UMIACS-TR-94-122 November, 1994
CS-TR-3371

Experiences with Constraint-based Array Dependence Analysis

William Pugh David Wonnacott
pugh@cs.umd.edu davew@cs.umd.edu
Institute for Advanced Computer Studies
Dept. of Computer Science Dept. of Computer Science
Univ. of Maryland, College Park, MD 20742

Array data dependence analysis provides important information for optimization of scientific programs. Array dependence testing can be viewed as constraint analysis, although traditionally general-purpose constraint manipulation algorithms have been thought to be too slow for dependence analysis. We have explored the use of exact constraint analysis, based on Fourier's method, for array data dependence analysis. We have found these techniques can be used without a great impact on total compile time. Furthermore, the use of general-purpose algorithms has allowed us to address problems beyond traditional dependence analysis. In this paper, we summarize some of the constraint manipulation techniques we use for dependence analysis, and discuss some of the reasons for our performance results.

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

Experiences with Constraint-based Array Dependence Analysis

William Pugh David Wonnacott
pugh@cs.umd.edu davew@cs.umd.edu

Department of Computer Science,
University of Maryland, College Park, MD 20742

Abstract. Array data dependence analysis provides important information for optimization of scientific programs. Array dependence testing can be viewed as constraint analysis, although traditionally general-purpose constraint manipulation algorithms have been thought to be too slow for dependence analysis. We have explored the use of exact constraint analysis, based on Fourier's method, for array data dependence analysis. We have found these techniques can be used without a great impact on total compile time. Furthermore, the use of general-purpose algorithms has allowed us to address problems beyond traditional dependence analysis. In this paper, we summarize some of the constraint manipulation techniques we use for dependence analysis, and discuss some of the reasons for our performance results.

1 Introduction

When two memory accesses refer to the same address, and at least one of those accesses is a write, we say there is a *data dependence* between the accesses. In this case, we must be careful not to reorder the execution of the accesses during optimization, if we are to preserve the semantics of the program being optimized. We therefore need accurate array data dependence information to determine the legality of many optimizations for programs that use arrays. Array dependence testing can be viewed as constraint analysis. For example, in Figure 1, determining whether or not any array element is both written by `A[i, j+1]` and read by `A[100, j]`, is equivalent to testing for the existence of solutions to the constraints shown on the right of the figure.

```
for i = 1 to n
  for j = i to n
    A[i, j+1] = A[n, j]
```

$1 \leq i_w \leq j_w \leq \mathrm{n}$ (write iteration in bounds)
$1 \leq i_r \leq j_r \leq \mathrm{n}$ (read iteration in bounds)
$i_w = \mathrm{n}$ (first subscripts equal)
$j_w + 1 = j_r$ (second subscripts equal)

Fig. 1. Dependence testing and associated constraints

Since integer programming is an NP-complete problem, ([GJ79]), production compilers employ techniques that are guaranteed to be fast but give conservative answers: they might report a possible solution when no solution exists. We have explored the use of exact constraint analysis methods for array data dependence analysis. We have gone beyond simply checking for satisfiability of conjunctions of constraints to being able to manipulate arbitrary Presburger formulas. This has allowed us to address problems beyond traditional dependence analysis.

In our previous papers [Pug92, PW93], we have presented timing results for our system on a variety of benchmark programs, and argued that our techniques are not prohibitively slow. In fact, using exact techniques to obtain standard kinds of dependence information requires about $1\% - 10\%$ of the total time required by simple workstation compilers that do no array data dependence analysis of any kind.

Our techniques are based on an extension of Fourier variable elimination to integers. Many other researchers in the constraints field [Duf74, LL92, Imb93, JMSY93] have stated that direct application of Fourier's technique may be impractical because of the number of redundant constraints generated. We have not experienced any significant problems with Fourier elimination generating redundant constraints, even though we have not implemented methods suggested [Duf74, Imb93, JMSY93] to control this problem. We believe that our extension of Fourier elimination to integers is much more efficient that described by [Wil76].

In this paper, we summarize some of the constraint manipulation techniques we use for dependence analysis, and discuss some of the reasons for our performance results.

2 The Omega Test

The Omega test [Pug92] was originally developed to check if a set of linear constraints has an integer solution, and was initially used in array data dependence testing. Since then, its capabilities and uses have grown substantially. In this section, we describe the various capabilities of the Omega test.

The Omega test is based on an extension of Fourier variable elimination [DE73] to integer programming. Other researchers have suggested the use of Fourier variable elimination for dependence analysis [WT92, MHL91b] but only as a last resort after exact and fast, but incomplete, methods have failed to give decisive answers. We proved [Pug92] that in cases where the fast but incomplete methods of Lam et al. [MHL91b] apply, the Omega test is guaranteed to have low-order polynomial time complexity.

2.1 Eliminating an Existentially Quantified Variable

The basic operation of the Omega test is the elimination of an existentially quantified variable, also referred to as shadow-casting or projection. For example, given a set of constraints P over x, y and z that define, for example, a dodecahedron, the Omega test can compute the constraints on x and y that define the shadow of the dodecahedron. Mathematically, these constraints are equivalent to $\exists z$ s.t. P. But the Omega test is able to remove the existentially quantified variables, and report the answer just in terms of the free variables (x and y).

Over rational variables, projection of a convex region always gives a convex result. Unfortunately, the same does not apply for integer variables. For example, $\exists y$ s.t. $1 \leq y \leq 4 \wedge x = 2y$ has $x = 2$, $x = 4$, $x = 6$ and $x = 8$ as solutions. Sometimes, the result is even more complicated. For example, the solutions for x in:

$$\exists i, j \text{ s.t. } 1 \leq i \leq 8 \wedge 1 \leq j \leq 5 \wedge x = 6i + 9j - 7$$

are all numbers between 8 and 86 (inclusive) that have remainder 2 when divided by 3, except for 11 and 83.

In general, the Omega test produces an answer in disjunctive normal form: the union of a finite list of clauses. A clause may need to describe a non-convex region. There are two methods for describing these regions:

Stride format The Omega test can produce clauses that consist of affine constraints over the free variables and stride constraints. A stride constraint $c|e$ is interpreted as "c evenly divides e". In this form, the above solution could be represented as:

$$x = 8 \ \vee \ (\ 14 \leq x \leq 80 \wedge 3|(x+1)\ ) \ \vee \ x = 86$$

Projected format Alternatively, the Omega test can produce clauses that consist of a set of linear constraints over a set of auxiliary variables and an affine 1-1 mapping from those variables to the free variables. Using this format, the above solution could be represented as

$$x = 8 \ \vee \ (\exists \alpha \text{ s.t. } 5 \leq \alpha \leq 27 \wedge x = 3\alpha - 1) \ \vee \ x = 86$$

These two representations are equivalent and there are simple and efficient methods for converting between them.

Our Extension of Fourier Elimination to Integers If $\beta \leq bz$ and $az \leq \alpha$ (where a and b are positive integers), then $a\beta \leq abz \leq b\alpha$. If z is a real variable, $\exists z$ s.t. $a\beta \leq abz \leq b\alpha$ if and only if $a\beta \leq b\alpha$. Fourier variable elimination eliminates a variable z by combining together all pairs of upper and lower bounds on z and adding the resulting constraints to those constraints that do not involve z. This produces a set of constraints that has a solution if and only if there exists a real value of z that satisfies the original set of constraints.

In [Pug92] and Figure 2 we show how to compute the "dark shadow" of a set of constraints: a set of constraints that, if it has solutions, implies the existence of an integer z such that the original set of constraints is satisfied. Of course, not all solutions are contained in the dark shadow.

For example, consider the constraints:

$$\exists y \text{ s.t. } 0 \leq 3y - x \leq 7 \wedge 1 \leq x - 2y \leq 5$$

Using Fourier variable elimination, we find that $3 \leq x \leq 27$ if we allow y to take on non-integer values. The dark shadow of these constraints is $5 \leq x \leq 25$. In fact, this equation has solutions for $x = 3, 5 \leq x \leq 27$ and $x = 29$.

In [Pug92] and Figure 2 we give a method for generating an additional sets of constraints that would contain any solutions not contained in the dark shadow. These "splinters" still contain references to the eliminated variable, but also contain an equality constraint (i.e., are flat). This equality constraint allows us to eliminate the desired variable exactly. For the example given previously, the splinters are:

$$\exists y \text{ s.t. } x = 3y \wedge 0 \leq 3y - x \leq 7 \wedge 1 \leq x - 2y \leq 5$$

$$\exists y \text{ s.t. } x + 1 = 3y \wedge 0 \leq 3y - x \leq 7 \wedge 1 \leq x - 2y \leq 5$$

$$\exists y \text{ s.t. } x - 5 = 2y \wedge y \text{ s.t. } 0 \leq 3y - x \leq 7 \wedge 1 \leq x - 2y \leq 5$$

Simplifying these produces clauses in projected form:

$$\exists y \text{ s.t. } x = 3y \wedge 1 \leq y \leq 5$$

$$\exists y \text{ s.t. } x = 3y - 1 \wedge 2 \leq y \leq 6$$

$$\exists y \text{ s.t. } x = 2y + 5 \wedge 5 \leq y \leq 12$$

```
**Eliminate $z$ from $C$, the conjunction of a set of inequalities**
$R$ = False
$C'$ = all constraints from $C$ that do not involve $z$
$C'' = C$
for each lower bound on $z$: $\beta \leq bz$
    for each upper bound on $z$: $az \leq \alpha$
        $C' = C' \wedge a\beta + (a-1)(b-1) \leq b\alpha$
        % Misses $a\beta \leq abz \leq b\alpha < a\beta + (a-1)(b-1)$
        % Misses $\beta \leq bz < \beta + \frac{(a-1)(b-1)}{a}$
    let $a_{\max}$ = max coefficient of $z$ in upper bound on $z$
    for $i = 0$ to $((a_{\max} - 1)(b-1) - 1)/a_{\max}$ do
        $R = R \vee C \wedge \beta + i = bz$
% $C'$ is the dark shadow
% $R$ contains the splinters
% $C' \vee (\exists$ integer $z$ s.t. $R) \equiv \exists$ integer $z$ s.t. $C$
```

Fig. 2. Extension of Fourier variable elimination to integers

2.2 Verifying the Existence of Solutions

The Omega test also provides direct support for checking if integer solutions exist to a set of linear constraints. It does this by treating all the variables as existentially quantified and eliminating variables until it produces a problem containing a single variable; such problems are easy to check for integer solutions. The Omega test incorporates several extensions over a naive application of variable elimination.

2.3 Removing Redundant Constraints

In the normal operation of the Omega test, we eliminate any constraint that is made redundant by any other single constraint (e.g., $x + y \leq 10$ is made redundant by $x + y \leq 5$). Upon request, we can use more aggressive techniques to eliminate redundant constraints. We use fast but incomplete tests that can flag a constraint as definitely redundant or definitely not redundant, and a backup complete test. This capability is used when verifying implications and simplifying formulas involving negation.

We also use these techniques to define a "gist" operator: informally, we say (gist P given Q) is what is "interesting" about P, given that we already know Q. More formally, we guarantee that ((gist P given Q) $\wedge Q) \equiv P \wedge Q$ and try to make the set of constraints produced by the gist operator as simple as possible.

2.4 Simplifying Formulas Involving Negation

There are two problems involved in simplifying formulas containing negated conjuncts, such as

$$-10 \leq i + j, i - j \leq 10 \ \wedge \ \neg(2 \leq i, j \leq 8 \wedge 2|i + j)$$

Naively converting such formulas to disjunctive normal form generally leads to an explosive growth in the size of the formula. In the worst-case, this cannot be prevented. But we [PW93] have described methods that are effective in dealing with these problems for the cases we encounter. One key idea to to recognize that we can transform $A \wedge \neg B$ to $A \wedge \neg$(gist B given A). Given several negated clauses, we simplify them all this way before choose one to negate and distribute.

Secondly, previous techniques for negating non-convex constraints, based on quasi-linear constraints [AI91], were discovered to be incomplete in certain pathological cases [PW93]. We [PW93] describe a method that is exact and complete for all cases.

2.5 Simplifying Arbitrary Presburger Formulas

Utilizing the capabilities described above, we can simplify and/or verify arbitrary Presburger formulas. In general, this may be prohibitively expensive. There is a known lower bound of $2^{2^{o(n)}}$ on the worst case nondeterministic time complexity, and a known upper bound of $2^{2^{2^{O(n)}}}$ on the deterministic time complexity, of Presburger formula verification. However, we have found that we are able to efficiently analyze many Presburger formulas that arise in practice.

For example, our current implementation requires 12 milliseconds on a Sun Sparc IPX to simplify

$$\begin{gathered} 1 \le \mathrm{i} \le 2\mathtt{n} \ \wedge\ 1 \le \mathrm{i}'' \le 2\mathtt{n} \wedge \mathrm{i} = \mathrm{i}'' \\ \wedge \neg(\ \exists \mathrm{i}', \mathrm{j}' \text{ s.t. } 1 \le \mathrm{i}' \le 2\mathtt{n} \wedge 1 \le \mathrm{j}' \le \mathtt{n} - 1 \wedge \mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' = \mathrm{i}'' \wedge 2\mathrm{j}' = \mathrm{i}''\ ) \\ \wedge \neg(\ \exists \mathrm{i}', \mathrm{j}' \text{ s.t. } 1 \le \mathrm{i}' \le 2\mathtt{n} \wedge 1 \le \mathrm{j}' \le \mathtt{n} - 1 \wedge \mathrm{i} \le \mathrm{i}' \wedge \mathrm{i}' = \mathrm{i}'' \wedge 2\mathrm{j}' + 1 = \mathrm{i}''\ ) \end{gathered}$$

to

$$(1 = \mathrm{i} = \mathrm{i}'' \le \mathtt{n}) \vee (1 \le \mathrm{i} = \mathrm{i}'' = 2\mathtt{n}) \vee (1 \le \mathrm{i} = \mathrm{i}'' \le 2 \wedge \mathtt{n} = 1)$$

Related Work

Other researchers have proposed extensions to Fourier variable elimination as a decision method for array data dependence analysis [MHL91a, WT92, IJT91]. Lam et al. [MHL91a] extend Fourier variable elimination to integers by computing a sample solution, using branch and bound techniques if needed. Michael Wolfe and Chau-Wen Tseng [WT92] discuss how to recognize when Fourier variable elimination may produce a conservative result, but do not give a method to verify the existence of integer solutions. These methods are decision tests and cannot return symbolic answers.

Corinne Ancourt and François Irigoin [AI91] describe the use of Fourier variable elimination for quantified variable elimination. They use this to generate loop bounds that scan convex polyhedra. They extend Fourier variable elimination to integers by introducing floor and ceiling operators. Although this makes their elimination exact, it may not be possible to eliminate additional variables from a set of constraints involving floor and ceiling operators. This limits their ability to check for the existence of integer solutions and remove redundant constraints.

Cooper [Coo72] describes a complete algorithm for verifying and/or simplifying Presburger formulas. His method for quantified variable elimination always introduces disjunctions, even if the result is convex. We have not yet performed a head-to-head comparison of the Omega test with Cooper's algorithm. However, we believe that the Omega test will prove better for quantified variable elimination when the result is convex and better for verification of a formula already in disjunctive normal form. Cooper's algorithm does not require formulas to be transformed into disjunctive normal form and may be better for formulas that would be expensive to put into disjunctive normal form (although our methods for handling negation address this as well).

The SUP-INF method [Ble75, Sho77] is a semi-decision procedure. It sometimes detects solutions when only real solutions exist and it cannot be used for symbolic quantified variable elimination.

H.P. Williams [Wil76] describes an extension of Fourier elimination to integers. His scheme leads to a much more explosive growth than our scheme. If the only constraints involving an eliminated variable x are $L \leq lx$ and $ux \leq U$, his scheme produces $\mathrm{lcm}(l,u)$ clauses, while ours produces

$$1 + \left\lceil \frac{(l-1)(u-1)}{\max(l,u)} \right\rceil$$

clauses. If there are p lower bounds $L_i \leq l_i x$ and q upper bounds $u_j x \leq U_j$, Williams' method produces a formula that, when converted into disjunctive normal form, contains

$$\prod_{1 \leq i \leq p \wedge 1 \leq j \leq q} \mathrm{lcm}(l_i, u_j)$$

clauses, while the number of clauses produced by our scheme is

$$1 + \min\left(\sum_{1 \leq i \leq p} \left\lceil \frac{(l_i - 1)(\max(u_j) - 1)}{\max(u_j)} \right\rceil, \sum_{1 \leq j \leq q} \left\lceil \frac{(\max(l_i) - 1)(u_j - 1)}{\max(l_i)} \right\rceil \right)$$

For example, if the l_i's are $\{1,1,1,2,3,5\}$ and the u_j's are $\{1,1,3,7\}$, Williams' method produces

$$23156852670000$$

clauses, while ours produces 12. It is almost certainly possible to improve Williams' method while using the same approach as Williams, but we know of no description of such an improvement.

Jean-Louis Lassez [LHM89, LL92, HLL92] gives an alternative to Fourier variable elimination for elimination of existentially quantified variables. However, his methods work over real variables, are optimized for dense constraints (constraints with few zero coefficients) and are inefficient when the final problem contains more than a few variables since they build a convex hull in the space of variables remaining after all quantified variables have been eliminated.

3 Constraint Based Dependence Analysis

Array dependence testing can be viewed as constraint analysis. Simply testing for the existence of a dependence (as in Figure 1) is equivalent to testing for solutions to a set of constraints.

We can also use constraint manipulation to obtain information about the possible differences in the values of the corresponding index variables at the times of the two accesses (this information can be used to test for the legality of some program transformations). To do so, we introduce variables corresponding to these differences, and existentially quantify and eliminate all other variables. Alternatively, we can choose to eliminate everything but the symbolic constants, and thus determine the conditions under which the dependence exists ([PW92]).

Figure 3 shows a relatively complicated example of constraint-based dependence analysis, from one of the NASA NAS benchmarks. Note that our techniques for eliminating equalities let us reduce both the number of variables and the number of constraints before resorting to Fourier elimination.

Program to be analyzed:

```
for j = 0 to 20 do
  for i = max(-j,-10) to 0 do
    for k = max(-j,-10)-i to -1 do
      for l = 0 to 5 do
        a(l,i,j) = ...a(l,k,i+j)...
```

Constraints before equality substitution:

$\exists j_w, i_w, k_w, l_w, j_r, i_r, k_r, l_r$ s.t.

$$\Delta i = i_r - i_w \wedge \Delta j = j_r - j_w$$
$$\Delta k = k_r - k_w \wedge \Delta l = l_r - l_w$$

$$l_w = l_r \wedge i_w = k_r \wedge j_w = j_r + i_r$$

$$\begin{gathered} 0 \le j_w \le 20 \\ -10, -j_w \le i_w \le 0 \\ -j_w - i_w, -10 - i_w \le k_w \le -1 \\ 0 \le l_w \le 5 \end{gathered}$$

$$\begin{gathered} 0 \le j_r \le 20 \\ -10, -j_r \le i_r \le 0 \\ -j_r - i_r, -10 - i_r \le k_r \le -1 \\ 0 \le l_r \le 5 \end{gathered}$$

Constraints after equality substitution:

$$\begin{gathered} \exists j_r, l_w \text{ s.t.} \\ 0 \le l_w \le 5 \\ 0 \le j_r \le 20 \\ 3\Delta j + 2\Delta i + \Delta k \le j_r \\ \Delta j \le j_r \le 20 + \Delta j \\ 2\Delta j + \Delta i \le j_r \\ 2\Delta j + 2\Delta i + \Delta k \le 10 \\ 1 \le \Delta j + \Delta i + \Delta k \\ 1 \le \Delta j + \Delta i \le 10 \\ 0 \le \Delta j \le 10 \\ 2\Delta j + \Delta i \le 10 \\ \Delta l := 0 \end{gathered}$$

Constraints after eliminating l_w and j_r:

$$\begin{gathered} 2\Delta j + \Delta i \le 10 \\ 0 \le \Delta j \le 10 \\ 3\Delta j + 2\Delta i + \Delta k \le 20 \\ 2\Delta j + 2\Delta i + \Delta k \le 10 \\ 1 \le \Delta j + \Delta i + \Delta k \\ 1 \le \Delta j + \Delta i \le 10 \\ \Delta l := 0 \end{gathered}$$

Fig. 3. Constraint-based dependence analysis

If we extend our constraint manipulation system to handle negated conjunctions of linear constraints, we can include constraints that rule out the dependences that are "killed" by other writes to the array, producing array data flow information ([PW93]). The analysis tells us the source of the value read at any particular point; standard array data dependence tests just tell us who had previously written to the memory location read at any particular point. We have also found that our use of constraints to represent dependences is useful for other forms of program analysis and transformation ([Pug91, PW94, KP93]).

4 Experiences

One of the main drawbacks of Fourier's method of variable elimination is the huge number of constraints that can be generated by repeated elimination, many of which could be redundant. Other researchers have found that Fourier's technique may be prohibitively expensive [HLL92, Imb93] for some sets of constraints, and have proposed either alternative methods for projection [HLL92] or methods to avoid generating so many redundant constraints [Imb93].

We have found Fourier's method to be efficient, and do not experience substantial increases in the number of constraints. Our empirical studies have shown that Fourier's method can be used in dependence analysis without a significant impact on total compile time [Pug92, PW93]. The average time required for memory-based analysis (as in Figure 1) was well under 1 millisecond per pair of references, and the average time for array data flow analysis a few milliseconds. These time trials were measured on a set of benchmarks that includes some of the NASA NAS kernels and some code from the Perfect Club Benchmarks ([B$^+$89]).

We believe this speed is the result of several attributes of the sets of constraints we produce for dependence analysis. First, loop bounds and array subscripts are often either constant or a function of a single variable. If all loop bounds and array subscripts have this form, all of our constraints will involve only one or two variables. Variable elimination is much less expensive within this restricted domain (known as LI(2)), even if we use the general algorithm. The number of constraints generated is bounded by a sub-exponential (though more than polynomial) function, rather than the $2^{n/2}$ of the general case [Cha93, Nel78].

Second, our constraints contain many unit coefficients. When the non-zero coefficients in a sparse set of constraints are all ± 1, projection ends up producing many parallel constraints, which can then be eliminated by our simple test for redundant constraints. Variable elimination in a LI(2) problem with unit coefficients preserves unit coefficients (after dividing through by the GCD of the coefficients). Under such situations, there cannot be more than $O(n^2)$ non-parallel constraints over n variables, and our method needs no more then $O(n^3)$ time to eliminate as many variables as desired [Pug92].

Finally, our constraint sets contain numerous equality constraints. Since we use these constraints to eliminate variables without resorting to projection, they help to keep down the size of the constraint sets that we must manipulate with Fourier's technique.

4.1 Empirical Studies of Dependence Analysis Constraints

We instrumented our system to analyze the types of constraints we deal with during dependence analysis. For each application of the Omega test, we analyzed the constraints that remained (a) after our initial removal of equality constraints and (b) after we had either eliminated all but two variables or run out of quantified variables to eliminate. In doing this analysis, we computed real shadows, as opposed to integer shadows (because the integer shadow may not be a simple conjunct). However, we still performed a number of other operations to rule out non-integer solutions (such as normalizing $2x + 4y \geq 3$ to $x + 2y \geq 2$).

When analyzing a set of constraints, we counted the number of variables, and counted (separately) the number of constraints that involved 1, 2 or 3+ variables. We then eliminated *all* redundant constraints, and recounted.

We performed these tests over our dataflow benchmark set [PW93], which includes some of the NASA NAS kernels and some code from the Perfect Club

Averages	when	# vars	kind	# of constraints involving 1 var	2 vars	3+ vars	total
	initial	5.6	as given	2.9	3.3	1.4	7.6
			nonredundant	2.0	2.1	0.9	5.0
	final	2.4	as generated	1.8	0.5	0.1	2.4
			nonredundant	1.2	0.3	0.07	1.6

a worst-case (but noncontrived)	when	# vars	kind	# of constraints involving 1 var	2 vars	3+ vars	total
example	initial	5	as given	6	5	4	15
encountered			nonredundant	4	2	3	9
in benchmarks	final	3	as generated	2	3	3	8
			nonredundant	1	2	2	5

Fig. 4. Characteristics of constraint sets used in dependence analysis

Benchmarks ([B$^+$89]). In total, we considered 1144 sets of constraints, and obtained the results shown in Figure 4.

Note that our methods always check for parallel constraints and eliminate the redundant one immediately (e.g., given $x+y \leq 5$ and $x+y \leq 10$, the second is eliminated). This can be done in constant time per constraint (through the use of a hash table).

Quite surprisingly, in *none* of the 1144 cases did the number of constraints increase as variables were eliminated (even though we did no elimination of non-parallel redundant constraints).

4.2 Empirical Studies of Random Constraints

To better understand the reasons for our good fortune in avoiding an explosion of constraints, we also studied the behavior of Fourier elimination on sets of random constraints. Figure 5 shows the results of these studies.

In each experiment, we fixed the number of constraints and variables, added one random non-zero to each constraint. When then projected the constraints onto the first two variables, and recorded the maximum number of constraints encountered during the elimination. We then added an additional nonzero coefficient to the original set of constraints, and repeated the projection. We continued doing this until the problem had no non-zeros left. Each line represents the median of 5-21 experiments. The key gives the elimination method used. All experiments shown here had 15 constraints on 5 variables, like the worst-case example from Figure 4.

The top graph compares the effectiveness of Fourier's method and the techniques described by Imbert on sets of constraints in which the non-zero coefficients had random integer values between -10 and +10. Our implementation of Imbert's method [Imb93] of redundant constraint detection uses Theorem 10 of

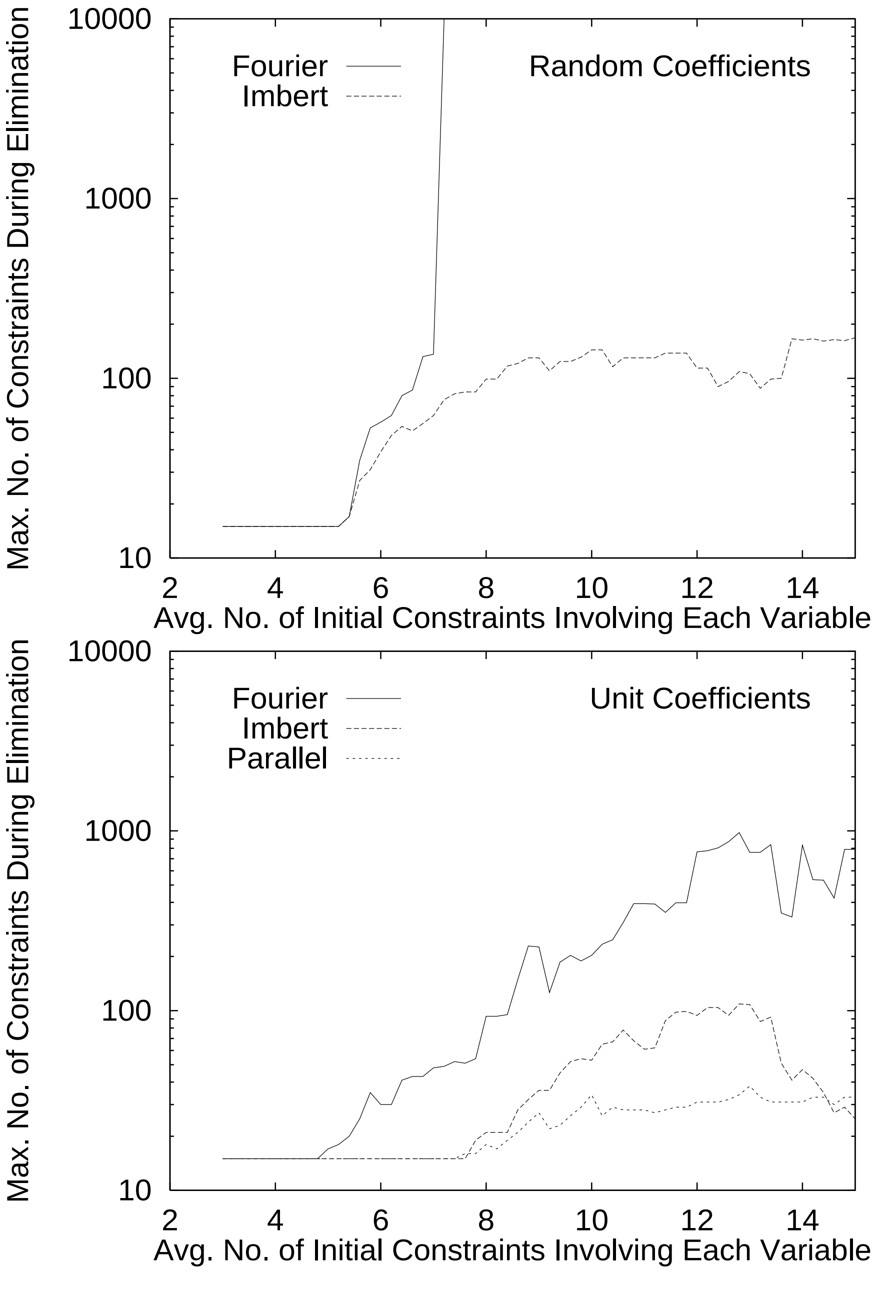

Fig. 5. Variable Elimination: 15 Constraints on 5 Variables

[Imb93] to determine that some constraints are redundant. However, we do not use the more expensive comparison or matrical tests.

Imbert's method is clearly important for problems of this size when the initial number of constraints per variable is above 7. When the initial density is below 5, even Fourier's original technique does not result in an increase in the number of constraints. However, our "worst-case" problem had an average of just under 6 initial constraints per variable, and we saw no increase in the number of constraints. Clearly the sparsity and size of the constraint sets were not sufficient to explain our results.

We therefore re-ran the tests on sets of constraints in which all the non-zero coefficients were ± 1, and included our techniques for detecting parallel redundant constraints. The results of this second set of tests are shown in the bottom graph of Figure 5. Note that the both our techniques and Imbert's do not produce an increase in the number of constraints when the initial number of constraints per variable is below 7. We therefore attribute our observations in Section 4.1 to a combination of constraint set size and sparsity and the high frequency of unit coefficients.

5 Conclusions

Other researchers [HLL92, Imb93] have been quite leary of applying Fourier variable elimination to sets of dense constraints. Our experience has lead us to believe that Fourier's method is quite efficient when applied to sparse constraints. Furthermore, we believe that sparse constraints arise in many applications.

We have extended our work beyond Fourier variable elimination: first to handling variable elimination for integer variables, and then to simplifying arbitrary Presburger formulas. We hope these extensions may be of interest to a broader community.

6 Availability

Technical reports about the Omega test and an implementation of the Omega test are available via anonymous ftp from `ftp.cs.umd.edu:pub/omega` or the world wide web `http://www.cs.umd.edu/projects/omega`.

References

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[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.

[Ble75] W. W. Bledsoe. A new method for proving certain presburger formulas. In *Advance Papers, 4th Int. Joint Conference on Artif. Intell.*, Tibilisi, Georgia, U.S.S.R, 1975.

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

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

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[Duf74] R. J. Duffin. On fourier's analysis of linear inequality systems. *Mathematical Programming Study*, pages 71–95, 1974.

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

[HLL92] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Practical issues on the projection of polyhedral sets. *Annals of mathematics and artificial intelligence*, November 1992.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proc. of the 1991 International Conference on Supercomputing*, pages 244–253, June 1991.

[Imb93] Jean-Louis Imbert. Fourier's elimination: Which to choose? In *PCPP 93*, 1993.

[JMSY93] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Projecting CLP(R) constraints. *New Generation Computing*, 11(3/4):449–469, 1993.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[LHM89] Jean-Louis Lassez, Tien Huynh, and Ken McAloon. Simplification and elimination of redundant linear arithmetic constraints. In *Proceedings of the North American Conference on Logic Programming*, pages 37–51, 1989.

[LL92] Catherine Lassez and Jean-Louis Lassez. Quantifier elimination for conjunctions of linear constraints via a convex hull algorithm. In Bruce Donald, Deepak Kapur, and Joseph Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*. Academic Press, 1992.

[MHL91a] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1991.

[MHL91b] 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 $o(n^{log n})$ algorithm for the two-variable-per-constraint linear programming satisfiablility problem. Technical Report AIM-319, Stanford University, Department of Computer Science, 1978.

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

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

[PW92] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

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

[Sho77] Robert E. Shostak. On the sup-inf method for proving presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[Wil76] H.P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.

[WT92] M. J. Wolfe and C. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.