Static and Dynamic Evaluation of Data Dependence Analysis Techniques

Paul M. Petersen and David A. Padua, *Senior Member, IEEE*

Abstract—Data dependence analysis techniques are the main component of today's strategies for automatic detection of parallelism. Parallelism detection strategies are being incorporated in commercial compilers with increasing frequency because of the widespread use of processors capable of exploiting instruction-level parallelism and the growing importance of multiprocessors. An assessment of the accuracy of data dependence tests is therefore of great importance for compiler writers and researchers. The tests evaluated in this study include the generalized greatest common divisor test, three variants of Banerjee's test, and the Omega test. Their effectiveness was measured with respect to the Perfect Benchmarks and the linear algebra libraries, EISPACK and LAPACK. Two methods were applied, one using only compile-time information for the analysis, and the second using information gathered during program execution. The results indicate that Banerjee's test is for all practical purposes as accurate as the more complex Omega test in detecting parallelism. However, the Omega test is quite effective in proving the existence of dependences, in contrast with Banerjee's test, which can only disprove, or break dependences. The capability of the Omega test of proving dependences could have a significant impact on several compiler algorithms not considered in this study.

Index Terms—Dependence analysis, automatic parallelization, parallelism detection, compiler optimizations, evaluation of compiler techniques.

1 Introduction

Data dependence analysis is at the core of current strategies for the automatic detection of implicit parallelism in programs written in conventional sequential languages, such as Fortran and C. Dependence analysis techniques estimate, at compile-time, the execution-time interactions between different statements or between different executions of the same statement. This estimation is conservative in that whenever the dependence analysis technique is unable to determine whether the interaction will take place, it assumes an interaction. An execution-time interaction exists between two statement executions when both access the same memory location and one of the accesses is a memory write. To illustrate how dependence analysis is used to detect implicit parallelism, consider a Fortran `DO` loop. If the dependence analysis tests determine at compile-time that there will be no interactions between the iterations, then it is clear that the `DO` loop can be straightforwardly transformed into a `PARALLEL DO` loop whose iterations can execute concurrently with each other.

Dependence analysis is not only necessary for automatic detection of parallelism, but also crucial for many other important compiler transformations, such as those for improving memory locality, load balancing, and reducing the overhead due to task initiation and synchronization. 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 select 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 [3], an integer programming test based on the Simplex method, and the Omega test [18]. Both static (compile-time) and dynamic (run-time) evaluations are presented.

The study has two major objectives. The first is to determine to what extent statically unknown loop limits affect the accuracy of the dependence analysis. The second objective is to determine whether the theoretically most accurate tests, the Simplex-based integer programming test and the Omega test, produce better results than more efficient approximations, such as Banerjee's test, when applied to real programs. Even though there have been other studies of dependence analysis [10], [14] to our knowledge this is the first work that uses run-time information to evaluate the accuracy of dependence analysis techniques. Run-time evaluation is the only method to accurately evaluate the effect of dependence analysis tests on program speedup.

The rest of this paper is organized as follows. In Section 2, we briefly discuss the dependence analysis techniques used in this study (we assume that the reader is familiar with the notion of dependence and 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.

- *P.M. Petersen is with Kuck & Associates, Inc., 1906 Fox Dr., Champaign, IL 61820. E-mail: petersen@kai.com.*
- *D.A. Padua is with the Center for Supercomputing Research and Development, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: padua@csrd.uiuc.edu.*

Manuscript received Aug. 15, 1994; revised Feb. 6, 1996. A preliminary version of this paper appeared in ICS'93.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95211.

Most of the dependence analysis techniques described in the literature (including all the tests considered in this paper) focus on statements with array references and assume that, when two different statements are analyzed, both are inside the same, possibly multiply-nested, DO loop.

Consider the loop in Fig. 1. Here X is an n-dimensional array, and f_i and g_i are functions from $\mathbf{Z}^d$ to $\mathbf{Z}$, where $\mathbf{Z}$ is the collection of all integers. The set of all possible values that the vector of loop indices, $\bar{I} = (I_1, I_2, \ldots, I_d)$, may assume is called the *iteration space*.

```
      DO I1 = L1, U1
        ...
        DO Id = Ld, Ud
Sv :      X(f1(I1, ..., Id), ..., fn(I1, ..., Id)) = ...
Sw :        ... = X(g1(I1, ..., Id), ..., gn(I1, ..., Id))
        END DO
        ...
      END DO
```

Fig. 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 $\bar{I}'' = (I''_1, I''_2, \ldots, I''_d)$. Both $\bar{I}'$ and $\bar{I}''$ are in the iteration space and, therefore,
$$(L_j \le I'_j, I''_j \le U_j, 1 \le j \le d).$$
2) $f_i(\bar{I}') = g_i(\bar{I}'')$ for all $(1 \le i \le n)$.
3) $\bar{I}' \le \bar{I}''$.

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

The task 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 $\bar{I}', \bar{I}'' \in \mathbf{Z}^2$, then the condtion $\bar{I}' < \bar{I}''$ can be decomposed into four cases:

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

In general, these cases are specified using *direction vectors* of the form $\Psi = (\psi_1, \ldots, \psi_d)$, where each ψ_k is either <, >, 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 set of *distance vectors* associated with a dependence are all the vectors $\bar{I}'' - \bar{I}'$ such that $\bar{I}'$ and $\bar{I}''$ satisfy conditions 1), 2), and 3) above.

We say there is a *potential flow dependence* from a statement S_v to a statement S_w, both in the body of the same, possibly multiply-nested, loop, if both statements refer to the same array with S_v referencing the array on the left-hand side and S_w referencing the array on the right-hand side. If the same array appears on both the right- and left-hand sides of a statement S, we say that there is a potential flow dependence of S with itself.

The same discussion presented above applies to the two other types of data dependences, namely *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 right-hand side in S_v and on the left-hand side of S_w. For *output* dependences, both occurrences of X should be on the left-hand side.

For each potential dependence we must 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. All the tests described below require that the f_i's and g_i's be linear in the loop indices with constant coefficients. Otherwise the potential dependence is assumed to be an *actual* dependence.

We say that a multidimensional array reference $X(f_1, f_2, \ldots, f_n)$ has *coupled subscripts* when a loop index appears in two or more of the f_i expressions. When a potential dependence involves coupled subscripts, it may be necessary to simultaneously consider all of the subscripts in a multidimensional array reference to break the dependence. As discussed below, three of the tests considered in this study (GCD, Simplex-based integer programming, and Omega tests) handle coupled subscripts, while Banerjee's tests conservatively analyze each subscript separately.

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

2.1 Simple and Approximate Dependence Tests

Approximate dependence techniques, especially those developed by Banerjee [2], [17] 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 tests [13], [11], [25].

The goal of the techniques presented in this section is to break potential dependences without incurring the cost associated with the integer programming tests discussed in Section 2.2. 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 $\bar{I}''$ within the iteration space that satisfy the equation $f_i(\bar{I}') = g_i(\bar{I}'')$, where f_i and g_i are the subscript expressions in Fig. 1. Performing the test independently for each subscript is conservative because, in the presence of coupled subscripts, the system of equations may not have a solution even if all the individual equations have solutions.

2.1.1 Simple Dependence Tests

The first test described here, the *constant test*, cannot only break dependences but also conclusively prove depend-

ences. 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 possible direction vectors. If any pair of corresponding subscripts are constant and different, then there is no data dependence regardless of any other subscript values. Loop invariant terms 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 that multiplies the components 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 [3]; this is an extension of the GCD method that simultaneously considers all subscripts in a multidimensional array. When the GCD method breaks a potential dependence, it simultaneously breaks all the direction vectors. The GCD method cannot prove dependence because it does not take into account the value of the loop limits.

2.1.2 Banererjee's Inequalities for Loops with Known Limits

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 function $\mathcal{F}(\bar{I}) = (a_1 I_1' - b_1 I_1'') + \cdots + (a_d I_d' - b_d I_d'')$ is continuous in $\mathbf{R}^{2d}$. Let $B_{\min}$, $B_{\max}$ denote any two values of $\mathcal{F}$ in a connected set $\Re \subset \mathbf{R}^{2d}$, which contains all possible values of the iteration space. Suppose also that $B_{\min} \le b_0 - a_0 \le B_{\max}$. Then, from the intermediate value theorem, we know that the equation $\mathcal{F}(\bar{I}) = b_0 - a_0$ has a solution $\bar{u} = (u_1, u_2, \ldots, u_2 d) \in \Re$ and Banerjee's test assumes that a dependence exists. This is a conservative assumption because $\bar{u}$ only belongs to the iteration space when it is an integer vector.

Conversely, with $B_{\min}$ and $B_{\max}$ defined as the minimum and maximum values, respectively, of $\mathcal{F}$ in $\Re$, if the relation $B_{\min} \le b_0 - a_0 \le B_{\max}$ does not hold, then no solution can exist and the potential dependence is broken. Data dependence tests based on the original formulation of Banerjee's test apply in two general cases [3], [24]: 1) when all the loop limits are constant and, therefore, the iteration space has a (d-dimensional) *rectangular* shape and 2) when the loop limits are either constant or 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*. Constant, in this context, refers to a constant value that is known to the compiler, perhaps after value analysis.

In case 1), the dependence test can use the additional constraints of a direction vector to aid in breaking the dependence arc. In case 2), there is no known closed-form expression that takes into account the constraints imposed by a direction vector [3]. For complex trapezoidal iteration spaces, we must use an involved algorithm to determine if the iteration space is empty. Thus, even though the trapezoidal formulation of Banerjee's inequalities appears as if it might be superior, the inability to consider direction vectors in a simple manner causes it to be a weak dependence test in practice. The implementation used for the studies presented here tries only to break a subset of the direction vectors to avoid the complexities just mentioned.

Banerjee's test cannot prove dependence because it is only able to show the existence of a real valued solution in the region of interest, which does not always imply the existence of an integer valued solution. Furthermore, Banerjee's test is applied one subscript at a time and therefore it may conservatively assume dependence in the presence of coupled subscripts.

2.1.3 Banerjee's Inequalities for Loops with Unknown Limits

This test, first described in [1], is similar to Banerjee's test when applied to rectangular iteration spaces, but it can also be applied to trapezoidal loops. 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. This differs from the traditional Banerjee's test only in that the arithmetic is done in the extended real number system [20] (i.e., $\mathbf{R} \cup \{-\infty, +\infty\}$) and the traditional extensions 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 whether there is an integer solution to a set of linear equalities and inequalities. The integer programming problem can be stated as follows: Does there exist $\bar{x}$ such that $A\bar{x} = \bar{b}$, $B\bar{x} \ge 0$, $\bar{x} \ge 0$, for integer $\bar{x}$ [21]. In this definition, matrix A contains the coefficients of the equalities, and matrix B refers to the coefficients of the inequalities that describe the limits of the iteration space.

General integer programming techniques have several advantages over the approximations mentioned in the preceding section. The ability to simultaneously consider all subscripts of an array reference allows this class of dependence tests to correctly analyze coupled subscripts. The affine loop limits of trapezoidal loops are naturally incorporated into the inequalities. Furthermore, execution constraints, such as covering conditionals, can be introduced into the equations. Finally, these tests work on the integer domain and therefore the existence of a noninteger solution does not make the test assume the existence of a dependence, as is the case for Banerjee's test. The importance of these advantages, however, must not be evaluated theoretically, but rather in the context of real programs and the overall analysis strategy. For example, one previous study, [14], determined that after some simple dependence tests were applied to the Perfect Benchmarks, a linear programming algorithm (without integer con-

straints) was always exact on the leftover dependences. In all cases either the linear programming algorithm broke the dependence or all the components of the obtained solution were integer.

2.2.1 Simplex Based Integer Programming Test

Several methods are available to solve the integer programming problem. One method of implementation uses a branch-and-bound algorithm which first applies a linear programming algorithm to find a real-valued solution. The linear programming algorithm used in our implementation is the Simplex method.

The solution is checked to see if all of its components are integers. The first noninteger 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 problem domain.

In effect, the branch-and-bound algorithm does an exhaustive search of the problem domain; however, it optimizes the search. Any region of the problem domain 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. Our implementation assumes that the problem domain is bound and that the coefficients of the equations are constants. In other words, our implementation can only break dependences when the limits of the iteration space and the coefficients of the subscript expressions are known at compile-time.

2.2.2 Omega Test

The Omega test is an extension of the Fourier-Motzkin algorithm with integer constraints on the solution vector [18]. It is a powerful method that subsumes the Simplex-based method just discussed and includes a number of additional capabilities. In the experiments presented below, although the Omega test is used only to determine whether there is an integer-valued solution to a dependence equation, the test can also eliminate value-based transitive dependences and accurately determine distance vectors. These additional capabilities are important for a number of compiler algorithms, including those for array privatization and register usage and cache locality optimization.

In addition to solving the integer-programming problem when all coefficients have known constant values, the Omega test also handles unknown additive terms. In particular, the loop limit values do not need to be known for the method to work. Consider, for example, 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 $[1 \ldots N]$.

For simple subscripts of the kind also handled by Banerjee's test, the Omega test is usually a small constant factor slower than Banerjee's test [18]. However, as the subscripts to be analyzed increase in complexity, the execution time of this test will greatly increase to a worst-case time exponential in the number of variables. We have specifically included the Omega test in this experiment to determine the impact advanced data dependence tests can have on the effectiveness of a compiler in the automatic detection of parallelism.

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 [8], a collection of 13 Fortran programs that represent some of the applications most frequently executed on parallel and vector computers, and LAPACK and EISPACK, two well-known linear algebra routine libraries.

We included in this study both programs and routines because they have different characteristics. For example, strides are more likely to be a variable in a library routine than in a program because library routines tend to be more flexible. As a consequence, dependence analysis, when performed on the subroutine, has less information available. We would, therefore, expect to see the effectiveness of data dependence analysis vary according to the type of code being analyzed. Another difference that motivated our inclusion of EISPACK and LAPACK is that linear algebra algorithms commonly involve coupled subscripts which are handled conservatively by Banerjee's tests.

In order to remove induction variables and thereby improve the effectiveness of the static dependence analysis by exposing more information to the dependence tests, the programs and routines were preprocessed by KAP/Concurrent (optimization level 4 with no loop unrolling) before doing static dependence analysis. No interprocedural analysis was performed. Intraprocedural constant propagation was done before the tests were applied.

Also, no other transformations were applied by the compiler. Specifically, reductions and other linear recurrences were not transformed into parallel form, and privatization and scalar expansion transformations were not applied.

Two experiments were conducted. The two differ in the order in which the dependence tests were applied. The results of these experiments for the Perfect Benchmarks are shown in Tables 1 and 2; those for EISPACK and LAPACK are shown in Tables 3, 4, 5, and 6. The order in which the dependence tests were applied is shown below the label *Proved Independent*. One order is reported in Tables 1, 3, and 5; a different order is reported in Tables 2, 4, and 6.

The tables show how many potential dependences are proven to be dependences by the tests and how many are broken (i.e., 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.

TABLE 1
DEPENDENCE ANALYSIS RESULTS FOR PERFECT BENCHMARKS

	Original Loop Limits		*Loop Limits Assumed Constant*		
Type	Count	(%)	Count	(%)	Difference
Proved Dependent					
Constant Test	116616	26.7%	116616	26.7%	0.0%
Branch and Bound Simplex	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%
Branch and Bound Simplex	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 2
DEPENDENCE ANALYSIS RESULTS FOR PERFECT BENCHMARKS

	Original Loop Limits		*Loop Limits 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%
Branch and Bound Simplex	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	1.0%	2282	0.5%	-0.4%
Branch and Bound Simplex	0	0.0%	0	0.0%	0.0%
Total Broken Dependences	231337	53.0%	233808	53.6%	0.6%
Total	436325		436320		

Obtained by Switching Banerjee's Test for Unbounded Loops with Banerjee's Test for Rectangular Loops, and the Omega Test with the Branch and Bound Simplex Test

TABLE 3
DEPENDENCE ANALYSIS RESULTS FOR EISPACK

	Original Loop Limits		*Loop Limits Assumed Constant*		
Type	Count	(%)	Count	(%)	Difference
Proved Dependent					
Constant Test	1077	1.7%	1077	1.7%	0.0%
Branch and Bound Simplex	0	0.0%	13322	20.5%	20.5%
Omega Test	18790	28.9%	8361	12.9%	-16.0%
Total Proved Dependent	19867	30.6%	22760	35.0%	4.4%
Assumed Dependent					
Unanalyzable Subscripts	1400	2.2%	1308	2.0%	-0.2%
Proved Independent					
Constant Test	3301	5.1%	3301	5.1%	0.0%
Greatest Common Divisor	2128	3.3%	2128	3.3%	0.0%
Banerjee's Test for Rectangular Loops	1571	2.4%	25865	39.8%	37.4%
Banerjee's Test for Trapezoidal Loops	0	0.0%	3007	4.6%	4.6%
Banerjee's Test for Unbounded Loops	27806	42.8%	1347	2.1%	-40.7%
Branch and Bound Simplex	0	0.0%	3766	5.8%	5.8%
Omega Test	8938	13.8%	1529	2.4%	-11.4%
Total Broken Dependences	43744	67.3%	40943	63.0%	-4.3%
Total	65011		65011		

TABLE 4
DEPENDENCE ANALYSIS RESULTS FOR EISPACK

	Original Loop Limits		*Loop Limits Assumed Constant*		
Type	Count	(%)	Count	(%)	Difference
Proved Dependent					
Constant Test	1077	1.7%	1077	1.7%	0.0%
Omega Test	18790	28.9%	21579	33.2%	4.3%
Branch and Bound Simplex	0	0.0%	0	0.0%	0.0%
Total Proved Dependent	19867	30.6%	22656	34.8%	4.2%
Assumed Dependent					
Unanalyzable Subscripts	1400	2.2%	1412	2.2%	0.0%
Proved Independent					
Constant Test	3301	5.1%	3301	5.1%	0.0%
Greatest Common Divisor	2128	3.3%	2128	3.3%	0.0%
Banerjee's Test for Unbounded Loops	29377	45.2%	29993	46.1%	0.9%
Banerjee's Test for Rectangular Loops	0	0.0%	0	0.0%	0.0%
Banerjee's Test for Trapezoidal Loops	0	0.0%	226	0.4%	0.4%
Omega Test	8938	13.8%	5295	8.1%	-5.7%
Branch and Bound Simplex	0	0.0%	0	0.0%	0.0%
Total Broken Dependences	43744	67.3%	40943	63.0%	-4.3%
Total	65011		65011		

Obtained by Switching Banerjee's Test for Unbounded Loops with Banerjee's Test for Rectangular Loops, and the Omega Test with the Branch and Bound Simplex Test

TABLE 5
DEPENDENCE ANALYSIS RESULTS FOR LAPACK

	Original Loop Limits		*Loop Limits Assumed Constant*		
Type	Count	(%)	Count	(%)	Difference
Proved Dependent					
Constant Test	798	0.8%	798	0.8%	0.0%
Branch and Bound Simplex	149	0.1%	11324	11.2%	11.1%
Omega Test	32716	32.4%	23364	23.2%	-9.2%
Total Proved Dependent	33663	33.4%	35486	35.2%	1.8%
Assumed Dependent					
Unanalyzable Subscripts	14018	13.9%	5795	5.7%	-8.2%
Proved Independent					
Constant Test	3325	3.3%	3325	3.3%	0.0%
Greatest Common Divisor	13164	13.0%	13428	13.3%	0.3%
Banerjee's Test for Rectangular Loops	1581	1.6%	28629	28.4%	26.8%
Banerjee's Test for Trapezoidal Loops	149	0.1%	4278	4.2%	4.1%
Banerjee's Test for Unbounded Loops	28370	28.1%	1529	1.5%	-26.6%
Branch and Bound Simplex	29	0.1%	3308	3.3%	3.2%
Omega Test	6596	6.5%	5117	5.1%	-1.4%
Total Broken Dependences	53214	52.7%	59614	59.1%	6.4%
Total	100895		100895		

TABLE 6
DEPENDENCE ANALYSIS RESULTS FOR LAPACK

	Original Loop Limits		*Loop Limits Assumed Constant*		
Type	Count	(%)	Count	(%)	Difference
Proved Dependent					
Constant Test	798	0.8%	798	0.8%	0.0%
Omega Test	32865	32.6%	34362	34.1%	1.5%
Branch and Bound Simplex	0	0.0%	0	0.0%	0.0%
Total Proved Dependent	33663	33.4%	35160	34.8%	1.4%
Assumed Dependent					
Unanalyzable Subscripts	14018	13.9%	6121	6.1%	-7.8%
Proved Independent					
Constant Test	3325	3.3%	3325	3.3%	0.0%
Greatest Common Divisor	13164	13.0%	13428	13.3%	0.3%
Banerjee's Test for Unbounded Loops	29786	29.5%	32979	32.7%	3.2%
Banerjee's Test for Rectangular Loops	0	0.0%	0	0.0%	0.0%
Banerjee's Test for Trapezoidal Loops	0	0.0%	1379	1.4%	1.4%
Omega Test	6625	6.6%	8503	8.4%	1.8%
Branch and Bound Simplex	0	0.0%	0	0.0%	0.0%
Total Broken Dependences	53214	52.7%	59614	59.1%	6.4%
Total	100895		100895		

Obtained by Switching Banerjee's Test for Unbounded Loops with Banerjee's Test for Rectangular Loops, and the Omega Test with the Branch and Bound Simplex Test

The notions of *prove* and *disprove* used in the tables refer only to the context of the available information. For example, if the upper limit, N, of a loop is a formal parameter to the subroutine containing the loop, we assume that N can have any value. The test may *prove* that there 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 same test could *prove* that there is no dependence

We used the following mechanism to obtain the values presented in the tables. Before we applied the sequence of tests in each of the experiments, the equations $f_i(\cdots) = g_i(\cdots)$ were simplified and loop invariant expressions on both sides of the equations were canceled. 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 associated with each dependence test gets 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 by 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 Limits* 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.

For the second part of each experiment (shown in the *Loop Limits Assumed Constant* column) we artificially changed all loop limits so that their values became known at compile-time. This step is intended to estimate the effect of unknown loop limits in dependence analysis. Any lower limit 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 limit is arbitrary and was chosen to maintain consistency with earlier experiments [22] 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 7 shows the percentage of lower limit, stride, and upper limit 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 limit is almost never known, and the lower limit is known most of the time.

Several important observations can be derived from Tables 1, 2, 3, 4, 5, and 6. First is the large number of dependences that are proven and broken by the constant dependence test in the Perfect Benchmarks. This result is consistent with the results obtained by other studies [10], [14]. However, the effectiveness of the constant test is significantly lower for EISPACK and LAPACK.

A second observation is the unexpected effectiveness of Banerjee's test for unbounded loops. Consider first the *Original Loop Limits* column of Tables 1, 3, and 5, where the loop limits are processed as they appear in the source program. 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%, 42%, and 28% of the cases, respectively, which is more than the other two implementations of Banerjee's test combined. In Tables 2, 3, and 5 it is shown that when the unbounded loop test is applied before the other two tests, it breaks dependences in more than 9%, 45%, and 29% of the cases respectively, and the application of the rectangular and trapezoidal loop tests contribute nothing. We conclude that Banerjee's test can be applied with maximum accuracy without the need for complete information about the loop limits. Also, only rarely the potential dependences removed by Banerjee's inequalities truly require the use of a trapezoidal dependence test. These results do not change substantially even when the loop limits are artificially set to the constant value of 40.

TABLE 7
PERCENTAGE OF STATICALLY UNKNOWN LOOP LIMITS

Benchmark	Lower Limit	Upper Limit	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%

A third observation is that the percentage of potential dependences that have not been broken or proven conclusively after the application of Banerjee's tests varies substantially among the three collections used in this study. For the Perfect Benchmarks, it is 20.9%; for LAPACK it is 53%; and for EISPACK it is 44.9%. Finally, the Omega test does an excellent job analyzing these remaining potential dependences. Practically all the potential dependences that were proven dependent in EISPACK and LAPACK are those identified by the Omega test. Also, in the case of the Perfect Benchmarks, over 16% of the dependences were proven conclusively by the Omega test. In fact, the percentage of potential dependences assumed dependent would be over five times higher in the Perfect Benchmarks if the Omega test had not been applied. For EISPACK and LAPACK, the percentage of assumed dependences in the absence of the Omega test would be over 20 and over three times higher, respectively. Furthermore, the percentage of assumed dependences is quite small in both the Perfect Benchmarks and EISPACK. In the case of LAPACK, the effectiveness of the Omega test on the potential dependences remaining after Banerjee's test is somewhat lower, and a relatively large percentage of the dependences (13.9%) has to be conservatively assumed. On the other hand, the percentage of "proven independent" dependences resulting from the Omega test (1.75% for the Perfect; about 20% for EISPACK; and about 12%

for LAPACK) shows that the Omega test has a negligible impact on the Perfect Benchmarks. However, despite the fact that it is applied as the last test in the sequence, it has an important impact on the static counts for the EISPACK and LAPACK collections.

3.1 Unanalyzable Subscripts

One way to improve dependence information is to apply 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 to determine where additional effort may prove beneficial. As can be seen from Table 8, in the case of the Perfect Benchmarks, the reasons for which a subscript is unanalyzable can be divided into three similarly sized categories.

TABLE 8
CLASSIFICATION OF STATICALLY UNKNOWN SUBSCRIPTS

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

The three categories describe a characteristic of the subscripts that makes them difficult to analyze. The reasons include linear subscripts with unknown loop invariant coefficients (`Linear`), nonlinear subscripts caused either by loop variant coefficients or nonlinear operators on the loop indices contained in the expressions (`Nonlinear`), and array references contained in the subscripts (`Array`). If two or more features are present in the same subscript pair, the left-most classification in the order {`Array`, `Nonlinear`, `Linear`} is chosen.

From Table 8, we see that the most common reason a potential dependence is unanalyzable is the presence of a nonlinear operator or a loop variant coefficient. The operators include division and exponentiation, as well as intrinsic function calls. More aggressive symbolic manipulation may reduce the size of this category by simplifying the expressions in some cases. The number of loop variant coefficients may be reduced by applying 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 coefficients. 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. Runtime techniques have been shown to be a viable alternative for this problem [19].

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], [23].

4 Dynamic Evaluation of Data Dependence Analysis

The second method of evaluation is based on information gathered dynamically. Dynamic data dependence evaluation uses the results of the conservative static data dependence analysis and compares them with 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 breaks only 1% of the potential dependences in the Perfect Benchmarks 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. Dynamic evaluation is the only method by which this effect can be shown.

An *ideal test* that calculates the flow dependences encountered during a program's execution will be used below to evaluate the effectiveness of different dependence tests. By restricting the analysis to flow dependences, this study will focus almost exclusively on the effect of transportation of data on parallelism and ignore the effects of memory-related dependences. We believe this is reasonable because the effect of memory-related dependences can usually be removed by compiler transformations such as renaming or expansion. To make the comparison meaningful, the dependence tests evaluated below will also analyze flow dependences exclusively. One possible problem with this approach is that it may negatively affect the evaluation of the dependence analysis techniques. The reason is that memory-related dependences can influence the number and type of flow dependences. Consider for example the loop in Fig. 2. Unless the array `A` is identified as loop-private (and therefore the memory-related dependences due to this array are removed), conventional tests, such as Banerjee's test, would consider that there is a dependence form `S1` to `S2` on the outer loop. However, as will be seen below, this does not seem to be a major problem for the programs we have measured because the average parallelism obtained by the ideal test is generally close to the average parallelism obtained by the dependence tests evaluated in this study.

```
      DO I=1,10
        DO J=1,10
S1:       A(J) = ...
        END DO
        DO J=1,10
S2:     ... = A(J)
        END DO
      END DO
```

Fig. 2. Code fragment illustrating the effect of privatization on the number of flow dependences identified by Banerjee's test.

We did not apply the Simplex-based integer programming test in this part of the study because it does not break any dependences after the Omega test is applied. Also, the rectangular and trapezoidal versions of Banerjee's test were not applied because they do not detect any dependences beyond those detected by the Banerjee's test for unbounded loops.

4.1 Overview of the Evaluation Method

The results of the dynamic evaluation of dependence analysis reported in this paper were obtained as part of a larger compiler evaluation project [16]. The purpose of this study was 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 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, with a real parallel machine it is difficult to experimentally isolate the effects of the architecture from those of the compiler strategy. Differences among the available resources 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 they enable 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 runtime, 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 honored to execute the program correctly. The critical path computation, detailed in [16], 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 total number of nodes to the length of the critical path is the value we use for average parallelism. This approach was originally introduced by Kumar [12] and later extended by Chen and Yew [7] with the purpose of measuring important characteristics of sequential programs.

To illustrate this idea, we will use the code fragment listed in Fig. 3 whose execution is graphically represented in Fig. 4. The solid arcs portray the flow dependences caused by values that flow between operations. The dashed arcs represent control dependences generated by conditional statements.

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 antidependent on S4 because A has to be fetched before it is modified to guarantee correct execution. However, such antidependence does not appear in the graph of Fig. 4. This omission is easy to justify because most memory-related dependences can be removed by parallelizing compilers using renaming, expansion, or privatization.

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

Fig. 3. Code fragment for dataflow graph.

All the measurements presented below were obtained by adding arcs that guarantee that only loop-level parallelism is exercised. The arcs of this type (called *constraining arcs*) that should be added to the graph in Fig. 4 are shown as grey arcs in Fig. 5. The use of constraining arcs preclude the possibility of statement reordering. However, other sources of parallelism will be exploited, including those that can be exposed by loop reverse, skewing, or transformation into doacross loops [15], [9].

The measurements of each dependence test were performed on graphs that included all the arcs used for the ideal test plus additional arcs corresponding to the potential flow dependences not broken by the test. These arcs are called *static flow dependences*, and the objective of our study is to determine to what extent these arcs increase the critical path of the directed graph. One implementation detail of these arcs is that the distance of any dependence that cannot be broken statically is assumed to be 1.

4.2 Effectiveness of Dependence Tests

A number of papers [18], [10], [14] show that each of the many different dependence tests are capable of solving a number of data dependence problems not solved by others.

For example, the Omega test is in general more accurate than Banerjee's test. One reason is that the Omega test can take into account the coupling of the different subscripts to break dependences.

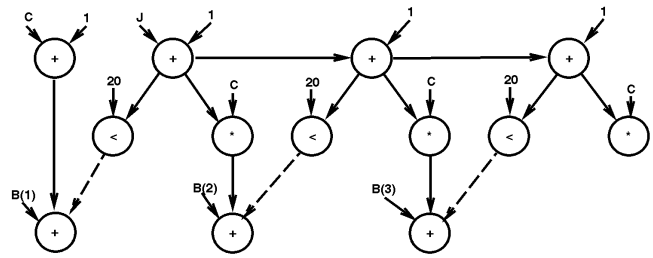


Fig. 4. Execution graph of program flow dependences.

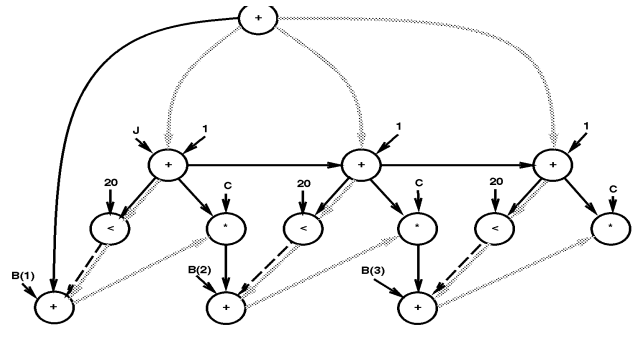


Fig. 5. Graph of flow dependences with constraining arcs.

The Omega test can potentially have a dramatic effect on the performance of a target parallel program. Consider the loop in Fig. 6. While Banerjee's test cannot break the dependences in this loop, the Omega test can. With the input of N = 100, the Omega test produces a speedup of 34.1 on our ideal machine. However, the evaluation of a test should be based on complete programs rather than isolated loops. The dynamic analysis described above makes it possible to automatically analyze large programs to determine the specific effects of any data dependence test.

```
DO I = 2, N
   DI = 1.0 / A(I,I)
   DO J = I+1, N
     SUM = 0.0
     DO K = 1, I-1
       SUM = SUM + A(K,I)*A(K,J)*A(K,K)
     END DO
     A(I,J) = DI*(A(I,J)-SUM)
    END DO
END DO
```

Fig. 6. Example loop from CHOFAC/dyfesm(SD).

For evaluation purposes, we need to use an ideal dependence test not just a better test. This ideal test should be able to determine dependence or independence for all possible subscripts on each reference to an array, it must be able to specify 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 of the flow dependence arcs computed statically.

The dynamic experiments were performed on the same collections used in the static evaluation. Before running the experiments, the programs were preprocessed in the same way they were for the static evaluation. That is, they were transformed by KAP/Concurrent and only induction variable elimination was applied. No other transformations were applied. In particular, scalar expansion, privatization, and linear recurrence substitution were not applied. The effect of these transformations, especially that of linear recurrence substitution, will be the subject of future work. We should point out that one important consequence of not performing linear recurrence substitution is that the average parallelism, even for the ideal test, is sometimes quite small. From our work on automatic detection of parallelism [6], we know that much better speedups are possible when reductions (a special case of linear recurrences) are recognized and replaced by equivalent parallel forms. We will first discuss the experiments on the Perfect Benchmarks which are summarized in Tables 9 and 10.

The first column in both tables lists the inherent parallelism obtained using the ideal dynamic dependence test just mentioned. The average parallelism in this column is the total number of nodes of the graph divided by the critical path length of the graph corresponding to the ideal test. The other columns list the results of three other dependence test sequences. The term GCD in these tables stands for the generalized GCD test. The values in the last three columns are computed in the same way as the values of the first column, except that the critical path length will usually differ because the graph now includes the static flow dependence arcs identified by each dependence sequence. Normally the results calculated by critical path analysis assume that all subroutines are effectively inlined. Table 9 gives the results for this assumption. Interprocedural parallelism allows loops with CALL statements to concurrently execute subject only to the restriction of the dynamic flow dependences. This means that for the parallelism reported in all columns of Table 9, we assume an ideal interprocedural dependence analysis test.

TABLE 9
AVERAGE INTERPROCEDURAL PARALLELISM IN PERFECT BENCHMARKS CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
adm(AP)	45.5	8.4	8.4	2.8
arc2d(SR)	336.0	330.8	330.8	3.1
bdna(NA)	139.5	73.4	72.6	2.4
dyfesm(SD)	17.9	10.5	10.5	3.6
flo52q(TF)	206.9	206.4	206.4	2.7
mdg(LW)	5.3	5.2	5.2	5.1
mg3d(SM)	1.3	1.3	1.3	1.1
ocean(OC)	272.4	2.4	2.4	2.0
qcd2(LG)	2.4	2.3	2.3	2.2
spec77(WS)	13.8	13.5	13.5	13.2
track(MT)	38.7	32.9	32.9	1.7
trfd(TI)	87.9	58.0	58.0	2.9

TABLE 10
AVERAGE INTRAPROCEDURAL PARALLELISM IN PERFECT BENCHMARKS CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
adm(AP)	45.5	3.0	3.0	1.9
arc2d(SR)	336.0	252.5	252.5	2.2
bdna(NA)	139.5	73.4	72.6	2.4
dyfesm(SD)	17.9	5.3	5.3	1.1
flo52q(TF)	206.9	206.2	206.2	2.7
mdg(LW)	5.3	4.6	4.6	3.5
mg3d(SM)	1.3	1.2	1.2	1.1
ocean(OC)	272.4	2.4	2.4	2.0
qcd2(LG)	2.4	1.5	1.5	1.4
spec77(WS)	13.8	2.3	2.3	2.2
track(MT)	38.7	1.7	1.7	1.6
trfd(TI)	87.9	26.6	26.6	1.5

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 thereby forcing only one subroutine call to be active at any time. Table 10 gives the results for this second assumption.

Several of the programs have a significant drop in parallelism over the ideal case 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. Simple symbolic dependence analysis makes it possi-

ble to show that the majority of the loops that were degraded, for these cases, are parallel [5]. In fact, KAP/Concurrent (using a different set of dependence tests), was able to statically 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 the necessity for dependence tests to consider not only linear subscripts with constant coefficients, but also special cases where loop invariant variables are used as coefficients.

A comparison of Tables 9 and 10 for the programs `flo52q(TF)`, `arc2d(SR)`, and `trfd(TI)` shows that they all have a minimal loss of parallelism when serializing subroutine calls and eliminating interprocedural parallelism. In fact, KAP/Concurrent, which currently does not have interprocedural analysis capabilities detects much of the parallelism in these programs. The conclusion reached is as one would expect: parallelizing compilers that do not perform interprocedural analysis work effectively only when the programs are easy to analyze and do not require interprocedural parallelism.

When Tables 9 and 10 are compared, the four programs, `track(MT)`, `spec77(WS)`, `dyfesm(SD)`, and `adm(AP)`, show a large loss of average parallelism. These programs also correspond to instances where KAP was unable to automatically exploit any meaningful parallelism.

Two other general observations should be made about the dynamic result on the Perfect Benchmarks. One is that in many cases the constant and GCD tests by themselves are not sufficient to extract a significant fraction of the ideal parallelism. The second observation is somewhat 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 loop parallelism.

For the dynamic analysis of EISPACK and LAPACK subroutines, we used the drivers and input datasets distributed with the routines. The results are presented in Tables 11, 12, 13, and 14. From examining these results, it is obvious that libraries have slightly different characteristics than do applications. For example, we see that for many of the subroutines we do see a slight performance change when applying the Omega test. Again, as with the Perfect Benchmarks, we see that the constant and GCD tests are not sufficient to uncover the available parallelism.

5 Conclusions

In this paper, we have evaluated three classes of important dependence analysis tests using both static and dynamic measurements. These classes are the constant and GCD tests, variants of Banerjee's test, and tests based on integer programming.

The dynamic evaluation was necessary because the statically computed 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 static infor-

TABLE 11
AVERAGE INTERPROCEDURAL PARALLELISM IN EISPACK CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
cgtest	37.9	36.1	32.0	7.7
chtest	5.0	5.0	5.0	4.5
rbltest	3.5	2.0	2.0	2.0
rggtest	7.6	4.6	4.4	1.6
rgtest	32.3	30.1	29.8	8.0
rltest	16.0	13.4	13.4	3.1
rsbtest	9.2	5.6	5.6	5.5
rsgabtest	6.0	5.9	5.7	4.5
rsgbatest	6.2	6.1	5.8	4.5
rsgtest	5.8	5.7	5.6	4.8
rsptest	6.0	6.0	6.0	5.1
rstest	2.8	2.7	2.6	1.7
rsttest	2.6	2.6	2.6	2.6
rttest	3.2	3.2	3.2	3.1

TABLE 12
AVERAGE INTRAPROCEDURAL PARALLELISM IN EISPACK CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
cgtest	37.9	5.4	5.0	1.4
chtest	5.0	2.1	2.1	1.5
rbltest	3.5	2.0	2.0	2.0
rggtest	7.6	4.5	4.3	1.6
rgtest	32.3	4.6	4.6	1.4
rltest	16.0	7.3	7.3	1.7
rsbtest	9.2	2.0	2.0	1.5
rsgabtest	6.0	2.6	2.1	1.3
rsgbatest	6.2	2.8	2.3	1.3
rsgtest	5.8	2.3	2.1	1.4
rsptest	6.0	2.7	2.7	1.7
rstest	2.8	2.5	2.4	1.5
rsttest	2.6	1.5	1.5	1.3
rttest	3.2	1.6	1.6	1.3

TABLE 13
AVERAGE INTERPROCEDURAL PARALLELISM IN LAPACK CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
sgeptim	11.6	10.7	10.6	2.0
sneptim	8.6	8.4	7.9	1.7
sseptim	37.3	35.3	35.3	3.0
ssvdtim	23.2	22.0	22.0	6.6
sband	109.2	8.5	8.5	1.9
sblasa	10.2	7.3	7.2	1.2
sblasb	28.6	23.2	20.3	1.8
sblasc	14.1	10.2	9.8	1.7

TABLE 14
AVERAGE INTRAPROCEDURAL PARALLELISM IN LAPACK CONSTRAINED BY STATIC DEPENDENCE ANALYSIS

Benchmark	Ideal test	Constant and GCD tests followed by		Constant and GCD tests only
		Omega test	Banerjee's test	
sgeptim	11.6	8.9	8.9	1.8
sneptim	8.6	6.7	6.4	1.3
sseptim	37.3	33.7	33.7	3.0
ssvdtim	23.2	10.7	10.7	4.9
sband	109.2	5.1	5.1	1.1
sblasa	10.2	7.0	6.8	1.2
sblasb	28.6	16.6	15.1	1.5
sblasc	14.1	9.4	9.0	1.5

mation is useful as a means to determine areas in which to concentrate further research.

It has been shown that the unbounded loop test is a practical implementation that can be used to replace the published implementations of Banerjee's rectangular and trapezoidal tests. The main advantage of the unbounded loop test is that it has semisymbolic properties. These properties allow it to be applied in cases where a full symbolic implementation of the standard Banerjee inequalities might be too expensive.

The Omega test had an important effect on the potential dependences that are not broken or conclusively proven by the constant, GCD, or Banerjee's test. In fact, the Omega test proved dependence in 70% of these leftover cases. Such proven dependences, with exact dependence distances in some cases, may prove valuable in transformations not studied here, such as array privatization and loop restructuring for register assignment and locality enhancement.

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 results of the dynamic evaluation shows that the Omega test does not significantly improve the average parallelism over the parallelism exposed by Banerjee's inequalities on the Perfect Benchmarks, and it only minimally improves the linear algebra routines. These results, however, could change if the conditions of the experiment were different. For example, if the compiler did a better job of induction variable substitution and symbolic value propagation, and if linear recurrence substitution were applied, the outcome of the analysis could be different. The Omega Project Software has capabilities for symbolic value propagation that interfaces naturally with the Omega test, but we did not use it in this study. Also, the results are a function of the programs analyzed and, for programs with a more complex structure, the impact of the Omega test could be much greater.

One important potential application of our evaluation technique is the ability to locate the exact places in the source code that are causing difficulty for the compiler. These locations might be useful to programmers as targets for hand transformations, or to compiler writers for identifying limitations in analysis and translation techniques.

ACKNOWLEDGMENTS

The research described was supported by Army Contracts No. DABT63-92-C-003 and DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the U.S. Army or the U.S. Government. This work was also supported by the NASA Ames Research Center under Grant No. NCC 2-559 (DARPA), and by a donation from MIPS Computer Systems.

REFERENCES

[1] J.R. Allen, "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations," PhD thesis, Rice Univ., Apr. 1983.

[2] U. Banerjee, "Speedup of Ordinary Programs," PhD thesis, Univ. of Illinois at Urbana-Champaign, Oct. 1979.

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

[4] W. Blume and R. Eigenmann, "An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks," *Proc. Int'l Conf. Parallel Processing*, pp. II.233–II.238, Aug. 1994.

[5] W. Blume and R. Eigenmann, "The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions," *Proc. Supercomputing '94*, pp. 528–537, Nov. 1994.

[6] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," *IEEE Parallel and Distributed Technology*, vol. 2, no. 3, pp. 37–47, Fall 1994.

[7] D.-K. Chen, "MaxPar: An Execution Driven Simulator for Studying Parallel Systems," Master's thesis, Univ. of Illinois at Urbana-Champaign, Oct. 1989.

[8] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer Performance Evaluation and the Perfect Benchmarks," Tech. Report 965, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing, Research & Development, Mar. 1990.

[9] R. Cytron, "DOACROSS: Beyond Vectorization for Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. 836–844, Aug. 1986.

[10] G. Goff, K. Kennedy, and C.-W. Tseng, "Practical Dependence Testing," *SIGPLAN Notices, Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation*, vol. 26, no. 6, pp. 15–29, June 1991.

[11] X. Kong, D. Klappholz, and K. Psarris, "The I Test: A New Test for Subscript Data Dependence," *Proc. 1990 Int'l Conf. Parallel Processing*, pp. II204–II211, Aug. 1990.

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

[13] Z. Li, P.-C. Yew, and C.-Q. Zhu, "Data Dependence Analysis on Multi-dimensional Array References," *Proc. Third ACM Int'l Conf. Supercomputing*, pp. 215–224, June 1989.

[14] D.E. Maydan, J.L. Hennessy, and M.S. Lam, "Efficient and Exact Data Dependence Analysis," *SIGPLAN Notices, Proc. ACM SIGPLAN '91 Conf. Programming Language Design and Implementation*, vol. 26, no. 6, pp. 1–14, June 1991.

[15] D.A. Padua, D.J. Kuck, and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *Special Issue on Parallel Processing, IEEE Trans. Computers*, vol. 29, no. 9, pp. 763–776, Sept. 1980.

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

[17] K. Psarris, D. Klappholz, and X. Kong, "On the Accuracy of the Banerjee Test," *J. Parallel and Distributed Computing*, vol. 12, no. 2, pp. 152–157, June 1991.

[18] W. Pugh, "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis," *Comm ACM*, vol. 35, no. 8, pp. 102-114, Aug. 1992.

[19] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization," *SIGPLAN Notices Proc. SIGPLAN'95 Conf. Programming Language Design and Implementation*, vol. 30, no. 6, pp. 218-232, June 1995.

[20] W. Rudin, *Principles of Mathematical Analysis*. International Series in Pure and Applied Mathematics, McGraw-Hill, 1964.

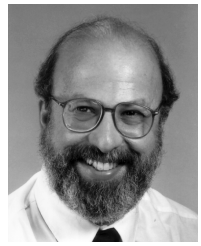
[21] H.M. Salkin and K. Mathur, *Foundations of Integer Programming*. North-Holland, 1989.

[22] Z. Shen, Z. Li, and P.-C. Yew, "An Empirical Study of FORTRAN Programs for Parallelizing Compilers," CSRD Report No. 983, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing, Research & Development, Apr. 1990.

[23] P. Tu and D. Padua, "Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers," *Proc. Ninth ACM Int'l Conf. Supercomputing*, pp. 414–423, July 1995.

[24] M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *Int'l J. Parallel Programming*, vol. 16, no. 2, p. 137, Apr. 1987.
[25] M. Wolfe and C.-W. Tseng, "The Power Test for Data Dependence," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 5, pp. 591–601, Sept. 1992.

Paul M. Petersen received his PhD and MS degrees in 1993 and 1989, respectively, from the University of Illinois at Urbana-Champaign, and his BS in computer science in 1986 from the University of Nebraska-Lincoln. Dr. Petersen is a lead developer for parallel processing at Kuck & Associates, Inc. His research interests include parallelizing compilers, dependence analysis, and dependence profiling techniques for optimization.



David A. Padua (M'88–SM'91) received the Licenciatura in computer science from the Universidad Central de Venezuela in 1973, and the PhD degree from the University of Illinois at Urbana-Champaign in 1980. From 1981 to 1984, he was with the Department of Computer Science at the Universidad Simon Bolivar, Venezuela. In 1985, he returned to the University of Illinois where, from 1990 to 1993, he was associate director for software at the Center for Supercomputing Research and Development and is now a professor of computer science. Dr. Padua has published more than 60 papers on different aspects of parallel computing, including machine organization, parallel programming languages and tools, and parallelizing compilers. His current research focuses on the experimental analysis of parallelizing compilers and on the development of the techniques needed to make these compilers more effective. In 1992, he received the Xerox Award for Faculty Research from the College of Engineering of the University of Illinois. He is a coorganizer of the workshops on Languages and Compilers for Parallel Computing, which have been held annually for the past eight years. Dr. Padua served as program committee chair of the 1990 and 1995 ACM Symposia on Principles and Practice of Parallel Programming. He was also program co-chair of the 1990 International Conference on Parallel Processing. He is an editor of the *International Journal of Parallel Programming* and serves on the editorial board of *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, and the *Journal of Programming Languages*. He is a member of the ACM and a senior member of the IEEE.