Efficient and Exact Data Dependence Analysis

Dror E. Maydan, John L. Hennessy and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Data dependence testing is the basic step in detecting loop level parallelism in numerical programs. The problem is equivalent to integer linear programming and thus in general cannot be solved efficiently. Current methods in use employ inexact methods that sacrifice potential parallelism in order to improve compiler efficiency.

This paper shows that in practice, data dependence can be computed *exactly* and *efficiently*. There are three major ideas that lead to this result. First, we have developed and assembled a small set of efficient algorithms, each one exact for special case inputs. Combined with a moderately expensive backup test, they are exact for all the cases we have seen in practice. Second, we introduce a memoization technique to save results of previous tests, thus avoiding calling the data dependence routines multiple times on the same input. Third, we show that this approach can both be extended to compute distance and direction vectors and to use unknown symbolic terms without any loss of accuracy or efficiency. We have implemented our algorithm in the SUIF system, a general purpose compiler system developed at Stanford. We ran the algorithm on the PERFECT Club Benchmarks and our data dependence analyzer gave an exact solution in all cases efficiently.

1 Introduction

Work on parallelizing programs written in Fortran-like languages has concentrated on parallelizing loops in the source code. To do this, one must be able to analyze array reference patterns. For example:

```
for i = 1 to 10 do
   a[i] = a[i+10]+3
end for
for i = 1 to 10 do
   a[i+1] = a[i]+3
end for
```

In the first loop all the iterations can be executed concurrently since the locations being written do not overlap those being read. In the second case, however, each read refers to the value written in the previous iteration, forcing sequential execution. Data dependence testing answers the question of whether two array references can refer to the same location across iterations. Since there are no integers i, i' such that $1 \leq i, i' \leq 10$ and $i = i' + 10$, we say that the two references in the first example are independent. Since there are integers i, i' such that $1 \leq i, i' \leq 10$ and $i + 1 = i'$, the two references in the second example are dependent.

Ideally, one would always like to know definitively whether or not the references are dependent. Assuming independence when the references are in fact dependent can lead to incorrect results. Assuming dependence can inhibit parallelism. In the worst case, though, the price of exactness is an unacceptably high computation cost for the compiler. As we will show, data dependence testing is equivalent to integer programming, and the most efficient integer programming algorithms known either depend on the value of the loop bounds or are order $O(n^{O(n)})$ where n is the number of loop variables [8][9][13]. Even for the small cases encountered in practice, these algorithms are too expensive.

Many algorithms have been proposed for this problem, each one selecting different tradeoffs between accuracy and efficiency. Traditional algorithms attempt to prove independence, assuming dependence if they fail [2][4][18][19]. If such an algorithm returns dependent, we do not know if an approximation was made. Some work has been done on algorithms which are guaranteed to be exact for special case inputs. Simple loop residue [15] and Li and Yew's work [10] fall into this category. Little work has been done to analyze either the accuracy

This research was supported in part by a fellowship from AT&T Bell Laboratories and by DARPA contract N00014-87-K-0828.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-428-7/91/0005/0001...$1.50

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

or the efficiency of these algorithms in practice. None of these algorithms, as far as we know, have been definitively shown to be both "accurate enough" and "efficient enough". In fact, Shen, Li and Yew found that cases such as coupled subscripts appear frequently and cannot be analyzed accurately using traditional algorithms [14].

Our approach is to use a series of special case exact tests. If the input is not of the appropriate form for an algorithm, then we try the next one. Using a series of tests allows us to be exact for a wider range of inputs. We evaluated our algorithms on the PERFECT Club [7], a set of 13 scientific benchmarks, and found the algorithms to be exact in every case.

Cascading exact tests can also be much more efficient than cascading inexact ones. By attempting our most applicable and least expensive test first, in most cases we can return a definitive answer using just one exact test, even the dependent ones. Even in the other cases, we only need to check the applicability of multiple tests. We never have to apply more than one. In contrast, cascading inexact algorithms would require using all the tests on at least all the truly dependent cases. As we will show, most cases encountered in practice are in fact dependent. In fact, most direction vectors tested are dependent as well.

Next we turn towards increasing the efficiency of our approach. Our basic tests range from inexpensive to moderately expensive. It has been said that dependence testing is performing a large number of tests on relatively small inputs [11]. We show that in actuality it is performing a small number of unique tests on small inputs repeatedly. There is little variation in array reference patterns found in real programs, and most bounds tend to go from 1 to n where n is the same throughout the program. Thus, one can save much computation by using memoization,[1] remembering the results of previous tests. We introduce a hashing scheme to do this efficiently.

We extend our approach to distance and direction vectors. We first use the standard hierarchical approach based on Burke and Cytron's work [5]. While our algorithms remain exact, we find that this approach leads to a large increase in computational cost. Simple pruning methods bring these costs back down.

Finally, we extend our approach to handle symbolic terms. This preserves the exactness property of the algorithms with very little increase in cost.

2 Problem Definition

We first give the standard definition for the general problem of data dependence testing. In the general case we allow multi-dimensional arrays and nested trapezoidal loops. We restrict ourselves to cases where loop bounds are integral linear functions of more outwardly nested loop variables and where all array references are integral linear functions of the loop variables. These condition do not necessarily have to be met in the source program. We use optimization techniques (constant propagation, induction variable and forward substitution [16]) to increase the applicability of these conditions.

Given the following general normalized (we normalize the step size to 1) loop:

for $i_1 = L_1$ **to** U_1 **do**
  for $i_2 = L_2(i_1)$ **to** $U_2(i_1)$ **do**
    ...
    for $i_n = L_n(i_1,\ldots,i_{n-1})$ **to** $U_n(i_1,\ldots,i_{n-1})$ **do**
      a$[f_1(\vec{i})][f_2(\vec{i})]\ldots[f_m(\vec{i})] = \ldots$
      $\ldots$=a$[f'_1(\vec{i})][f'_2(\vec{i})]\ldots[f'_m(\vec{i})]$
    end for
  end for
end for

such that all the L, U, f, f' are known linear functions. The two references are dependent iff

$$\begin{aligned}&\exists \text{ integer } i_1,\ldots,i_n,i'_1,\ldots,i'_n \text{ such that}\\ &\quad f_1(\vec{i}) = f'_1(\vec{i'}),\ \ldots,\ f_m(\vec{i}) = f'_m(\vec{i'})\\ &\quad L_1 \le i_1, i'_1 \le U_1\\ &\quad \ldots\\ &\quad L_n(i_1,\ldots,i_{n-1}) \le i_n, i'_n \le U_n(i_1,\ldots,i_{n-1})\end{aligned}$$

In matrix form this is equivalent to

$$\begin{aligned}&\exists \text{ integral } \vec{x}\\ &\quad \text{such that } A_1\vec{x} = \vec{b_1}, A_2\vec{x} \le \vec{b_2}\end{aligned} \tag{1}$$

where integer matrices A_1 and A_2 and vectors $\vec{b_1}$ and $\vec{b_2}$ are given. By replacing any equality $ax = b$ in (1) by the two inequalities $ax \le b$ and $-ax \le -b$ we see that this is equivalent to

$$\begin{aligned}&\exists \text{ integral } \vec{x}\\ &\quad \text{such that } A\vec{x} \le \vec{b}\end{aligned} \tag{2}$$

2.1 Equivalence of Data Dependence to Integer Programming

Ideally one would like the data dependence test to be exact. Unfortunately, data dependence in general is exactly equivalent to integer programming, a well studied problem. The standard integer programming problem is [13]

$$\text{find the max } \vec{x} \text{ such that } A\vec{x} < \vec{b},\ \vec{x} \text{ integral} \tag{3}$$

[1] Memoization is a technique to remember the results of previous computations, previously used to implement call-by-need arguments in LISP compilers.[1].

It is clear that (2) is a special case of (3) so data dependence can be reduced to integer programming. Another polynomially equivalent version of integer programming is [13]

$$\exists\ \vec{x} \text{ such that } A\vec{x} = \vec{b},\ \vec{x} \geq 0,\ \vec{x} \text{ integral} \qquad (4)$$

One can reduce (4) to data dependence testing by constructing the following program

for x_1 = 0 **to** *unknown* **do**
  ...
  for x_n = 0 **to** *unknown* **do**
    a[$A_{1,1}x_1 + \ldots A_{1,n}x_n$]...[$A_{m,1}, x_1 + \ldots$] = ...
    ... = a[$-A_{1,n+1}x_1 + \ldots - A_{1,2n}x_n$]...[...]
  end for
end for

All integer programming algorithms that we know of are too expensive in the worst case. IP is NP-Complete, any existing algorithm either depends exponentially on the number of variables and constraints or depends linearly on the size of the coefficients. Typical program have very few array dimensions and do not have deeply nested loops. Therefore being exponential in the number of variables and constraints could be acceptable. The size of the coefficients, though, could conceivably be very large.

The complexity of most common algorithms (branch and bound, cutting plane) depend on the size of the coefficients in the worst case. Lenstra [9] and Kannan [8] have developed algorithms that do not depend on the coefficients, but in the worst case, Kannan's algorithm is $O(n^{O(n)})$ where n is the number of variables. Unfortunately, this is much too expensive to use in general data dependence testing. Therefore, we do not believe it is possible to develop a practical test that will apply to every conceivable case. Nonetheless, as we next show, special case algorithms which are efficient, apply to all the examples we have found in our benchmarks.

3 Data Dependence Tests

In this section, we describe the individual tests used in our approach.

3.1 Extended GCD Test

We use Banerjee's Extended GCD test [4] as a preprocessing step for our other tests. While the test itself is not exact, it allows us to transform our problem into a simpler and smaller form, increasing the applicability of our other tests. This test solves the simpler question: Ignoring the bounds, is there an integral solution to the set of equations. From equation (1) we see that this is equivalent to: does there exist an integer vector $\vec{x}$ such that $A\vec{x} = b$. If this system of equations is independent then we know that the original system is also independent since the loop bounds merely introduce additional constraints. If it is dependent, the total system may be either independent or dependent. In this case, though, we are able to use the results of the extended GCD test to make a change of variables which simplifies the original problem. The original GCD test is derived from number theory. The single equation $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$ has an integer solution iff gcd(a_i) divides b. This gives us an exact test for single-dimensional arrays ignoring bounds. Banerjee shows how this can be extended to multi-dimensional arrays. The system of equations $\vec{x}A = \vec{c}$ can always be factored into a unimodular[2] integer matrix U and an echelon[3] matrix D (with $d_{11} > 0$) such that $UA = D$. The factoring is done with an extension to Gaussian elimination. Then, the system $\vec{x}A = \vec{c}$ has an integer solution $\vec{x}$ iff there exists an integer vector $\vec{t}$ such that $\vec{t}D = c$. Since D is an echelon matrix this back substitution can be done very simply. If no such $\vec{t}$ exists, then the total system is independent. Otherwise, ignoring the bounds, there is a dependence. We then add in the bounds and continue.

If such a $\vec{t}$ exists then the solution $\vec{x}$ is given by $\vec{x} = \vec{t}U$. If the system is not of full rank, and it usually is not, then $\vec{x}$ will have some degrees of freedom. For example: $\vec{t} = (1, t_1)$ $U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ then $\vec{x} = (1, t_1)$ where t_1 (and therefore x_2) can take on any integral value. Wolfe showed that the bounds constraints on $\vec{x}$ can be expressed as constraints on $\vec{t}$ [19]. For example:

for i = 1 **to** 10 **do**
  a[i+10] = a[i]
end for

The initial dependence problem is to find

integers i,i' such that
  $i + 10 = i'$ and
  $1 \leq i, i' \leq 10$

The extended GCD test tells us that $(i,\ i') = (t_1,\ t_1 + 10)$. Transforming the constraints to be in terms of t_1 gives us:

does there exist integer t_1 such that
  $1 \leq t_1 \leq 10$ and
  $1 \leq t_1 + 10 \leq 10$

[2] a unimodular matrix is a matrix whose determinant is ± 1.

[3] An echelon matrix is an $m * n$ matrix such that if the first non zero element in row i is in column j, then the first non zero element in row i+1 is in column k > j

This transformation is valuable for several reasons. First, we have reduced the number of variables. In general, each independent equation will eliminate one variable. Second, we have reduced the number of constraints. Before, each lower and upper loop bound generated one constraint each, while each dimension of the array generated one equality constraint. The equality constraint $a\vec{x} = b$ had to be converted into the two inequality constraints $a\vec{x} \leq b$ and $a\vec{x} >= b$. Therefore we had $2*l+2*d$ constraints (where l is the number of enclosing loops, and d is the number of array dimensions). Now all the equality constraints are folded into the bounds constraints. Thus we are left with only $2*l$ constraints.

The complexity of most integer programming algorithms depends on the number of constraints and the number of variables. Thus, in general GCD preprocessing should make the other algorithms more efficient, but more importantly, the form of the new constraints is typically simpler than the original. We have eliminated equality constraints, a necessity for our Acyclic Test discussed below, and we have cut down on the number of variables per constraint. In the previous example, some constraints (the equality ones) contained two variables, while after the transformation all constraints contain one variable. As we shall see, our first exact test, the Single Variable Per Constraint Test, only applies in cases where each constraint has at most one variable.

3.2 Single Variable Per Constraint Test

Banerjee shows that if the solution to the generalized GCD test has at most 1 free variable then one can solve the exact problem easily [4]. Each constraint is merely an upper or lower bound for the free variable. One merely goes through each constraint calculating the appropriate lower or upper bound and storing the best ones found. If after going through all the constraints, the lower bound is greater than the upper bound, then the test returns "independent". Otherwise the equations are dependent. Banerjee notes that this can be extended to the case where there are an arbitrary number of variables in the system but at most one variable per constraint. Each constraint is merely an upper or lower bound for one of the free variables. One goes through each constraint and remembers the tightest bound for each variable. If after going through all the constraints, $lb_i > ub_i$ for any variable t_i, the system is independent. Otherwise it is dependent. This test is a superset of the well known single loop, single dimension exact test [3]. It is quite clear that with one loop and single-dimensional arrays one cannot get constraints with more than one free variable. This test, though, also applies to many common multi-dimensional cases, including those with coupled subscripts, as shown below.

for i_1 = L_1 **to** U_1 **do**
  for i_2 = L_2 **to** U_2 **do**
    a[i_1][i_2] = a[$i_1 + const_1$][$i_2 + const_2$]
  end for
end for

for i_1 = L_1 **to** U_1 **do**
  for i_2 = L_2 **to** U_2 **do**
    a[i_1][i_2] = a[$i_2 + const_1$][$i_1 + const_2$]
  end for
end for

To demonstrate the algorithm, we cover the following example in detail.

for i_1 = 1 **to** 10 **do**
  for i_2 = 1 **to** 10 **do**
    a[i_1][i_2] = a[$i_2 + 10$][$i_1 + 9$]
  end for
end for

The GCD test will set $i_1 = t_1$, $i_1' = t_2$, $i_2 = t_2 + 9$ and $i_2' = t_1 - 10$. Expressing the constraints in terms of the t variables we get the following:

$1 \leq t_1 \leq 10$
$1 \leq t_2 \leq 10$
$1 \leq t_2 + 9 \leq 10$
$1 \leq t_1 - 10 \leq 10$

The first constraint sets the lower bound of t_1 to 1 and its upper bound to 10. The second does the same for t_2. The third constraint resets t_2's upper bound to 1. Finally, the last constraint resets t_1's lower bound to 11. Since the lower bound on t_1 is greater than its upper bound, the system is independent.

This algorithm is very efficient. It requires $O(num\_constraints + num\_vars)$ steps with very few operations per step. Even if certain constraints have more than one variable, applying this test to the applicable ones will eliminate constraints for the proceeding algorithms.

3.3 Acyclic Test

We have developed the Acyclic Test for cases where at least one constraint has more than one variable. We use the constraints involving more than one variable to create a directed graph. We add two nodes to the graph for each variable t_i; one labeled i and one labeled $-i$. We examine each pair of variables, t_i and t_j, occurring in a single constraint. We first express the constraint as $a_i t_i \leq \ldots + a_j t_j$. If both a_i and a_j are greater than zero, we add an edge to our graph from node i to node j. We then express the constraint as $-a_j t_j \leq \ldots - a_i t_i$ and add an edge from node $-j$ to node $-i$. If a_i is less than zero, we would use node $-i$ for the first edge and node i for

the second. Similarly, if a_j is less than zero, we would use node $-j$ for the first edge and node j for the second. Two nodes are needed for each variable to distinguish the case $t_i + t_j + \ldots \leq 0$ from the case $t_i - t_j + \ldots \leq 0$. Looking at an example, let us assume that we have one constraint $t_1 + 2t_2 - t_3 \leq 0$. We create a graph with six nodes. The constraint implies that $t_1 \leq -2t_2 + t_3$. We add an edge to the graph from node 1 to node -2 and another from node 1 to node 3. Similarly, the constraint can be expressed as $2t_2 \leq -t_1 + t_3$ so we add edges from node 2 to nodes -1 and 3. Finally, expressing the constraint in terms of t_3 we add edges from node -3 to nodes -1 and -2. If we had multiple constraints, we would repeat this step for each one.

If the resulting graph has no cycles, then we can solve the system exactly using a simple substitution method. If there is no cycle, there exists a node i (there is no loss of generality in assuming that $i > 0$) such that there are no edges entering node i (this node is a leaf in the depth-first search tree of the graph). From the method used to construct the graph, this implies that there are no constraints of the form $a_j t_j \leq a_i t_i + \ldots$ where $a_i \geq 0$. Thus all constraints involving t_i are of the following form:

$$
\begin{aligned}
&a_{i,1} t_i \leq f_1(t) \\
&a_{i,2} t_i \leq f_2(t) \\
&\quad \ldots \\
&a_{i,n} t_i \leq f_n(t)
\end{aligned}
$$

where $f(t)$ is a linear function involving any of the variables except t_i and where all $a_{i,k} > 0$, $1 \leq k \leq n$. Thus t_i is only constrained in one direction, to be smaller than some function of the other variables. Thus we can set t_i to L_i (the lower bound that the Single Variable Per Constraint Test have previously calculated for t_i). It is possible that $L_i = -\infty$ if there is no lower bound for t_i. If there is a solution with $t_i = L_i$ then that solution is a solution to the original system. If there is a solution to the original system with $t_i > L_i$ then clearly setting t_i to the lower value of L_i will not violate any constraints. Thus there is a solution with $t_i = L_i$ iff there is a solution to the original system.

If our initial node was $-i$ rather than i, then all constraints on t_i would be in the other direction, i.e. all $a_{i,k} < 0$, $1 \leq k \leq n$, and we would set t_i to U_i.

Once we set t_i, we continue on our depth-first search and set another variable. We continue until we reach a contradiction, a lower bound larger than an upper one, or until no variables are left. If we eliminate all the variables without finding a contradiction, then the system is dependent. Otherwise it is independent. Below we show an example of the algorithm.

$$
\begin{aligned}
&1 \leq t_1, t_2 \leq 10 \\
&0 \leq t_3 \leq 4 \\
&t_2 \leq t_1 \\
&t_1 \leq t_3 + 4
\end{aligned}
$$

t_1 is constrained in both directions, but we can set t_2 to $L_2 = 1$. This leaves us with

$$
\begin{aligned}
&1 \leq t_1 \leq 10 \\
&0 \leq t_3 \leq 4 \\
&1 \leq t_1 \\
&t_1 \leq t_3 + 4
\end{aligned}
$$

Now either t_1 or t_3 can be set. If we set t_1 to $L_1 = 1$ we are left with

$$
\begin{aligned}
&0 \leq t_3 \leq 4 \\
&t_3 \geq 1 - 4
\end{aligned}
$$

t_3 can be set to any value between 0 and 4. There are no contradictions so the system is dependent.

Even if there is a cycle in the graph, this algorithm can solve for all variables which are not in the cycle. This simplifies the system for the next stages.

It is not absolutely necessary to create the graph. One can instead simply search for variables which are only constrained in one direction and then set them. This approach has a higher complexity than creating a depth-first search tree, but it is easier to implement.

This algorithm requires GCD preprocessing to eliminate equality constraints. The constraint $i_1 = i_2$ would otherwise be represented as the two constraint $i_1 \leq i_2$ and $i_1 \geq i_2$. These two constraints alone create a cycle in the graph ($i_1 \leq i_2 \leq i_1$).

3.4 Simple Loop Residue Test

If there is a cycle in our graph, we attempt the Simple Loop Residue Test. Pratt developed a simple algorithm for data dependence testing which works when all constraints are of the form $t_i \leq t_j + c$ [12]. One creates a graph with a node for each variable. For this inequality, we place a directed arc with value c from node t_i to node t_j. Assume we have another constraint $t_j \leq t_k + d$. By transitivity, this implies that $t_i \leq t_k + c + d$. We define the value of a path in the graph to be equal to the sum of the values of the edges on the path. In the above example, the value of the path from node i to node k is $c + d$. So the value of the path constrains its endpoints in the same way that an edge does. Thus, if there is a path from node n_1 to n_2 with value v, we know that $n_1 \leq n_2 + v$. Constraints with only one variable are also acceptable. We create a special node, n_0. The constraint $t_i \leq c$ is represented with an edge from i to n_0 with value c. Similarly, the constraint $t_i \geq c$ is represented with an edge from n_0 to i with value $-c$. A cycle in the graph represents a constraint of the form $i - i \leq c$ or $0 \leq c$. We check every cycle in the graph. If any cycle has a negative value, the system is independent. Otherwise it is dependent.

Shostak [15] extends this algorithm first to deal with inequalities of the form $at_i \leq bt_j + c$ and then to handle cases with more than two variables. Unfortunately these extensions make the algorithm inexact. However, the algorithm can be extended to the case $at_i \leq at_j + c$ without losing exactness. This case is equivalent to $a(t_i - t_j) \leq c$. Let d be the largest integer such that $d \leq c$ and d is a multiple of a. We can replace the inequality with $t_i - t_j \leq d/a$ where d/a is an integer.

As an example of the Loop Residue Test, assume we have the following constraints:

$1 \leq t_1, t_2 \leq 10$
$0 \leq t_3 \leq 4$
$t_2 \leq t_1$
$2t_1 \leq 2t_3 - 7$

Figure 1 shows the graph after converting the last constraint to $t_1 \leq t_3 - 4$.

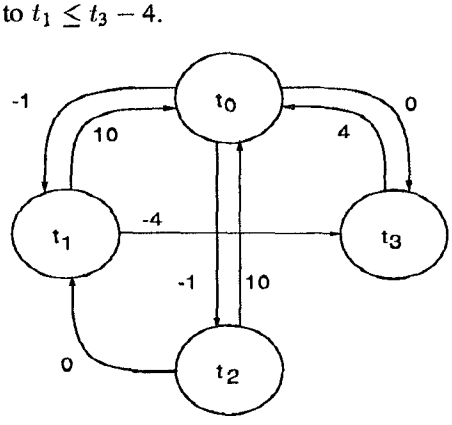

Figure 1: Example graph for Loop Residue Test

There is a cycle from t_1 to t_3 to t_0 to t_1 with value $-4 + 4 - 1 = -1$. Therefore the system is independent.

3.5 Fourier-Motzkin

Our last algorithm, Fourier-Motzkin [6], is a backup inexact test. It solves the general non-integer linear programming case exactly. If it returns independent, we know that the integer case is also independent. If it returns dependent, it also returns a sample solution. If this sample solution is integral, then the integral case is dependent. Otherwise we are not sure. We extend this algorithm to deal with a few cases where there is a real solution but no integral one. Even if there is a sample non-integer solution, we can sometimes prove independence. In the few cases where our first four algorithms did not apply, and we were required to call Fourier-Motzkin, the algorithm was always exact (i.e., it either returned independent or the sample solution was integral). The cost of this algorithm is a matter of debate. Theoretically it can be exponential. Experimentally, Triolet has implemented this approach and seems to be satisfied with its efficiency [17], but Li, Yew and Zhu consider Triolet's numbers to be too expensive [11]. Nonetheless, we are required to call this algorithm so few times that its accrued expense is very reasonable.

In the first step, we eliminate the first variable, x_1, from the set of constraints. All the constraints are first normalized so that their coefficient for x_1 is 0, 1 or -1. The set of constraints is then partitioned into three sets depending on the value of the coefficient.

$x_1 \geq D_1(\vec{x}_{2,\ldots,n}), \ldots, x_1 \geq D_p(\vec{x}_{2,\ldots,n})$
$x_1 \leq E_1(\vec{x}_{2,\ldots,n}), \ldots, x_1 \leq E_q(\vec{x}_{2,\ldots,n})$
$0 \leq F_1(\vec{x}_{2,\ldots,n}), \ldots, 0 \leq F_r(\vec{x}_{2,\ldots,n})$

This system has a solution iff

$\exists \vec{x}_{2,\ldots,n}$ such that
$D_i(\vec{x}_{2,\ldots,n}) \leq E_j(\vec{x}_{2,\ldots,n})$, $i = (1,\ldots,p)$, $j = (1,\ldots,q)$
$0 \leq F_k(\vec{x}_{2,\ldots,n})$, $k = (1,\ldots,r)$

A proof can be found in [6]. Thus, one can eliminate one variable at a time until there are none left. At each step, the number of constraints grows by $pq - p - q$. While this can lead to exponential behavior in the worst case, when p and q are small, each step might actually eliminate constraints. If there is a solution $\vec{x}_{2,\ldots,n}$ then the original system will be satisfied with any x_1 such that $\max(D(\vec{x}_{2,\ldots,n})) \leq x_1 \leq \min(E(\vec{x}_{2,\ldots,n}))$. Thus one can back substitute to find sample solutions for all the x variables. We would like the sample solution to be integral. As a heuristic, at each step of the back substitution, we set x_i to be the integer at the middle of the allowed range.

At each step i in the back substitution, we know that there exists a real solution for x_i such that $\max(D(\vec{x}_{i+1,\ldots,n})) \leq x_i \leq \min(E(\vec{x}_{i+1,\ldots,n}))$, where we have already substituted in values for $\vec{x}_{i+1,\ldots,n}$. Having no integer in the allowable range for x_i does not necessarily imply that there is no integer solution. If we had substituted a different sample value for some $x_k, i+1 \leq k \leq n$, we might have found an integral solution. One special case, though, does turn out to be useful. Suppose there is no allowable integer for x_n. Since we have not constrained ourselves by selecting a value for any other x_j, we can be assured that there is no integer solution.

In general, if the sample solution is not integral, one can use branch and bound techniques. Say for example that $x_i = 3.5$. Then one sets up two companion systems. One with the added constraint that $x_i \leq 3$ and another with the constraint $x_i \geq 4$. If neither system has a solution then the original system is independent. It is possible that after this step one is still left with a non-integral solution. One can then repeat the branch and bound step. Conceivably, one might be required to

branch and bound many times (proportional to the size of the region). In such a case, one might have to cut off the process after an arbitrary number of steps and assume dependence. We have not found any cases which require us to use explicit branch and bound. In four cases, we are required to use it implicitly (see Section 6).

4 Effectiveness of Data Dependence Tests

We have implemented these algorithms in the SUIF system [16], a general purpose compiler system developed at Stanford. We then ran them on the PERFECT Club benchmarks. These are a set of 13 scientific Fortran programs ranging in size from 500 to 18,000 lines collected at the University of Illinois at Urbana-Champaign [7]. We feel that these are a fair set of benchmarks. Non-scientific codes do not exhibit much loop level parallelism, and we feel that library routines tend to be simpler than full programs.

Table 1 shows how many times each dependence test was called per program. The first column represents array constants, for example a[3] versus a[4]. These cases are handled without dependence testing. We merely include them to show that their frequency can skew statistics if we apply general dependence routines to them. The second column represents the cases where GCD returns independent. For these cases, we do not need to call the exact routines. The other columns correspond to the number of successful applications of the tests. As one can see, the vast majority of cases are handled with the Single Variable Per Constraint Test.

5 Memoization To Improve Efficiency

We have shown that algorithms are exact in every instance of our benchmarks. Now we consider efficiency. Many references are very simple and thus tend to be repeated frequently. Bounds are frequently of the form $i = 1$ to n where n is either a constant or unknown. While n may vary across programs, in the same program n might always be the same. There is no need to call the dependence routines multiple times on the same data. By saving previous results, we can eliminate most calls. The use of a hash table allows us to find a duplicate call very quickly.

A simple memoization scheme does not repeat a test if the input exactly matches a previous one. We can improve the effectiveness of the algorithm by eliminating the loop bound constraints on unused loop indices. Both of the following programs

(a)
```
for i = 1 to 10 do
  for j = 1 to 10 do
    a[i+10] = a[i]+3
  end for
end for
```

(b)
```
for i = 1 to 10 do
  for j = 1 to 10 do
    a[j+10] = a[j]+3
  end for
end for
```

collapse to this one:

```
for i = 1 to 10 do
  a[i+10] = a[i]+3
end for
```

Two cases which before appeared to be different, now are identical.

In Table 2 we show the results of both our simple scheme and our improved one applied to the PERFECT Club. We use two hash tables; one using loop bounds and one not. The GCD test does not make use of bounds. Thus, if a particular reference matches ignoring the bounds, we are not required to repeat the GCD test.

We use a simple-minded open table hashing scheme. It performs well enough for our purposes. If necessary, we are certain that a more optimal one could be found. Treating the input data, array reference equations and loop bounds, as one long vector, $\vec{x}$, of integers, our hashing function is $h(\vec{x}) = size(\vec{x}) + \sum_i 2^i x_i$ This function was chosen so that symmetrical or partially symmetrical references would not collide. Because of the low number of unique cases, random collisions are not much of a problem.

In Table 3 we show how memoization improves the results of Table 1. The Total Cases column gives the total number of exact tests from Table 1. The remaining columns show how many of each exact test is left after memoization. Memoization reduces the total from 5,679 to 332 tests!

Further optimizations are possible. For example, one can eliminate symmetrical cases. Assume there are two references in a loop; r_1 and r_2. We wish to know if this case is equivalent to a pair of references in our table; r_1' and r_2'. Assume the bounds are the same. Our simple scheme would say the two cases are equivalent iff $r_1 = r_1'$ and $r_2 = r_2'$, but the cases are actually also equivalent if $r_1 = r_2'$ and $r_2 = r_1'$. For example comparing a[i] to a[i-1] is the same as comparing a[i-1] to a[i]. This can be taken farther. a[i][j] versus a[i+1][j+1] is equivalent to a[j][i] versus a[j+1][i+1].

One other possible improvement is to store the hash table across compilations. This will eliminate the data

Dependence Test Frequency							
Program	#Lines	Constant	GCD	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	229	91	613	0	0	0
CS	18,520	50	0	127	15	0	0
LG	2,327	6,961	0	73	0	0	0
LW	1,237	54	0	34	43	0	0
MT	3,785	49	0	326	0	0	0
NA	3,976	45	0	679	202	1	2
OC	2,739	2	7	36	0	0	0
SD	7,607	949	0	526	17	5	12
SM	2,759	1,004	98	264	0	0	0
SR	3,970	1,679	0	1,290	0	0	0
TF	2,020	801	6	826	0	0	0
TI	484	0	0	4	42	0	0
WS	3,884	36	182	378	4	0	160
TOTAL	59,412	11,859	384	5,176	323	6	174

Table 1: Number of times each test called for each program in the PERFECT Club

Percentage of Unique Cases						
Program	Without Bounds (GCD)			With Bounds		
	Total	Unique		Total	Unique	
		Simple	Improved		Simple	Improved
AP	704	7.0%	4.4%	613	6.4%	4.4%
CS	142	7.7%	7.0%	142	16.2%	14.1%
LG	73	32.9%	13.7%	73	47.9%	31.5%
LW	77	11.7%	10.4%	77	23.4%	22.1%
MT	326	3.4%	2.5%	326	6.4%	4.3%
NA	884	4.2%	3.4%	884	7.9%	6.9%
OC	43	27.9%	20.9%	36	19.4%	13.9%
SD	560	6.6%	6.1%	560	9.5%	8.8%
SM	362	5.5%	3.6%	264	4.9%	3.0%
SR	1,290	1.1%	0.9%	1,290	1.6%	1.1%
TF	832	2.2%	1.7%	826	2.9%	2.4%
TI	46	30.4%	19.6%	46	34.8%	23.9%
WS	724	11.9%	11.0%	542	14.2%	11.6%
TOT	6,063	5.7%	4.4%	5,679	7.3%	5.8%

Table 2: Percentage of unique cases for memoization scheme with simple scheme and with unused variables eliminated

Dependence Test Frequency For Unique Cases						
Program	#Lines	Total Cases	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	613	27	0	0	0
CS	18,520	142	14	6	0	0
LG	2,327	73	23	0	0	0
LW	1,237	77	15	2	0	0
MT	3,785	326	14	0	0	0
NA	3,976	884	48	11	1	1
OC	2,739	36	5	0	0	0
SD	7,607	560	36	6	3	4
SM	2,759	264	8	0	0	0
SR	3,970	1,290	14	0	0	0
TF	2,020	826	20	0	0	0
TI	484	46	3	8	0	0
WS	3,884	542	35	1	0	27
TOTAL	59,412	5,679	262	34	4	32

Table 3: Number of times each test was called looking only at unique cases

dependence cost of incremental compilation. In addition, if there is similarity across programs, one could use a set of benchmarks to set up a standard table which would be used by all programs.

6 Direction and Distance Vectors

Typically, we are not just interested in knowing if two references are independent [19]. In case of dependence, we frequently also want to know the relationship of the dependence. For example:

```
for i = 1 to 10 do
  a[i+1] = a[i]+7
end for
for i = 1 to 10 do
  a[i] = a[i]+7
end for
```

In both cases, the two references are dependent, but while the second loop can run in parallel, the first cannot. The amount of information to exactly describe the dependence is large. We need to enumerate every pair of iterations (i, i') for which there is a dependence. In the first case, there is a dependence when (i, i') is $(1,2),(2,3),\ldots,(10,11)$. In the second case, there is a dependence when (i, i') is $(1,1),(2,2),\ldots,(10,10)$. Differentiating the two examples does not require us to use all the information. The difference between the two examples is that in the first case $i < i'$ while in the second $i = i'$. Direction vectors are a commonly used technique to summarize this relationship between the loop variables $\vec{i}$ and their corresponding $\vec{i'}$. We define a direction ψ to be one of '<', '=', '>'. There is a dependence with direction '<' if there is a dependence such that $i < i'$. We allow combinations such as '$\leq$' to imply that the direction is '<' or is '='. We use '$*$' to represent any possible direction. A direction vector $\vec{\psi}$ is a vector of directions $\psi_1, \psi_2, \ldots, \psi_n$. We say that two references with loop index variables $\vec{i}$ and $\vec{i'}$ are dependent with direction vector $\vec{\psi}$ iff

$$
\begin{array}{l}
i_1 \ \psi_1 \ i'_1 \\
i_2 \ \psi_2 \ i'_2 \\
\quad \ldots \\
i_n \ \psi_n \ i'_n
\end{array}
$$

Two references can be dependent with more than one direction vector.

```
for i = 0 to 10 do
  for j = 0 to 10 do
    a[i][j] = a[2i][j]+7
  end for
end for
```

The two references are dependent with both (<, =) and (=, =). The dependence test should therefore return all the direction vectors with which the two references are dependent.

Another, more detailed, form of summarizing the dependence information is distance vectors. Two references are dependent with distance $\vec{d}$ if $\vec{i} - \vec{i'} = \vec{d}$.

```
for i = 0 to 10 do
  a[i] = a[i-3]+7
end for
```

In this example, we know more than just $i < i'$. We know that $i - i' = -3$. The extended GCD test provides us with an easy way to compute distance vectors. In the above example, GCD tells us that $i = t_1$ and $i' = t_1 + 3$. We merely subtract these two expressions. Since GCD does not use bounds, this method does not work for cases where the distance is only constant because of the bounds. For example:

```
for i = 1 to 8 do
  for j = 1 to 10 do
    a[10i+j] = a[10(i+2)+j]+7
  end for
end for
```

We will not discover that the distance vector is (2,0). Short of enumerating all possible dependences, we know of no way to compute distance vectors in every case. Nonetheless, GCD should work for the common constant-distance cases.

On the other hand, direction vectors can be computed in all cases. A simple method is to enumerate all possible direction vectors[4] and ask if the references are independent subject to the current vector. Each direction vector is a set of simple linear constraints on the loop variables. We simply add these constraints to our system and solve as before. The standard approach, based on Burke and Cytron [5], uses a hierarchical system to prune. Rather than testing each possible vector it first tests $(*, *, \ldots, *)$ If this returns independent, we know there are no direction vectors for which the references are dependent. If it returns dependent, we then preform the tests with each of the following vectors $(<, *, \ldots, *)$, $(=, *, \ldots, *)$ and $(>, *, \ldots, *)$. If, for example, the first vector returns independent we know that there is no dependence with any vector whose first component is $<$. We can prune any such vector. If any vector returns dependent, we continue to expand its $*'s$.

The addition of direction vector constraints can conceivably limit the applicability of our tests. For example:

$$1 \leq t_1, t_2, \leq 10$$
$$t_1 \leq t_2$$

can be solved with the Acyclic Test. A direction vector may add the constraint $t_2 < t_1$. The Acyclic Test is no longer applicable and we must use the Loop Residue Test. Similarly, there could be cases where direction vectors force us to use the Fourier-Motzkin Test. In practice, we have observed a greater need for Acyclic and Loop Residue, but in no case have we been forced to use Fourier-Motzkin due to the addition of the extra constraints.

A more serious problem is that direction vectors require the dependence tests to be applied multiple times for a pair of references. The number of possible vectors is potentially exponential in the loop nesting. Even using hierarchical pruning, without further optimizations we still have problems in practice. In Table 4, we repeat Table 3 with direction vectors, counting every direction tested. This overestimates the cost since certain fixed costs such as the GCD test and the overhead of transforming the bounds to be in terms of the t variables do not depend on how many vectors are tried per test.

Even allowing for a generous overestimate, calculating direction vectors has greatly increased the number of tests performed. Before, the compiler called 332, mostly SVPC, tests. Now, it needs to call about 12,500, mostly Acyclic and Loop Residue tests. The number of times Fourier-Motzkin is applied has gone from 32 to 157 (note that this is solely due to checking multiple vectors for the same references).

Some simple pruning methods can bring these costs back down dramatically. In discussing memoization, we mentioned that we need not include unused variables. For example:

```
for i = 1 to 10 do
  for j = 1 to 10 do
    a[j] = a[j+1]
  end for
end for
```

Since i does not appear in either the array expression nor in a loop bound, we know that direction for $i - i'$ is "$*$". Thus we run the tests for j and then prepend a "$*$" to the resultant direction vectors.

Calculating distance vectors can also help. If, for example, we have

```
for i = ...
  a[i+1] = a[i]
end for
```

we know from the GCD test that $i - i' = -1$. We therefore know that $i < i'$ and need not try out any other directions for i.

Table 5 shows our results with unused variables eliminated and with distance vector pruning.

We now have to call the tests only about 900 times. If we need better results, Burke and Cytron suggest as an optimization that *nice* cases can be treated on a dimension by dimension basis rather than as a system. For example:

[4]Note, with distance vectors we would have had to enumerate all possible dependences. With direction vectors we only need to enumerate all possible directions, a large but much smaller number.

Dependence Test Frequency For Direction Vectors					
Program	#Lines	SVPC	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	363	104	100	0
CS	18,520	127	48	34	0
LG	2,327	1,067	1,138	4,619	0
LW	1,237	132	73	59	0
MT	3,785	120	32	16	0
NA	3,976	295	124	172	23
OC	2,739	37	8	4	0
SD	7,607	309	106	120	28
SM	2,759	355	110	169	0
SR	3,970	130	30	18	0
TF	2,020	169	16	11	0
TI	484	780	267	703	0
WS	3,884	303	105	52	106
TOTAL	59,412	4,187	2,161	6,077	157

Table 4: Number of times each test was called only looking at unique cases and computing direction vectors

Unused Variables and Distance Vector Pruning					
Program	#Lines	SVPE	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	27	6	6	0
CS	18,520	14	16	14	0
LG	2,327	44	6	6	0
LW	1,237	15	12	5	0
MT	3,785	14	0	0	0
NA	3,976	48	59	118	7
OC	2,739	5	0	0	0
SD	7,607	54	20	55	28
SM	2,759	8	0	0	0
SR	3,970	14	0	0	0
TF	2,020	23	0	0	0
TI	484	3	38	72	0
WS	3,88 4	35	15	0	106
TOTAL	59,412	304	172	276	141

Table 5: Number of times each test was called using distance vector pruning and pruning away all unused variables

```
for i = 1 to 10 do
  a[i + 1][j] = a[i][j]
end for
```

i and j are not interrelated and we can compute each component of the direction vector independently.

Direction vectors also introduce an implicit branch-and-bound step. It is possible for the tests to return "unknown" when not calculating direction vectors but to return independent for every possible direction vector. In these cases, we can clearly set the references to independent. This occurs four times in our test suite. In each case, there is a real dependence with distance greater than zero but less than one.

7 Discussion

We have shown that the algorithms can be exact in practice. We have not shown how being exact compares with other, inexact, approaches. Looking at pairs of references does not give the entire picture. In a loop with a thousand independent pairs, being inexact in just one test could have a devastating effect on the amount of parallelism discovered. Ideally, one would like a standard model to measure the parallelism found. Then one could say how much faster a program ran due to exact data dependence. Unfortunately no such system yet exists.

Nonetheless, to give some comparison, we implemented the simple GCD test (algorithm 5.4.1 in [4]) and the Trapezoidal Banerjee Test (algorithm 4.3.1 in [4]). Not computing direction vectors, these algorithms found 415 out of 482 independent pairs, missing 16%. For direction vectors, we used the simple GCD test followed by Wolfe's extension to Banerjee's rectangular test (2.5.2 in [19]). We eliminated unused variables so that a[i] versus a[i-1] would return the one direction vector (* <) and not for example the three direction vectors ((< <) (= <) (> <)) which would be returned if these references were also enclosed by a second, unused outer loop. These algorithms returned 8,314 direction vectors which is 22% more than the exact answer of 6,828.

We do not believe that our particular choice of tests is very important. The key concept is the use of a suite of special case exact tests. It is quite possible that other tests could be added and some eliminated without significantly changing our results.

Extended GCD was chosen because it increases the applicability of the other tests. The other tests were chosen for several reasons. They all expect their data in the same form: $A\vec{x} \leq \vec{b}$. Thus there is no need to convert data from one form to another. Some tests, like the lambda test [11] expect their data in a different form.

All the tests succeed in finding independent references in practice. We checked how many times each test returned independent (counting each direction vector tested) for the tests in Table 5. SVPC returned independent in 40 out of 308 cases, Acyclic in 14 out of 172 cases, Loop Residue in 131 out of 276 cases and Fourier-Motzkin in 82 out of 141 cases.

The ordering of the tests is by cost. Timing them on a MIPS R2000 based machine (a 12 MIPS machine), SVPC averaged about 0.1 msec/test, Acyclic about 0.5 msec/test, Loop Residue about 0.9 msec/test and Fourier-Motzkin about 3 msec/test.

Finally, in Table 6 we timed our dependence tests and compared them to standard scalar optimizing compilers (f77 -O3). We wish to show that being exact adds very little cost to compilation time. The timings do not include the set up time required for dependence testing, for example expressing a reference in terms of the loop variables. This setup time, while possibly significant, is equivalent for all methods. The timings therefore should be looked upon as an upper bound for the extra time required to use our approach. The standard compilation time used full scalar optimizations. Our approach added only about 3% on average to the compile time.

8 Extension to Symbolic Testing

Our tests expect all references and bounds to be linear functions of the induction variables. We mentioned before that we use optimization techniques (constant propagation, induction variable and forward substitution) to increase the applicability of these conditions. For example:

```
n = 100
i2 = 0
...
for i = 1 to 10 do
  i2 = i2 + 2
  a[i2+n] = a[i2+2n+1]+3
end for
```

will be converted by our optimizer to:

```
for i = 1 to 10 do
  a[2i+100] = a[2i+201]+3
end for
```

which meets our conditions for analysis.

Nonetheless, there are cases where the unknown variables can not be expressed as functions of the induction variables.

```
read(n)
...
for i = 1 to 10 do
  a[i+n] = a[i+2n+1]+3
end for
```

Dependence Testing Cost		
Program	Dep. Test Cost (in secs)	f77 -O3 (in secs)
AP	2.2	151.4
CS	*	485.0
LG	4.0	65.4
LW	1.1	33.0
MT	1.0	45.0
NA	3.6	136.3
OC	0.3	38.2
SD	2.7	62.1
SM	3.5	102.5
SR	3.8	118.5
TF	2.6	116.6
TI	0.7	12.6
WS	3.6	110.0

Table 6: Total Cost of Dependence Testing. * too small to measure

As long as we know that n does not vary inside the loop, we can add it to our system as if it were an induction variable without bounds. For this example, our system would ask the following question.

does there exist integers i, i' and n such that
$1 \leq i, i' \leq 10$ and
$i + n = i' + 2 * n + 1$

Table 7 shows the results of adding symbolic testing to our system. Our tests are now called about 1,060 times compared with about 900 times before. This should add little to our total cost. We speculate that the low cost is because our prepass optimizations are quite powerful.

9 Conclusion

Data dependence analysis is a fundamental component in any parallelizing compiler. Previous techniques have required approximations. We have presented a definitive solution to the problem by using a combination of simple, easy to implement techniques; cascading special case exact tests, memoization and better direction vector pruning. In practice, our tests have found independent references which could not be found with currently used techniques. Running large benchmarks, we demonstrate empirically that our method is both exact and inexpensive.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[4] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[5] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 162–175, 1986.

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

[7] M. Berry et al. The PERFECT Club benchmarks: effective performance evaluation of supercomputers. Technical Report UIUCSRD Rep. No. 827, University of Illinois Urbana-Champaign, 1989.

[8] R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, August 1987.

[9] H.W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.

[10] Z. Li and P. Yew. Practical methods for exact data dependency analysis. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.

Symbolic Testing					
Program	#Lines	SVPE	Acyclic	Loop Residue	Fourier-Motzkin
AP	6,104	33	22	6	0
CS	18,520	20	24	19	0
LG	2,327	48	6	6	0
LW	1,237	15	12	5	0
MT	3,785	19	0	0	0
NA	3,976	55	149	101	7
OC	2,739	5	1	0	0
SD	7,607	54	20	55	28
SM	2,759	8	0	0	0
SR	3,970	21	1	2	0
TF	2,020	43	0	0	0
TI	484	3	38	72	0
WS	3,884	35	19	0	106
TOTAL	59,412	359	292	266	141

Table 7: Number of times each test was called computing direction vectors and adding symbolic constraints

[11] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan 1990.

[12] V.R. Pratt. Two easy theories whose combination is hard. Technical report, Mass Institue of Technology, Sept. 1977.

[13] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[14] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependencies. In *Proceedings of 1989 International Conference on Parallel Processing*, pages II–145 to II–152, 1989.

[15] R. Shostak. Deciding linear inequalities by computing loop residues. *ACM Journal*, 28(4):769–779, Oct 1981.

[16] S. Tjiang, M. Wolf, M.S. Lam, K. Pieper, and J.L. Hennessy. An overview of the SUIF compiler system. 1990.

[17] R. Triolet. Interprocedural analysis for program restructuring with parafrase. Technical Report CSRD Rep. No. 538, University of Illinois Urbana-Champaign, Dec. 1985.

[18] D. R. Wallace. Dependence of multi-dimensional array references. In *Proceedings of 1988 International Conference on Parallel Processing*, pages 418–428, 1988.

[19] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.