DIMACS Technical Report 2000-38
November, 2000

Polynomial Time Array Dataflow Analysis [1]

by

Robert Seater
Haverford College
Haverford, PA 19041
rseater@haverford.edu

David Wonnacott [2]
Department of Computer Science
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu

[1]This work is supported by NSF grant CCR-9808694
[2]Visiting DIMACS Fall 1998, with support from DIMACS (NSF grant STC–91–19999)

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs-Research, Bell Labs, NEC Research Institute and Telcordia Technologies (formerly Bellcore).

DIMACS was founded as an NSF Science and Technology Center, and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

Array dataflow analysis is a valuable tool for supercomputer compilers. However, the worst-case time complexities for array dataflow analysis techniques are either not well understood or alarmingly high. For example, the Omega Test uses a subset of the $2^{2^{2^{O(n)}}}$ language of Presburger Arithmetic for analysis of affine dependences; its use of uninterpreted function symbols for non-affine terms introduces additional sources of complexity. Even traditional data dependence analysis of affine dependences is equivalent to integer programming, and is thus NP-complete. These worst-case complexities have raised questions about the wisdom of using array dataflow analysis in a production compiler, despite empirical data that show that various tests run quickly in practice.

In this paper, we demonstrate that a polynomial-time algorithm can produce accurate information about the presence of loop-carried array dataflow. We first identify a subdomain of Presburger Arithmetic that can be manipulated (by the Omega Library) in polynomial time; we then describe a modification to prevent exponential blowup of the Omega Library's algorithm for manipulating function symbols. Restricting the Omega Test to these polynomial cases can, in principle, reduce the accuracy of the dataflow information produced. We therefore present an investigation of the effects of our restrictions on the detection of loop-carried array dataflow dependences (which prevent parallelization). The restriction to polynomial-time algorithms blocks parallelization of only three unimportant loop nests in the approximately 18000 lines of benchmark code we studied.

1 Introduction

Array dataflow analysis is an important tool for supercomputer compilers. This analysis provides information that can be used to determine the legality of array privatization, a critical technique for automatic parallelization [EHLP91]. It can also serve as the basis for optimization of communication [Fea94] or memory locality [Won00]. Unfortunately, the worst-case time complexities for array dataflow analysis techniques are either not well understood or alarmingly high.

Array dataflow analysis can be seen as an extension of traditional array dependence analysis to produce information about the flow of values rather than simple memory aliasing. We thus refer to it as a *value-based* analysis, in contrast to the traditional *memory-based* analysis. The Omega Test implements both memory-based [Pug92] and value-based [PW93, PW98] analyses by generating and simplifying sets of constraints that describe dependences. The complexity of these constraints depends on the subscripts and control flow expressions (hereafter referred to as *dependence expressions*) and the kind of analysis being performed. Memory-based dependence analysis in the presence of arbitrary affine (i.e., linear plus a constant) dependence expressions is equivalent to integer programming, and thus is NP-complete [MHL91]. Value-based analysis includes negated constraints, and thus requires a richer constraint language; the Omega Test uses Presburger Arithmetic [KK67], which appears to have time complexity $2^{2^{2^{O(n)}}}$ [FR74, Opp78].

Empirical studies of benchmark programs have found that affine dependence expressions are generally very simple, and the full power of a test based on arbitrary affine constraints is almost never needed for memory-based analysis [May92]. This finding suggests that we can either (a) get generally accurate results with a provably fast (i.e. polynomial-time) test, or (b) get generally quick results from the (provably accurate) Omega Test. Pugh's empirical studies of the memory-based Omega Test found that it always ran quickly in practice, with most tests taking less than a few hundred microseconds, and two outlying data points at just over 2 milliseconds. (And this was in 1991, on a Decstation 3100.) However, worst-case possibility of unacceptably long compile time has caused some to question the wisdom of using such a test in a production compiler, and option (a) remains a popular approach to memory-based analysis.

The use of simple memory-based tests is unfortunate, not because the memory-based Omega Test eliminates any key dependences, but because it produces an exact description of the dependences (as a set of constraints). This exact description makes it possible to perform exact or conservative negation of dependences (for value-based analysis) or determine conditions under which a dependence exists. Studies of automatic parallelization [May92, EHLP91] show that analysis of value-based and conditional dependences is much more important than exact affine memory-based testing. In our prior work on value-based and conditional analysis [PW93, PW98], we provided empirical evidence that our techniques run quickly in practice: Value-based analysis took less than twice as long as memory-based analysis, and conditional analysis took between two and three times as long. Other research on array dataflow analysis has also provided substantial empirical evidence of efficient be-

havior in practice (see Section 6), but the lack of a polynomial bound still raises questions about the use of such techniques in production compilers.

In this paper, we demonstrate that a polynomial-time algorithm can produce accurate information about the presence of loop-carried array dataflow. We begin, in Section 2, with a review of the principles of constraint-based program analysis. Section 3 then describes our polynomial subdomain of Presburger Arithmetic and our technique for preventing exponential blowup of the Omega Library's algorithm for manipulating function symbols. Section 4 discusses the kinds of programs that produce constraints in our polynomial subdomain. Section 5 presents our investigation of the impact of our restrictions on parallelization. Section 6 covers related work, and Section 7 presents our conclusions.

2 Review of Constraint-Based Dependence Analysis

In this section, we review our previous work on constraint-based techniques for memory-based and value-based dependence analysis. Our goal is to provide insight into the nature of the constraints we use rather than a definitive discussion of dependence analysis; readers who are interested in the latter may want to refer to [Wol89] for a general discussion, and [PW98, Won95] for details of our techniques. These techniques constitute the Omega Test for data dependences, which has been implemented in the Petit system [KMP+95b], and performs constraint manipulation with the Omega Library [KMP+95a].

Dependences represent restrictions on our ability to reorder statements or executions of statements (for example, if we wish to reorder the iterations of a statement in a loop). Data dependences arise from the definitions and uses of variables, and are classified by the types of array accesses connected by the dependence, and by the *carrying loop*. A dependence from a write to a subsequent read is called a *flow* dependence; a dependence from a write to a write an *output* dependence, and a dependence from a read to a write an *anti*-dependence. The carrying loop is the outermost loop for which the loop index variables differ (this loop generally cannot be parallelized unless the dependence is eliminated somehow). A dependence that does not cross any iterations is said to be *loop independent*.

For example, consider Figure 1. In iteration $n1 = 3, k1 = 2$, the statement that defines `c(k1,nn)` writes to `c1(2,2)`; in iteration $n1' = 3, k1' = 3$, this statement reads this same array element (with the read from `c(k1-1,nn+1)`). Thus, if $n \geq 3$ (so that these iterations occur), there is a memory-based flow dependence from the write to the read. There is also a value-based flow dependence between these iterations, since the value produced in iteration $n1 = 3, k1 = 2$ is not overwritten before iteration $n1' = 3, k1' = 3$. Since $n1 = n1'$ and $k1 < k1'$, this dependence is carried by the `k1` loop. Loop-carried value-based flow dependences represent dataflow that fundamentally inhibits loop parallelization, and will be the focus of this paper (although our techniques can be applied to other dependences).

```
do n1 = 2, n
{
    nn = n1-1
    c(1,n1) = 1.0
    do k1 = 2, n1
    {
        c(k1,nn) = c(k1-1,nn+1) * ...
        nn = nn-1
    }
}
```

Figure 1: Excerpt from CNSTNT in MDG [B+89]

2.1 Memory-Based Dependence Analysis

Dependence analyzers are generally asked to perform a conservative test of whether a particular kind of dependence can exist for any values of the symbolic constants. This analysis can be described in terms of satisfiability tests on sets of constraints on integer variables that represent the values of the loop indices and other program variables. For example, we might wish to test whether `c` creates any memory-based flow dependences that are carried by the `k1` loop in Figure 1. We can perform this analysis by producing constraints on the variables $n1$, $k1$ (the values of the loop indices when the write occurs), $n1'$, $k1'$ (the values of the loop indices when the read occurs), nn, nn' (the values of `nn` at the time of the write and read), and n (the value of `n`, which is constant in this loop nest). The constraints must describe the dependence (i.e., the fact that the statements are executed and the subscripts are equal) and the desired relationship between the loop indices. To determine whether or not any pair of iterations might be connected by a dependence for any values of the symbolic constraints, we existentially quantify the loop indices and constants and test these constraints for satisfiability:

$$\begin{aligned}
&\exists\, n1, k1, n1', k1', n \text{ s.t.} \\
&\qquad \underbrace{2 \le n1, n1' \le n \wedge 2 \le k1 \le n1 \wedge 2 \le k1' \le n1'}_{\textit{statements are executed}} \\
&\qquad \wedge\ \underbrace{k1 = k1' - 1 \wedge nn = nn' + 1}_{\textit{subscripts are equal}} \\
&\qquad \wedge\ \underbrace{n1 = n1' \wedge k1 < k1'}_{\textit{flow dep. carried by } \texttt{k1} \textit{ loop}}
\end{aligned}$$

Standard scalar analysis techniques such as constant propagation and induction variable detection can often be used to eliminate free variables such as nn and nn' in the above, producing:

$$\begin{aligned}&\exists\, n1, k1, n1', k1', n \text{ s.t.}\\ &\quad 2 \le n1, n1' \le n \wedge 2 \le k1 \le n1 \wedge 2 \le k1' \le n1'\\ &\quad \wedge\, k1 = k1' - 1 \wedge (n1 - k1 + 1) = (n1' - k1' + 1) + 1\\ &\quad \wedge\, n1 = n1' \wedge k1 < k1'\end{aligned}$$

If the dependence constraints are not satisfiable, there can be no dependence. If they are satisfiable (as in this case), the Omega Test saves a "simplified" version of the constraints as a *dependence relation*, which can be combined with other constraints in later analysis to avoid re-building and re-analyzing the same constraints. The relation for the above dependence is:

$$\{\ [\ n1, k1\ ] \rightarrow [\ n1, k1 + 1\ ] \mid 2 \le k1 < n1 \le n\ \}$$

When a dependence can be expressed as a conjunction of affine constraints on the symbolic constants and loop indices, memory-based dependence testing is equivalent to integer programming [MHL91], which is NP-Complete [GJ79]. When free variables cannot be replaced with affine combinations of loop indices and constants, and in the presence of non-affine expressions, the Omega Test introduces uninterpreted function symbols [Sho79, PW95, PW98].

2.2 Value-Based Dependence Analysis

The Omega Test performs value-based analysis by subtracting, from the set of memory-based dependences, the dependences in which the array element is overwritten between the original write and the read. There are two ways to formulate this:

- We can perform a *kill* test, adding an "and not overwritten" clause to the the constraints for memory-based dependence above.
- In certain conditions, we can factor out the shared work for many dependences to the same array read by performing a *cover* analysis of the read. We start with constraints describing the iterations in which the read occurs, and subtract out those that are covered by each dependence in turn. The remaining set, which describes the iterations with upward-exposed reads, can then be combined with other memory-based dependences to produce value-based information.

In practice, the Omega Test performs the cover analysis when it is possible, and resorts to the kill test to resolve other cases. Regardless of the test we use, we must negate a conjunction of constraints that may involve existentially quantified variables.

For the code in Figure 1, we use the cover test to describe the effect of the dependence from Section 2.1 on all other dependences to this array read. To do this, we first subtract, from the set of all iterations in which the read occurs, those iterations reached by this dependence:

$$\underbrace{\{\ [\ n1, k1\ ]\ |\ 2 \leq k1 \leq n1 \leq n\ \}}_{\textit{iterations executed}} - \underbrace{\{\ [\ n1, k1+1\ ]\ |\ 2 \leq k1 < n1 \leq n\ \}}_{\textit{iterations reached by the dependence}}$$

Determining whether all iterations are covered is equivalent to testing whether the constraints $\exists\ n1, k1$ s.t. $2 \leq k1 \leq n1 \leq n \wedge \neg(2 \leq k1-1 < n1 \leq n)$ are satisfiable (they are, e.g. for $n1 = 3, k1 = 2$). The remaining exposed set, like the results of memory-based analysis, is saved in simplified form (in this case, $\{\ [\ n1, 2\ ]\ |\ 2 \leq n1 \leq n\ \}$). It is used to update other flow dependences to this read; this step entails one satisfiability test (without any new negation) per dependence.

The need to negate constraints increases the complexity of our constraint system beyond the level of integer programming (NP-complete) to a larger subclass of Presburger Arithmetic [KK67]. Presburger Arithmetic appears to have time complexity $2^{2^{2^{O(n)}}}$ [FR74, Opp78].

2.3 Conditional Dependence Analysis

Since our simplified dependence relations contain constraints on the symbolic constants, we could use variable elimination (selectively) to detect conditions on these constants that are necessary for any dependence to exist. For example, if we existentially quantify all variables *except* n in the dependence constraints for the flow dependence discussed above, we find it exists iff $n \geq 3$. We could therefore parallelize this loop when there is no dependence, i.e. when $n < 3$.

To avoid wasting time trying to parallelize single-trip loops and other useless cases, we search for conditions that are not redundant with a set of constraints that define the non-useless cases (such as $n \geq 3$). We perform this test with the *gist* operation [PW92, Won95]. Intuitively, given two conjunctions of equality and inequality constraints p and q, *gist* p *given* q yields constraints in p which are not implied by q. That is, (*gist* p *given* q) $\wedge\, q$ is equivalent to $p \wedge q$. In the worst case, finding a gist forces us to perform one satisfiability test of $p \wedge \neg e$ for each equation e in q, and thus has exponential time complexity.

2.4 Non-Affine Terms

The Omega Test provides two approaches to dealing with a term that cannot be expressed as an affine function of the symbolic constants and loop indices. First, such terms (or, indeed, any term) can safely be replaced with *unknown*, a special term that holds the place of any fact that cannot be represented any other way. In some cases, it is still possible to disprove a dependence; in others, the *unknown* remains in the simplified dependence relation, indicating an approximate answer. Second, each non-affine term can be replaced with an application of an *uninterpreted function symbol*. Any program expression produces a value each time it is executed, and can thus be viewed as a function of the loop indices. For example, in the absence of a closed-form definition, nn and nn' could be written $nn(n1, k1)$ and $nn(n1', k1')$,

though in this case this does not provide any advantage. Sometimes an expression can be viewed as a function of other arguments, though the current implementation of the Omega Test does not include general techniques for detecting such cases.

Presburger Arithmetic with arbitrary uses of uninterpreted function symbols is undecidable [Dow72]. The algorithms in [Won95] allow each argument to a function to be any affine combination of loop indices and symbolic constants; the current implementation requires that the list of arguments be a prefix of the set of loop indices (i.e. in a loop nest `i`, `j`, `k`, a binary function could be only be applied to the argument list i, j). If the Omega Library is forced to apply a function to some other argument, it simply replaces it with *unknown*.

The Omega Library simplifies a formula containing a function symbol by splitting it into cases in which the arguments are provably equal or distinct. That is, if the formula F contains $f(i, j)$ and $f(i', j')$, then F is replaced with

$$(F \wedge i = i' \wedge j = j') \vee (F \wedge \neg(i = i' \wedge j = j')).$$

When the arguments are equal, the two function applications can be treated as a single scalar variable; otherwise, they can be treated as independent scalars [Sho79].

This simple technique greatly increases the accuracy of the Omega Test (see [PW98] or Section 5). Unfortunately, this makes the size of a formula with u uses of a function grow by a factor equal to the number of ways to partition a set of size u, or *Bell number* [Com74], which is exponential (note that every subset defines a partition). All but one of the terms in the resulting disjunction will have at least one negated set of a equalities (for a function of arity a), and the Omega Library treats these as as a 2^a-way disjunction, multiplying the size of the formula $O(2^a)$. (For example, F above is split into five formulas, with $(i = i' \wedge j = j')$, $(i < i')$, $(i > i')$, $(i = i' \wedge j < j')$, and $(i = i' \wedge j > j')$, respectively). Even with the limitations of the current implementation, which allow at most two distinct sets of arguments for each function, a formula with any number of functions of arity up to a can increase in size by this factor of $O(2^a)$.

3 An Efficient Subdomain

Many of the Presburger Formulas that arise during dependence analysis fall into the following subdomain of Presburger Arithmetic, and can be tested for satisfiability in polynomial time:

a. The individual "atomic" constraints are drawn from the *LI(2)-unit* subdomain (which allows only affine equality and inequality constraints with unit coefficients and no more than two variables per constraint).

b. These constraints are only combined in the form $\exists\ v_1, v_2, ...v_n$ s.t. $C_0 \wedge \neg C_1 \wedge \neg C_2 ... \wedge \neg C_m$, where each C_i term is a conjunction of atomic constraints involving $v_{0..n}$ and possibly additional existentially quantified variables local to C_i.

c. $C_1..C_m$ can be ordered such that at most one *negatable equation* in C_1 is not redundant with respect to C_0, at most one negatable equation in C_2 is not redundant with respect to $C_0 \wedge \neg C_1$, etc. A negatable equation is any atomic constraint that we can negate without introducing disjunction, specifically, either (i) an inequality, (ii) a stride constraint, (iii) a disequality (i.e. $x \neq y + 1$), or (iv) if the system is *splayed*, an equality constraint. (Splayed systems are defined below).

Property (b) holds for all formulas for memory-based analysis (when $m = 0$) and value-based analysis (whether using the kill or cover test). Property (a) ensures that the Omega Library will perform satisfiability tests on individual conjuncts in polynomial time, and Property (c) defines the cases in which the Omega Library can handle negation in polynomial time. It is possible to produce programs with affine loops bounds and subscripts that still violate Property (a) or (c), though these appear to be very rare in practice, as we show in Section 5.

3.1 Conjunctions of LI(2)-unit Equations

A system of equations is considered to be LI(2)-unit if all elements are in the form $ax + by + c = 0$ or $ax + by + c \geq 0$ where $a, b \in \{+1, -1, 0\}$. The Omega Library uses repeated variable elimination to test satisfiability of a conjunction of arbitrary affine inequalities on existentially quantified integer variables (a formula with no variables is either $true$ or $false$). Substitution is used to eliminate a variable that is involved in an equality constraint, and Pugh's extension of Fourier's method of variable elimination is used to eliminate the remaining variables [Pug92]. Fourier's method produces all combinations of one upper bound and one lower bound on each variable being eliminated. In the worst case, this causes the number of inequalities to grow exponentially (even without the added complexity of Pugh's extension).

The LI(2) and LI(2)-unit subdomains are both closed under variable elimination. For LI(2), Fourier's method is sub-exponential, and other algorithms are faster still [Cha93, Nel78]. For LI(2)-unit, all constraints on a given pair of variables (x, y) must be parallel to either $x+y = 0$ or $x-y = 0$. This fact ensures that Pugh's techniques for parallel redundant constraint detection will be highly effective, and allows the removal of v variables in $O(v^3)$ time [1] [Pug92]. Thus, satisfiability testing of a formula of size n is done in $O(n^3)$ time.

3.2 Extension of the LI(2)-unit subdomain

Our domain can be extended to include sets of constraints which are, in one way or another, "almost" LI(2)-unit, The method we discuss here occurs in a large fraction of the non-LI(2)-unit cases we encountered in benchmarks. However, since the number of non-LI(2)-unit cases

[1] If a formula includes large numbers of redundant constraints, satisfiability testing may take $O(v^3 + c)$ for v variables and c LI(2)-unit constraints.

is so small, the frequency of occurence of these cases as a whole is very small. We have not yet implemented these extensions.

The LI(2)-unit domain can, of course, be extended to any system of equations which, though it is not in an LI(2)-unit form, can be transformed into LI(2)-unit. The Omega Library will sometimes produce equations of the form $ax + ay + k \geq 0$. Such a case can be normalized (using Pugh's extension of Fourier's method [Pug92])into the LI(2)-unit equation $x + y + (k)mod(a) \geq 0$. While normalization changes the real solutions to a constraint, it will not change the integer solution [Pug92].

We found that 41% (525/1282) of the non-unit LI(2) dependence equations resulting from our five benchmarks could be normalized in this fashion. However, since the number of non-unit LI(2) equations is only .069% (1282/1851169) of the total number of lines of dependence equations, performing such normalization may not prove to be efficient. It is, however, constant time complexity to normalize and therefore linear time complexity (with respect to the number of equations) to find and correct all of them.

3.3 Gist Operations and Conditional Analysis

In the worst case, the gist operation may need to perform $O(\mid q \mid)$ satisfiability tests to produce *gist* p *given* q (where $|q|$ is the number of equations in q). When p and q are both in LI(2)-unit, these each take $O(v^3)$ time, for a total of $O(v^3 \mid q \mid)$ steps.

3.4 Negation

Our value-based analysis produces queries of the form

$$\exists\ v_1, v_2, ...v_n \text{ s.t. } C_0 \wedge \neg C_1 \wedge \neg C_2 ... \wedge \neg C_m$$

as described above. The Omega Library tests such a system for satisfiability by converting it into disjunctive normal form (hereafter, we refer to a formula in this form as "a DNF") and applying Pugh's extension of Fourier's method to test the satisfiability of each term of the disjunction.

Consider first the case of $m = 1$, in which C_1 is a conjunction of g inequalities. Simply negating C_1 and distributing it over C_0 would produce a DNF of size $g \mid C_0 \mid$. However, any inequality of C_1 that is redundant with respect to C_0 (that is, $x \wedge C_0 \equiv C_0$) would produce an unsatisfiable conjunction, which would then be eliminated by the subsequent satisfiability tests. The Omega Library uses the gist operation to improve the efficiency of this process by eliminating such redundant terms before converting to DNF, replacing C_1 with *gist* C_1 *given* C_0. For this step, it uses a less complex variant of the gist operation [PW92, Won95] that may leave some redundant constraints in C_i. This test will detect any constraint that is redundant with respect to a single constraint or pair of constraints in C_0 (we term such constraints *obviously redundant* with respect to C_0), and runs in time $O(|C_0|^2|C_1|)$.

A simple stride constraint, such as $\exists\ \alpha$ s.t. $i = 3\alpha$, can be negated without introducing disjunction. However, other equality constraints are treated as a pair of inequalities. Property (c) ensures that, after our simple gist test, $\neg C_1$ will not include disjunction, so checking any $C_0 \wedge \neg C_1$ in our domain requires a single simple gist operation and a single satisfiability test.

When $m > 1$, it may be the case that, for each C_i, $\neg(gist\ C_i\ given\ C_0)$ does not introduce disjunction. In this case, the Omega Library simply processes the C_i as above, using m simple gist operations and m simplifications (a single simplification at the end would suffice for this case). However, it may be the case that a constraint in some C_i is made redundant by the negation of a constraint in some other C_j. If $j < i$, $\neg C_j$ will be incorporated into C_0 before C_i is processed, and then C_i may not produce disjunction. If $i < j$, the Omega Library does not incorporate $\neg C_i$ immediately, as this would introduce disjunction. Instead, it performs repeated passes of gist operations, incorporating only those terms that do not introduce disjunction. For any formula in our domain, this will produce a single conjunction with at most $\frac{m(m+1)}{2}$ gists and m satisfiability tests. Thus, the total time is $O(m^2((|C_0| + m)^2\ max_i(|C_i|)))$ for gists plus $O(mv^3)$ for simplifications, which is $O(n^5)$ for a formula of size n.

For queries outside of our domain, the Omega Library will eventually perform a pass in which no new C_i can be negated without introducing disjunction. At this point, it introduces disjunction: negating a C_i with g inequalities, s strides, and e equalities produces a $(g + s + 2e)$-way disjunction. Doing this for m such negated conjuncts produces a DNF with $(g + s + 2e)^m$ terms, each of which must be tested for satisfiability (without Property (a), each of these tests may be exponential).

Note that this restriction does not take into account the possibility of doing analysis with a different set of constraints that could fit within our domain. For example, it is always legal to replace the cover test with a collection of kill tests, which sometimes produce simpler negations.

3.5 Non-affine Terms

Property (a) allows only affine terms, so no dependence problem within this domain will involve function symbols. Unfortunately, a significant number of dependences involve non-affine terms. As with the full Omega Test, we have the option of treating each non-affine term as an *unknown* or an uninterpreted function; the former introduces a large number of false dependences [PW98, Table I].

To retain our polynomial time bound while allowing the use of function symbols, we modify the algorithm for manipulating uninterpreted function symbols to replace the constraints describing the conditions under which the arguments are distinct (e.g. $\neg(i = i' \wedge j = j')$) with *unknown*. Furthermore, if there are functions of different arities, we use a single formula with *unknown* to represent all cases in which arguments differ. Under this restriction and the restriction that any given function may only be applied to two distinct lists of arguments (as in the current implementation), the presence of any number of applications of any number

of functions can only increase the size of a formula by a factor of the largest arity of any function. This modification does not restrict the domain of our test, but it could force our test to produce an approximate answer.

4 Programs Fitting This Domain

To ensure that all dependence expressions are LI(2)-unit, we limit control flow to allow only **for** loops, and limit loop bounds and subscript expressions to only LI(1)-unit expressions (i.e. variable + constant or just a constant) of the symbolic constants and loop indices. We can also allow structured **if**'s, as long as they perform equality or inequality tests involving LI(2)-unit expressions. For programs in this domain, Property (a) must hold for all dependence tests.

Note that the restriction to LI(1)-unit dependence expressions is sufficient, but not necessary, for exact polynomial time memory-based analysis: programs outside of this domain may have LI(2)-unit constraints, and constraint sets outside of LI(2)-unit may be solved efficiently by the Omega Library. For example, the `trip` routine of the `ARC2D` benchmark contains references with non-LI(1) subscripts `j1+j2-i` and `j2`, which produce the LI(2)-unit constraint $j1 - i = 0$.

We can still perform dependence analysis in the presence of non-LI(1)-unit dependence expressions, by handling them as if they were not affine. That is, we can introduce an *unknown* or a function symbol. The latter does not seem to provide any significant advantage in practice (and would probably be slower than simply applying the Omega Test with a few non-affine terms). Note that we can still eliminate some dependences that involve constraints outside of our domain, since the remaining constraints may be enough to prove the system is unsatisfiable.

As stated in Section 2, all of the constraint sets produced by our analysis have Property (b).

Describing the set of programs that do not violate Property (c) is somewhat tricky. We believe that this property will hold in any procedure in which each array use is covered by a single write statement, or when an array is defined as a collection of chunks, organized along in a single dimension of the array. This domain should include many programs, but it also omits some interesting cases. Instead of attempting to avoid building constraints that violate Property (c), we simply build the constraints and introduce *unknown*s when we would otherwise have to introduce disjunction (this can still be done in polynomial time).

Our current implementation does not include any system to determine that an alternate set of equations might have Property (c). For example, the `sshap4` routine of the `DYFESM` benchmark contains an assignment to each of the first four elements of the `b` array, followed by a loop nest that repeatedly reads elements `1..nnpes` of `b`. The cover test generated for this read attempts to subtract element 4 before element 1; this would introduce disjunction, and thus we produce an approximate answer (though there is in fact no loop-carried dataflow). False dependences such as this one can often be eliminated by applying the kill test rather than the cover test, or by reordering the sequence in which the covering dependences are

applied when it is legal to do so. In this particular case, interprocedural constant propagation that determines that $nnpes = 4$ would also do the trick.

5 Empirical Results

Identifying a domain in which our analysis is both fast and exact is only valuable if real programs produce analysis problems from this domain. We have developed a restricted version of the Omega Test, and applied it to a set of benchmarks, to measure the impacts of the restrictions given in Sections 3 and 4. As we noted above, this produces an overestimate of the number of dependences that could be found in polynomial time by a more sophisticated test. We therefore manually examined the new (false) dependences to produce a more accurate count of the number that could be detected by a less conservative polynomial-time test.

5.1 Experimental System

We based our restricted Omega Test on a modified version of Release 1.2 of Petit and the Omega Library. Release 1.2 includes our modifications to implement the restrictions on Presburger Formulas from Section 4, under control of the new "-X" (no eXponential cases) flag to Petit. Specifically, with this flag, the part of Petit that determines whether or not a dependence expression is affine treats any non-LI(1)-unit term as an *unknown*, and the Omega Library gives *unknown* as the result of any negation of more than one constraint. For our experiments, we fixed the restriction on negation to allow only a single inequality or stride constraint, and modified the algorithm for uninterpreted function symbols as per Section 3.5. Finally, we had to increase the variable `maxVars` to 56 to handle all dependences from the benchmark set we used. These modifications will be included in the next release.

5.2 Programs to be Tested

We applied our modified version of the Omega Test to the five programs from the Perfect Club Benchmarks [B+89] that were studied by Maydan [May92] and in our previous work [Won95, PW98]. To allow comparison with earlier results, we reinstated the old definition of reduction dependence (via a new compile-time option in Petit 1.2). We also collected data only for dependences due to arrays of type real, and only dependences that fundamentally inhibit loop-level parallelism: loop-carried value-based flow dependences that are part of a dependence cycle (via the Petit flags "-g -r -j -Fvcf"). In this study, as in the past, we only consider only intra-procedural dependences.

With these flags set, the full Omega Test prints 271 dependences. These correspond exactly to the 188 inter-line dependences given in [PW98, Table I]; the difference occurs because the Omega Test reports dependences by the pair of references and the carrying loop, not by the pair of lines. (Our full techniques for handling uninterpreted function

Table 1: Number of Affine but not LI(2)-unit Dependence Expressions, by Benchmark

Benchmark	Dependence expressions	Affine but not LI(2)-unit
QCD2	1535	80 (5%)
MDG	602	74 (12%)
DYFESM	1760	25 (1%)
ARC2D	3906	72 (2%)
TRFD	252	0 (0%)
Total	8055	251 (3%)

Table 2: Number of Parallelism-Blocking Dependences Found by Various Tests, by Benchmark

	Dependence Test		
Benchmark	Full	Restricted	Manual
QCD2	65	65	65
MDG	64	78	78
DYFESM	53	184	58
ARC2D	89	114	97
TRFD	0	0	0
Total	271	441	298

symbols, which are not currently implemented, allow the elimination of five more inter-line dependences.)

5.3 Results

Tables 1 and 2 give our measurements of the effects of our restrictions. Table 1 shows the total number of dependence expressions and how many of these expressions were affine but not LI(1)-unit. Overall, only about three percent of dependence expressions fall into this category. This is consistent with observations by Maydan [May92] and others that memory-based dependence testing based on arbitrary affine constraints produces little improvement over simple tests.

Table 2 shows the number of intra-procedural loop-carried value-based flow dependences that arise from arrays of type real and are part of a dependence cycle, for various restrictions on the Omega Test. The "Full" column gives the number of dependences found without our restrictions. The "Restricted" column gives the number of dependences found when the "-X" flag is used to restrict dependence testing. The "Manual" column gives the number that remain after we manually identified and removed cases in which our restrictions are overly conservative. Specifically, we found 16 cases in which an LI(2)-unit dependence arises from a

set of non-LI(1) dependence expressions, and 127 cases in which the use of the kill equations would produce a set of constraints within our restricted domain, though the cover equation did not. This leaves only 27 false dependences that are introduced by the restriction to polynomial time analysis.

The 27 false dependences that we could not avoid while using our polynomial algorithms had the following effects on parallelization:

Eight inhibit parallelization of three loops in `ARC2D` that are already blocked by some of the 271 dependences found by the unrestricted test.

Two block the parallelization of the loop from `cnstnt` shown in Figure 1 (these arise from the non-LI(2) constraint $n1 - k1 + 1 = n1' - k1' + 2$).

Twelve block parallelization of a pair of nested loops from `initia` (these all include the single non-LI(2)-unit constraint $3i + j - 3 = 3i' + j' - 3$).

Five block outer loop parallelism in the `compl` subroutine of the `DYFESM` benchmark.

The `cnstnt` and `initia` subroutines are each called only once, during initialization of the `MDG` benchmark, and are not among the loops that constitute a significant fraction of the work done by this program [EHLP91]. The `compl` subroutine is also not critical to parallelization [EHP98].

As a point of comparison, restricting the Omega Test to use only affine constraints (i.e. treat all functions as *unknown*) introduces 944 false dependences. For these benchmarks, information about non-affine terms is far more important than information about affine terms that lie outside of our polynomial subdomain.

We did not perform any new measurements of analysis times, as our earlier work on the unrestricted Omega Test [Won95, PW98] included timings of a superset of the analysis done here. On a Sun Sparcstation 10/51, for all five benchmarks (totaling about 18000 lines of code), the unrestricted test required 25 seconds for memory-based analysis, 40 seconds for value-based analysis, and 71 seconds for conditional analysis. By limiting our analysis to dependences that might influence array privatization (the main use of value-based analysis in parallelization), we can reduce this total analysis time from 136 to 36 seconds [Won95].

6 Related work

Most traditional memory-based dependence tests, such as Banerjee's inequalities or the GCD test [Wol89], run in polynomial time. Early work on array dataflow analysis was not exact over our entire polynomial domain. Brandes [Bra88] and Ribas [Rib90] each developed extensions of array dependence analysis that eliminate killed dependences. Brandes's analysis requires rectangular loop nests, and Ribas's techniques require perfectly nested loops. Rosene [Ros90, Chapter 3] and Gross and Steenkiste [GS90] extended scalar dataflow analysis to describe regions of arrays. Gross and Steenkiste restrict their analysis to represent only

rectangular regions in arrays, which are a special case of the regions that can be represented in our system. Rosene described a system with a somewhat more general representation, but his system was never implemented.

More recent work, including our previous work on array dataflow [PW93, PW98], has focused on producing more general results, without provably polynomial time bounds. Feautrier's array dataflow analysis [Fea91] also involves extending an integer programming system to allow negated constraints. His techniques appear to be more expensive than our general system [PW93]. Maydan and Lam [May92] developed algorithms to produce Feautrier's results more efficiently in certain common cases, but did not focus on finding a polynomial subdomain or give a bound on the complexity of their algorithm (which, in the worst case, resorts to Feautrier's techniques).

Li's techniques [Li92, GLL95, GLL97], like those of Rosene and Gross and Steenkiste, produce information about exposed regions of arrays (but with a focus on the information needed for privatization). Li gives extensive empirical evidence that his techniques are efficient [GLL97], but does not give a polynomial time bound. Tu and Padua [TP92, Tu95] convert code into gated single assignment form and then perform a demand-driven backward substitution to detect inter-iteration flow of values. They do not provide any polynomial bound on the amount of backward substitution that must be performed.

Another possible approach to polynomial-time analysis is to use an interrupt timer or a limit on the size or number of terms in a dependence, recording any dependence that exceeds the limit as *unknown*. This would allow us to produce exact results for many dependences that are slightly outside of the domain given in this paper, such as the dependences from `CNSTNT` shown in Section 2. However, this solution lacks the elegance of a true polynomial bound, the results might vary significantly for different limits, and a system based on a timer could even produce unpredictable results from run to run.

7 Conclusions

Most of the affine loop bounds and subscripts that occur in real programs are quite simple. This allows polynomial-time dependence analysis techniques to be almost as accurate as those that handle arbitrary affine constraints. Prior work with polynomial-time analysis has focused on memory-based tests, which must be followed by a separate (typically exponential) dataflow analysis algorithm to produce information for advanced optimizations such as privatization.

We have described a subdomain of Presburger Arithmetic in which satisfiability testing can be performed in polynomial time. The existing algorithms in the Omega Library are polynomial in this subdomain. Thus, we can perform provably efficient and exact memory-based, value-based (dataflow), and conditional analyses with the Omega Test for any dependence within this subdomain. In the benchmarks we studied, most dependences that fell outside of this subdomain did so because of non-affine terms (rather than complex yet still affine terms).

We can perform analysis on dependences outside of our domain with either of the techniques used in previous work on the Omega Test (*unknown*s or uninterpreted function symbols). We maintain our polynomial time bound when manipulating function symbols by making a slight modification to the Omega Library's original algorithm.

Either technique for handling terms outside of our domain can introduce approximations, which may produce false dependences. However, our studies of benchmark codes indicate that the number of false dependences is quite small (just under ten percent of the total number of dependences). Furthermore, the false dependences do not have a significant impact on parallelization: in almost 18000 lines of code in five benchmarks, this restriction blocked parallelization of only three unimportant loop nests.

There are several practical benefits of the use of a single set of algorithms that can be applied to either the complete Presburger domain or our polynomial-time subdomain. Techniques that have been developed to derive or apply program transformations based on the Omega Test's dependence relations can still be applied (although they may receive less accurate information when analysis is restricted to polynomial cases). Additionally, any compiler based on this approach can easily provide a run-time switch to allow or disallow potentially exponential analysis, as is done in the Petit system.

References

[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.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelism. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 407–417, July 1988.

[Cha93] Vijay Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, 1993.

[Com74] Louis Comtet. *Advanced Combinatorics*. D. Reidel Publishing Company, Dordrecht, Holland, 1974.

[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 *Proceedings of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991. Also Technical Report 1193, CSRD, Univ. of Illinois.

[EHP98] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the Perfect benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 9(1), January 1998. Also Technical Report 1392, CSRD, Univ. of Illinois.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, September 1994.

[FR74] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. *SIAM-AMS Proceedings*, 7:27–41, 1974.

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

[GLL95] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing '95*, San Diego, Ca, December 1995.

[GLL97] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, Las Vegas, Nevada, June 1997.

[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.

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

[KMP+95a] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[KMP+95b] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *Petit: a tool for analyzing and transforming array calculations*. Dept. of Computer Science, University of Maryland, College Park, 1995. see http://www.cs.umd.edu/projects/omega/index.html.

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

[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.

[Nel78] C. G. Nelson. An $o(n^{logn})$ algorithm for the two-variable-per-constraint linear programming satisfiablility problem. Technical Report AIM-319, Stanford University, Department of Computer Science, 1978.

[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*, 35(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 *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, August 1993. Also available as Tech. Report CS-TR-3196, Dept. of Computer Science, University of Maryland, College Park.

[PW95] William Pugh and David Wonnacott. Nonlinear array dependence analysis. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998. http://www.acm.org/pubs/citations/journals/toplas/1998-20-3/p635-pugh/.

[Rib90] Hudson Ribas. Obtaining dependence vectors for nested-loop computations. In *Proceedings 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.

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

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, September 1992.

[Tu95] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Mass., 1989.

[Won95] David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995. Available as ftp://ftp.cs.umd.edu/pub/omega/davewThesis/davewThesis.ps.

[Won00] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.