UMIACS-TR-93-137 December, 1993
CS-TR-3196

An Exact Method for Analysis of Value-based Array 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

Standard array data dependence testing algorithms give information about the aliasing of array references. If statement 1 writes `a[5]`, and statement 2 later reads `a[5]`, standard techniques described this as a flow dependence, even if there was an intervening write. We call a dependence between two references to the same memory location a memory-based dependence. In contrast, if there are no intervening writes, the references touch the same value and we call the dependence a value-based dependence.

There has been a surge of recent work on value-based array data dependence analysis (also referred to as computation of array data-flow dependence information). In this paper, we describe a technique that is exact over programs without control flow (other than loops) and non-linear references. We compare our proposal with the technique proposed by Paul Feautrier, which is the other technique that is complete over the same domain as ours. We also compare our work with that of Tu and Padua, a representative approximate scheme for array privatization.

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

1 Introduction

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). This can reduce the effectiveness of several optimizations: First, it can inhibit the use of techniques (such as privatization, renaming, and array expansion) that eliminate storage-related dependences. These techniques cannot be applied if they appear to affect the flow dependences of a program. Second, it reduces the compiler's ability to make effective use of caches or distributed memories. Flow dependences represent the flow of information in the program being compiled; accurate information about this flow is critical for memory use optimizations.

Array flow dependences in which the value written at the source is actually used at the sink of the dependence are referred to as *value-based* flow dependences [PW92] or *data-flow dependences* [May92, MAL93]. Dependence tests that are based on only the first four of the above criteria are said to find *memory-based* dependences [PW92].

Data dependence testing is undecidable in general. Within the restricted domain described in Section 2, performing exact memory-based array dependence testing is NP-complete [Pug92]. Paul Feautrier [Fea91] showed that performing exact value-based dependence analysis of arrays within this restricted domain is decidable, and gave an algorithm for doing so. Many other researchers have also studied value-based array data dependence analysis, however no other work is complete over the same domain as Feautrier's, with the exception of recent work by Maslov [Mas94]. Some of these other methods handle larger domains (such as complicated control flow) but are not complete within that domain [Bra88, Rib90, GS90, Ros90, Li92, Fea91, MAL92, MAL93, DGS93].

We describe an extension of our previous work on dependence analysis [Pug92, PW92]. This extension makes us complete over the same domain as Feautrier's technique, and produces the same information (although using a different computational method and a different dependence abstraction). In this paper, we review this extension and compare it with Feautrier technique.

Dror Maydan developed [May92, MAL93] a variant of Feautrier's techniques that apply in a more restricted domain, but appears faster and to apply to most real cases. Maydan's techniques integrate well with Feautrier's and in cases in which Maydan's techniques do not apply, Feautrier's algorithms can be called. We believe that when Maydan's methods apply, our methods will be fast as well.

We evaluate our methods on the examples evaluated by Feautrier [Fea91] and find that our methods are 40-90 times faster than Feautrier's current implementation.

2 Dependence Analysis

For the methods we describe to apply, we must be able to determine which loops and conditionals control the execution of each statement. This can be done in a straightforward manner for code that uses only structured loops and **if**'s for control flow.

We can produce exact dependence information for any single structured procedure in which the expressions in the subscripts, loop bounds, and conditionals are affine functions of the loop indices and loop-independent variables, and the loop steps are known constants.

We start our dependence analysis by producing traditional dependence difference summaries. Dependence difference is equivalent to the more traditional "dependence distance" when loops are normalized, as they are in all examples in this paper. The term "dependence distance" is not well defined for unnormalized loops – see [Pug93] for details.

Dependence difference summaries do not describe which iterations are involved in the dependence, and do not describe the effect of the values of symbolic constants. We therefore represent dependences with *dependence relations* [Pug91]. A dependence relation is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that represent the values of the loop indices at the source and sink of the dependence and the values of the symbolic constants (e.g., `n`). The notation we use in the constraints is adapted from [ZC91]:

$A, B, \ldots$	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 $\mathcal{I}$
$A(\mathcal{I}) \overset{sub}{=} B(\mathcal{I}')$	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 $B(\mathcal{I}')$
$A(\mathcal{I}) \ll_D B(\mathcal{I}')$	The dependence difference from $A(\mathcal{I})$ to $B(\mathcal{I}')$ is described by the direction/difference vector D

The dependence relation below describes exactly the iterations and values of symbolic constants for which $A(\mathcal{I})$ and $B(\mathcal{I}')$ refer to the same element of the array, and $A(\mathcal{I})$ is executed before $B(\mathcal{I}')$ (i.e., it describes the memory-based dependence from A to B).

$$\{ \ \mathcal{I} \rightarrow \mathcal{I}' \mid \mathcal{I} \in [A] \wedge \mathcal{I}' \in [B] \wedge A(\mathcal{I}) \ll B(\mathcal{I}') \wedge A(\mathcal{I}) \overset{sub}{=} B(\mathcal{I}') \ \} \tag{1}$$

For example, the flow dependence from `b(i)` to `b(j)` in Example 1 is described by the direction vector $(+)$, and the dependence relation:

$$\{ \ [\mathrm{i}] \rightarrow [\mathrm{i}', \mathrm{j}'] \mid \overbrace{1 \leq \mathrm{i} \leq \mathtt{n}}^{\mathcal{I} \in [A]} \wedge \underbrace{1 \leq \mathrm{j}' < \mathrm{i}' \leq \mathtt{n}}_{\mathcal{I}' \in [B]} \wedge \overbrace{\mathrm{i} < \mathrm{i}'}^{A(\mathcal{I}) \ll B(\mathcal{I}')} \wedge \underbrace{\mathrm{i} = \mathrm{j}'}_{A(\mathcal{I}) \overset{sub}{=} B(\mathcal{I}')} \ \}$$

This can be simplified to $\{ \ [\mathrm{i}] \rightarrow [\mathrm{i}', \mathrm{j}'] \mid 1 \leq \mathrm{i} = \mathrm{j}' < \mathrm{i}' \leq \mathtt{n} \}$.

```
for i := 1 to n do
  s := 0
  for j := 1 to i-1 do
    s := s + a(i,j)*b(j)
  endfor
  b(i) := b(i) - s
endfor
```

Example 1

```
1: for i := 1 to 2*n do
2:    a(i) := ...
3:    for j := 1 to n-1 do
4:       a(2*j)   := ...
5:       a(2*j+1) := ...
6:    endfor
7:    ... := a(i)
8: endfor
```

Example 2

2.1 Value-based dependences

We can calculate value-based flow, output, and anti-dependences, but in this paper we are not concerned with the latter two. There is a value-based flow dependence between an instance of a write $A(\mathcal{I})$ and an instance of a read $C(\mathcal{I}'')$ if and only if $C(\mathcal{I}'')$ reads the value that was written by $A(\mathcal{I})$. For this to occur, $A(\mathcal{I})$ and $C(\mathcal{I}'')$ must access the same element of the array, and that element must not be overwritten between $A(\mathcal{I})$ and $C(\mathcal{I}'')$. If the element is overwritten by $B(\mathcal{I}')$, we say $B(\mathcal{I}')$ kills the dependence from $A(\mathcal{I})$ to $C(\mathcal{I}'')$. Let B_1, B_2, ..., B_p be the array writes that might kill the dependence (note that A might be included in the list of B_q's). The value-based flow dependence from A to C is described by the relation:

$$
\begin{aligned}
\{\, \mathcal{I} \to \mathcal{I}'' \;\mid\; & \mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll C(\mathcal{I}'') \wedge A(\mathcal{I}) \overset{sub}{=} C(\mathcal{I}'') \\
& \wedge\, \forall q, 1 \le q \le p, \neg\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B_q] \\
& \qquad \wedge\, A(\mathcal{I}) \ll B_q(\mathcal{I}') \ll C(\mathcal{I}'') \wedge B_q(\mathcal{I}') \overset{sub}{=} C(\mathcal{I}'') \,\}
\end{aligned}
\tag{2}
$$

For example, consider the flow dependence from the write at line 2 to the read at line 7 in Example 2. We build the dependence relation by expanding the $\forall q, \ldots$ with a set of constraints for each B_q. The relation will have two "kill" terms (B_q's): the writes at lines 4 and 5 (there is no kill term for the write on line 2 because there is no self output dependence for this write). The unsimplified version of the relation is:

$$
\begin{aligned}
\{\, [\mathrm{i}] \to [\mathrm{i}''] \mid\; & \overbrace{1 \le \mathrm{i} \le 2\mathrm{n}}^{\mathcal{I}\in[A]} \wedge \overbrace{1 \le \mathrm{i}'' \le 2\mathrm{n}}^{\mathcal{I}''\in[C]} \wedge \overbrace{\mathrm{i} = \mathrm{i}''}^{A(\mathcal{I})\ll C(\mathcal{I}'')} \wedge \overbrace{\mathrm{i} = \mathrm{i}''}^{A(\mathcal{I})\overset{sub}{=}C(\mathcal{I}'')} \\
& \underbrace{\begin{aligned}
&\wedge \neg(\, \exists[\mathrm{i}',\mathrm{j}'] \text{ s.t. } (1 \le \mathrm{i}' \le 2\mathrm{n} \wedge 1 \le \mathrm{j}' \le \mathrm{n}-1) \wedge (\mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' \le \mathrm{i}'') \wedge (2\mathrm{j}' = \mathrm{i}'') \,) \\
&\wedge \neg(\, \exists[\mathrm{i}',\mathrm{j}'] \text{ s.t. } (1 \le \mathrm{i}' \le 2\mathrm{n} \wedge 1 \le \mathrm{j}' \le \mathrm{n}-1) \wedge (\mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' \le \mathrm{i}'') \wedge (2\mathrm{j}'+1 = \mathrm{i}'') \,)
\end{aligned}}_{\forall q, 1\le q\le p, \neg\exists\mathcal{I}' \text{ s.t. } \mathcal{I}'\in[B_q] \wedge A(\mathcal{I})\ll B_q(\mathcal{I}')\ll C(\mathcal{I}'') \wedge B_q(\mathcal{I}')\overset{sub}{=}C(\mathcal{I}'')} \,\}
\end{aligned}
$$

By using techniques described in Section 3, we need 10 milliseconds on a Sun Sparc IPX to simplify this to:

$$
\{\, [\mathrm{i}] \to [\mathrm{i}''] \mid\; (1 = \mathrm{i} = \mathrm{i}'' \le \mathrm{n}) \vee (1 \le \mathrm{i} = \mathrm{i}'' = 2\mathrm{n}) \vee (1 \le \mathrm{i} = \mathrm{i}'' \le 2\mathrm{n} \wedge \mathrm{n} = 1) \,\}
$$

Thus, we have discovered that there is a dependence from the first write of Example 2 to the read only during the first iteration and last iteration (if $\mathrm{n} = 1$, there are only 2 iterations).

2.2 Implementation details

Equation 2 is best thought of as a denotational description of how array kills are computed. There are a number of tricks we can use that will improve our efficiency while still computing the exact same results as if we had used Equation 2.

When computing value-based dependences, we use characterizations of the memory based dependences (such as the level that carries the dependence, or a direction/distance vector) that describes the dependence. The dependence between two variable accesses might need to be described by several such descriptions (for example, a dependence might be carried by several different levels). We treat each such description as a separate dependence, and use $A(\mathcal{I}) \xrightarrow{x} B(\mathcal{I}')$ to describe the dependence from $A(\mathcal{I})$ to $B(\mathcal{I}')$ that is characterized by x – the type of characterization does not matter.

To compute the value-based version of a flow dependence $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$, we need to consider all kills of the form $A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'')$. For the dependence $B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'')$, we can ignore any dependences that we have already proven to not be value-based. We can perform a quick check based only on f, o and f' to see if we can prove that $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'') \cap A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'') = \emptyset$. If so, we need not consider this kill combination. Otherwise, we can use o and f' to enforce the $\ll$ restrictions in the kill clause:

$$\begin{aligned}
\{\, \mathcal{I} \to \mathcal{I}'' \quad | \quad & \mathcal{I} \in [A] \wedge \mathcal{I}'' \in [C] \wedge A(\mathcal{I}) \ll_f C(\mathcal{I}'') \wedge A(\mathcal{I}) \overset{sub}{=} C(\mathcal{I}'') \\
& \ldots \wedge\ \neg\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \\
& \qquad \wedge\ A(\mathcal{I}) \ll_o B(\mathcal{I}') \ll_{f'} C(\mathcal{I}'') \wedge B(\mathcal{I}') \overset{sub}{=} C(\mathcal{I}'') \,\}
\end{aligned}$$

While this seems to increase the number of clauses we need to consider, it actually saves us work. Equation 2 uses $\ll$ constraints, which are non-linear. Expanding these out to be linear would introduce a larger expansion than the one we get here by using the output dependences to the kill and the flow dependences from the kill.

2.2.1 Partial cover

With respect to $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$, the dependence $B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'')$ is a partial cover iff there does not exist a dependence $B(\mathcal{I}') \xrightarrow{o'} A(\mathcal{I})$ such that

$$B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'') \cap B(\mathcal{I}') \xrightarrow{o'} A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'') \neq \emptyset$$

We try to prove this based only by quick checks on f, o', and f'. If we can prove this, then if a memory location is touched by both $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$ and $B(\mathcal{I}') \xrightarrow{f} C(\mathcal{I}'')$, then the write by $B(\mathcal{I}')$ comes after $A(\mathcal{I})$, and there can't be any flow of values from $A(\mathcal{I})$ to $C(\mathcal{I}'')$. In other words, we can ignore the dependence $A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}')$ in considering the effects of the kill. We can calculate the iterations $\mathcal{I}''$ of $[C]$ that might be involved in the dependence with the relation:

$$\mathcal{I}'' \in [C] \wedge \neg\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge\ B(\mathcal{I}') \ll_{f'} C(\mathcal{I}'') \wedge B(\mathcal{I}') \overset{sub}{=} C(\mathcal{I}'')$$

This has the effect of eliminating from $\mathcal{I}''$ any iterations that are supplied a value by B. Since the effects of this kill do not depend on $A(\mathcal{I})$, we can compute it just once and use this information for any dependence for which $B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'')$ is a partial cover.

Being somewhat more aggressive, we can sort the complete list of flow dependences to $C(\mathcal{I}'')$, so that the ones that are closest in time (according to the level that carries the dependence, the direction/distance vector, ...) are at the head of the list.

In computing each value-based dependence to $C(\mathcal{I}'')$, some dependences on this list will be legal to use as partial covers. Assume the first p are legal and the $p+1^{\text{st}}$ is not (there may be others after the $p+1^{\text{st}}$ that are legal, but we will ignore them). We can now compute the effects of the partial kills by the first p dependences on the list, and use that in computing the dependence. If we had previously computed the effects for the first q partial covers ($q < p$), we can simply extend that information by taking into account the $q+1^{\text{st}}$ through the p^{th} element of the list.

Of course, we may find that after considering the first r partial covers, the read is completely covered. In this case, any dependence that can use the first r partial covers has no value-based component.

2.2.2 Partial termination

With respect to $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$, the dependence $A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}')$ is a partial terminator iff there does not exist a dependence $C(\mathcal{I}'') \xrightarrow{a} B(\mathcal{I}')$ such that

$$A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}') \cap A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'') \xrightarrow{a} B(\mathcal{I}') \neq \emptyset$$

If we can prove this, then if a memory location is touched by both $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$ and $A(\mathcal{I}) \xrightarrow{o} B(\mathcal{I}')$, then the read by $C(\mathcal{I}'')$ comes after $B(\mathcal{I}')$, and there can't be any dependence from $A(\mathcal{I})$ to $C(\mathcal{I}'')$. In other words, we can ignore the dependence $B(\mathcal{I}') \xrightarrow{f'} C(\mathcal{I}'')$ in considering the effects of the kill, and the iterations $\mathcal{I}$ of A that might be involved in the dependence are:

$$\mathcal{I} \in [A] \wedge \neg\exists \mathcal{I}' \text{ s.t. } \mathcal{I}' \in [B] \wedge \ A(\mathcal{I}) \ll_o B(\mathcal{I}') \wedge A(\mathcal{I}) \stackrel{sub}{=} B(\mathcal{I}')$$

2.2.3 Putting it all together

Now, when considering a kill of $A(\mathcal{I}) \xrightarrow{f} C(\mathcal{I}'')$, we get the appropriate information from the partial termination list for $A(\mathcal{I})$ and the partial coverage list for $C(\mathcal{I}'')$, and only need to directly consider the kills that we not handled as either a partial cover or as a partial terminator.

When considering kills, it is probably best consider all the dependences to a single read at a time, and to consider the flow dependencies in the order they appear on the partial cover list.

We describe this technique as partial cover and termination, since it is similar to the cover and termination tests described in [PW92], but records the effect of dependences that only partially cover or termination an array reference (just as Equation 2 is an extension of the kill test described in [PW92]).

3 Simplifying Formulas Containing Negation

When performing array kill analysis, we have to simplify formulas of the form:

$$\ldots \vee (C_0 \wedge \neg(\exists V_1 \text{ s.t. } C_1) \wedge \ldots \wedge \neg(\exists V_n \text{ s.t. } C_n)) \vee \ldots$$

Here, the C_i's are conjunctions of linear constraints, and the V_i's are (possibly empty) sets of variables. Techniques described in our previous papers ([Pug92, PW92]) allow us to eliminate existentially quantified variables, check for the feasibility of a conjunction of constraints, and perform other simplifications, but these techniques do not address negation. There are two problems involved in simplifying formulas containing negations:

- We must transform a formula into disjunctive normal form in order to verify the existence of solutions. A straightforward transformation of a formula containing negation into disjunctive normal form may lead to a huge explosion in the number of terms. We describe in Section 3.1 a method for replacing the formula with an equivalent form that, typically, will not suffer from as large an increase. By only evaluating the negation for one term at a time and reapplying the transformation in Section 3.1, additional savings may be obtained.

- If a negated term $\exists V_i$ s.t. C_i represents non-convex constraints (e.g., $\exists\alpha$ s.t. $x = 5\alpha$), we can not directly evaluate the negation. When standard Fourier-Motzkin variable elimination cannot eliminate an integer variable exactly, we can sometimes eliminate the variable exactly by introducing quasi-linear constraints (constraints containing floor and ceiling operators) [AI91]. For example, $(\exists\alpha \text{ s.t. } x = 5\alpha) \equiv \lceil x/5 \rceil \leq \lfloor x/5 \rfloor$. In these cases, negation is easy to apply (e.g., $\neg(\lceil x/5 \rceil \leq \lfloor x/5 \rfloor) \equiv (\lceil x/5 \rceil > \lfloor x/5 \rfloor)$).

  Although we can always eliminate one variable this way, we may not be able to eliminate multiple variables this way, since we may not be able to apply Fourier-Motzkin variable elimination to a set of constraints containing floor and ceiling operators. In these cases, we apply the technique given in Section 3.2.1, which is complete.

The complete set of steps we apply is described in Section 3.3, and some examples are given in Section 3.4.

3.1 Using Gist to Simplify Negations

The goal of the gist operator is to simplify a term B as much as possible, given than A is known to be true. In [PW92], we define gist B given A as a minimal subset of the constraints of B such that $(A \wedge (\text{gist } B \text{ given } A)) \equiv (A \wedge B)$. When performing negations, we rely on the fact that $(A \wedge \neg B) = (A \wedge \neg(\text{gist } B \text{ given } A))$ (see Step 1 of our algorithm in Section 3.3).

$$\begin{aligned} A \wedge \neg B &\equiv A \wedge \neg(A \wedge B) \\ &\equiv A \wedge \neg(A \wedge (\text{gist } B \text{ given } A)) \\ &\equiv A \wedge \neg(\text{gist } B \text{ given } A) \end{aligned}$$

Since gist B given A will often have fewer constraints than B, the disjunctive normal form of $A \wedge \neg(\text{gist } B \text{ given } A)$ will often have fewer clauses than $A \wedge \neg B$.

3.2 Negating Non-Convex Constraints

To eliminate an integer variable x with Fourier-Motzkin variable elimination, we combine each upper and lower bound on x. In general, a lower bound $\alpha \leq ax$ and an upper bound $bx \leq \beta$ produce $\lceil \alpha/a \rceil \leq \lfloor \beta/b \rfloor$. If $a = 1$ or $b = 1$, then $\lceil \alpha/a \rceil \leq \lfloor \beta/b \rfloor$ is equivalent to $b\alpha \leq a\beta$, which is preferred as it does not introduce floor and ceiling operations.

Constraints involving floor and ceiling operations are called quasi-linear constraints [AI91]. When a set of constraints involves quasi-linear constraints, we cannot verify the existence of solutions, and our ability to eliminate redundant constraints is diminished. We normally avoid the introduction of quasi-linear constraints by simply avoiding the elimination of variables that would introduce them. Variables that we would like to eliminate but cannot because they would introduce quasi-linear constraints are called *wildcard* variables. When we need to verify the existence of solutions to sets of constraints, methods described in [Pug92] allow us to do so accurately in the presence of wildcard variables.

However, it is easier to negate sets of constraints containing quasilinear constraints than sets containing wildcards. We therefore use quasilinear constraints, when possible, to perform negation.

To eliminate a ceiling operation $\lceil \alpha/a \rceil$ we introduce a new wildcard variable c, add the constraints $ac - a < \alpha \leq ac$ and replace $\lceil \alpha/a \rceil$ with c. To eliminate a floor operation $\lfloor \beta/b \rfloor$ we introduce a new wildcard variable f, add the constraints $bf \leq \beta < bf + b$ and replace $\lfloor \beta/b \rfloor$ with f.

3.2.1 Quasi-linear Constraints are not Complete

Unfortunately, using quasi-linear constraints is an incomplete method. Given a set of constraints involving ceiling and floor operators, we do not know of a general purpose method to eliminate an existentially quantified variable that appears inside a ceiling or floor operator. For example, consider

$$\exists x, y \text{ s.t. } 0 \leq x, y \leq 10 \wedge 4x + 7z \leq 3y \wedge 2y \leq 3x + z$$

After the elimination of y, this becomes

$$\exists x \text{ s.t. } 0 \leq x \leq 10 \wedge \lceil (4x + 7z)/3 \rceil \leq \lfloor (3x + z)/2 \rfloor \wedge 4x + 7z \leq 30 \wedge 0 \leq 3x + z$$

Current methods for eliminating existential quantifiers cannot eliminate variables inside floor or ceiling functions.

Rather than introducing ceiling and floor operators, we can eliminate variables by using splintering [Pug92]. Splintering performs exact elimination by producing a set of problems, the union of which exactly describe the result of the quantifier elimination. The subproblems produced by splintering may contain wildcards (existentially quantified variables). In this case, the constraints in the final subproblems are of the form:

$$\{\ \vec{x}, \vec{y} \mid \exists \vec{\alpha} \text{ s.t. } \begin{bmatrix} \vec{x} \\ \vec{y} \end{bmatrix} = T \begin{bmatrix} \vec{x} \\ \vec{\alpha} \end{bmatrix} + \begin{bmatrix} 0 \\ \vec{c} \end{bmatrix} \ \wedge\ A \begin{bmatrix} \vec{x} \\ \vec{\alpha} \end{bmatrix} \leq \vec{b}\ \}$$

where T has the form:

$$\begin{bmatrix} I & 0 \\ & S \end{bmatrix}$$

Here, $\vec{y}$ represent the variables that have been eliminated from the subproblem via substitution, $\vec{x}$ represent the variables that remain in the problem and $\vec{\alpha}$ represent wildcard variables. Due to the way substitutions are performed, the mapping from $\vec{x}$ and $\vec{\alpha}$ to $\vec{x}$ and $\vec{y}$ is 1-1, so the above equation is equivalent to the following: (where T^{+} is the pseudoinverse [Str88] of T):

$$\begin{aligned} \{\vec{x}, \vec{y} \mid \exists \vec{\alpha} \text{ s.t. } & \begin{bmatrix} \vec{x} \\ \vec{y} - \vec{c} \end{bmatrix} \in \text{RowSpace}(T) \\ \wedge & \begin{bmatrix} \vec{x} \\ \vec{\alpha} \end{bmatrix} = T^{+} \begin{bmatrix} \vec{x} \\ \vec{y} - \vec{c} \end{bmatrix} \ \wedge\ AT^{+} \begin{bmatrix} \vec{x} \\ \vec{y} - \vec{c} \end{bmatrix} \leq \vec{b}\ \} \end{aligned}$$

Each wildcard now appears in only one equality constraint that enforces a modulo constraint (e.g., $\exists\beta$ s.t. $3\beta = x+2y$). Such constraints can be easily negated (e.g., $\exists\beta$ s.t. $3\beta < x+2y < 3\beta+3$). Since we can negate each constraint in the subproblem, we can negate the entire subproblem.

It is unclear how often it will be necessary to resort to the handling of negation via splintering, or how expensive it will be to apply. However, we feel that it is important to have a complete method for handling negation.

3.3 Detailed algorithm

We convert such formulas to disjunctive normal form by repeatedly choosing a clause that contains a negated term, and applying the following steps:

1. We simplify each of the $\exists V_i$ s.t. C_i terms using the *gist* operation we defined in [PW92]. We replace each $\exists V_i$ s.t. C_i term with gist ($\exists V_i$ s.t. C_i) given C_0. This step is justified in Section 3.1. In doing this simplification, we use Fourier-Motzkin variable elimination to remove as many existentially quantified variables as is possible to do exactly.

   If we find that the simplified term contains a single inequality constraint, we immediately negate it and add it to C_0, discarding the C_i term. We repeatedly simplify terms until each negated term has been checked at least once since the last time C_0 changed.

   If a C_i simplifies to TRUE, we know that the entire clause is unsatisfiable.

2. Of the remaining negated terms, we pick a term that is likely to cause the least amount of combinatorial explosion when negated. A simple and effective estimate of the expansion factor is the number of inequality constraints needed to express the term (i.e., number of inequality constraints plus twice the number of equality constraints).

3. For the term we pick, we eliminate exactly *all* existentially quantified variables (see Section 3.2).

4. We then negate the term, producing a disjunction of constraints.

5. We remove any floor and ceiling operators from the constraints in the disjunction by introducing additional constraints and wildcards (see Section 3.2).

6. We now convert the entire clause into disjunctive normal form and simplify, producing a list of clauses that replace the original clause.

We repeat this process until there are no more clauses involving negation. This process may generate redundant clauses. If desired, we can eliminate many of them by testing for pairs of clauses C_i, C_j such that $C_i \Rightarrow C_j$ (in this case, C_i is redundant and can be removed).

3.4 Examples

In the following examples, we show examples of negation that would arise from direct use of Equation 2, ignoring the techniques described in Section 2.2. This is purely for illustrative purposes; the techniques described here for handling negation work with either a direct implementation of Equation 2, or the more sophisticated techniques described in Section 2.2.

The relation for Example 2 given in Section 2.1 contains negations. Figure 1 shows the results of each step (except #2) in our simplification of this relation. Step 2 involves selection of a term, and thus has no visible result in our table. After the initial simplification, each term can be represented

$\{\ [i] \rightarrow [i''] \mid 1 \leq i \leq 2n \wedge 1 \leq i'' \leq 2n \wedge i = i'' \wedge i = i''$ $\quad \wedge\neg(\exists [i',j'] \text{ s.t. } 1 \leq i' \leq 2n \wedge 1 \leq j' \leq n-1 \wedge i \leq i' \wedge i' \leq i'' \wedge 2j' = i'')$ $\quad \wedge\neg(\exists [i',j'] \text{ s.t. } 1 \leq i' \leq 2n \wedge 1 \leq j' \leq n-1 \wedge i \leq i' \wedge i' \leq i'' \wedge 2j'+1 = i'')\ \}$	Unsimplified relation from Example 2
$\{\ [i] \rightarrow [i''] \mid 1 \leq i = i'' \leq 2n \wedge \neg(\exists \alpha \text{ s.t. } i = 2\alpha \wedge i+2 \leq 2n)$ $\quad \wedge\neg(\exists \alpha \text{ s.t. } i = 2\alpha - 1 \wedge 3 \leq i)\ \}$	Simplify (1)
$\{\ [i] \rightarrow [i''] \mid \cdots \wedge \neg(\lceil (i+1)/2 \rceil \leq \lfloor (i+1)/2 \rfloor \wedge 3 \leq i)\ \}$	Introduce $\lfloor\ \rfloor$, $\lceil\ \rceil$ (3)
$\{\ [i] \rightarrow [i''] \mid \cdots \wedge\ (\lceil (i+1)/2 \rceil > \lfloor (i+1)/2 \rfloor \vee 3 > i)\ \}$	Apply negation (4)
$\{\ [i] \rightarrow [i''] \mid \cdots \wedge (3 > i$ $\quad \vee \exists f, c \text{ s.t. } c > f \wedge 2f \leq i+1 < 2f+2 \wedge 2c-2 < i+1 \leq 2c)\}$	Introduce wildcards (5)
$\{\ [i] \rightarrow [i''] \mid\ \ 1 \leq i = i'' \leq 2n \wedge (\exists \beta \text{ s.t. } i = 2\beta) \wedge \neg(\exists \alpha \text{ s.t. } i = 2\alpha \wedge i+2 \leq 2n)$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge i \leq 2 \wedge \neg(\exists \alpha \text{ s.t. } i = 2\alpha \wedge i+2 \leq 2n)\ \}$	Convert to DNF and simplify (6)
$\{\ [i] \rightarrow [i''] \mid\ \ 1 \leq i = i'' \leq 2n \wedge (\exists \beta \text{ s.t. } i = 2\beta) \wedge \neg(i+2 \leq 2n)$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge i \leq 2 \wedge \neg(\lceil i/2 \rceil \leq \lfloor i/2 \rfloor \wedge i+2 \leq 2n)\ \}$	Simplify (1) and Introduce $\lfloor\ \rfloor$, $\lceil\ \rceil$ (3)
$\{\ [i] \rightarrow [i''] \mid\ \ 1 \leq i = i'' \leq 2n \wedge (\exists \beta \text{ s.t. } i = 2\beta) \wedge (i+2 > 2n)$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge i \leq 2 \wedge (\lceil i/2 \rceil > \lfloor i/2 \rfloor \vee i+2 > 2n)\ \}$	Apply negation (4)
$\{\ [i] \rightarrow [i''] \mid\ \ 1 \leq i = i'' \leq 2n \wedge (\exists \beta \text{ s.t. } i = 2\beta) \wedge (i+2 > 2n)$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge i \leq 2 \wedge (\exists f, c \text{ s.t. } c > f \wedge 2f \leq i < 2f+2 \wedge 2c-2 < i \leq 2c)$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge i \leq 2 \wedge i+2 > 2n\ \}$	Introduce wildcards (5) and convert to DNF (6)
$\{\ [i] \rightarrow [i''] \mid\ \ 1 \leq i = i'' = 2n$ $\quad \vee\ 1 = i = i'' \leq 2n$ $\quad \vee\ 1 \leq i = i'' \leq 2n \wedge n = 1\ \}$	Simplify (6)

Figure 1: Evaluating negations produced by Example 2

as three inequalities, and thus we can choose either one (in the example, we choose the second). After the second application of step 1, each clause contains only one negated term, so there is no need to apply step 2.

When a group of assignments kills a dependence, but no single assignment from the group does so, we say the dependence is killed by a comb. Combs often involve a set of subscripts that differ only in the constant term. Figure 2 shows a comb used in the Perfect Club program `MDG`. This code demonstrates the advantages of the repeated simplification in step 1 of our technique.

To the right of the code is the dependence relation for the value-based flow dependence from the write of `xl(j)` on line 18 to the read of `xl(j)` on line 17. Our initial application of step 1

```
 1: for i := 1 to n do
 2:    xl(1) := ...
 3:    xl(2) := ...
     ...
15:    xl(14) := ...
16:    for j := 1,14 do
17:      if (abs(xl(j))>boxh)
18:        xl(j) := xl(j)-...
19:      endif
20:    endfor
21: endfor
```

$$
\begin{aligned}
\{\ [i,j] \rightarrow\ & [i'',j''] \mid \\
& 1 \leq i < i'' \leq n \wedge 1 \leq j = j'' \leq 14 \\
& \wedge\ \neg(\exists i' \text{ s.t. } i < i' = i'' \wedge j'' = 1) \\
& \wedge\ \neg(\exists i' \text{ s.t. } i < i' = i'' \wedge j'' = 2) \\
& \wedge\ \ldots \\
& \wedge\ \neg(\exists i' \text{ s.t. } i < i' = i'' \wedge j'' = 14)\ \}
\end{aligned}
$$

$$
\begin{aligned}
\{\ [i,j] \rightarrow\ & [i'',j''] \mid \\
& 1 \leq i < i'' \leq n \wedge 1 \leq j = j'' \leq 14 \\
& \wedge\ \neg(j \leq 1) \wedge \neg(j = 2) \wedge \ldots \wedge \neg(j \geq 14)\ \}
\end{aligned}
$$

Dependence relation before and after step 1

Figure 2: Example of dependence analysis in the presense of a comb kill

eliminates all of the existentially quantified variables. If we were to perform all of the negations at this time, we would produce twelve terms containing disjunctions, yielding 2^{12} clauses when we convert to disjunctive normal form. We would have to simplify all of these clauses to show that there is no value-based flow dependence.

We avoid this problem with our re-application of step 1: On our first pass, we negate $(j \leq 1)$ and $\neg(j \geq 14)$, producing the new $C_0 = 2 \leq j \leq 13$. Our second application of Step 1 reduces $\neg(j = 2)$ to $\neg(j \leq 2)$, and $\neg(j = 13)$ to $\neg(j \geq 13)$, allowing both of these terms to be negated without introducing disjunction. Thus, we never produce more than one clause, and perform fewer simplifications than we would perform without repeated use of step 1. Furthermore, a simplification may reduce the size of a term, speeding up future simplifications of that term.

3.5 Simplifying Arbitrary Presburger Formulas

The ability to negate conjunctions of linear constraints with wildcards or with quasilinear constraints gives us a complete method for simplifying Presburger formulas.

We propagate negations inwards, but not over quantifiers. We eliminate universal quantifiers by replacing formulas of the form $\forall x, P$ with formulas of the form $\neg\exists x$ s.t. $\neg P$. Given a quantified expression $\exists x$ s.t. P where P contains no embedded quantifiers, we convert P into disjunctive normal form, and then apply the methods of [Pug92] to eliminate the existential quantifier (possible leaving wildcards). When applying negation, we use the techniques of this section.

The best known upper bound on the performance of an algorithm for verifying Presburger formulas is $2^{2^{2^n}}$ [Opp78], and we have no reason to believe that this method be provide better worst-case performance. However, our method may be more efficient for many simple cases that arise in many applications.

4 Related Work

The method described by Feautrier [Fea88b, Fea91] was the first method for computing exact value-based dependence information over the restricted domain of programs with structured control flow and affine subscripts, guards and loop bounds. Dror Maydan and Monica Lam developed an alternative way of computing the same information as Feautrier does. This method is faster, but does not apply in some cases (in which case the falls back to Feautrier's method). Vadim Maslov [Mas94] has recently described a new framework for value-based dependence information that is exact over the same domain.

In previous work we described exact methods for computing memory-based dependences [Pug92] and methods for identifying some, but not all, dependences that were not value-based [PW92].

4.1 Feautrier's and Maydan's approach

Feautrier and Maydan compute a decision tree (called a *quast* or a *last write tree*(LWT)) to describe a dependence. This decision tree allows the computation of the source of any particular read. The internal nodes represent tests to be performed. The left branch corresponds to a false result, a right branch to a true result. The leaves are either a description of the statement and iteration that wrote the value read in the iteration of interest, or $\perp$, corresponding to a read of an uninitialized location. In the method described by Paul Feautrier and Dror Maydan, there are three basic steps to dependence analysis:

1. Computing the data-flow dependence from a single write to a single read, assuming there are no other writes.
2. Combining the dependences generated in step 1 from multiple writes to describe the data-flow dependences from many writes to a single read.
3. Check each leaf of the quast/LWT to verify that it is feasible. (Note: this check can be done on-the-fly while building the quast/LWT).

4.1.1 Parametric Integer Programming with the Omega test

Paul Feautrier discusses parametric integer programming [Fea88a], which is the problem of finding the optimal/maximal solution to a set of linear constraints over integer variables. For example, $\max\{i \mid i \leq j \wedge i \leq k\} =$ if $j \leq k$ then j else k. Parametric integer programming is done with respect to finding the minimum or maximum lexical value for a vector of variables.

Parametric integer programming can be restated in a form of Presburger arithmetic that we can handle in the Omega test:

$$\max\{\mathcal{I} \mid P(\mathcal{I},\mathcal{I}'')\} \equiv P(\mathcal{I},\mathcal{I}'') \wedge \neg\exists\mathcal{I}' \text{ s.t. } \mathcal{I} \prec \mathcal{I}' \wedge P(\mathcal{I}',\mathcal{I}'')$$

4.1.2 Computing dependences from single writes to a read

Paul Feautrier uses parametric integer programming to compute the value-based dependences from a single write to a read, ignoring all other writes. Dror Maydan and Monica Lam noted that in many common situations, faster techniques would suffice.

4.1.3 Computing dependences from many writes to a read

When quasts or LWT's from multiple writes are combined to give a single description of which writes reach a read, the size of the quast/LWT can grow exponentially [Fea91, May92]. There is not yet enough experimental evidence to evaluate the growth of quast's/LWT's vs. the growth in the number of conjunctions produced by our methods. We suspect that the requirement that the quast/LWT be a decision tree will tend to make it grow faster. Whatever condition is tested as the root of the tree becomes part of the conditions of every leaf, even it is not relevant to some leaves.

4.1.4 Checking feasibility and handling negation

The quasts or Last Write Trees constructed by combining quast's/LWT's may contain infeasible paths [Fea91, MAL93]. To enable compile-time transformations such as privatization, it is necessary to determine which of these paths are feasible. This cost is likely to be substantial, particularly since the number of leaves in a quast/LWT grows exponentially when multiple writes are considered.

Dror Maydan suggested that if we check the feasibility interior and leaf nodes, using a depth-first-search, we can avoid checking all nodes and leaves below any infeasible node found. No studies have yet been done to see what improvements could be obtained this way.

Determining which of the paths are feasible requires checking the feasibility of a problem such as:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \wedge \neg N_1 \wedge \neg N_2 \wedge \cdots \wedge \neg N_m$$

where the P_i's are the conditions for the nodes where we take the true branch and the N_i's are the conditions for the nodes where we take the false branch. Each of these conditions is a conjunction of linear constraints, and may include non-convex constraints (e.g., constraints such as "i is even" specified using wildcards or quasi-linear constraints). Directly converting these expressions into disjunctive normal form would be infeasible for many real problems. The methods we describe in Section 3 should reduce this blow-up.

The methods described by [MAL93] can handle only special cases of negated non-convex constraints. Paul Feautrier uses quasi-linear constraints (constraints containing floor and ceiling operations) to handle negation. Unfortunately, this technique is not complete for all cases. In Section 3.2.1, we describe techniques that do not suffer this incompleteness.

It is our belief that the cost of checking the feasibility of all leaves of a quast/LWT is likely to be the major expense in an implementation of Feautrier's or Maydan's scheme.

4.1.5 A question of form

One advantage of the quasts/LWT's computed by Feautrier and by Maydan is that they are represented as a set of constraints over the read iteration and the symbolic variables, which can be easily tested at run-time, and by simple formulas that, given the values of the read iteration and symbolic variables, determine the write iteration.

If all constraints are affine, Equation 2 is guaranteed to produce a dependence relation such that, for any value of the read iteration of symbolic variables, there is exactly one write iteration that satisfies the constraints. However, we can make few guarantees about the form of the constraints in the dependence relation. For example, we might obtain a dependence relation of the form:

$$\{[i_w, i_r - 2i_w] \rightarrow [i_r] \mid 0 \leq i_w \leq m \wedge q \leq i_r \leq p \wedge i_r - 1 \leq 2i_w \leq i_r \wedge i_r \leq n + 2i_w\}$$

In order to produce these properties of quasts/LWT's, we will need to recognize constraints that are equivalent to integer division (and perhaps integer remainder?). This would allow us to recognize the above dependence relation as:

$$\{[(i_r \div 2), i_r - 2(i_r \div 2)] \rightarrow [i_r] \mid 0 \leq (i_r \div 2) \leq m \wedge q \leq i_r \leq p \wedge i_r \% 2 \leq n\}$$

It is currently an open question as to whether this extension will allow us to obtain these properties in all cases.

4.2 Maslov's approach

Vadim Maslov [Mas94] suggested that using just Equation 2 to compute value-based dependences would be inefficient. His observation was that it is wasteful to consider all possible killers for every possible dependence; instead, we can keep track of the upwards exposed iterations of a read, and utilize this information. Since the upwards exposed information can be calculated once per killer-read pair, as opposed to once per write-killer-read triple, this could lead to a substantial performance improvement. He also suggested reordering computations in a more lazy fashion, so as to avoid performing computations that might be rendered useless or irrelevant by later computations.

We have incorporated this idea into our system with the idea of partial covers (Section 2.2). We have extended it by also keeping track of the downwards exposed iterations of each write (using partial termination).

There are two other significant differences between this work and Maslov's. First, our method uses memory-based dependences to calculate the value-based dependences, while Maslov's does not require that memory-based dependences be calculated (which, in some cases, can save time). Second, Maslov uses a lexicographical maximum calculation instead of Equation 2.

It is our belief that any system using value-based dependence information will also need memory-based dependence information. Thus, we have not tried to avoid the cost of computing memory-based dependence information. However, our methods for computing value-based dependences work just fine if we start from a conservative approximation to the memory based dependences. If fact, if we start from the crude approximation that there is a dependence carried at every level between any two references to the same variable, our algorithm works in a fashion very similar to that of Maslov's, and Equation 2 and Maslov's lexicographical maximum performance almost identical computational steps. The main difference is that our use of partial terminators may save us some work.

In the case where we use better information about memory-based dependences, we may be able to save additional work. We may be able to handle more killers as partial covers and terminators and we may be able to avoid considering some killers at all.

Maslov utilizes our previous work on the Omega test [Pug92, PW92] and the techniques described here for handling negation (Section 3).

4.3 Tu's and Padua's Approach

We will briefly compare our scheme with that of Tu and Padua [TP92, PEH+93], as a representative example of other related work on array privatization [GS90, Ros90, Li92]. Their scheme handles control flow, while ours currently does not. We believe that a naive implementation of the methods described in their papers would be equivalent, in our scheme, to performing only those kills where the output dependence from the write to the kill or the flow dependence from the kill to the read was loop independent. Equivalently, we could perform the upward-exposed and downward-exposed calculations of Section 2.2. They [PEH+93] note that this is imprecise:

> ... a naive aggregation of $USE_b(L)$ may exaggerate the exposed use set....

They do not describe in [TP92, PEH+93] the algorithm used to perform the more sophisticated aggregation. It is our belief that the effect of the more sophisticated aggregation will be equivalent, in our scheme, to using only partial covers in computing kills.

The primary effects of these differences are that while they correctly determine whether or not a flow dependence is carried by a loop, they are approximate when determining the exact source of a dependence (i.e., which iteration of which write). Since their scheme is targeted at privatizing arrays, the information they determine is sufficient. For other purposes, such as analyzing communications [AL93] and scalar replacement, additional information is needed.

Another difference is that Tu's and Padua's method is based on determining which array elements are covered, while Feautrier's and our methods are based on determining which read iterations are covered. when all array references are linear, this does not make a difference. It could make a difference when non-linear subscripts occur, but it is unclear which would be advantageous.

Tu and Padua use an extension of regular sections [CK88] to represent used, defined and exposed array sections. Their intersection (and their difference?) operators are approximate. If exact calculations are desired, our algorithms for simplifying Presburger formulas may be useful.

5 Performance evaluation

In Table 1, we report Feautrier's performance evaluation of his techniques, and a performance evaluation of our techniques on the same problems. For our work, we list the times required to perform a standard, memory-based dependence analysis and to perform a value-based dependence analysis. Our times are on a SPARC IPX (a SPECint89 rating of 21.7), Feautrier's are on a SPARC ELC (a SPECint89 rating of 18.0). So that our results can be compared with Feautrier's, we analyze both array and scalar variables. We also report the time required to analyze just the array variables.

In analyzing memory based dependences as a pre-pass, we calculated conservative approximate memory-based dependences for scalar variables and for dependences with no common loops. Using approximate memory-based dependences still allows us to compute exact value-based dependences.

Our current implementation uses the partial covers described in Section 2.2; we do not currently use partial terminators. We found that using partial covers give a factor of 2-4+ improvement in analysis time, compared with use of Equation 2 alone. We also experimented with using Equation 2 with the complete cover and termination checks described in [PW92] (but not partial cover and termination). For a few programs, computing partial covers gave a nearly a factor of 2 improvement over doing only full cover and termination checks. But for most programs, they do not lead to a major improvement and in some cases even slows down the analysis. However, since partial covers and terminators reduce the number of situations in which we see worst-case "cubic number of terms" behaviour of Equation 2, we think they are a valuable idea.

Some of the dependence relations we calculate for `olda` and the NASA NAS kernels are conservative since they contain (non-loop) control flow and non-linear terms. We currently do not attempt to perform a kill using a non-linear dependence. However, a dependence might be linear unless carried by the outer loop (e.g., if the dependence involved a variable that was changing unpredictably in the outermost loop). We detect such cases and handle the linear components of the dependence.

The times reported for Feautrier's algorithm are from an implementation of his algorithm that has not been engineered for efficiency. While the PIP algorithm is implemented in `C`, the remainder of his algorithm is implemented in Lisp. Work is underway to recode Feautrier's algorithms more efficiently. Feautrier hopes that this will result in a significant speed-up.

Dror Maydan [MAL93] notes that his techniques require 100 milliseconds on a Decstation 3100 (a SPECint89 rating of 11.8) to evaluate the `relax` example and to calculate the dependence direction/distance vectors from the LWT's. The `relax` example does not require merging LWT's.

6 Implementation Status and Benchmark Availability

The techniques described here are being implemented in our extended version of Michael Wolfe's `tiny` tool [Wol91], which is available for anonymous ftp from `ftp.cs.umd.edu:pub/omega`. The programs analyzed in Table 1 come from a set of benchmark programs for comparing the performance and coverage of algorithms for analyzing value-based flow dependences between array references. Send email to `omega@cs.umd.edu` to receive a copy of the benchmarks and be added to the dataflow benchmarks mailing list.

7 Conclusion

The cost of performing exact value-based flow dependence analysis for arrays appears to be 2-7 times that required to do exact memory-based exact array dependence analysis using integer programming techniques [Pug92]. We believe that these methods are suitable for use in production compilers. However, we may wish to avoid applying them blindly. It may be cost effective to determine when it might be profitable to have exact value-based dependence information, and

Analyzing array variables only

From	Code	M	V	$M+V$	`f77`	D	$\frac{M+V}{D}$
[Fea91]	across	3	6	9	200	13	0.69
	burg	14	58	72	600	32	2.25
	relax	4	20	24	400	8	3.00
	gosser	5	52	58	700	16	3.63
	choles	4	7	12	600	12	1.00
	lanczos	29	78	107	1700	136	0.79
	jacobi	379	621	999	1600	255	3.92
[MAL93]	ocean (extract)	11	14	25	500	16	1.56
Perfect	olda (simplified)	51	338	389	3300	48	8.10
NASA	btrix	330	1036	1369	8600	771	1.78
NAS	cfft2d1	33	484	516	1500	52	9.92
Kernels	cholsky	86	156	242	2900	106	2.28
	emit	34	87	121	3700	129	0.94
	gmtry	25	70	95	3700	91	1.04
	vpenta	262	163	425	5700	1501	0.28

Analyzing array and scalar variables

From	Code	M	V	$M+V$	[Fea91]	D	$\frac{M+V}{D}$
[Fea91]	across	3	6	9	600	13	0.69
	burg	15	76	91	5600	46	1.98
	relax	4	19	24	1700	8	3.00
	gosser	6	56	62	2800	24	2.58
	choles	6	25	32	2600	32	1.00
	lanczos	28	91	119	12600	148	0.80
	jacobi	386	718	1104	81900	374	2.95
[MAL93]	ocean (extract)	10	15	25		16	1.56
Perfect	olda (simplified)	65	732	796		142	5.61
NASA	btrix	331	1175	1515		809	1.87
NAS	cfft2d1	37	540	577		103	5.60
Kernels	cholsky	84	162	246		107	2.30
	emit	37	144	181		181	1.00
	gmtry	34	146	180		161	1.12
	vpenta	253	220	473		1757	0.27

M - Memory-based dependence analysis time
V - Value-based flow dependence analysis time (needs M)
`f77` - Time required to compile with f77 -c -O3
[Fea91] - Times reported by Feautrier
D - # of dependences for which value-based dependences are calculated
(*all times in milliseconds*)

Table 1: Evaluation of times required to perform analysis

apply them only in those cases. Some methods for doing this are described in [PW93]. Also, the methods described here are more susceptible to bad worst-case performance than the methods described in [Pug92, PW92]. We might want to be able to detect when computing exact value-based dependence information is going to be very expensive, and use some approximation.

Neither Feautrier nor Maydan has does an analysis of which components of their algorithms are expensive. Therefore, we can only speculate on the reasons why our scheme appears 40-75 times faster than Feautrier's.

Use of the algorithm described by Maydan (which falls back to Feautrier's when the special cases handled by Maydan do not apply) is probably a reasonable way to determine the exact value-based dependence from a single write to a read. More work is needed to compare the two, but we do not expect more than an order of magnitude difference between our scheme and Maydan's.

The algorithm used by Feautrier and Maydan for merging quasts/LWT may be subject to problems with exponential growth in the number of leaves. Compared with our scheme, two factors might lead to a larger blow-up:

- We use the dependence direction/distance vectors for the flow *and* the output dependences to and from a kill to determine when a kill is feasible. This allows us to rule out more kills.
- Tests irrelevant to a particular dependence may increase the branching factor. For example, if there are three possible sources of a dependence (s_1, s_2 and s_3), the leaves for the dependence to s_3 will occur under both branches of a test that determines if the write by s_1 or s_2 is most recent.

The papers by Feautrier and Maydan have not addressed the issue of efficiently checking the feasibility of formulas containing negation. It is our belief that this is responsible for a substantial portion of the time required by Feautrier's algorithm. Some efficient scheme for testing the feasibility of a set of linear constraints is needed, such as [MHL91, Pug92]. In is unclear how effective PIP [Fea88a] is at checking feasibility (as opposed to parametric integer programming). In addition, some method like the one we describe here will be required to handle negation efficiently.

It is unclear if the exact information computed by our scheme and by Feautrier will ever be required, or if approximate information, computed by schemes such as [TP92, PEH+93], will suffice. For array privatization, approximate schemes may suffice but more advanced transformations [AL93, PW93] may require more exact information. We have pursued exact analysis methods because we want to determine exactly how expensive it will be to compute, and because we find that it gives us a better insight into the problem. If exact methods do not cost significantly more than approximate methods, then the justification for using approximate methods is weaker. If we find that exact methods are too expensive, we can decide to cut corners and know exactly what information we may be loosing.

The techniques we have described are impractical for real programs, since they do not handle control flow (other than loops) and procedure calls. We are currently exploring ways of extending our methods to deal with these cases. We expect that we will have to abandon our goal of being exact in all cases to deal with these features.

References

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Proc. of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[AL93] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelism. In *Proc of 1988 International Conference on Supercomputing*, pages 407–417, July 1988.

[CK88] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[Fea88a] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[Fea88b] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.

[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.

[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.

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

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

[MHL91] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[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.

[PEH+93] David A. Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. CSRD Rpt. 1306, Dept. of Computer Science, University of Illinois at Urb ana-Champaign, June 1993.

[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. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-2993, Dept. of Computer Science, University of Maryland, College Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[PW93] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. Technical Report CS-TR-2994.2, Dept. of Computer Science, University of Maryland, College Park, June 1993.

[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.

[Str88] Gilbert Strang. *Linear Algebra and its Applications.* Harcourt Brace Jovanovich, Publishers, 1988.

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, September 1992. appeared in ACM SIGPLAN Notices January 1993.

[Wol91] 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.