Polynomial Time Array Dataflow Analysis

Robert Seater and David Wonnacott

Haverford College, Haverford, PA 19041
davew@cs.haverford.edu
http://www.cs.haverford.edu/people/davew/index.html

Abstract. Array dataflow analysis is a valuable tool for supercomputer compilers. However, the worst-case asymptotic time complexities for modern 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 the results of our investigation of the effects of these restrictions on the detection of loop-carried array dataflow dependences (which prevent parallelization). These restrictions block parallelization of only a few unimportant loop nests in the approximately 18000 lines of benchmark code we studied. The use of our subdomain of Presburger Arithmetic also gives a modest reduction in analysis time, even with our current unoptimized implementation, as long as we do not employ our modified algorithms for function symbols. The data collected in our empirical studies also suggest directions for improving both accuracy and efficiency.

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 [1]. It can also serve as the basis for optimization of communication [2] or memory locality [3]. Unfortunately, the worst-case asymptotic 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 [4] and value-based [5, 6] 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 [7]. Value-based analysis includes negated constraints, and thus requires a richer constraint language; the Omega Test uses Presburger Arithmetic [8], which appears to have time complexity $2^{2^{2^{O(n)}}}$ [9, 10].

Empirical studies of benchmark programs have found that affine dependence expressions are generally very simple. As we shall see, this lets the Omega Test run quickly. It also lets simple memory-based tests provide fairly accurate information. Compiler writers thus choose between the risk of missing an optimization and the risk of a non-terminating compilation due to an exponential case in a more general test. However, the simple memory-based tests carry a hidden cost, in that they do not provide the Omega Test's exact description of each dependence. More accurate information is needed for value-based and conditional analyses, which can be much more important than exact affine memory-based testing [11, 1]. Empirical studies have shown that these analyses can be efficient practice (see Section 6), but the lack of a polynomial bound still raises questions about the use of such techniques in production compilers.

In addition to leaving open the possibility that some unusual program may require years of dependence analysis, empirical studies provide very little information about *why* an exponential algorithm works well in practice. An $O(2^n)$ algorithm may run well because the test data do not exercise the exponential behavior of the algorithm, or because n is small: a compiler can afford to perform some tasks with 2^{10} operations (taking about a microsecond), or even a few with 2^{20}. In contrast, $2^{2^{2^n}}$ is disastrous, even for small n: A petaflop computer would have to run for about 10^{45} times the age of the universe to do $2^{2^{2^3}}$ operations. Clearly there is some other factor working in our favor here.

In this paper, we identify a subdomain of Presburger Arithmetic for which we can perform polynomial-time satisfiability testing; this lets us define a polynomial-time array dataflow algorithm by simply introducing an approximation for any term outside of this subdomain. We begin, in Section 2, with a brief restatement of the constraint-based approach to dependence 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 the speed and accuracy of our analysis. Section 6 covers related work, and Section 7 presents our conclusions.

2 Constraint-Based Dependence Analysis

In this section, we review the Omega Test's constraint-based approach to dependence analysis. Space limitations prevent a full review; readers who are not familiar with this work may wish to consult [12, 6].

In its most general form, a dependence is a relation between tuples of integers that correspond to the values of the loop index variables at the source and sink of the dependence. Since we may not have a bound on the number of loop iterations, we cannot hope to enumerate them, and thus we represent dependence relations with constraints on integer variables. For example, in Figure 1, the memory-based flow dependence from iteration $[\ n1, k1\ ]$ to iteration $[\ n1', k1'\ ]$ is

$$\{\ [\ n1, k1\ ] \to [\ n1', k1'\ ] \mid n1' = n1 \wedge k1' = k1 + 1 \wedge 2 \le k1 < n1 \le n\ \}$$

The constraints under which memory aliasing occurs can be obtained from the loop bounds and array subscripts, as well as the fact that the source of any dependence must precede its sink. In some cases, other program analyses are also useful: For the relation above, we substituted the closed form `n1-k1+1` for the induction variable `nn`.

```
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
    }
}
```

Fig. 1. Excerpt from CNSTNT in MDG [13]

When a dependence relation 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 [7], which is NP-Complete [14].

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. This can be done by eliminating iterations in which the dependence is *killed* or, in some cases, by eliminating iterations in which the sink of the dependence is *covered*. In either case, we must negate a conjunction of constraints that may involve existentially quantified variables. For example, we can find the set of iterations that are exposed above the aforementioned dependence as:

$$\underbrace{\{[n1,2] \mid 2 \le n1 \le n\}}_{iterations\ exposed} = \underbrace{\{[n1,k1] \mid 2 \le k1 \le n1 \le n\}}_{iterations\ executed} - \underbrace{\{[n1,k1{+}1] \mid 2 \le k1 < n1 \le n\}}_{iterations\ reached\ by\ dependence}$$

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 [8]. Presburger Arithmetic appears to have time complexity $2^{2^{2^{O(n)}}}$ [9, 10].

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 such 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 [15, 16]. 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.

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* [17, 18, 6]: Any program expression produces a value each time it is executed, and can thus be viewed as a function of the loop indices (or, in some cases, a smaller set of parameters).

Presburger Arithmetic with arbitrary uses of uninterpreted function symbols is undecidable [19]. The algorithms in [16] 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 function applications can be treated as a single scalar variable; otherwise, they can be treated as independent scalars [17].

This simple technique greatly increases the accuracy of the Omega Test (see [6] or Section 5.4). 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* [20], 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 (i.e. constraints with up to two variables and unit coefficients).
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 C_1 contains at most one constraint that is not *obviously redundant* (see Section 3.3) with respect to C_0, and which must be an inequality or simple stride constraint (e.g. x is a multiple of 3); C_2 contains at most one such constraint that is not obviously redundant with respect to $C_0 \wedge \neg C_1$; C_3 contains at most one such constraint that is not obviously redundant with respect to $C_0 \wedge \neg C_1 \wedge \neg C_2$; etc.

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

An equation is said to be *LI(2)* if it has the form $ax + by + c = 0$ or $ax + by + c \geq 0$. It is *LI(2)-unit* if $a, b \in \{+1, -1, 0\}$. The satisfiability testing algorithm of the Omega Library can process any conjunction of LI(2)-unit equations on existentially quantified integer variables in polynomial time.

The Omega Library uses repeated variable elimination to test satisfiability (a formula with no variables is either *true* or $false$). Substitution is used for a variable that is involved in an equality constraint, and Pugh's extension of Fourier's method of variable elimination is used for the remaining variables [4]. Fourier's method produces all combinations of an upper bound and a lower bound on the given variable. This can cause the number of inequalities to grow exponentially with the number of variables eliminated.

Within the LI(2)-unit domain, all constraints on a pair of variables (x, y) must be parallel to either $x + y = 0$ or $x - y = 0$. Furthermore, the LI(2)-unit domain is, like LI(2), closed under variable elimination. These facts ensure that Pugh's techniques for parallel redundant constraint detection will succeed in limiting the number of inequalities produced during each variable elimination, allowing the removal of v variables in $O(v^3)$ time[1] [4]. Thus, satisfiability testing of an LI(2)-unit formula of size n is done in $O(n^3)$ time.

Note that constraints like $j1 + j2 - i = j2$ and $2x \leq 7$ can easily be made LI(2)-unit, since they can be rewritten as $j1 - i = 0$ and $x \leq 3$ (respectively). These conversions are performed during the normal processing by the Omega Library. We have investigated techniques for converting other classes non-LI(2)-unit constraints into this domain, but we have not found this to be important in practice. The details of these conversions are covered in [12].

3.2 Gist Operations and Conditional Analysis

In the worst case, the gist operation discussed in Section 2 may need to perform $O(|q|)$ 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 $O(v^3\ |q|)$ steps.

3.3 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 $g\ |C_0|$. 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.

[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.

For this step, it uses a less complex variant of the gist operation [15, 16] 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). In other cases, the Omega Library makes up to m passes over these terms to find cases in which a constraint in some C_i is made redundant by the negation of a constraint in some other C_j. For any formula in our domain, this will produce a single conjunction after 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.

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.

3.4 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 [6, 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 memory-based dependence tests involve conjunctions of LI(2)-unit constraints, we restrict programs as follows: Control flow statements must be only `for` loops with bounds that are LI(1)-unit expressions (i.e. variable + constant or just a constant) of the symbolic constants and outer loop indices, `if`'s that perform LI(2)-unit inequality tests, or `else`-less `if`'s with LI(2)-unit equality tests. Furthermore, all pairs of corresponding subscript expressions for a given array must have a difference that is LI(2)-unit.

It is tempting to restrict each individual subscript to the LI(1)-unit domain, but this would be overly severe: Subscript expressions that are not LI(1)-unit can still produce LI(2)-unit constraints. For example, the `trip` routine of the `ARC2D` benchmark contains references with the subscripts `j1+j2-i` and `j2`, which produce the LI(2)-unit constraint $j1 + j2 - i = j2$.

We can still perform dependence analysis in the presence of expressions outside of our domain by handling them as if they were not affine (i.e. by introducing an *unknown* or a function symbol).

As stated in Section 2, all of the constraint sets for our analysis have Property (b). Thus there is no need to restrict programs to produce this property.

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).

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 Section 3.

5.1 Experimental System

We based our restricted Omega Test on a modified version of Release 1.2 of Petit and the Omega Library (our modifications will be included in the next bug fix release). We use the "-X" flag (no eXponential cases) of this release to cause Petit to introduce an *unknown* for any non-LI(2)-unit constraint, any conditional statement that introduces disjunction, or any negation that would introduce disjunction. This flag also enables our modifications of algorithm for uninterpreted function symbols given in Section 3.4.

We have not implemented a system for using the dependence killing test selectively when the cover test loses accuracy due our limitations on negation. To factor out this effect, we disabled the cover analysis during our experiments (this has no impact on the accuracy of the full Omega Test). We also made several minor enhancements and bug fixes to Release 1.2, and increased the variable `maxVars` to 56 to handle all dependences from the benchmark set we used. Our timing results were produced on a 400 MHz Intel Linux system.

5.2 Programs to be Tested

We applied our modified version of the Omega Test to the five programs from the Perfect Club Benchmarks [13] that were studied by Maydan [11] and in our previous work [16, 6]. 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.

5.3 Results

Table 1 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, given various restrictions on the Omega Test.

Table 1. Number of Parallelism-Blocking Dependences (and Newly Blocked Loops)

Benchmark	Dependence Test				
	Exp	Exp Cond	Exp UFS	Poly	LI(1)
QCD2	65	65	151	151 (3/1)	151
MDG	64	78	78	78 (3/2)	78
DYFESM	53	72	112*	111 (11/5)	223
ARC2D	89	89	89	89 (0/0)	114
TRFD	0	0	0	0 (0/0)	0
Total	271	304	430*	429 (17/8)	566

* Numbers marked * are 1 higher than they should be due to an Omega Library bug.

The "Exp" column gives a baseline of information about the full (exponential) test. The "Exp Cond" and "Exp UFS" columns give data for runs in which we applied several, but not all, of our restrictions. In "Exp Cond", we prevented all forms of exponential growth except those created by a disjunction in a program term (such as the conditions describing the `else` clause of the conditional `if (i==1)`); In "Exp UFS", we restricted our test to the polynomial subdomain of Presburger Arithmetic, but did not restrict disjunction from uninterpreted function symbols beyond the limitations of the current implementation. The "Poly" column gives the results for the fully polynomial algorithm. For this column, we have also given the number of loops and loop nests for which parallelization is blocked by a dependence in the restricted test, but not in the original test. Finally, the "LI(1)" column gives the results if we restrict our test to LI(1)-unit dependence expressions rather than LI(2)-unit dependence constraints.

```
do iblock = 1, nblock {
  iloc = pptr(iblock)
  do i = 1, iblen(iblock) {
    xdplus(iloc+i-1) = xd(iloc+i-1) +
         deltat * .5 * xdd(iloc+i-1)
    xplus(iloc+i-1)  = x(iloc+i-1) +
         deltat * xdplus(iloc+i-1)
  } }
```

Fig. 2. Excerpt from HOP0 in DYFESM [13]

The new dependences introduced by our restriction to the polynomial-time test have the following effects on parallelization of the benchmarks:

`QCD2` **(86 new dependences)** 10 new dependences block the parallelization of a loop nest in the main routine of `qcd2` (this nest contains I/O statements). 74 of the remaining 76 are due to statements in the `else` branches of two conditionals of the form `if (i==1)` in `choos` (this loop already carries a conditional dependence).

`MDG` **(14 new dependences)** Two of these block the parallelization of the loop from `cnstnt` shown in Figure 1, and twelve block parallelization of a pair of nested loops from `initia` (these all include the single non-LI(2)-unit constraint $3i + j - 3 = 3i' + j' - 3$).

`DYFESM` **(58 new dependences)** 15 of these dependences arise from a single pattern of dependences that occur in `hop`, `hop0`, and `solxdd`. This pattern, shown in Figure 2, is responsible for dependences carried by 8 loops in 3 nests in this benchmark. These dependences involve the expression $iloc + i - 1 = iloc' + i' - 1$, which is not LI(2)-unit. However, for the loop-independent dependence, $i = i'$ and $iloc = iloc'$, and the expression is simply *true*. We are investigating techniques for encorporating this type of information.

The remaining dependences include 4 that block outer loop parallelism in `compl`, 3 that block a pair of loops in `matinv`, and 36 that block the outer loop of `matinv`, which already carries conditional dependences.
This program also contained a dependence that brought to light a subtle bug in the Omega Library, causing the system to produce an approximate answer when it should have been able to disprove a dependence.

In total, the 158 new dependences introduced by the fully polynomial test block the parallelization of nine loop nests, two of which were already blocked by conditional dependences. According to [1, Table 3], none of the newly blocked loops contains important parallelism.

5.4 Discussion

Most of the dependences introduced by our restriction to polynomial-time analysis come from our refusal to incorporate information from conditions that could introduce disjunction (i.e. the restriction from "Exp Cond" to "Poly"). However, most of the newly blocked loops were a consequence of our restriction to LI(2)-unit constraints. The restriction to LI(1)-unit subscripts, rather than LI(2)-unit differences between pairs of subscripts, would almost double the number of false dependences introduced. Our techniques for controlling disjunction created by uninterpreted function symbols had no impact whatsoever on dependence detection or parallelism (except by uncovering a bug in the library).

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 outside of our polynomial subdomain.

5.5 Analysis Time

To determine the practical value of our polynomial-time algorithm, we measured the speed of our implementation. Table 2 gives the results of our timing measurements (in seconds). Our restriction to a polynomial time subdomain of Presburger Arithmetic gives a clear speed advantage: The "Exp UFS" column has the fastest analysis time of all. However, the inclusion of our restrictions on function symbols increases the analysis time (in the "Poly" column) beyond the time needed for full analysis.

We believe there are three significant factors slowing down our analysis: First, our technique for preventing exponential blowup during the processing of function symbols is not a practical benefit. Recall that we replace a potentially large disjunction with one *unknown*; when each element of the disjunction contains an easily discovered contradiction, eliminating it appears to be faster than testing the single conjunction containing the *unknown*. Second, our test for LI(2)-unit constraints appears to be too expensive: when we run the simpler LI(1)-unit test on the dependence expressions, we once again find a significant speedup in analysis time (though at a cost of significantly more dependences). Finally, the

Table 2. Analysis Times (in Seconds) for Various Tests

Benchmark	Dependence Test				
	Exp	Exp Cond	Exp UFS	Poly	LI(1)
QCD2	11.8	29.6	6.6	17.5	9.1
MDG	10.5	16.4	10.4	16.1	2.8
DYFESM	14.1	16.1	6.1	6.8	12.8
ARC2D	6.8	6.8	6.8	6.7	6.7
TRFD	1.6	3.0	1.5	2.9	1.0
Total	44.8	71.9	31.4	50.0	32.4

failure to accurately describe a loop-independent dependence that covers a read can force us to spend a lot of time analyzing loop-carried dependences that it would have killed. For example, our refusal to introduce terms like $i \neq 1$ in the analysis of `choos` in the `QCD2` benchmark more than doubles both the number of loop-carried dependences and the analysis time for this benchmark.

At the moment, the most practical result of our work is the combination of the polynomial subset of Presburger Arithmetic and the current Omega Library implementation of function symbols ("Exp UFS"). This guarantees polynomial-time analysis of affine dependences and consistently speeds up our analysis.

6 Related Work

Most traditional dependence tests, such as Banerjee's inequalities or the GCD test [21], run in polynomial time, but do not produce dataflow information.

Early work on array dataflow analysis was not exact over our entire polynomial domain. Brandes [22] and Ribas [23] 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 [24, Chapter 3] and Gross and Steenkiste [25] extended scalar dataflow analysis to describe regions of arrays. Gross and Steenkiste restrict their analysis to represent only rectangular regions in arrays, allow only LI(1) subscripts for which each loop counter appears in only one subscript expression (which must be the same subscript in all references to each array in a given loop). 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 [5, 6], has focused on producing more general results, without provably polynomial time bounds. Feautrier's array dataflow analysis [26] also involves extending an integer programming system to allow the removal of iterations from which dependences are killed. His techniques appear to be more expensive than our general system, primarily due to differences in our techniques for negated constraints [5].

Maydan and Lam [11] developed algorithms to produce Feautrier's results more efficiently in certain common cases. However, they note that, in general, "the complexity of our algorithm grows exponentially with the number of write statements" [11, Section 4.5].

More recently, Hall, Murphy, Amarasinghe, Liao, and Lam [27, 28] developed an array dataflow algorithm based on the use of affine constraints to represent exposed regions of an array. They produce efficient results by introducing approximations in some cases where negation is required, but use Fourier's (exponential) algorithm to test for satisfiability [28, Section 5.4.1]. Moon, Hall, and Murphy [29, 30] have modified this system to produce information about conditional dependences, but did not limit the asymptotic complexity.

Li's techniques [31–33], like those of Rosene and Gross and Steenkiste, produce information about exposed regions of arrays. Li gives extensive empirical evidence that his techniques are efficient [33], but does not give a polynomial time bound. Tu and Padua [34, 35] 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 property allows polynomial-time analysis that is generally accurate in practice, or exact analysis that is generally efficient in practice, for programs with affine dependence expressions. We have produced a modified version of the Omega Test that allows the user of a compiler to switch between these two options with a command-line argument. Our restricted algorithm improves on recent (post-1990) work in that it has a polynomial-time upper bound; it improves on earlier (pre-1991) systems with exact analysis for cases that had been handled approximately (i.e. imperfectly nested or triangular loops, certain kinds of coupled subscripts, or LI(2)-unit equations that stem from non-LI(1) subscripts).

Our algorithm is based on a subdomain of Presburger Arithmetic for which satisfiability testing can be performed in polynomial time. The existing algorithms in the Omega Library are polynomial in this subdomain. Thus, we can select either "approximate but provably fast" or "exact but potentially slow" options for all our existing array analysis algorithms (memory-based, value-based (dataflow), and conditional analyses) and process the results with a single set of algorithms.

We can maintain our polynomial time bound when manipulating function symbols by making a slight modification to the Omega Library's original algorithm, though this does not appear to be useful in practice.

In the benchmarks we studied, most dependences that fell outside of our subdomain of Presburger Arithmetic did so because of non-affine terms rather than complex yet still affine terms. Dependences with "complex yet still affine" terms often include only a single deviant constraint, such as the subscript equality in the `initia` routine, or the conditional in `choos`. By restricting ourselves to our polynomial subdomain of Presburger arithmetic, we can reduce total analysis time for the benchmarks we studied from 44.8 to 31.4 seconds (as long as we do *not* use our modified function symbol algorithm). Most of the loop-carried dataflow dependences that were introduced were carried by loops that already carried conditional dataflow. None of the previously dependence-free loops that were blocked by false dependences contained significant parallelism.

This work is essentially a formalization and experimental validation of some of the intuitions that have developed in the constraint-based array analysis community over the past decade. It is our hope that this formalization can serve as a guide to help identify other cases in which constraint-based techniques can be used effectively.

8 Acknowledgments

This work is supported by NSF grant CCR-9808694.

References

1. 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.
2. Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, September 1994.
3. 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.
4. 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.
5. William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on 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.
6. 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/`.
7. 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.
8. G. Kreisel and J. L. Krevine. *Elements of Mathematical Logic*. North-Holland Pub. Co., 1967.
9. Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In Richard M. Karp, editor, *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, Providence, RI, 1974. AMS.
10. 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.

11. Dror Eliezer Maydan. *Accurate Analysis of Array References*. PhD thesis, Computer Systems Laboratory, Stanford U., September 1992.
12. Rob Seater and David Wonnacott. Polynomial time array dataflow analysis. Technical Report 2000-38, Center for Discrete Mathematics and Theoretical Computer Science, November 2000.
13. M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, March 1989.
14. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
15. 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.
16. 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`.
17. Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2):351–360, April 1979.
18. 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.
19. P. Downey. Undecidability of presburger arithmetic with a single monadic predicate letter. Technical Report 18-72, Center for Research in Computing Technology, Havard Univ., 1972.
20. Louis Comtet. *Advanced Combinatorics*. D. Reidel Publishing Company, Dordrecht, Holland, 1974.
21. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Mass., 1989.
22. Thomas Brandes. The importance of direct dependences for automatic parallelism. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 407–417, July 1988.
23. 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.
24. Carl Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, March 1990.
25. 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.
26. Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.
27. M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural analysis for parallelization. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, August 1995.
28. Saman P. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Computer Systems Laboratory, Stanford U., January 1997.
29. Sungdo Moon, Mary W. Hall, and Brian R. Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the 1998 International Conference on Supercomputing*, July 1998.
30. Sungdo Moon and Mary W. Hall. Evaluation of predicated array data-flow analysis for automatic parallelization. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 84–95, May 1999.
31. Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 313–322, July 1992.
32. 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.
33. 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.
34. 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.
35. Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.