An Empirical Study of Precise Interprocedural Array Analysis

MICHAEL HIND[1,2], MICHAEL BURKE[2], PAUL CARINI[2], AND SAM MIDKIFF[2]

[1]*State University of New York, New Paltz, NY 12561*
[2]*IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598*

ABSTRACT

In this article we examine the role played by the interprocedural analysis of array accesses in the automatic parallelization of Fortran programs. We use the PTRAN system to provide measurements of several benchmarks to compare different methods of representing interprocedurally accessed arrays. We examine issues concerning the effectiveness of automatic parallelization using these methods and the efficiency of a precise summarization method. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

Effective program parallelization, like any compiler optimization, can benefit from increased precision during its analysis phase. However, increased precision often implies an increase in compilation time and/or storage, forcing a tradeoff between precision and efficiency. If the benefits of increased precision outweigh the degradation in efficiency, a precise analysis should be utilized.

In this article we assess the effectiveness and efficiency of a precise form of interprocedural array analysis for automatic parallelization. Specifically, we examine a method employed to represent the interprocedural accesses of arrays. Using the PTRAN system [1–3], we introduce a number of metrics to help ascertain:

1. How much additional parallelism can be obtained from a precise array access representation?
2. How much time and space overhead is incurred by this technique?

Received November 1992
Revised May 1994

© 1994 by John Wiley & Sons, Inc.

In Section 2 we provide the background for our experiment. Section 3 describes our precise approach and how it differs from previous approaches. In Section 4 we present our experiment and discuss the results. Section 5 describes related work and Section 6 contains our conclusions and discusses future work.

2 BACKGROUND

Traditionally, compilers have processed programs at the subroutine level. In the absence of subroutine calls, standard intraprocedural analysis techniques [4, 5] can be applied. However, due to the use of modular programming techniques, programs are often written with multiple subroutines. When an intraprocedural analysis encounters a subroutine call, information regarding how the called routine accesses its parameters and global variables is absent. Without this information, conservative assumptions must be made. For a parallelizing compiler, this can imply superfluous dependencies that lead to a loss of parallelism.

Thus, it seems imperative that as much information as possible be captured regarding the side effects of subroutine calls. Interprocedural analysis attempts to provide this information.

Procedure integration or inlining can be viewed as an alternative to interprocedural analysis. When inlining is performed, the body of a called subroutine is substituted for the call statement with appropriate changes made to the naming of the formal parameters. Although selectively performing inlining can be beneficial, the cost of an enlarged program renders it infeasible as a general solution to handling all subroutines calls [6, 7]. Thus, inlining is used as a complement, rather than an alternative, to interprocedural analysis. The relationship between inlining and our precise summary method is discussed in Section 3.1.

Traditionally, to determine the side effects of a call statement, two prior analyses of the called routine are performed. For definitions, a flow-insensitive analysis is computed for the routine, recording nonlocal variables that may be defined. In contrast, to determine what uses should be created by a call, a flow-sensitive analysis is employed to find upward-exposed uses (a use on a definition-free path from the subroutine entry) of nonlocal variables in the called routine [8]. A flow-sensitive analysis of definitions, which can determine which variables must be defined, can be used to supplement the flow-insensitive analysis.

The results of these side-effect analyses are represented by two sets for each routine. The *PMOD*(*P*) set contains all global variables and parameters of routine *P* that may be defined. The *PUSE*(*P*) set contains all global variables and parameters of routine *P* that have an upward-exposed use.

This approach is illustrated in Figure 1. As subroutine P contains definitions of the formal parameter A and the global B, calls to P assume that both of these variables are modified ($PMOD(P) = \{A, B\}$). Although both variables are referenced in subroutine P, only A is upward exposed with respect to the subroutine entry; no definition-free path exists from the subroutine entry to the use of B. Thus, a use is created for A, but not B, at the call site of P ($PUSE(P) = \{A\}$).

Consider the example in Figure 2 where A and B are arrays. Because an array access only references one element of the array, array definitions are treated as preserving, i.e., they do not kill any definitions that reach them. Thus, the use of B is viewed as upward exposed in P.

```
                        Subroutine P(A)
  Common // B             Common // B
      :                         = A
  Call P(A)                   B =
      :                       A = B
                        Return
                        End
```

$$PMOD(P) = \{A, B\}$$
$$PUSE(P) = \{A\}$$

FIGURE 1 Scalar interprocedural Mod and Use.

In the interest of efficiency, classical interprocedural analysis represents array accesses by treating them in the same manner as scalars. Thus, it regards an access to an element of the array as an access to the whole array. Although this method retains efficiency, it suffers a loss of precision.

For example, in Figure 2 only the first element of A is used, while a proper subset (1, . . . , 100) of the elements are defined. Likewise, parts of B are neither modified (odd elements) nor referenced (elements > 100) in P. Because A and B are arrays, simply stating that they are modified or used disregards subscript information describing which part of the array is accessed.

To address the loss in precision of the classical approach, several approaches have been suggested to represent portions of an accessed array. These techniques differ in the amount of precision they provide, as well as the storage and time required in processing the suggested representations. The spectrum of Figure 3 summarizes these

```
                          Subroutine P(A)
  Common // B(200)          Common // B(200)
  Dimension A(200)          Dimension A(200)
        :                        = A(1)
                            Do 10, i = 1, 100
  Call P(A)                      B(2 * i) =
        :                        A(i) = B(i)
                            10 Continue
                          Return
                          End
```

$$PMOD(P) = \{A(*), B(*)\}$$
$$PUSE(P) = \{A(*), B(*)\}$$

FIGURE 2 Classical array interprocedural Mod and Use.

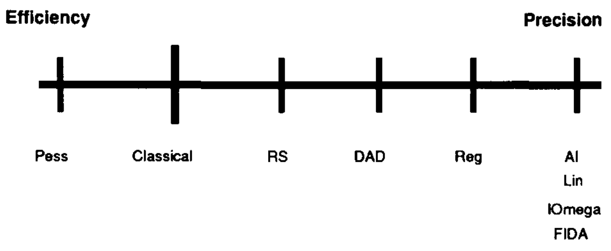


FIGURE 3 The interprocedural array summary spectrum (spacing is not significant).

methods. Movement to the right on the spectrum represents improved precision as well as diminished efficiency. The following is a list of the techniques, with a brief description of each in terms of precision:

1. RS—Regular Sections [9–12]: Several variants have been described, some of which include strides and bounds information using triplet notation. Others allow for diagonal references and triangular sections.
2. DAD—Data Access Descriptors [13, 14]: More general than all RS variants because trapezoidal shapes can be represented.
3. Reg—Regions [15]: Allows more general shapes than DAD by using linear inequalities to describe the shape's boundaries.
4. AI—Atom Images [9, 16, 17]: Represents full subscript information for each dimension as a linear combination of iteration variables and formal parameters. Loop bounds are also retained.
5. Lin—Linearization [18]: Similar to AI except all subscripts are linearized into one dimension.
6. IOmega—Interprocedural Omega Test [19]: Full subscript information is captured in the form of an integer programming projection so that the Omega exact dependence test [20] can be applied. Although multiple projections are merged into a single projection, the size of the integer projection is increased by adding extra variables.
7. FIDA—Full Interprocedural Dependence Analysis [21]: Combines AI and Lin techniques.

In addition to these techniques, and the classical technique just discussed, we include a pessimistic approach in our spectrum, which performs no interprocedural analysis. For correctness, it assumes that each routine modifies and uses all parameters and global variables. Although this scheme is imprecise, it can be highly efficient because no summary information needs to be recorded. In fact, most production compilers perform this type of analysis by default. Furthermore, this method must be selectively employed when some routines of a program are not available for analysis.

To the right of the pessimistic approach is the classical mod/exposed-use approach utilized for scalars [8] and described above. In our experiment we compare these two approaches with a more precise, but less efficient, approach called FIDA [21]. An overview of FIDA is given in Section 3.

A number of advanced techniques (RS, DAD, Reg) lie between classical and FIDA on the spectrum. These techniques offer more precision (at the cost of less efficiency) than the classical approach, yet they are more efficient (and less precise) than the more precise techniques (AI, Lin, IOmega, FIDA).

The key difference between these two groups of advanced techniques is how they handle multiple accesses to the same array in a routine. Information about each access is retained in full with the precise techniques. For AI, Lin, and FIDA, this information is represented by a list of descriptors. For IOmega it is represented by modifying the projection function. By contrast, the more efficient advanced techniques represent multiple accesses with one descriptor. Thus, no matter how many accesses to a variable are made in a routine, only one descriptor is retained. However, there are two disadvantages to the less efficient techniques. For efficiency, they place more restrictive constraints on the expressiveness of their descriptors than what is employed for intraprocedural array accesses. This results in a less precise representation than is used for intraprocedural accesses. Moreover, the union of two descriptors cannot always be performed precisely (i.e., union is not closed over the descriptors). Representing the union approximately introduces further imprecision.

The FIDA approach combines the functionality of Lin proposed by Burke and Cytron [18] and AI suggested by Li and Yew [16, 17, 22, 23]. It is more precise than these two approaches because it draws from the benefits of both: simultaneity by coupling subscript positions (Lin) and more opportunities for proving independence by recording subscript expressions separately (AI). The distinguishing characteristic between each of these approaches and the previous ones is that multiple access descriptors are not combined, thereby making the union operation closed. Although this improves precision, it also implies that a list of accesses is associated with a call site. The result is that a dependence test of a particular variable between two calls can require $l_1 * l_2$ dependence tests, where l_1 and l_2 are the descriptor list lengths corresponding to the first and second calls, respectively.

3 FIDA

FIDA, like Lin [18] and AI [16, 17], is a precise interprocedural array summary scheme motivated by the information required for standard dependence analysis. Our motivation for developing FIDA is to assess an upper bound on precision and efficiency of array access repreentations. This approach captures the same type of dependence information that is available intraprocedurally. This allows all array accesses to be analyzed in a uniform manner regardless of whether they are intraprocedural or interprocedural. In particular, standard dependence analysis techniques can be employed. The next section describes the information retained in each descriptor. In Section 3.2 we present some of the implementation highlights of FIDA in the PTRAN system, leaving the full details to Hind [21].

3.1 Functionality

As mentioned in Section 2, each nonlocal array access is described by an access descriptor. An access descriptor contains information about: subscripts, loop nests and bounds, and the declared shape of the array. As with intraprocedural dependence analysis in PTRAN, we allow a linear combination of induction variables in the subscripts and loop bounds. To capture the effects of arguments, we also allow a linear combination of unmodified formal parameters in the subscripts and loops bounds, and in the dimension statement defining the shape of the accessed array. When processing a call site, the corresponding arguments will be substituted for these formal parameters.

Consider Figure 4a where subroutine P contains a definition of the array parameter A. When summarizing P, the context of this definition (subscripts, loop nest and bounds, and dimension information) is retained. At a call site of P, this information is propagated, substituting actual parameters for their corresponding formals. This method provides functionally similar information to that obtained from data dependence analysis after inlining. It differs in that only the information of interest is "inlined"; superfluous information (for the purposes of the dependence test) is not collected.

```
                          Subroutine P(A, K)
                            Dimension A(100, 100)
Do 30, i = 2, 100           A(K, 1) =
      Call P(A, i)          Do 20, i = 1, 50

      = A(i-1, 3)              :
30 Continue                     A(K, 2 * i) =

                               :
                            20 Continue
                          Return
                          End
```

(a)

```
Do 30, i1 = 2, 100
   A(i1,1) =
   Do 20, i2 = 1, 50
      A(i1,2 * i2) =
   20 Continue
      = A(i1 - 1,3)
30 Continue
```

(b)

FIGURE 4 FIDA functionality.

Figure 4b represents a functional view of the information that would be present using FIDA. (No code modification is actually performed.) By using FIDA, we can detect that the outer loop surrounding the call in Figure 4a can be executed in parallel. Less precise interprocedural analysis would force serial execution of this loop.

Where the shapes of references are consistent, both Lin and subscript-by-subscript analysis are performed. Furthermore, Lin [18] is employed to handle cases where array dimensions and sizes are not consistent across routines, or where offsets into array arguments are used.

Note that this method allows traditional dependence testing schemes to be employed. In particular, we utilize the Burke-Cytron hierarchical dependence method [18] as well as the following dependence tests: GCD, Banerjee-Wolfe, and trapezoidal Banerjee-Wolfe [24, 25].

3.2 Implementation Highlights

In this section we present a high level description of FIDA (Fig. 5), which is broken into three phases for each routine being analyzed.* A FIDA descriptor is one of two types: access or call site. An access descriptor represents an actual reference (read or write) to the array. A call site descriptor is created when a nonlocal array access exists due to a call site, i.e., an access descriptor exists at a call site.

During the def/use generation phase, definitions (uses) are created at call sites in the classical manner using the *PMOD*(*PUSE*) set. However, when a definition (use corresponds to a variable for which FIDA descriptors exist, this definition (use) is marked as a special FIDA def (use). This maintains the number of definitions (uses) as the same number as in the classical approach, leaving data flow analysis unaffected by FIDA.

A FIDA def (use) is used to communicate with the dependence analysis phase. During this phase, the context of a FIDA descriptor (subscript reference, loop information, and dimension information) may be required. When this is the case, we utilize the FIDA description information by substituting references to formal parameters with their corresponding actuals.

In the PTRAN system, dependence analysis is performed on demand as determined by a cost model of the target architecture. Under this approach only dependencies that will provide useful parallelism if disproven are tested. If breaking a dependence will not result in any useful parallelism, the dependence is not tested. For example, once a loop is marked sequential due to either insufficient granularity or some other dependence that cannot be disproven, dependence analysis of other loop-carried dependencies is not beneficial and is not performed.

This technique increases the efficiency of dependence analysis by eliminating some dependencies from consideration. It is also beneficial in the context of FIDA, as descriptor translation is directly tied to dependence analysis. If dependence analysis information is not required for a particular call site, translation is not performed.

For each routine, *P*, in a bottom-up traversal of the call graph:

1. **Def/Use Generation**
   - **For each call site in *P*:**
     Create a FIDA def (use) for each array argument and global variable if it is in the *PMOD* (*PUSE*) set for the called routine.
2. **Dependence Analysis** (Performed on demand)
   - **If a FIDA def/use is involved:**
     Translate the FIDA (call site or access) descriptor(s) to the call site environment using the appropriate arguments. This may require propagating through multiple call site descriptors.
3. **Summarization**
   - **For each non-local array reference:**
     Create an *access descriptor* (subscript expressions, loop bound and nesting information, and dimension information).
   - **For each call site with a summarized non-local array reference:**
     Create a *call site descriptor* (argument expressions, loop bound and nesting information, and dimension information).
   - **Collect the FIDA descriptors created in the previous two steps into lists associated with each non-local array variable.**

FIGURE 5 An overview of the FIDA algorithm.

This characteristic distinguishes FIDA from all other previous methods. For each routine, translated descriptors are cached to avoid redundant translations.

During the summarization phase the "context" for each nonlocal (formal or global) array access is captured in a FIDA descriptor (access or call site). An access descriptor represents an explicit reference. A call site descriptor represents an implicit reference via a call site.

Callahan [9] states that the amount of summary information can grow exponentially with the depth of the call graph. We avoid this potential exponential increase of storage by postponing the propagation of call site descriptors until the information is required by dependence analysis. Thus, the number of FIDA descriptors can grow (at worst) linearly with respect to the program.

* Currently the FIDA algorithm is limited to Fortran 77 as it does not handle recursion. However, as it is similar to AI, we anticipate that techniques to handle recursion with this approach [16] will apply, as well as those mentioned in Havlak and Kennedy [12].

4 THE EXPERIMENT

The PTRAN parallelization system [1–3] was used for our experiment. In addition to detecting parallelism, PTRAN has also been shown to be a useful vehicle for gathering experimental data [26]. We ran several Fortran benchmarks, varying the levels of interprocedural analysis and recording various metrics.

The benchmarks we ran are:

1. Perfect [27]: The Perfect Club benchmarks are a collection of applications that were contributed by various large system vendors and that have been used to characterize supercomputer performance.
2. SPEC [28]: The System Performance Evaluation Cooperative benchmark programs are designed to establish a fair method of evaluating workstation performance on typical customer applications. The experiment includes members of the Fortran subset of Release 1.
3. LINPACK [29]: The LINPACK library is a collection of linear algebra subroutines. We modified the main subroutines to give values to their parameters if they are used in a dimension statement.

As the environment in which an experiment is performed affects the results obtained, we present an overview of our environment in the next section.

4.1 The Environment

PTRAN takes a Fortran 77 program and automatically detects parallelism, producing a parallel Fortran program. In this section we describe the environment by specifying the target model, the analysis and transformations performed by PTRAN, and two Fortran 77 language issues.

Parallelism Model

The PTRAN target model of parallelism allows loops to be designated as parallel (DOALL) or sequential. In addition to loop-level parallelism, nonloop parallelism is allowed in a ''cobegin. . . coend'' style, with a DAG of sequencing constraints allowed among parallel begin. . . end blocks [30]. IBM Parallel Fortran [31] and PCF [32] are examples of languages that fit our model.

Analysis

The PTRAN system includes a rich collection of program analyses. As a description of these analyses is beyond the scope of this article, we refer the reader to the cited articles for details, and list a summary below:

1. Interprocedural analysis [1]: alias analysis (see "Standard Fortran Versus Fortran Practice" in Section 4), constant propagation, and mod and exposed use
2. Program dependence graph for nonloop parallelism [33]
3. SSA-based data flow analysis [34] and the sparse evaluation graph [35]
4. Demand-driven dependence analysis
5. Dependence tests using the Burke-Cytron hierarchical framework [18]: GCD, Banerjee-Wolfe, Trapezoidal Banerjee-Wolfe [24, 25]
6. Standard intraprocedural analysis: constant propagation, induction variable analysis, loop normalization
7. Static cost analysis for architecture-specific effective parallelization [3]

In Section 4.3 we describe how the cost analysis phase is used in one of the metrics.

Transformations

Privatization is the only transformation (other than constant propagation) implemented in the version of PTRAN used in the experiment.† This fact, combined with our target loop model, implies that only loops that are parallelizable in their original form (with the aid of loop privatization) are marked parallel.

Scalar privatization for loops and nonloops is performed [30, 37]. To enhance the effect of privatization, interprocedural analysis includes flow-sensitive kill information for formal parameters. We also perform array privatization when dependence analysis can prove its legality.‡ This privatization may require run-time support or additional storage to ensure proper "copy out" semantics.

† Although a general loop distribution algorithm has been implemented in PTRAN [36], its interface with the cost model is not complete. Thus, it is not included in this experiment.

‡ This is not as powerful as the element-level data flow analysis for arrays previously described [38–41].

Standard Fortran Versus Fortran Practice

The benchmarks we measure are written in Fortran 77. Although the Fortran 77 standard does not allow them, two well-known programming practices appear in these benchmarks. The first makes use of the underlying storage model most often implemented by Fortran compilers, which allow arrays to exceed their declared bounds. Although this is not legal Fortran 77, it is nevertheless done in practice.

Fortran 77 also prohibits assignment to an interprocedurally aliased parameter or common variable [42]. Several examples in the Fortran 77 benchmarks violate this prohibition.

A parallelizing compiler for Fortran 77 must make a decision whether to recognize the standard or to accept common practices that are prohibited by it. PTRAN handles this problem by defining two switches that can be set to allow either of these features. As these features are present in the test programs we analyze, we allow both features in this experiment.

4.2 Interprocedural Analysis Parameters

The PTRAN system computes classical mod/exposed use interprocedural analysis by default. For our experiments we implemented two additional levels of interprocedural analysis. The levels of interprocedural analysis are:

1. Pessimistic: No interprocedural information is known. To uphold safety, all globals and formal parameters are assumed to be both modified and used by a called routine. Likewise, conservative alias information is assumed (see "Standard Fortran Versus Fortran Practice").
2. Classical: Flow-insensitive mod and flow-sensitive use analysis is performed on each called routine before call sites are processed as described in Section 2.
3. FIDA: The precise scheme for arrays described in Section 3. Classical interprocedural analysis is used for scalars.

4.3 Metrics

Because the goal of the experiment is to measure the effectiveness and efficiency of various approaches, our metrics fall into two categories: those that measure parallelism detection and those that measure compilation overhead. We describe each in the next two sections.

Effectiveness for Parallelization

We utilize two metrics to measure the effectiveness of all levels of interprocedural array analysis and a third metric to measure the effectiveness of FIDA. The first two metrics are the number of parallelized loops and the ideal speedup.

Ideal speedup is a static measure of the parallelized program, which disregards the costs associated with parallelism overhead (startup and management) and assumes an unlimited number of processors. It is found by statically estimating the cost of instructions along the critical path in both sequential and parallel cases and computing the ratio of the two [3]. Therefore, it is an upper bound on the amount of obtainable speedup.

To obtain a more detailed measure of effectiveness, we inspect the results of dependence analysis. Of particular interest are those dependence tests where the information provided by our interprocedural analyses differ—dependence candidates involving call site array accesses. We refer to these candidates as the target dependence candidates. These candidates are used in our third effectiveness metric, the independence success rate, which is defined as the number of target dependence candidates proven independent divided by the number of target dependence candidates.

As dependence testing in our experiment is the same for all forms of interprocedural array analysis, only the precision of the input information can affect its result. In the case of the target dependence candidates, both pessimistic and classical analyses are not precise enough to prove independence. This results in a success rate of 0% for these approaches. In contrast, FIDA can provide enough information to prove independence, making a non-zero success rate achievable. Thus, this metric captures how often precise information is potentially beneficial. Unlike the previous metrics, it is not dependent on the transformations that are performed.

Efficiency

We measure two types of efficiency for FIDA: storage and time. We assess storage efficiency by measuring: the number of access descriptors for formal parameters, the number of access descriptors for common blocks, and the number of call site descriptors.

These metrics give an estimate of the amount of storage required by this technique regardless of whether the information is used or not. Recall that the number of access descriptors corresponds to the total number of definitions and uses to nonlocal arrays in the program. A call site descriptor is created when a nonlocal array is accessed through a call. This number is bounded by the number of call sites in the program.

We capture time efficiency by recording two pairs for each nonlocal array variable, the maximum and average length of all access lists and the examined part of all access lists.

The first pair of metrics describes the amount of information associated with a FIDA def/use. This information is composed of a list of access descriptors linked by call site descriptors. The maximum length represents an upper bound on the amount of translation that can be performed for any definition or use in the program. The average length represents the average upper bound on translation for the definitions and uses of the program.

The second pair of metrics identifies how much of this information is actually processed. As FIDA performs both the translation of arguments to formal parameters and dependence analysis with the resulting information, on demand, the second pair of metrics is a good measure of the time efficiency of our approach.

4.4 Results

In this section we present the results of our experiment using the parameters and metrics described in the previous section.[§] We ran the Perfect, SPEC, and LINPACK benchmarks and report the results in two parts: effectiveness and efficiency. Three programs that are not included are FPPPP (unrelated compilation error) and SPICE (irreducible flow graph) in the SPEC benchmarks, and SPEC77 (storage overflow) in the Perfect benchmarks.

Effectiveness

The second column of Table 1 reports the number of loops in each program. The next three columns describe how many of these loops are parallelized using the three forms of interprocedural analysis. A comparison of the results of the first two forms of analysis seems to suggest an error, as in some cases the difference in the number of parallelized loops is greater than the number of loops with calls. However, recall that the pessimistic analysis does not capture any interprocedural information. Thus, not only must it assume that all call sites modify their arguments and global variables, but also that worst case aliasing exists (see "Standard Fortran Versus Fortran Practice"). As the numbers suggest, this conservative aliasing assumption has a drastic effect on the number of parallelizable loops.

Excluding pessimistic analysis, the levels of interprocedural analysis differ only in how they summarize interprocedurally accessed arrays. As interprocedural accesses arise only at call sites, only loops with calls are affected by whether classical interprocedural analysis or FIDA is performed. Thus, these loops represent an upper bound on the potential increase of parallelized loops due to a more precise interprocedural analysis. The sixth column of Table 1 identifies the number of loops that contain subroutine or function calls. To the right of this column is the number of these loops that are parallelized for the three levels of interprocedural array analysis.

Comparing the classical approach (where arrays are treated like scalars) with FIDA, we see three routines in the LINPACK benchmarks where additional parallel loops are detected: SGEDI, SPODI, and SSVDC. Each of these loops contains calls to the much documented routine SAXPY, where independent columns of a matrix are modified on different loop iterations.[‖]

The number of parallel loops can be a misleading metric, as some loops are more critical to the running time of a program than others. Table 2 presents ideal speedup figures for the three interprocedural analysis techniques. Although some programs exhibited a dramatic increase in the number of parallel loops between pessimistic and classical analysis, the increase in ideal speedup is sometimes more modest. We attribute this to the fact that some loops that are parallelized are not critical to a program's execution. Nevertheless, some programs show a substantial increase (FLO52Q, DYFESM) in ideal speedup when classical interprocedural analysis is used. Once again, the benefit of a precise technique is limited to the

[§] These results correct an earlier version of this article [43].

[‖] A slight modification to the SAXPY code was performed to simulate constant folding of the value returned by the MOD built-in function. Similar modifications were made in [11, 12].

Table 1. Number of Parallel Loops for the Perfect Club, SPEC, and LINPACK Benchmarks

Program	No. Loops	Parallel Loops Pess	Parallel Loops Class	Parallel Loops FIDA	No. Loops w/Calls	Parallel Loops Pess	Parallel Loops Class	Parallel Loops FIDA
Perfect								
ARC2D	219	1	104	104	1	0	0	0
BDNA	219	13	67	67	9	0	0	0
DYFESM	203	9	67	67	21	0	0	0
FLO52Q	186	22	135	135	9	0	7	7
MDG	52	9	11	11	7	0	0	0
MG3	155	2	3	3	12	0	0	0
OCEAN	135	45	73	73	12	0	0	0
QCD2	157	25	87	87	40	0	0	0
TRACK	87	16	39	39	18	0	1	1
TRFD	38	1	1	1	6	0	0	0
SPEC								
DODUC	280	17	220	220	19	0	2	2
MATRIX300	17	2	5	5	11	0	0	0
NASA7	130	3	48	48	8	0	0	0
TOMCATV	19	12	12	12	0	0	0	0
LINPACK								
SGBCO	27	2	8	8	7	0	0	0
SGBFA	13	3	6	6	2	0	0	0
SGECO	24	2	5	5	7	0	0	0
SGEDI	15	1	4	6	4	0	0	2
SGEFA	10	0	3	3	2	0	0	0*
SGESL	10	0	3	3	4	0	0	0
SPBCO	24	3	9	9	7	0	0	0
SPOCO	24	3	9	9	7	0	0	0
SPODI	11	0	3	5	4	0	0	2
SPPCO	24	2	8	8	7	0	0	0
SQRDC	19	0	4	4	4	0	1	1
SSIDI	17	0	3	3	3	0	0	0
SSIFA	13	0	0	0	2	0	0	0
SSPCO	26	2	5	5	3	0	0	0
SSVDC	36	8	11	13	11	0	0	2
STRCO	15	2	8	8	3	0	0	0
STRDI	11	0	3	3	4	0	0	0

* One parallel loop can be found using range analysis on the function call isamax.

three LINPACK programs, two of which show significant improvement.

Table 3 illustrates the effect of FIDA on target dependence candidates, i.e., dependencies involving a call site where the corresponding formal or common block element is an array. This table reports the number of target dependence candidates (CAND) and the number of these proven independent due to the additional information provided by FIDA. The last column gives the success rate. As no subscript information is present using the pessimistic or classical approach, each of these candidates would be classified as a dependence (success rate = 0%).

In 6 of the 32 programs a nonzero success rate is found; dependencies were eliminated solely due to the more precise array access information provided by FIDA. However, in three of these programs, the removal of these dependencies did not result in an increase in parallelism.

In 26 of the 32 programs, using a precise interprocedural analysis technique does not enhance automatic parallelization. This does not imply that automatic parallelization of these programs cannot benefit from precise interprocedural array analysis. By transforming loops with dependencies, parallelization can often be obtained. For example, in Blume and Eigenmann [44] the au-

Table 2. **Ideal Speedup for the Perfect Club, SPEC, and LINPACK Benchmarks**

Program	No. Stmts	Ideal Speedup Pess	Class	FIDA
Perfect				
ARC2D	2,544	1.13	1.95	1.95
BDNA	3,825	3.46	3.87	3.87
DYFESM	2,619	1.05	11.39	11.39
FLO52Q	2,325	1.17	1,322.81	1,322.81
MDG	1,049	1.22	2.23	2.23
MG3	2,550	1.01	1.02	1.02
OCEAN	2,050	1.02	1.49	1.49
QCD2	1,987	1.06	1.23	1.23
TRACK	1,823	1.08	1.13	1.13
TRFD	384	1.04	1.04	1.04
SPEC				
DODUC	5,066	1.34	2.97	2.97
MATRIX300	208	1.11	4.17	4.17
NASA7	773	1.03	1.37	1.37
TOMCATV	225	48.71	48.71	48.71
LINPACK				
SGBCO	400	1.12	1.45	1.45
SGBFA	183	1.09	1.15	1.15
SGECO	346	1.08	1.28	1.28
SGEDI	191	1.09	1.15	37.70
SGEFA	149	1.06	1.10	1.10
SGESL	130	1.09	1.55	1.55
SPBCO	316	1.13	1.18	1.18
SPOCO	295	1.12	1.17	1.17
SPODI	135	1.07	1.13	74.96
SPPCO	314	1.13	1.17	1.17
SQRDC	334	1.13	1.30	1.30
SSICO	544	1.10	1.27	1.27
SSIDI	168	1.10	1.12	1.12
SSIFA	249	1.06	1.11	1.11
SSPCO	603	1.12	1.29	1.29
SSVDC	612	1.09	1.34	1.73
STRCO	217	1.28	3.86	3.86
STRDI	154	1.12	1.18	1.18

thors show significant speedup in ARC2D by performing some sophisticated transformations by hand. To perform these transformations automatically, precise interprocedural analysis is usually required. Simple transformations such as loop distribution can also benefit from precise information [17].

FIDA Efficiency

In this section we present efficiency results for FIDA using the metrics described in "Efficiency" in Section 4. The metrics concerning space are given in Table 4: the number of access descriptors and the number of call site descriptors. Access descriptors are divided into accesses of formal parameters (FP) and common blocks (CB).

Recall that an access descriptor is created for each access to a nonlocal array. A call site descriptor is created for each call site that is associated with at least one FIDA def/use. These descriptors, which are created regardless of whether they are used, represent the amount of space overhead for FIDA. The size of each access descriptor is dependent on the number of dimensions, the number of formal parameters occurring in the descriptor, and the depth of the loop nest. The size of the call site descriptor is dependent on these characteristics as well as the number of arguments in the call.

The number of access descriptors does not necessarily correlate to the program size. For example, the ratios of statements to access descriptors in ARC2D and OCEAN differ by about a factor of 6. Furthermore, the proportion between formal parameter and common block descriptors varies widely. This proportion is an attribute of the method of data communication between subroutines. In ARC2D, an average of almost seven formal parameters per routine is found. In DYFESM, where common blocks are more prevalent, this ratio is less than one [26].

Whereas Table 4 represents information overhead, Table 5 illustrates how this information is used. In the second and third columns of Table 5 we capture the access descriptor list length associated with a particular formal parameter or common block. We report the maximum and average lengths. They do not correspond to any additional storage (except the pointer required to link them together), but do capture the magnitude of information associated with each nonlocal array.

The large list length associated with the Perfect program MG3 requires explanation. Through a chain of calls, portions of an array of 60,000 elements are passed through many routines. Each routine accesses parts of the array and calls several other routines that also access it. At the end of these call chains is a routine, CPASSM, which makes 208 references to the array. As CPASSM is called eight times by each of several routines, the list of descriptors grows quickly.

As this list comprises several duplicate sublists, each of which contains a potentially unique call site descriptor, it does not require a lot of storage. Thus, no storage or performance penalty is paid for this excessive size, unless it is examined. Moreover, a simple optimization can be per-

Table 3. FIDA Independence Success Rate for the Perfect Club, SPEC, and LINPACK Benchmarks

	Total			Parameters			Common Blocks		
Program	No. Candidates	No. Independent	Independent Rate (%)	No. Candidates	No. Independent	Independent Rate (%)	No. Candidates	No. Independent	Independent Rate (%)
Perfect									
ARC2D	152	0	0	139	0	0	13	0	0
BDNA	197	0	0	187	0	0	10	0	0
DYFESM	164	0	0	108	0	0	56	0	0
FLO52Q	54	0	0	34	0	0	20	0	0
MDG	46	0	0	44	0	0	2	0	0
MG3	64	0	0	58	0	0	6	0	0
OCEAN	1,048	0	0	1,045	0	0	3	0	0
QCD2	177	22	12	142	20	14	35	2	6
TRACK	145	0	0	107	0	0	38	0	0
TRFD	10	0	0	8	0	0	2	0	0
SPEC									
DODUC	245	30	12	102	0	0	143	30	21
MATRIX300	46	0	0	46	0	0	0	0	0
NASA7	70	0	0	41	0	0	29	0	0
TOMCATV	3	0	0	3	0	0	0	0	0
LINPACK									
SGBCO	76	0	0	76	0	0	0	0	0
SGBFA	10	0	20	10	2	20	0	0	0
SGECO	75	0	0	75	0	0	0	0	0
SGEDI	28	21	75	28	21	75	0	0	0
SGEFA	7	0	0	7	0	0	0	0	0
SGESL	7	0	0	7	0	0	0	0	0
SPBCO	65	0	0	65	0	0	0	0	0
SPOCO	65	0	0	65	0	0	0	0	0
SPODI	31	24	77	31	24	77	0	0	0
SPPCO	61	0	0	61	0	0	0	0	0
SQRDC	11	1	9	11	1	9	0	0	0
SSICO	87	0	0	87	0	0	0	0	0
SSIDI	35	0	0	35	0	0	0	0	0
SSIFA	9	0	0	9	0	0	0	0	0
SSPCO	85	0	0	85	0	0	0	0	0
SSVDC	31	6	19	31	6	19	0	0	0
STRCO	19	0	0	19	0	0	0	0	0
STRDI	8	0	0	8	0	0	0	0	0

formed to prevent this list from growing this large without a loss in precision [21].

As dependence analysis is performed on demand, an element on the list is inspected only if all preceding elements have proven independence. The last two columns of Table 5 record the number of list elements inspected. Notice that even though some programs have a large list maximum, the length of the list that is actually inspected is usually small.

This result is consistent with the effectiveness results reported in the previous section. Once independence cannot be proven for a FIDA reference, whatever remains of its list is not translated or tested. As the previous section showed that few programs exhibited an increase in parallelism using FIDA, a limited amount of list inspection is expected. This illustrates an important advantage of this approach when parallelization is the goal: much of the overhead—list traversal and translation—is incurred only when it may be beneficial. When independence cannot be proven, the compilation performance penalty is negligible.

An inspection of QCD2 explains the large list length examined. In addition to the maximum list of 193, list of 108 elements exists and is fully examined. This program contains a subroutine (MULT) with three array parameters of size 18 (A,B,C). In the subroutine, C is computed as a function of A and B. Although this computation could have been performed in a nest of loops, it was written as 18 assignment statements, each of which contained six uses to both arrays A and B. As each of these references generates an access descriptor, 108 descriptors are created for each of the arrays A and B.

This routine is then called twice within a loop (in routine ROTMEA). The first column of an 18 by 2 matrix is written by the first call, and the second column is written in the second call. As no overlap exists between these two calls, each of the 108 references of both lists is translated and tested. Although this results in an exorbitant number of dependence tests, a simple program transformation could reduce both lists to one element. This transformation would not only reduce the number of translations and dependence tests (by a factor of 108 to 2 and 1, respectively), but also reduce

Table 4. FIDA Storage Efficiency for the Perfect Club, SPEC, and LINPACK Benchmarks

		No. Descriptors				
		Access				Total
	No.				Call	Call
Program	Statements	Parm	CB	Total	Site	Sites
Perfect						
ARC2D	2,544	1,449	177	1,626	35	53
BDNA	3,825	1,015	694	1,709	74	87
DYFESM	2,619	154	512	666	98	127
FLO52Q	2,325	665	210	875	13	94
MDG	1,049	230	37	267	20	34
MG3	2,550	1,590	16	1,606	42	55
OCEAN	2,050	189	18	207	37	242
QCD2	1,987	495	152	647	41	109
TRACK	1,823	501	284	785	22	68
TRFD	384	41	14	55	1	13
SPEC						
DODUC	5,066	130	2,109	2,239	42	117
MATRIX300	208	9	0	9	10	27
NASA7	773	76	374	450	17	24
TOMCATV	225	1	0	1	0	8
LINPACK						
SGBCO	400	76	0	76	3	20
SGBFA	183	38	0	38	0	3
SGECO	346	73	0	73	3	20
SGEDI	191	52	0	52	0	4
SGEFA	149	38	0	38	0	3
SGESL	130	32	0	32	0	4
SPBCO	316	59	0	59	1	18
SPOCO	295	59	0	59	1	18
SPODI	135	32	0	32	0	4
SPPCO	314	59	0	59	1	18
SQRDC	334	79	0	79	0	9
SSICO	544	119	0	119	7	27
SSIDI	309	70	0	70	0	14
SSIFA	249	44	0	44	0	7
SSPCO	603	119	0	119	7	18
SSVDC	612	91	0	91	0	24
STRCO	217	40	0	40	0	8
STRDI	154	32	0	32	0	4

the list size, and hence, the number of access descriptors from 108 to 1.

Consider the results for LINPACK in Table 5. In 2 of the 17 routines (SGEDI and SPODI), the average examined list length is relatively close to the average list length. This contrasts to the other programs, where the average length examined is close to one. Tables 1 to 3 show that independence and new parallelism are detected for these two routines although it is not for the other 16. Once again, the extra processing implied by examining the access lists is only paid when it might be beneficial.

To further assess the frequency of dependence tests involving interprocedurally accessed array references, we provide two additional tables. Table 6 records the number of interprocedural dependence candidates involving arrays, partitioning these into arrays passed by parameters and arrays that reside in common blocks.

The second column reports the total number of dependence candidates tested in our demand-driven approach. The third and fourth columns give the number of dependence candidates for formal parameters and common blocks, respec-

Table 5. FIDA Performance Efficiency for the Perfect Club, SPEC, and LINPACK Benchmarks

	List Length		Examined List Length	
Program	Maximum	Average	Maximum	Average
Perfect				
ARC2D	305	13.7	1	1.0
BDNA	761	11.4	15	1.1
DYFESM	16	3.9	8	1.2
FLO52Q	88	8.3	3	1.1
MDG	243	11.6	10	1.9
MG3	6,682	475.7	361	6.5
OCEAN	56	7.8	1	1.0
QCD2	193	29.3	193	5.7
TRACK	82	9.7	15	2.9
TRFD	2	1.2	1	1.0
SPEC				
DODUC	140	19.2	91	2.8
MATRIX300	24	14.5	1	1.0
NASA7	97	24.7	17	1.6
TOMCATV	1	1	1	1.0
LINPACK				
SGBCO	19	7.4	2	1.0
SGBFA	7	6.3	6	2.0
SGECO	17	7.3	3	1.1
SGEDI	7	6.0	6	5.2
SGEFA	7	6.4	2	1.1
SGESL	7	6.4	3	1.3
SPBCO	8	7.2	2	1.0
SPOCO	8	7.2	3	1.2
SPODI	7	6.1	6	5.3
SPPCO	8	7.2	3	1.1
SQRDC	13	7.4	6	1.3
SSICO	8	7.0	1	1.0
SSIDI	9	7.5	1	1.0
SSIFA	6	5.4	1	1.0
SSPCO	8	7.0	1	1.0
SSVDC	13	6.5	7	2.4
STRCO	8	7.2	1	1.0
STRDI	7	6.5	1	1.0

tively, duplicating these columns from Table 3. The percentage of interprocedural arrays tested is small. This illustrates that the early focus on intraprocedural dependence analysis has been justified.

To better judge the magnitude of the FIDA success ratios, Table 7 reports the success ratios for dependence tests with at least one FIDA reference and those with no FIDA references.

4.5 Discussion

The results of the previous two sections illustrate two points: (1) Precise interprocedural analysis, alone, is generally not enough to improve automatic parallelization. (2) Parallelization under FIDA is a "pay for what you get" technique and therefore can be efficient.

The sophisticated interprocedural and intraprocedural analyses used in this experiment did not dramatically affect program parallelization. From this, we do not feel that one may conclude there is no reason to perform a precise interprocedural analysis. In fact, recent work [44, 45] has shown that advanced transformations can significantly improve parallelization and has called for precise interprocedural analysis information. In particular, we feel that loop distribution and array

Table 6. Dependence Tests Using Interprocedurally Accessed Arrays

Program	Total Candidates	IPA Arrays	
		Parm	Common
Perfect			
ARC2D	7,771	139	13
BDNA	5,937	187	10
DYFESM	5,475	108	56
FLO52Q	6,012	34	20
MDG	1,380	44	2
MG3	3,567	58	6
OCEAN	4,929	1,045	3
QCD2	4,955	142	35
TRACK	2,796	107	38
TRFD	303	8	2
SPEC			
DODUC	6,346	102	143
MATRIX300	116	46	0
NASA7	3,330	41	29
TOMCATV	272	3	0
LINPACK			
SGBCO	312	76	0
SGBFA	125	10	0
SGECO	264	75	0
SGEDI	173	28	0
SGEFA	98	7	0
SGESL	76	7	0
SPBCO	270	65	0
SPOCO	245	65	0
SPODI	121	31	0
SPPCO	265	61	0
SQRDC	199	11	0
SSICO	413	87	0
SSIDI	169	35	0
SSIFA	153	9	0
SSPCO	477	85	0
SSVDC	390	31	0
STRCO	187	19	0
STRDI	100	8	0

privatization would benefit greatly from our analysis. Other interprocedural transformations have also been suggested [46].

If a precise form of analysis is required to perform these transformations, the efficiency of such an analysis is paramount. Due to its demand-driven implementation, FIDA is reasonably efficient in the context of automatic parallelization.

5 RELATED WORK

In previous work [9–19], emphasis has been placed on improving the precision of interprocedural analysis for array accesses. Although experimental results appear in some of these articles, most of it has been limited to the parallelization of LINPACK with little empirical results concerning efficiency.

Li and Yew [16, 17] evaluate the effectiveness of their approach by reporting the number of parallel loops containing calls using the LINPACK benchmarks. No information is presented pertaining to the efficiency of their approach except stating that it runs 2.6 times faster than the Paraphrase implementation of [15].

Havlak and Kennedy [11, 12] evaluate their implementation of bounded regular sections using LINPACK and a collection of other programs. They measured the efficiency of their implementation in real-time as part of PFC. They report the number of calls in parallel loops as well as the number of dependencies removed using their approach.

Although a direct comparison with these two works would be illustrative, it is not possible as only Li and Yew report the number of parallel loops with calls without loop distribution. Under this scenario, they report a total of six parallel loops in five LINPACK routines, all but one of which we parallelize.¶ The failure to parallelize the loop in SGEFA is not a result of the precision of dependence analysis, but rather a consequence of PTRAN not being able to evaluate a function ISAMAX at compile-time. Furthermore, in programs SQRDC and SGEDI, we detect an additional parallel loop containing a call. Both of these loops were not parallelized by Li and Yew [17].

The observation that more precise interprocedural analysis alone is not enough for effective parallelization was also made by Irigoin et al. [47] but no experimental numbers were presented. They call for better programming practice as well as new compilation techniques like array privatization.

6 SUMMARY

This work has presented an experiment designed to capture the effectiveness and efficiency of interprocedural analysis of array accesses in the context of parallelization. It has shown that classical interprocedural analysis can provide a significant improvement over pessimistic interproce-

¶ Their original work incorrectly reported two additional loops in SSIFA as being parallel without loop distribution [Li, Personal Communication, 1992].

Table 7. Overall Independence Success Rate for the Perfect Club, SPEC, and LINPACK Benchmarks

	Total			Non-FIDA			FIDA		
Program	No. Candidates	No. Independent	Independent Rate (%)	No. Candidates	No. Independent	Independent Rate (%)	No. Candidates	No. Independent	Independent Rate (%)
Perfect									
ARC2D	7,771	3,608	46	7,619	3,608	47	152	0	0
BDNA	5.937	935	16	5,740	935	16	197	0	0
DYFESM	5,475	2,176	40	5,311	2,176	41	164	0	0
FLO52Q	6,012	3,437	57	5,958	3,437	58	54	0	0
MDG	1,380	93	7	1,334	93	7	46	0	0
MG3	3,567	86	2	3,503	86	2	64	0	0
OCEAN	4,929	297	6	3,881	297	8	1,048	0	0
QCD2	4,955	3,217	65	4,778	3,195	67	177	22	12
TRACK	2,796	981	35	2,651	981	37	145	0	0
TRFD	303	0	0	293	0	0	10	0	0
SPEC									
DODUC	6,346	790	12	6,101	760	12	245	30	12
MATRIX300	116	1	1	70	1	1	46	0	0
NASA7	3,330	2,627	79	3,260	2,627	81	70	0	0
TOMCATV	272	131	48	269	131	49	3	0	0
LINPACK									
SGBCO	312	47	15	236	47	20	76	0	0
SGBFA	125	53	42	115	51	44	10	2	20
SGECO	264	46	17	189	46	24	75	0	0
SGEDI	173	80	46	145	59	41	28	21	75
SGEFA	98	44	45	91	44	48	7	0	0
SGESL	76	28	37	69	28	41	7	0	0
SPBCO	270	67	25	205	67	33	65	0	0
SPOCO	245	68	28	180	68	38	65	0	0
SPODI	121	74	61	90	50	56	31	24	77
SPPCO	265	66	26	204	66	32	61	0	0
SQRDC	199	47	24	188	46	24	11	1	9
SSICO	413	45	11	326	45	14	87	0	0
SSIDI	169	9	5	134	9	7	35	0	0
SSIFA	153	4	3	144	4	3	9	0	0
SSPCO	477	45	9	392	45	11	85	0	0
SSVDC	390	64	16	359	58	16	31	6	19
STRCO	187	65	35	168	65	39	19	0	0
STRDI	100	46	46	92	46	50	8	0	0

dural analysis. It has also shown that a precise analysis (FIDA), without the support of sophisticated transformations, provides a limited benefit over classical analysis.

A demand-driven analysis is more efficient than the corresponding exhaustive analysis when complete information is not needed. When a demand-driven analysis requires interprocedural information, FIDA is an attractive approach. For demand-driven dependence analysis, the performance overhead is commensurate with benefits. Further improvements in efficiency can be obtained by simple optimizations [21].

Although FIDA was developed to test the effects of interprocedural analysis in the context of automatic parallelization, it can also be used in other contexts. Wherever subscript analysis is required, FIDA can be used to capture precise interprocedural information. Some other applications are: analysis to reduce communication costs in distributed memory machines [48–50], automatic data partitioning for distributed memory machines [51], array privatization [38–40], and analysis of locality to benefit cache performance [52, 53]. Determining the effectiveness and efficiency of these applications of FIDA.

ACKNOWLEDGMENTS

The authors thank Fran Allen for her encouragement and support of this work. The PTRAN system served as a useful vehicle for our experiments. We wish to thank all members, past and present, of the PTRAN team. We would also like to thank the referees for their useful comments.

REFERENCES

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, *1987 International Conference on Supercomputing*. New York, NY: Springer-Verlag, 1987, pp. 194–211. (Also published in *J. Parallel Distributed Comput.*, vol. 55, pp. 617–640, 1988.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, J. Ferrante, V. Sarkar, D. Shields, *The 4th International Conference of Supercomputing*. Santa

Clara, CA: International Supercomputing Institute, Inc., 1989, pp. 89–93. Extended Abstract.

[3] V. Sarkar, *Parallel Functional Programming Languages and Compilers.* New York: ACM Press Frontier Series, 1991, pp. 309–391.

[4] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986, pp. 653–660.

[5] K. Kennedy, *Program Flow Analysis: Theory and Applications.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981, pp. 5–54.

[6] S. Richardson, M. Ganapathi, "Interprocedural analysis versus procedure integration," *Information Processing Lett.*, vol. 32, pp. 137–142, 1989.

[7] K.D. Cooper, M.W. Hall, L. Torczon, "Unexpected side effects of inline substitution: A case study," *ACM Lett. Programming Languages Systems,* vol. 1, pp. 22–32, 1992.

[8] J. Banning, "A method for determining the side effects of procedure calls." PhD thesis, Stanford, University, 1978.

[9] D. Callahan, "A global approach to detection of parallelism." PhD thesis, Rice University, Rice COMP TR87-50, April 1987.

[10] D. Callahan, K. Kennedy, 1987 *International Conference on Supercomputing.* New York, NY: Springer-Verlag, 1987. (Also published in *J. Parallel Distributed Comput.*, vol. 55, pp. 517–550, 1988.

[11] P. Havlak, K. Kennedy, *Supercomputing '90.* IEEE Computer Society and ACM SIGARCH. Los Alamitos, CA: IEEE Computer Society Press. 1990, pp. 952–961.

[12] P. Havlak, K. Kennedy, "An implementation of interprocedural bounded regular section analysis." *IEEE Trans. Parallel Distributed Systems,* vol. 2, pp. 350–360, 1991.

[13] V. Balasundaram, K. Kennedy, *SIGPLAN '89 Conference on Programming Language Design and Implementation.* Portland, Oregon: ACM, 1989, pp. 41–53.

[14] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor." *J. Parallel Distributed Comput.*, vol. 9, pp. 154–170, 1990.

[15] R. Triolet, F. Irigoin, P. Feautrier, *SIGPLAN '86 Symposium on Compiler Construction.* Palo Alto, CA: ACM, 1986, pp. 176–185.

[16] Z. Li, "Intraprocedural and interprocedural data dependence analysis for parallel computing," PhD thesis, University of Illinois, CSRD Report No. 910, 1989.

[17] Z. Li, and P.-C. Yew, *ACM/SIGPLAN PPEALS Conference.* New Haven, CT: ACM, 1988, pp. 85–99.

[18] M. Burke, R. Cytron, *SIGPLAN '86 Symposium on Compiler Construction.* Palo Alto, CA: ACM, 1986, pp. 162–175.

[19] P. Tang, *1993 International Conference on Supercomputing.* Tokyo, Japan: ACM, 1993, pp. 137–146.

[20] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications ACM,* vol. 35, pp. 102–115, 1992.

[21] M. Hind, "Full interprocedural dependence analysis." Technical Report, IBM T.J. Watson Research Center, 1994. In preparation.

[22] Z. Li, P.-C. Yew, *International Conference on Parallel Processing.* University Park, PA: Pennsylvania State University Press, 1988, pp. 221–228.

[23] Z. Li, P.-C. Yew, "Program parallelization with interprocedural analysis," *J. Supercomput.*, vol. 2, pp. 225–244, 1988. (Also at the 1988 Workshop on Programming Languages and Compilers for Parallel Computing.)

[24] U. Banerjee, *Dependence Analysis for Supercomputing.* The Kluwer International Series in Engineering and Computer Science. Boston, MA: Kluwer Academic Publishers, 1988, pp. 101–148.

[25] M.J. Wolfe, *Optimizing Supercompilers for Supercomputers.* Cambridge, MA: MIT Press, 1989, pp. 6–53.

[26] M. Burke, P. Carini, "Compile-time measurements of interprocedural data-sharing in Fortran programs." Technical Report RC 17389 76684, IBM - T.J. Watson Research Center, November 1991.

[27] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Waler, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, J. Martin, "The Perfect Club benchmarks: Effective performance evaluation of supercomputers." Technical Report, University of Illinois at Urbana-Champaign - Center for Supercomputing Research & Development, November 1988.

[28] J. Uniejewski, "SPEC benchmark suite: Designed for today's advanced systems." Technical Report, Systems Performance Evaluation Cooperative, Fall 1989. SPEC Newsletter, Volume 1, Issue 1.

[29] J.J. Dongarra, J.R. Bunch, C.B. Moler, G.W. Stewart, *Linpack Users' Guide.* Philadelphia, PA: SIAM Press, 1979, pp. 1.1–11.23.

[30] R. Cytron, M. Hind, W. Hsieh, *SIGPLAN '89 Conference on Programming Language Design and Implementation,* Portland, Oregon: ACM, 1989, pp. 54–68.

[31] IBM, *Parallel Fortran - Language and Library Reference.* Armonk, NY: IBM, March 1988, pp. 1–474. First Edition.

[32] Parallel Computing Forum, "PCF parallel For-

tran extensions," Fortran Forum. New York: ACM Press, vol. 10, 1991, pp. 1–57.

[33] J. Ferrante, K.J. Ottenstein, J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programming Languages Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[34] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, "An efficient method for computing static single assignment form and the control dependence graph," *ACM Trans. Programming Languages Systems*, vol. 14, pp. 451–490, 1991.

[35] J.-D. Choi, R. Cytron, J. Ferrante, *18th Annual ACM Symposium on the Principles of Programming Languages.* Orlando, FL: ACM, 1991, pp. 55–66.

[36] B.M. Hsieh, M. Hind, R. Cytron, *Supercomputing '92,* Minneapolis, MN: IEEE Computer Society Press, November 1992, pp. 204–213.

[37] M. Burke, R. Cytron, J. Ferrante, W. Hsieh, "Automatic generation of nested, fork-join parallelism," *J. Supercomput.* vol. 2, pp. 71–88, 1989.

[38] P. Feautrier, *1988 International Conference on Supercomputing.* St. Malo, France: ACM, 1988, pp. 429–441.

[39] Z. Li, *1992 International Conference on Supercomputing.* Washington, DC: ACM, 1992, pp. 313–322.

[40] D.E. Maydan, S.P. Amarsighe, M.S. Lam, "5th Workshop on Languages and Compilers for Parallel Computing." Technical Report YALEU/DCS/RR-915, Yale University, August 1992.

[41] P. Tu, D. Padua, *6th Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon: Springer-Verlag, 1993, pp. 500–521.

[42] American National Standards Institute: "Programming Language Fortran." Technical Report, ANSI X3.9-1978. American National Standards Institute, January 1978.

[43] M. Hind, M. Burke, P. Carini, S. Midkiff, *3rd Workshop on Compilers for Parallel Computers*, Volume 2. Vienna: University of Vienna, 1992.

[44] W. Blume, R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs." Technical Report, University of Illinois at Urbana-Champaign, May 1992. (To appear in IEEE. Trans. Parallel Distributed Systems.)

[45] R. Eigenmann, J. Hoeflinger, Z. Li, D. Padua, *Languages and Compilers for Parallel Computing.* Santa Clara, CA: Springer-Verlag, 1991, pp. 65–83.

[46] M.W. Hall, K. Kennedy, K.S. McKinley, *Supercomputing '91.* Albuquerque, NM: IEEE Computer Society Press, 1991, pp. 424–434.

[47] F. Irigoin, P. Jouvelot, R. Triolet, *1991 International Conference on Supercomputing.* Cologne, Germany: ACM Press, 1991, pp. 244–251.

[48] K. Knobe, J.D. Lukas, G.L. Steele Jr, "Data optimization: Allocation of arrays to reduce communication on SIMD machines," *J. Parallel Distributed Comput.*, vol. 8, pp. 102–118, 1990.

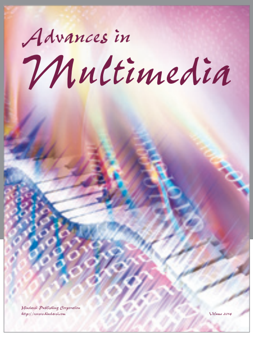
[49] J. Li, M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distributed Systems*, vol. 2, pp. 361–376, 1991.

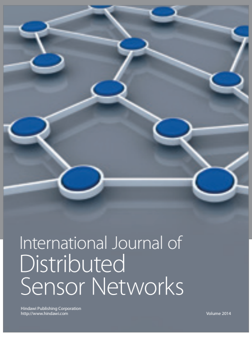
[50] M. Gupta, E. Schonberg, *6th Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon: Springer-Verlag, 1993, pp. 216–233.

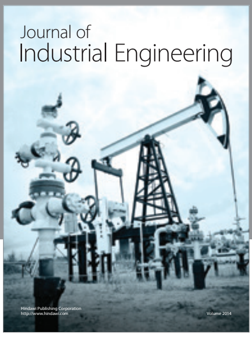
[51] M. Gupta, P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers for multicomputers," *IEEE Trans. Parallel Distributed Systems,* vol. 3, pp. 179–193, 1992.

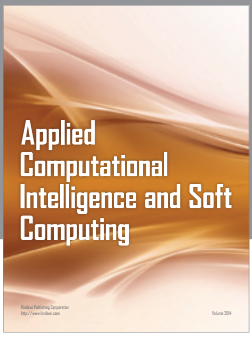
[52] J. Ferrante, V. Sarkar, W. Thrash, Lecture Notes in Computer Science, no. 589, 1991. *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, California, August 1991.

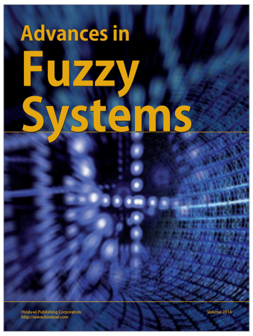
[53] M.E. Wolf, M.S. Lam, *SIGPLAN '91 Conference on Programming Language Design and Implementation.* SIGPLAN. Toronto, Canada: ACM, 1991, pp. 30–44.











Hindawi

Submit your manuscripts at
http://www.hindawi.com