linear terms in loop bounds and **if**'s. They also give a precise description of the set of programs for which they can provide exact dependence information. However, their system cannot be applied to programs with nonlinear array subscripts, and according to Section 5.1 of their work, extending it to handle this case is "very difficult". Furthermore, there is no discussion of any way of introducing information about the values of non-linear expressions, as we describe in Section 3.1. We have not been able to come up with a precise mathematical comparison of our methods, but have so far not found an example in which one method cannot disprove a dependence that is disproved by the other, without resorting to non-linear subscripts or the techniques in Section 3.1.

We believe that the most significant distinction between our work and [CF94] is our ability to relate the non-linear terms in our dependence relations to expressions in the program, and thus discuss the program with some external agent (such as the programmer), as described in Section 4. The techniques described in [CF94] produce information about the possible sources of a value that is read from an array, but do not provide information about which expressions in the program control which source actually produces the value. In other words, they provide no mechanism for deriving the fact that Example 5 can be parallelized (after array expansion) if `L` is nondecreasing.

Procedure	time for our analysis	time for `f77 -fast`
BTRIX	2.3s	3.8s
CFFT2D1	9.0s	1.0s
INTERF	10.2s	2.2s
NLFILT	4.6s	1.4s
OLDA	4.2s	2.1s

Figure 1: Analysis times for array data dependences in several procedures

```
for i := 1 to nmol1 do
  for j := 1 to nmol do
    kc := 0
    for k := 1 to 9 do
      rs[k] := ...
      if rs[k] > cut2 then kc := kc + 1
    if kc < 9 then
      for k := 2 to 5 do
        if rs[k] <= cut2 then
          rl[k+4] := ...
      if kc = 0 then
        for k := 11 to 14 do
          ... := ... rl[k-5] ...
```

Example 12: MDG INTERF 1000

```
for i := 1 to n do
  for j := 1 to i do
    a[p[i]+j] := ...
```

Example 13: TRFD INTGRL 540

```
for i := 1 to n do
  for j := 1 to b[i] do
    a[p[i]+j] := ...
```

Example 14: DYFESM HOP 20

Vadim Maslov's work on lazy dependence analysis ([Mas94]) can handle some non-linear constraints. The value-based dependences his system can disprove are a strict subset of the ones we can disprove, but the dependences his system fails to disprove do not appear to be common. However, his system cannot determine the conditions that would disprove a dependence and cannot utilize the all of the optimizations described in Section 3.1.

5.3 Timing Results

We have implemented techniques to manipulate relations in which functions are applied to a prefix of the input or output tuple of the relation (this is sufficient to handle everything described in this paper except a small subset of the cases described in Section 3.1). Figure 1 shows the amount of time required on a Sparcstation 10/51 for our implementation to perform memory and value-based dependence analysis on the array variables of several routines from the Perfect Club Benchmark Suite ([B+89]). The routines shown in this table all contain many non-linear terms.

While these times may be to great to allow the use of our techniques during regular compilation, they are not excessive for a system that interactively assists programmers in the detection and elimination of dependences that prevent parallelism (this task has traditionally been done manually, in time measured in minutes, hours, or days).

6 Real world examples

In this section, we describe the results of applying the techniques describes here to a number of loops from the Perfect club that have been identified by other researchers as being parallel but not made parallel by current compilers [EHLP91, BE94, BEH+94, RP94].

MDG INTERF 1000 (Example 12) The difficult issue in the analysis of this program is proving that the `rl` array can be privatized. Our analysis cannot prove this, but can determine that the condition that needs to be proven to verify this is:

$$\forall i, j, k, 1 \leq i \leq nmol1 \wedge 1 \leq j \leq nmol \wedge 1 \leq k \leq 9 \wedge C_{11}(i,j) \Rightarrow C_9(i,j,k)$$

where C_{11} and C_9 are the guards at lines 11 and 9. It is not easy to generate a fast run-time test to verify this, and we doubt that this could be automatically proved except by building a special case for this example. However, this condition is guaranteed to be true and can be easily verified by a programmer.

TRFD INTGRL 540 Example 13 is a very simplified version of the difficulty here. Our methods determine that there is an output dependence iff:

$$\forall i, i', 1 \leq i < i' \leq \mathtt{n} \Rightarrow (p(i') \geq p(i) + i \vee p(i) \geq p(i') + i')$$

We can use the techniques in Section 4.3 to show that a sufficient (but not necessary) test to disprove the dependence is:

$$\forall i, 1 \leq i < \mathtt{n} \Rightarrow p(i+1) \geq p(i) + i \vee \forall i, 1 \leq i < \mathtt{n} \Rightarrow p(i) \geq p(i+1) + i + 1$$

Aggressive interprocedural analysis can prove that $p(i) = i(i-1)/2$, which would disprove the dependence.

DYFESM Example 14 shows a very simplified version of the tricky dependence here. Our methods determine that there is an output dependence iff:

$$\forall i, i', 1 \leq i < i' \leq \mathtt{n} \Rightarrow (p(i') \geq p(i) + b(i) \vee p(i) \geq p(i') + b(i'))$$

If we try to show that $\forall i, 1 \leq i < \mathtt{n} \Rightarrow p(i+1) \geq p(i) + b(i)$ is a sufficient condition to disprove the dependence, we find that we must also establish $\forall i, 1 \leq i < \mathtt{n} \Rightarrow b(i) \geq 0$. The fact that $p(i+1) = p(i) + b(i)$ might be verified by advanced interprocedural analysis, but the fact that $b(i)$ is nonnegative depends on input data and would thus need to be asserted by the programmer or verified by a run-time test.

BDNA ACTFOR 240 Our methods don't work on this example. We are unable to disprove the loop carried flow dependence on the arrays `XDT` and `IND`, and do not generate exact constraints that could disprove it. We are investigating ways of handling this example.

ARC2D FILERX 290 Our methods prove that there is no loop carried value-based flow dependence, allowing us to privatize the work array and run the loop in parallel. We do not prove that no copy-in is required, but do generate an exact set of conditions that describe when it isn't (advanced interprocedural analysis could determine that these conditions are always satisfied).

MDG POTENG 2000 We disprove all value based loop carried flow dependences, except for some accumulations into scalars, which can be handled by parallel reductions, allowing the loop to be run in parallel.

TRFD OLDA 100 and 300 We prove that the work arrays (`XRSIQ`, `XIJ`, `XIJKS`, and `XKL`) can be privatized. We also need to show that there are no output dependences for the writes to `XIJRS` and `XIJKL`. It is quite easy to determine from the code that the subscripts of these writes are strictly increasing (if checked before induction variable recognition). Since facts of this kind can be easily incorporated into our framework, we suggest such information be recorded and used. After non-linear induction variable recognition, the subscripts are replaced with complicated polynomials of the loop variables, and verifying that these polynomial are non-decreasing is more challenging. Generating a run-time test is not particularly easy or useful for this example.

7 Conclusions

Standard array dependence analysis techniques produce conservative approximations by ignoring any constraints arising from non-linear terms in the program. This approach is not adequate for tests that perform value-based dependence analysis or provide information about the conditions under which a dependence exists.

We use uninterpreted function symbols to represent non-linear terms in our constraint system. This approach lets us calculate conditions that are sufficient and necessary for a memory-based dependence to exist. Furthermore, it seems to be at least as powerful as other methods for computing approximate value-based array dependences in the presence of non-linear terms. We can determine the conditions under which a value-based dependence exists, although these conditions will not be tight if the dependence is approximate (i.e. we will produce separate sets of necessary and sufficient constraints).

Our techniques provide information that is useful in determining that some code from the Perfect Club Benchmark programs can be run in parallel. This information is not provided by standard analysis techniques. Some of this information might be derived by advanced interprocedural analysis techniques [BE94], but it may be more efficient to derive such information in a demand-driven way, rather than trying to derive all interprocedural information that can be proven.

A partial implementation of our approach will be distributed with version 0.9 of the Omega library and dependence analyzer (expected Nov. 20th, 1994), and most of the techniques described here should be implemented by spring 1995.

A Manipulating Presburger Formulas with Uninterpreted Function Symbols

Our techniques for manipulating Presburger formulas with uninterpreted function symbols are based on our previous work with regular Presburger formulas ([PW93]). Our approach is based on conversion to disjunctive normal form and testing each conjunct via Fourier's method of elimination, adapted for integer variables ([Pug92]). We have developed a number of techniques to ensure that the number and sizes of conjuncts produced do not get out of hand for the formulas that we encounter in practice.

Our techniques, like those of Shostak, rely on the fact that when $i = j$, $f(i)$ must equal $f(j)$, and when $i \neq j$, $f(i)$ is unrelated to $f(j)$. We therefore convert each formula into a specialized disjunctive normal form, in which every conjunct that contains multiple uses of a function also contains either an equality constraint between or the function arguments or some constraint that contradicts this equality. In other words, a conjunct containing both $f(i_1, i_2, i_3)$ and $f(j_1, j_2, j_3)$, could contain either $i_1 = j_1 \wedge i_2 = j_2 \wedge i_3 = j_3$ or $i_2 > j_2$ (or some similar inequality).

Note that the relations we generate to describe data dependences are often in this form already, because the functions are typically applied to a prefix of the input or output tuple and we use distinct relations to describe dependences carried at different levels.

We simplify each individual conjunct, treating function symbols as follows:

- If a conjunct contains only one application of f, treat the function application as a scalar constant.
- If a conjunct contains multiple function applications whose arguments are equal, we unify them, treating the result as a single scalar constant.
- We treat function applications with arguments that are known to be distinct as independent distinct scalar constants.

When we manipulate relations with the relational operators supported by the Omega test, we handle appearances of f as follows:

- For some operations, such as intersection, union, or inversion, the input and output tuples of all input relations become the input or output tuple of the result. Such operations must always produce relations that we can represent exactly.
- Operations such as range or composition, in which the input or output tuple of a relation becomes existentially quantified in the result, may produce results that we cannot handle. In this case, we approximate any constraint that contains a function application that cannot be written as $f(In)$ or $f(Out)$ in the result.

When we approximate, we may wish to produce either an upper or a lower bound on the relation. We can relax any conjunction of constraints containing $f(x)$, where x is existentially quantified, by replacing $f(x)$ with a new existentially quantified scalar. We can tighten such a conjunction by replacing it with false.

During value-based dependence analysis, we may perform operations on approximate relations. Specifically, we need to analyze $A \wedge \neg B$, where A is exact formula and B is a formula that is an upper bound on what B is intended to express. This upper bound could arise due to an inexact composition. If $A \wedge B$ is infeasible, we know that A is an exact answer, even though B is an inexact upper bound. Otherwise, A is an inexact upper bound on the answer. To analyze $A \wedge \neg B \wedge \neg C \wedge \neg D \wedge \ldots$, we first handle the exact negated clauses. This makes it more likely that we will be able to use the technique above to get an exact answer.

References

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1–11, January 1988.

[B+89] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.

[BE94] William Blume and Rudolf Eigenmann. Symbolic analysis techniques needed for effective parallelization of the Perfect benchmarks. Technical Report 1332, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994.

[BEH+94] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Lawrence Rauchwerger, and Peng Tu. Automatic detection of parallelism: A grand challenge for high-performance computing. Technical Report 1349, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994.

[CF94] Jean-François Collard and Paul Feautrier. Fuzzy array dataflow analysis. Technical Report Research Report N° 94-21, Laboratoire de l'Informatique du Parallélisme, Ecolo Normal Supérieure de Lyon, Instiut IMAG, July 1994. Postscript available as `lip.ens-lyon.fr:pub/LIP/RR/RR94/RR94-21.ps.Z`.

[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.

[Dow72] P. Downey. Undeciability of presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Center for Research in Computing Technology, Havard Univ., 1972.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proc. of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991.

[Fea88] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freemand and Company, 1979.

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

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In *ACM '94 Conf. on Principles of Programming Languages*, January 1994.

[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.

[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.

[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. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[PW94] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 14(3):1248–1278, July 1994.

[RP94] Lawrence Rauchwerger and David Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. Technical Report 1383, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 12–27, January 1988.

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.

[SW94] Eric Stoltz and Michael Wolfe. Detecting value-based scalar dependence. In *Proc. of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*. Cornell University, August 1994.