# Static and Dynamic Evaluation of Data Dependence Analysis*

Paul M. Petersen David A. Padua

Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign,
465 CSRL, 1308 W. Main Street,
Urbana, IL 61801-2932, USA
{petersen,padua}@csrd.uiuc.edu

## Abstract

This paper discusses the effectiveness of several dependence tests in the Perfect Benchmarks. The tests analyzed include the generalized greatest common divisor test, Banerjee's test and the Omega test. Two methods are applied. One uses only compile-time information for the analysis. The other uses information gathered during program execution. It is shown that, for the codes considered, the Omega test improved the accuracy of the analysis by only 1% when codes are analyzed statically. Furthermore, the dynamic analysis shows that the Omega test does not improve the detected inherent parallelism.

## 1 Introduction

Data dependence analysis is a central part of today's strategies for the automatic detection and exploitation of implicit parallelism. For this reason, experimental evaluation to determine the accuracy of dependence analysis techniques is very important. Such evaluation is necessary to guide research and to help compiler writers in the selection of a dependence analysis strategy.

In this paper we present an experimental evaluation of several dependence analysis techniques, including the constant test, the GCD test, three variants of Banerjee's inequalities [1], and integer-programming based tests such as the Omega test [2]. Both static (compile-time) and dynamic (run-time) experimental studies are presented.

Our study has two major objectives. The first is to determine how much the accuracy of the dependence analysis is affected if the value of the loop limits is not known. The second objective is to determine whether the tests based on integer programming produce better results than traditional approximations such as Banerjee's test when applied to real programs. Even though there have been other studies of dependence analysis that have presented and analyzed compile-time information [3, 4], to our knowledge this is the first work that focuses on these two objectives. In addition, there is no other work using run-time information to evaluate the accuracy of dependence analysis techniques. Run-time evaluation is the only practical way of determining the effects of different dependence analysis tests on program speedup.

The balance of this paper is organized as follows. In Section 2, we discuss briefly the dependence analysis techniques used in this study (we assume that the reader is familiar with the notion of dependence and with some of the techniques mentioned below). In Sections 3 and 4 we discuss the static and dynamic studies respectively. Finally in Section 5, we present our conclusions.

## 2 Background

In this study, we will concentrate on the analysis of data dependence when the interaction involves array elements. Most of the work published on dependence analysis (including all the tests considered in this paper) focus on statements with array references and assume that the two statements to be analyzed are both inside the same, possibly multiply-nested, DO loop.

*The research described is supported by Army contract #DABT63-92-C-003. This work is not necessarily representative of the positions or policies of the Army or the Government. This work was also supported by the NASA Ames Research Center Grant No. NCC 2-559 (DARPA), and a donation from MIPS Computer Systems.



**ICS-7/93 Tokyo, Japan**
**© 1993 ACM 0-89791-600-X/93/0007/0107...$1.50**

Consider the loop in Figure 1. Here $X$ is an $n$-dimensional array, and $f_i$ and $g_i$ are functions from $\mathbf{Z}^d$ to $\mathbf{Z}$.

```
      DO I_1 = L_1, U_1
        ...
        DO I_d = L_d, U_d
S_v :     X(f_1(I_1, ..., I_d), ..., f_n(I_1, ..., I_d)) = ...
S_w :       ... = X(g_1(I_1, ..., I_d), ..., g_n(I_1, ..., I_d))
        END DO
        ...
      END DO
```

Figure 1: Generic loop nest.

To decide whether there is a *flow dependence* from $S_v$ to $S_w$ that is, to determine whether a value computed by $S_v$ is used by $S_w$ it is necessary to determine whether there are two executions of $S_v$ and $S_w$ such that

(1) The execution of $S_v$ takes place in iteration $\bar{I}' = (I'_1, I'_2, \ldots, I'_d)$ and the execution of $S_w$ takes place in iteration $\bar{I}'' = (I''_1, I''_2, \ldots, I''_d)$ with $\bar{I}' \leq \bar{I}''$

(2) $f_i(\bar{I}') = g_i(\bar{I}'')$ for all $(1 \leq i \leq n)$ and for some $I'_1, I'_2, \ldots, I'_d, I''_1, I''_2, \ldots, I''_d$ within the loop limits specified in the program.

The conditions that determine a dependence if $S_v$ is located lexically after $S_w$ are the same except that condition (3) is replaced by (3'): $\bar{I}' < \bar{I}''$.

The problem of determining dependence is sometimes decomposed into several subproblems, one for each possible ordering relationship between the components of the vectors $\bar{I}'$ and $\bar{I}''$. For example, if $\bar{I}', \bar{I}'' \in \mathbf{Z}^2$, the condition $\bar{I}' < \bar{I}''$ can be decomposed into four cases:

$$\begin{array}{lll} I'_1 < I''_1 & \text{and} & I'_2 < I''_2 \\ I'_1 < I''_1 & \text{and} & I'_2 = I''_2 \\ I'_1 < I''_1 & \text{and} & I'_2 > I''_2 \\ I'_1 = I''_1 & \text{and} & I'_2 < I''_2 \end{array}$$

In general, these cases are specified using direction vectors which are of the form $\Psi = (\psi_1, \ldots, \psi_d)$ where each $\psi_k$ is one of $<$, $>$, or $=$ and represents the ordering relation between $I'_k$ and $I''_k$. In the case of the previous example, the feasible direction vectors are $(<,<)$, $(<,=)$, $(<,>)$, and $(=,<)$.

The same discussion presented above applies to *anti-* and *output* dependences. The difference is that for an *anti*-dependence to exist, it is necessary that the element of array $X$ be on the righthand side in $S_v$ and on the lefthand side of $S_w$. For *output* dependences, both occurrences of $X$ should be on the lefthand side.

We say there is a *potential dependence* for each pair of references to the same array within the body of the same, possibly multiply-nested, loop if at least one reference is on the lefthand side of an assignment statement. The two references could appear on the same statement. In particular, there is a potential dependence between the two different instances of an array reference that appear on the lefthand side of a statement. For each potential dependence it is necessary to invoke a dependence test to determine whether an actual dependence exists. When the test determines that no actual dependence exists, we say that it *breaks* the potential dependence.

A potential dependence involves *coupled subscripts* if it can only be broken by simultaneously considering the subscripts in a multidimensional array reference. Of the tests described in this paper, only the generalized GCD and the integer programming tests are able to handle coupled subscripts.

The rest of this section consists of two subsections. One discusses simple and approximate tests and the other describes integer-programming based tests.

## 2.1 Approximate Dependence Tests

Approximate dependence techniques, especially those developed by Banerjee [5, 6], have been widely adopted in both experimental and commercial compilers. In the last few years, a renewed interest in the subject of dependence analysis has arisen, and techniques have been developed that are in some cases more accurate than Banerjee's [7, 8, 9].

In addition to the published variations of Banerjee's inequalities, for rectangular and trapezoidal iteration spaces, we have extended the implementation to handle dependences correctly in loops with unknown upper and lower limits. This extension was developed after some preliminary results from the experiments in this paper. We later found that J. R. Allen [10] describes a method that works in a similar manner, and we believe that in practice Banerjee's inequalities are implemented in a similar manner. This variation has been analyzed separately because it has intrinsic interest and also to determine empirically the necessity of knowing the loops bounds.

The goal of all the approximate techniques presented in this section is to break potential dependences without incurring the cost associated with the exact tests discussed in the next section. Except for the generalized GCD test, these techniques analyze one subscript at a time. A potential dependence is broken only if the test shows that, for some subscript position $i$, there are no index vectors $\bar{I}'$ and $\bar{I}''$ that satisfy the equation $f_i(\bar{I}') = g_i(\bar{I}'')$, where $f_i$ and $g_i$ are the functions defined in Figure 1. Doing the test independently for each

subscript is conservative because the system of equations may not have a solution even if all the individual equations have solutions.

### 2.1.1 Simple Dependence Tests

The constant test is the only approximate method presented here that not only can break dependences but can also conclusively prove dependences. If all the subscripts in the two array references are loop invariant and have the same value, then there will be a data dependence for all potential direction vectors. If any pair of corresponding subscripts are constant and different, then there is no data dependence regardless of the values of any other subscript. Loop invariant expressions that are common to both subscripts in the potential dependence are canceled before the comparison is made.

The greatest common divisor (GCD) test establishes an existence criterion for the solution to the equation $f_i(\bar{I}') = g_i(\bar{I}'')$ mentioned above. This test is based on the fact that when both $f_i$ and $g_i$ are linear, a solution to the equation exists if the greatest common divisor of the coefficients of $\bar{I}'$ and $\bar{I}''$ also divides the constant term. Conversely, if it does not divide the constant term, then no solution can exist. In the work reported here we used the generalized GCD method, described in [1]; it is an extension of the GCD method that considers all subscripts in a multidimensional array simultaneously. When the GCD method breaks a potential dependence, it breaks all the direction vectors simultaneously. The GCD method cannot prove dependence because it does not take into account the value of the loop limits.

### 2.1.2 Banerjee's Inequalities for Loops with Known Limits

Traditionally the most widely studied dependence tests are those based on Banerjee's inequalities. A formalization of these ideas is presented in [1]. If we assume that $f_i(\bar{I}) = a_1 I_1 + a_2 I_2 + \cdots + a_d I_d + a_0$ and $g_i(\bar{I}) = b_1 I_1 + b_2 I_2 + \cdots + b_d I_d + b_0$, then the equation $f_i(\bar{I}') = g_i(\bar{I}'')$ can be written as $(a_1 I_1' - b_1 I_1'') + \cdots + (a_d I_d' - b_d I_d'') = (b_0 - a_0)$. The mathematical basis for Banerjee's inequalities [11] is derived from the *Intermediate Value Theorem* that states: Let $\mathcal{F}$ be defined as $(a_1 I_1' - b_1 I_1'') + \cdots + (a_d I_d' - b_d I_d'')$ which is continuous and can be considered as a real valued function in $\mathbf{R}^{2d}$. Let $B_{\min}, B_{\max}$ denote any two values of $\mathcal{F}$ on a connected set $\Re \subset \mathbf{R}^{2d}$, which is determined by the loop limits and the direction vectors. Suppose also that $B_{\min} \leq b_0 - a_0 \leq B_{\max}$. Then the equation $\mathcal{F}(x) = b_0 - a_0$ has a solution $x \in \Re$. Conversely with $B_{\min}$ and $B_{\max}$ the minimum and maximum values respectively of $\mathcal{F}$ in $\Re$, if $B_{\min} \leq b_0 - a_0 \leq B_{\max}$ is not true, then no solution can exist and independence has been proven. Data dependence tests based on this formulation of the original test apply in two general cases; (a) where all the loop bounds are known, and therefore the iteration space is rectangular, and (b) where the inner loop limits are linear functions of the indices of the outer loops and all the coefficients are constants. In this latter case the iteration space has the form of a trapezoid.

Banerjee's test cannot prove dependence because only the existence of a real valued solution in the region of interest has been proven, which does not always imply the existence of an integer valued solution.

### 2.1.3 Banerjee's Inequalities for Loops with Unknown Limits

This test is similar to Banerjee's test when it is applied to rectangular iteration spaces. If the stride of the loop is a positive constant, then whenever a loop lower limit is not known we assume $-\infty$ as its value, and whenever the loop upper limit is not known we assume $+\infty$ as its value. The only difference from the traditional Banerjee's test is that the arithmetic is done in the extended real number system [12] (i.e., $\mathbf{R} \bigcup \{-\infty, +\infty\}$) and the traditional conventions are made on the operations (e.g., $x + \infty = +\infty$, $x/\infty = 0$ if $x$ is real).

## 2.2 Integer Programming Based Tests

As discussed above, data dependence analysis of linear array references is equivalent to deciding if there is an integer solution to a set of linear equalities and inequalities [13]. The integer programming problem can be stated as follows: Does there exist $\bar{x}$ such that $A\bar{x} = \bar{b}$, $B\bar{x} \geq 0$, $\bar{x} \geq 0$, for integer $\bar{x}$. In this definition, matrix $A$ contains the coefficients of the equalities, and matrix $B$ refers to the coefficients of the inequalities that describe the bounds of the iteration space.

General integer programming techniques have many advantages over the approximations mentioned in the preceding section. The ability to consider simultaneously all subscripts of an array reference allows this dependence test to analyze coupled subscripts correctly. Affine loop bounds are incorporated naturally into the inequalities. Furthermore, execution constraints such as covering conditionals can be introduced into the dependence equations. It has also been reported that the linear programming approximation to integer programming is sufficient in most cases [4].

### 2.2.1 Simplex Based Integer Programming Test

Several methods are available to solve the integer programming problem. One method of implementation is the branch-and-bound algorithm. This algorithm works by first solving the real valued linear programming problem using the simplex method. The solution is checked

to see if all of its components are integers. The first non-integer component ($x_i$) is selected and used to create two new problems with new constraints. The first problem is the same as the original problem with the additional constraint $x_i \leq \lfloor \text{value of } x_i \rfloor$. The second problem is the same as the original problem with the additional constraint $x_i \geq \lceil \text{value of } x_i \rceil$. This constraint process generates a binary tree of problems that repeatedly divide the iteration space.

In effect, the branch-and-bound algorithm does an exhaustive search of the iteration space; however, it optimizes the search. Any region of the iteration space that does not have at least a real valued solution will not be searched for an integer solution. Once an integer solution is found, the process stops and reports success. If, on the other hand, all branches of the exhaustive search lead to empty sets, then the process reports that no solution exists.

### 2.2.2 Omega Test

The Omega test [2] is an extension of the Fourier-Motzkin linear-programming algorithm allowing integer constraints on the solution vector. In addition to supporting the full capabilities of integer-programming, the Omega test also permits the systematic handling of unknown additive terms. Consider the subscripts $X(I+N)$ and $X(I')$ where $1 \leq I, I' \leq N$ ( $N$ is the loop upper limit ). The Omega test is capable of analyzing such expressions involving unknown additive constrained variables. After the addition of the loop limit, we find that the system of equations is inconsistent since $I + N \neq I'$ for all $I, I'$ in $[1 \ldots N]$.

# 3 Static Evaluation of Dependence Analysis Techniques

For the static evaluation of the dependence tests described in Sections 2.1 and 2.2, we used a subset of the Perfect Benchmarks [14], a collection of 13 Fortran programs that represent some of the applications most frequently executed on parallel and vector computers. All of the programs were preprocessed by KAP/Concurrent (optimization level 4 with no loop unrolling) before doing static dependence analysis, in order to remove induction variables and in this way improve the effectiveness of the static dependence analysis by exposing more information to the dependence tests. No interprocedural analysis was performed. Intraprocedural constant propagation was done before the tests were applied.

Two experiments were conducted. In each, a different sequences of dependence tests was applied. The results of these experiments are shown in Tables 1 and 2. The order of application of the dependence tests is that shown below the label *Proved Independent.*

The tables show how many potential dependences are proven to be dependences by the tests and how many are broken (i.e., are proven independent). Additionally, the contribution of each dependence test to each category is presented. In the *Assumed Dependent* row, we see the number of potential dependences that had to be assumed dependent because of the lack of compile-time information or because the subscripts could not be expressed as linear functions of the loop indices.

For example, if the upper bound, `N`, of a loop is a function of a formal parameter to the subroutine containing the loop, we assume that `N` can have any value. The test may *prove* that is is a dependence under this assumption. However, it is possible that if the value of `N` were known (for example via interprocedural constant propagation) the dependence tests could *prove* the opposite result. Thus, the notion of *prove* and *disprove* are only in the context of the available information.

We used the following mechanism to arrive at these numbers. Before we applied the sequence of tests in each of the experiments, the expressions $f_i(\cdots) = g_i(\cdots)$ generated from Figure 1 of each potential dependence, were simplified and loop invariant expressions on both sides of the equations were cancelled. The potential dependence is passed to the sequence of tests only if the equations are linear and the coefficients of the subscript expression are known at compile-time. Otherwise, the counter for the assumed dependences is incremented by the number of feasible directions of the potential dependence. In this way we keep a count of the potential dependences that cannot be analyzed because compile-time information is lacking.

An accumulator is associated with each dependence test and it is incremented each time the corresponding test is the first to detect independence. The accumulators associated with Banerjee's tests and the integer programming tests are incremented by one each time the corresponding test breaks a potential dependence for a given direction vector. The constant and GCD test accumulators are incremented by the number of feasible direction vectors for the potential dependence since these two tests apply to all direction vectors. Notice that counting in this way, the weight of each potential dependence grows with its level of nesting.

In this work, only dependences *within the same loop-nest* are considered. Furthermore, as mentioned above, only dependences caused by references to array elements are considered. Each of the two experiments consists of two parts. For the first part (shown in the *Original Loop Bounds* column), the loop limits are those of the original source program. In this case, the rectangular and trapezoidal versions of Banerjee's test and the Simplex-based integer programming test cannot be applied to all loops because they require that the loop limits be known at compile-time.

| | *Original Loop Bounds* | | *Loop Bounds Assumed Constant* | | |
|---|---|---|---|---|---|
| Type | Count | (%) | Count | (%) | Difference |
| Proved Dependent | | | | | |
| Constant Test | 116616 | 26.7% | 116616 | 26.7% | 0.0% |
| Integer Programming | 1553 | 0.4% | 30697 | 7.0% | 6.6% |
| Omega Test | 70770 | 16.2% | 39684 | 9.1% | -7.1% |
| Total Proved Dependent | 188939 | 43.3% | 186997 | 42.9% | -0.4% |
| Assumed Dependent | | | | | |
| Unanalyzable Subscripts | 16049 | 3.7% | 15515 | 3.6% | -0.1% |
| Proved Independent | | | | | |
| Constant Test | 150716 | 34.5% | 150716 | 34.5% | 0.0% |
| Greatest Common Divisor | 34645 | 7.9% | 34675 | 7.9% | 0.0% |
| Banerjee's Test for Rectangular Loops | 13320 | 3.1% | 42781 | 9.8% | 6.7% |
| Banerjee's Test for Trapezoidal Loops | 15 | 0.1% | 3418 | 0.8% | 0.7% |
| Banerjee's Test for Unbounded Loops | 28567 | 6.5% | 0 | 0.0% | -6.5% |
| Integer Programming | 92 | 0.1% | 1491 | 0.3% | 0.2% |
| Omega Test | 3982 | 0.9% | 727 | 0.2% | -0.7% |
| Total Broken Dependences | 231337 | 53.0% | 233808 | 53.6% | 0.6% |
| Total | 436325 | | 436320 | | |

Table 1: Dependence results for the Perfect Benchmarks

| | *Original Loop Bounds* | | *Loop Bounds Assumed Constant* | | |
|---|---|---|---|---|---|
| Type | Count | (%) | Count | (%) | Difference |
| Proved Dependent | | | | | |
| Constant Test | 116616 | 26.7% | 116616 | 26.7% | 0.0% |
| Omega Test | 72323 | 16.6% | 70381 | 16.1% | -0.5% |
| Integer Programming | 0 | 0.0% | 0 | 0.0% | 0.0% |
| Total Proved Dependent | 188939 | 43.3% | 186997 | 42.9% | -0.4% |
| Assumed Dependent | | | | | |
| Unanalyzable Subscripts | 16049 | 3.7% | 15515 | 3.6% | -0.1% |
| Proved Independent | | | | | |
| Constant Test | 150716 | 34.5% | 150716 | 34.5% | 0.0% |
| Greatest Common Divisor | 34645 | 7.9% | 34675 | 7.9% | 0.0% |
| Banerjee's Test for Unbounded Loops | 41902 | 9.6% | 44901 | 10.3% | 0.7% |
| Banerjee's Test for Rectangular Loops | 0 | 0.0% | 0 | 0.0% | 0.0% |
| Banerjee's Test for Trapezoidal Loops | 0 | 0.0% | 1234 | 0.3% | 0.3% |
| Omega Test | 4074 | 0.9% | 2282 | 0.5% | -0.4% |
| Integer Programming | 0 | 0.0% | 0 | 0.0% | 0.0% |
| Total Broken Dependences | 231337 | 53.0% | 233808 | 53.6% | 0.6% |
| Total | 436325 | | 436320 | | |

Table 2: Switch Banerjee's Test for Unbounded Loops with Banerjee's Test for Rectangular Loops, and the Omega Test with the Integer Programming Test.

For the second part of each experiment (shown in the *Loop Bounds Assumed Constant* column) we artificially changed all loop limits so that their values became known at compile-time. This step is intended to show the maximum effect of unknown loop limits in dependence analysis. Any lower bound of a loop that was not a linear function of the indices in the enclosing loop nest was replaced by 1, and any upper limit that was also not a linear function of the indices in the loop nest was replaced by the constant 40. The choice of 40 as the upper bound is arbitrary and was chosen to maintain consistency with earlier experiments [15] at Illinois. The stride or step of the loop was defined to be 1 if it was not an integer constant. Notice that the total number of potential dependences decreases when constants replace loop limits because, in the programs we analyzed, a few loop limits are expressions involving array elements that generate potential dependences. These potential dependences disappear when the expression is replaced by a constant.

The loops in the Perfect Benchmarks vary widely in the amount of information available at compile time. Table 3 shows the percentage of lower bound, stride, and upper bound values that cannot be detected at compile time after (intraprocedural) constant propagation, induction variable elimination, and dead-code elimination have been applied. When averaged over the entire collection, it is significant to note that the stride is almost always known at compile-time, the upper bound is almost never known, and the lower bound is known most of the time.

| Benchmark | Lower Bound | Upper Bound | Stride |
|---|---|---|---|
| adm(AP) | 5.0% | 97.7% | 0.0% |
| arc2d(SR) | 77.6% | 95.1% | 0.0% |
| bdna(NA) | 4.2% | 62.7% | 0.0% |
| dyfesm(SD) | 0.9% | 73.6% | 0.0% |
| flo52q(TF) | 2.2% | 89.2% | 1.6% |
| mdg(LW) | 3.8% | 66.0% | 3.8% |
| mg3d(SM) | 14.2% | 100.0% | 36.1% |
| ocean(OC) | 3.7% | 93.4% | 6.6% |
| qcd2(LG) | 7.0% | 66.9% | 0.0% |
| spec77(WS) | 2.6% | 22.1% | 0.0% |
| track(MT) | 7.7% | 42.9% | 0.0% |
| trfd(TI) | 4.1% | 64.9% | 0.0% |
| TOTAL | 11.9% | 71.0% | 3.2% |

Table 3: Statically unknown loop bounds.

Several important observations can be derived from Tables 1 and 2. First is the large number of dependences that are proved and broken by the constant dependence test. This result is consistent with the the results obtained by other studies [3, 4]. Second is the unexpected effectiveness of Banerjee's test for unbounded loops. Consider first the *Original Loop Bounds* column of Table 1 where the loop limits are processed as they appear in the source program. In this column we observe that, when the test for unbounded loops is applied after the tests for rectangular and trapezoidal loops, it breaks potential dependences in more than 6% of the cases, more than in the two other tests combined. In Table 2, it is shown that when the unbounded loop test is applied before the other two tests, it breaks dependences in more than 9% of the cases, and the application of the rectangular and trapezoidal loop tests contribute nothing extra. This dependence sequence illustrates that Banerjee's test can be performed with total accuracy without the need for complete information about the loop limits, and that none of the potential dependences removed by Banerjee's inequalities truly require the use of a trapezoidal dependence test. A third important conclusion is that after all the traditional tests have been applied, less than 1.0% of the analyzable cases are detected as independent by the integer programming methods.

## 3.1 Unanalyzable Subscripts

One way to improve dependence information is by applying the data dependence tests to a larger percentage of the potential dependences. Classifying the reason a potential dependence is unanalyzable by the techniques discussed in this paper is useful in determining where additional effort may prove useful. As can be seen from Table 4, the reasons for which a subscript is unanalyzable can be divided into three similarly sized categories.

| Type of Unknown Subscript | Count |
|---|---|
| Linear | 5242 |
| Nonlinear | 6503 |
| Array References | 4304 |
| Total | 16049 |

Table 4: Classification of unknown subscripts.

The three categories describe a characteristic of the subscripts that makes them difficult to analyze. The reasons include linear subscripts with unknown loop invariant information, nonlinear subscripts caused by loop variant variables or difficult operators contained in the expressions, and array references contained in the subscripts. The subscripts in each category are placed into the most restrictive set based on the classification order {Array, Nonlinear, Linear}. If two or more features are present in the same subscript pair, the more restrictive (leftmost) classification is chosen.

From Table 4, we see that the most common reason a potential dependence is unanalyzable is the presence

of a nonlinear operator or the use of a loop variant variable. The operators include division and exponentiation as well as intrinsic function calls. More aggressive symbolic manipulation may be able to reduce the size of this category by simplifying the expressions in some cases. The loop variant variables may be reduced by more aggressive algorithms to remove any remaining induction variables and by performing interprocedural analysis to make the assumptions at call sites less conservative.

The second most common category includes linear subscripts with unknown but loop invariant information. Assertions about the relations among these variables may allow more aggressive analysis. Finally, the last category of unanalyzable subscripts include references to array elements in the subscript. The use of subscripted subscripts is a difficult problem in dependence analysis, as it is comparable to using pointers to reference an element.

All the categories can be reduced by having more information about the subscripts. Interprocedural analysis and symbolic value propagation are two methods of collecting more information about the calculations that go into the variables involved in a subscript expression.

# 4 Dynamic Evaluation of Data Dependence Analysis

The second method of evaluation is based on information gathered dynamically. Dynamic data dependence evaluation uses the difference between the conservative static data dependence analysis and what happens during execution to evaluate the effectiveness of the static analysis. Our objective here is to relate a dependence test to the program speedup it makes possible. In other words, the fact that the Omega test only breaks 0.9% of the potential dependences seems to indicate that this test does not have a profound impact on the resulting speedup. However, if the few dependences that the Omega test breaks happen to be the critical ones, then the conclusion should be the opposite. Using a dynamic test is the only way by which this effect can be shown.

For the purposes of this evaluation we will be concerned only with *flow* dependences. The restriction to *flow* dependences allows us to focus our attention on the transportation of data rather than the effects of memory management that can be attenuated by compiler transformations such as renaming or expansion.

## 4.1 Overview of the Evaluation Method

The results of the dynamic evaluation of dependence analysis reported in this paper were one of many results obtained as part of a compiler evaluation project we have been conducting at Illinois [16] as an extension of previous work [17]. The purpose of this study is to evaluate parallelizing compilers and parallelization techniques in a machine-independent form.

By measuring the performance of a particular program on a single machine, it is easy to determine which of two compilers is more effective. All we have to do is to create an executable program with each compiler, run the programs, and record the execution times. The compiler that produces the fastest program is obviously better. However, using a real parallel machine introduces several problems. It is difficult to isolate experimentally the effects of the architecture from those of the compiler strategy. Differences in the resources available on real parallel machines may make it impossible to observe any potential differences in parallelism.

We have chosen, therefore, to evaluate a compiler or compilation technique using as the metric the program speedup made possible by them on an ideal target parallel machine. This ideal parallel machine consists of an unlimited number of processors. Each has unit time access to a common shared memory. Conflict-free memory access is assumed. Also, each arithmetic operation takes one time unit to execute.

The execution time on the ideal machine is computed by viewing the program as a directed graph, generated at run-time, where the nodes represent the operations that the program executes for a particular input data set, and the arcs represent the *flow* and *control* dependences that have to be obeyed to execute the program correctly. The critical path computation, detailed in [18], dynamically determines the length of the longest path in the directed graph corresponding to an execution of the program under study. The ratio of the number of operations executed to the length of the critical path is the value we use for speedup. This approach was originally introduced by Kumar [19] and later extended by Chen and Yew [17] with the purpose of measuring important characteristics of sequential programs.

```
    ...
S1: A=C+1
    DO I=1,3
S2:   J = J + 1
S3:   IF J < 20 THEN
S4:     B(I) = A + B(I)
      ENDIF
S5:   A = J * C
    ENDDO
```

Figure 2: Code fragment for dataflow graph.

To illustrate this idea, we will use the code fragment listed in Figure 2 whose execution is graphically represented by the digraph in Figure 3. The solid arcs portray

the *flow* dependences caused by values that flow between operations. The dashed arcs represent control dependences that are generated by conditional statements.

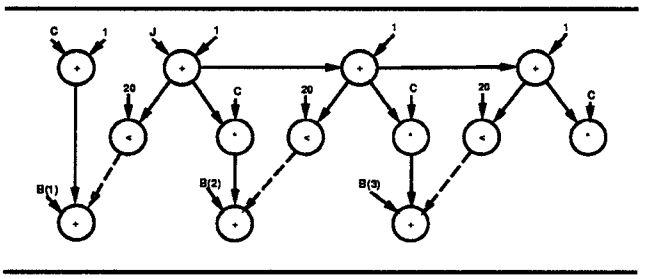

Figure 3: Execution graph of program *flow* dependences.

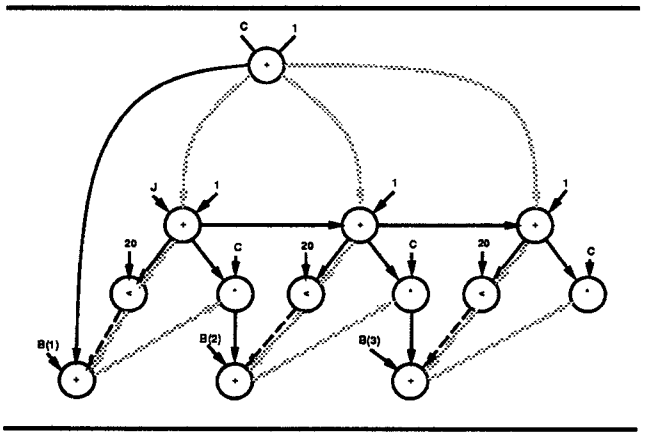

Figure 4: Graph of *flow* dependences with *constraining* arcs.

Notice that if we assign a weight of one to each node, the critical path through the graph corresponds to the shortest execution time on the ideal machine. Notice also that the memory-related dependences (i.e., *anti-* and *output* dependences) do not influence the critical path of the program. Thus, in the loop above, S5 is *anti*-dependent on S4 because A has to be fetched before it is modified to guarantee correct execution. However, such *anti-* dependence does not appear in the graph of Figure 3. This omission is easy to justify because many memory-related dependences can be removed by parallelizing compilers using expansion or privatization.

To evaluate the effect of a dependence test, we add arcs to the directed graph discussed above. One collection of arcs is used to guarantee that only loop-level parallelism is exercised. The arcs of this type (called *constraining arcs*) that should be added to the graph in Figure 3 are shown as grey arcs in Figure 4. The use of constraining arcs preclude the possibility of statement

| PROGRAM | Optimal Loop-Level | KAP-Obtained Loop-Level |
|---|---|---|
| adm(AP) | 45.5 | 3.0 |
| arc2d(SR) | 336.0 | 66.3 |
| bdna(NA) | 139.5 | 1.3 |
| dyfesm(SD) | 17.9 | 3.9 |
| flo52q(TF) | 206.9 | 76.7 |
| mdg(LW) | 5.3 | 1.4 |
| mg3d(SM) | 1.3 | 1.2 |
| ocean(OC) | 272.4 | 1.3 |
| qcd2(LG) | 2.4 | 1.2 |
| spec77(WS) | 13.8 | 1.1 |
| track(MT) | 38.7 | 1.1 |
| trfd(TI) | 87.9 | 10.3 |

Table 5: Inherent parallelism and parallelism obtained through automatic methods.

| PROGRAM | Omega Test CST & GCD | Banerjee CST & GCD | CST & GCD |
|---|---|---|---|
| adm(AP) | 8.38 | 8.38 | 2.77 |
| arc2d(SR) | 330.81 | 330.81 | 3.10 |
| bdna(NA) | 73.36 | 72.55 | 2.38 |
| dyfesm(SD) | 10.51 | 10.49 | 3.61 |
| flo52q(TF) | 206.36 | 206.36 | 2.66 |
| mdg(LW) | 5.22 | 5.22 | 5.12 |
| mg3d(SM) | 1.25 | 1.25 | 1.14 |
| ocean(OC) | 2.40 | 2.40 | 2.03 |
| qcd2(LG) | 2.27 | 2.27 | 2.15 |
| spec77(WS) | 13.52 | 13.52 | 13.20 |
| track(MT) | 32.90 | 32.90 | 1.74 |
| trfd(TI) | 58.02 | 58.02 | 2.89 |

Table 6: Average interprocedural parallelism: constrained by static dependence analysis

| PROGRAM | Omega Test CST & GCD | Banerjee CST & G CD | CST & GCD |
|---|---|---|---|
| adm(AP) | 3.00 | 3.00 | 1.89 |
| arc2d(SR) | 252.52 | 252.52 | 2.18 |
| bdna(NA) | 73.35 | 72.55 | 2.38 |
| dyfesm(SD) | 5.31 | 5.30 | 1.13 |
| flo52q(TF) | 206.16 | 206.15 | 2.66 |
| mdg(LW) | 4.61 | 4.61 | 3.50 |
| mg3d(SM) | 1.18 | 1.18 | 1.08 |
| ocean(OC) | 2.40 | 2.40 | 2.03 |
| qcd2(LG) | 1.49 | 1.49 | 1.42 |
| spec77(WS) | 2.25 | 2.25 | 2.20 |
| track(MT) | 1.73 | 1.73 | 1.59 |
| trfd(TI) | 26.60 | 26.60 | 1.52 |

Table 7: Average intraprocedural parallelism: constrained by static dependence analysis.

reordering. However, other sources of parallelism will be exploited including those that can be exposed by loop interchanging, skewing, or transformation into **doacross** loops.

The other collection of additional arcs correspond to the potential *flow* dependences not broken by a particular test. These arcs are called *static flow dependences*, and the objective of our study is to determine by how much these arcs increase the critical path of the directed graph. One implementation detail of these arcs is that they enforce static dependences to be of distance one. More details on how this is implemented can be found in [16].

## 4.2 Effectiveness of Dependence Tests

A number of papers [2, 3, 4] show that each of the many different dependence tests are capable of solving a number of data dependence problems. For evaluation purposes we want to use an *ideal* dependence test. This *ideal* test should be able to determine dependence or independence for all possible subscripts. It must be able to say on each reference to an array, exactly which set of definitions could have contributed to the value placed in this location. To approach the effect of an *ideal* dependence test, we use dynamic information reflected in a directed graph without imposing any static flow dependence arcs.

The dynamic experiments were performed on the same programs used in the static experiments. The first column of Table 5 lists the optimal inherent parallelism obtained through the use of an *ideal* dynamic dependence test [18]. This is the speedup for each program if no static flow dependence arcs are added to the directed graph. The second column, shown for informational purposes, is the measured performance of the codes transformed by the KAP/Concurrent restructuring compiler on the ideal machine. The performance of KAP is, as expected, lower than any of the other comparable measurements because it does both static intraprocedural dependence analysis and mapping of the parallelism.

Tables 6 and 7 list the results of our dynamic analysis study. Normally the results calculated by critical path analysis assume that all subroutines are effectively inlined. Table 6 gives the results for this assumption. Interprocedural parallelism allows loops with `CALL` statements to execute concurrently subject only to the restriction of the dynamic *flow* dependences. This means that for the parallelism reported in Table 6, we assume an *ideal* interprocedural dependence analysis test.

We define intraprocedural parallelism to be the parallelism inherent in a program when loops that contain subroutine calls are serialized. Instead of effectively inlining the subroutine at each call site, *artificial* dependence arcs connecting all `CALL` statements are added forcing only one subroutine call to be active at any time. Table 7 gives the results for this second assumption.

The last three columns in Table 6 list the interprocedural speedup, which is computed the same as the ideal speedup except that certain restrictions are applied. The restrictions are based on the potential *flow* dependences that are not proven independent by the data dependence tests listed in the heading for the column. The token `CST` stands for the constant test, and `GCD` stands for the generalized gcd test. Table 7 reports the effects of intraprocedural parallelism for each of the three types of dependence tests.

Several of the programs have a significant drop in parallelism when they are constrained by static dependence analysis. The most notable are `ocean(OC)` and `adm(AP)`. The reason for this degradation is that none of the dependence tests can handle the case where the coefficients of the index variables are loop invariant variables whose values are unknown at compile-time. Using a simple pattern matching technique, it is possible to show that the majority of the loops that were degraded, for these cases, are parallel. In fact, KAP/Concurrent was able to determine that most of these loops were parallel. The source of the problems in these programs are subscript references that have been linearized, which illustrates that one must consider not only linear subscripts with constant coefficients, but also special cases where loop invariant variables are used as coefficients.

One observation from the dynamic dependence evaluation is unexpected. The measured inherent parallelism, when constrained by static dependence arcs, does not change significantly when the more powerful dependence analysis techniques used in the Omega test are applied. Thus, for the Perfect Benchmarks, Banerjee's inequalities seem to be sufficient to detect **doall** and **doacross** parallelism.

Sometimes the correlation between the static and dynamic evaluations is as expected. The three best programs for average KAP parallelism are: `flo52q(TF)`, `arc2d(SR)`, and `trfd(TI)`. A comparison of Tables 6 and 7 for the same three programs shows that they all have a minimal loss of parallelism when serializing subroutine calls and eliminating interprocedural parallelism. The conclusion reached is as one would expect. Using a parallelizing compiler that does not do interprocedural analysis works effectively only when the programs are easy to analyze and do not require interprocedural parallelism.

Sometimes the correlation is not as encouraging. Four programs, `track(MT)`, `spec77(WS)`, `dyfesm(SD)`, and `adm(AP)`, show a large loss of average parallelism when Tables 6 and 7 are compared. These programs also correspond to instances where KAP was unable to exploit automatically any meaningful parallelism.

# 5 Conclusions

The statistical summary of static dependence information may not relate to the speedup that is obtained for a program. A single dependence may be responsible for precluding the parallelization of a loop. But we believe that statistical information is useful as a means to determine areas in which to concentrate further research.

It has also been shown that the infinity test seems to be a practical implementation that complements the published implementations of Banerjee's rectangular and trapezoidal tests. The main advantage of the infinity test is that it has semi-symbolic properties. These properties allow it to be applied in cases where a full symbolic implementation of the standard Banerjee inequalities might be too expensive.

Finally, the results from the Omega test indicate that only a small number of potential dependences require its power. Most of the applicability of this test is in proving the dependence of potential arcs that would otherwise be assumed dependent.

In Section 4 we extended the evaluation of data dependence analysis into the dynamic domain by considering the effects of the dependence analysis on the parallelism exploited in a program. This method is similar to performing the actual parallel compilation and execution on a parallel machine. However, our method attempts to isolate the important features of a parallel architecture without being constrained by artificial limitations such as cache/memory bandwidth or a limited number of processors.

The major result from the dynamic evaluation shows that the Omega test does not improve the average parallelism over the parallelism exposed by Banerjee's inequalities on these codes. Another benefit of this evaluation technique is the capability to locate exactly the places in the source code that are causing difficulty for the compiler. These locations might be beneficial as targets for hand transformations by the user or as test cases to identify further paths for development of the compiler.

# References


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

[2] W. Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," *Supercomputing'91*, 1991.

[3] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical dependence testing," *SIGPLAN Notices*, vol. 26, pp. 15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[4] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," *SIGPLAN Notices*, vol. 26, pp. 1–14, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

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

[6] K. Psarris, D. Klappholz, and X. Kong, "On the accuracy of the Banerjee test," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 152–157, June 1991.

[7] Z. Li, P.-C. Yew, and C.-Q. Zhu, "Data Dependence Analysis on Multi-dimensional Array References," in *Proc. 3rd International Conf. on Supercomputing*, pp. 215–224, June 1989.

[8] X. Kong, D. Klappholz, and K. Psarris, "The I Test: A New Test for Subscript Data Dependence," in *Proc. 1990 International Conf. on Parallel Processing*, August 1990.

[9] M. Wolfe and C.-W. Tseng, "The Power Test for Data Dependence," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 591–601, September 1992.

[10] J. R. Allen, *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[11] M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Processing*, October 1987.

[12] W. Rudin, *Principles of Mathematical Analysis.* International Series in Pure and Applied Mathermatics, McGraw-Hill Book Company, 1964.

[13] H. M. Salkin and K. Mathur, *Foundations of Integer Programming.* North-Holland, 1989.

[14] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer Performance Evaluation and the Perfect Benchmarks," Tech. Rep. 965, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., March 1990.

[15] Z. Shen, Z. Li, and P.-C. Yew, "An Empirical Study of FORTRAN programs for Parallelizing Compilers," CSRD Report no. 983, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., April 1990.

[16] P. M. Petersen, *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques.* PhD thesis, University of Illinois at Urbana-Champaign, January 1993.

[17] D.-K. Chen, "MaxPar: An Execution Driven Simulator for Studying Parallel Systems," Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., October 1989.

[18] P. Petersen and D. Padua, "Machine-Independent Evaluation of Parallelizing Compilers," in *Advanced Compilation Techniques for Novel Architectures*, January 1992.

[19] M. Kumar, "Measuring Parallelism in Computation-Intensive Science / Engineering Applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 5–40, 1988.