INTERPROCEDURAL PARALLELIZATION ANALYSIS: PRELIMINARY RESULTS

MARY W. HALL
SAMAN P. AMARASINGHE
BRIAN R. MURPHY
SHIH-WEI LIAO
MONICA S. LAM

Technical Report: CSL-TR-95-665

March 1995

Supported in part by DARPA contracts N00039-91-C-0138 and DABT63-91-K-0003, an NSF Young Investigator Award, an NSF CISE postdoctoral fellowship, a fellowship from Intel Corporation, and a fellowship from AT&T Bell Laboratories.

Interprocedural Parallelization Analysis: Preliminary Results

Mary W. Hall
Saman P. Amarasinghe
Brian R. Murphy
Shih-Wei Liao
Monica S. Lam

Technical Report No.: CSL-TR-95-665

March 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
pubs@shasta.stanford.edu

Abstract

This paper describes a fully interprocedural automatic parallelization system for Fortran programs, and presents the results of extensive experiments obtained using this system. The system incorporates a comprehensive and integrated collection of analyses including dependence, privatization and reduction recognition for both array and scalar variables, and scalar symbolic analysis to support these. All the analyses have been implemented in the SUIF (Stanford University Intermediate Format) compiler system, with the aid of an interprocedural analysis construction tool known as FIAT. Our interprocedural analysis is uniquely designed to provide the same quality of information as if the program were analyzed as a single procedure, while managing the complexity of the analysis.

We have implemented a robust system that has parallelized, completely automatically, loops containing over a thousand lines of code. This work makes possible the first comprehensive empirical evaluation of state-of-the-art automatic parallelization technology. This paper reports evaluation numbers on programs from standard benchmark suites. The results demonstrate that all the interprocedural analyses taken together can substantially advance the capability of current automatic parallelization technology.

Key Words and Phrases: automatic parallelization, interprocedural data-flow analysis

Interprocedural Parallelization Analysis: Preliminary Results

Mary W. Hall
California Institute of Technology

Saman P. Amarasinghe   Brian R. Murphy
Shih-Wei Liao   Monica S. Lam
Stanford University

Abstract

This paper presents results of extensive experiments with a fully interprocedural automatic parallelization system for Fortran programs. The system incorporates an integrated collection of interprocedural analyses, including privatization and reduction recognition for both array and scalar variables, and a supporting scalar symbolic analysis. The analyses were implemented in the SUIF (Stanford University Intermediate Format) compiler system, with the aid of an interprocedural analysis construction tool known as FIAT. The analysis framework is uniquely designed to provide results as precise as full inlining, while managing the complexity of the analysis. The implemented system is robust, and has parallelized, completely automatically, loops containing hundreds of lines of code.

This work makes possible the first empirical evaluation that demonstrates positive results from automatic parallelization, particularly interprocedural analysis, presenting measurements of a number of programs from standard benchmark suites. The results demonstrate that a combination of interprocedural analyses can substantially advance the capability of automatic parallelization technology.

1 Introduction

As small to moderate-scale multiprocessors have become increasingly common, the demand for compilers capable of extracting parallelism from existing codes is also increasing. Today's commercial parallelizers tend to be successful only in identifying parallelizable innermost loops. While this may be sufficient for vector and SIMD (Single Instruction Multiple Data) machines, it is inadequate when compiling for multiprocessors. Code outside of innermost loops represents a significant fraction of the execution time and must be parallelized to achieve high performance. Because multiprocessors are capable of following different threads of control, they can exploit such parallelism when available by executing iterations of outer parallel loops. Outer parallel loops provide coarse-grain computations that exhibit very small overhead costs from synchronization relative to the amount of time spent doing useful work in parallel.

If compilers are to successfully locate coarse-grain parallelism, procedure boundaries must not pose a barrier to analysis. One way to eliminate procedure boundaries is to perform inline substitution — replacing each procedure call by a copy of the called procedure — and perform program analysis in the usual way. This is not a practical solution for large programs, which may expand to an unmanageable size. Other techniques for interprocedural parallelization analysis have been much studied [8, 12, 20, 9, 10, 15], but the results have been inconclusive and sophisticated interprocedural analysis has not been adopted in practice.

The primary obstacle to progress in this area has been that effective interprocedural compilers are substantially harder to build than their intraprocedural counterparts. The compiler must be able to handle program features not encountered in an intraprocedural setting, such as parameter passing and variable scoping. In addition, when extending traditional data-flow analysis to the interprocedural setting, we must ensure that the analysis maintains reasonable efficiency without sacrificing too much precision. For example, a simple interprocedural adaptation of iterative analysis might converge very slowly and introduce imprecision by propagating information

---

This research was supported in part by DARPA contracts N00039-91-C-0138 and DABT63-91-K-0003, an NSF Young Investigator Award, an NSF CISE postdoctoral fellowship, a fellowship from Intel Corporation, and a fellowship from AT&T Bell Laboratories.

Technical Report CSL–TR–95–665, Computer Systems Laboratory, Stanford University, March 1995

along *unrealizable paths* — paths in the program representation that do not correspond to executable paths. Finally, interprocedural analysis must be very robust, avoiding gross approximation even when it encounters program constructs that are difficult to analyze; such constructs appear frequently when parallelizing computations that span thousands of lines of code.

This paper presents an interprocedural parallelizing compiler that performs a comprehensive suite of parallelization analyses. Our research goal in developing this system has been to provide parallelization results comparable to what would be possible with full inlining of the program, while managing the complexity of the analysis. In this way, we avoid penalizing the programmer for writing programs in a modular style, in which important computations in the program might span multiple procedures. This paper makes the following contributions:

Extensive empirical evaluation. This paper is the first to provide extensive measurements of a state-of-the-art automatic parallelization system. These are the first results measuring the effectiveness of aggressive interprocedural parallelization analysis on a large collection of real programs. They show that a purely automatic parallelization analysis can be successful at finding the parallelism necessary to obtain good speedups for some programs. This evaluation provides a critical missing piece of information needed to advance the state of the art in parallelization technology.

Interprocedural reductions. This system is the first to incorporate interprocedural reduction recognition. Our algorithm is simple yet powerful enough to recognize reductions to array regions, even in the presence of arbitrarily complex data dependences. Our experiments suggest that interprocedural reduction is very important.

Precise analysis framework. We use a pair of techniques to obtain precise results while managing the complexity of the analysis. First, to obtain path-specific interprocedural information at a procedure, we rely on *selective procedure cloning*, whereby the compiler replicates information for a procedure so that it may be analyzed in light of each distinct calling context. We manage the amount of replication by performing it *selectively*, based on the unique information contributed by a path.

Second, we use *interprocedural interval analysis* to perform data-flow analysis over the entire program with only two passes over the code. This interval-based approach is not only more efficient than interprocedural iterative data-flow analysis, but it also avoids propagating information along unrealizable paths.

Mathematical formulation of array reshapes at procedure calls. A system for interprocedural array analysis must address the common occurrence in Fortran programs of array reshaping across procedure boundaries. We formulate the array reshape problem as a rewrite of a system of linear inequalities in terms of different variables. This approach is more powerful than pattern matching or special-case solutions.

A comprehensive and robust experimental platform. We have an implementation of a large suite of interprocedural analyses that is sufficiently robust to perform this empirical evaluation. The system incorporates a comprehensive and integrated suite of analyses, including array and scalar dependence, privatization and reduction recognition and symbolic analysis of integer variables. In developing a robust platform, we found it necessary to extend existing approaches to meet the demands of compiling real programs. For example, our array privatization algorithm extends the scope of previous implementations by including arrays that need to be initialized before execution. The determination of the specific analysis requirements for this collection of programs provides valuable information to designers of parallelization systems.

The paper is organized as follows. We start by describing briefly other related work in the area. We then introduce the basic interprocedural framework used in the implementations of all our analyses. Due to space constraints, we are able to only include an overview of the analysis on scalar variables (Section 4) and analysis on array variables (Section 5). We then present a series of experimental results on both static and dynamic measurements of contributions of the various analyses, including speedups.

2 Related Work

In the late 1980s, a series of papers presented results on interprocedural parallelization analysis [8, 12, 20]. Their common approach was to determine the sections of arrays that are modified or referenced by each procedure call, enabling parallelization of some loops containing calls whenever each invocation modifies array elements distinct from those that are referenced or modified in other invocations. These techniques were shown to be effective in parallelizing linear algebra libraries. More recently, the FIDA system was developed at IBM to obtain more precise array sections through partial inlining of array accesses [9] (see Section 6).

Irigoin et al. have developed the PIPS system, an interprocedural analysis system that is part of an environment for parallel programming [10]. More recently, PIPS has been extended to incorporate interprocedural array privatization. PIPS is the most similar to our work, but lacks three important features: (1) path-specific interprocedural information such as obtained through selective procedure cloning, (2) interprocedural reductions, and (3) extensive interprocedural scalar data-flow analysis such as scalar privatization. Further, to the best of our knowledge, the PIPS project has never published experimental results on the effectiveness of their system.

The Polaris system at University of Illinois is also currently being developed to advance the state of the art in parallelization technology [2]. The most fundamental difference between our system and Polaris is that Polaris performs no interprocedural analysis, instead relying on *full inlining* of the programs to obtain interprocedural information. The Polaris group has demonstrated that good coverage results (% of the program parallelized) can be obtained automatically (with some hand-modification). Although they report that full inlining is feasible on eight medium-sized programs, this approach will have difficulty parallelizing large loops containing thousands of lines of code. The Polaris group has not published execution-time results on the programs parallelized by their system.

A few commercial parallelizing compilers have initial interprocedural analysis systems. Most notably, the Convex Applications Compiler performs flow-insensitive array analysis and interprocedural constant propagation and obtains some path-specific information through inlining and procedure cloning [15]. However, we know of no commercial system that currently employs any flow-sensitive array analysis, particularly interprocedural array privatization [1].

3 Interprocedural Framework

Interprocedural parallelization depends upon the solution of a large number of interprocedural data-flow analysis problems. These problems share many commonalities. We have utilized FIAT [6], a tool which encapsulates these common features, in combination with the Stanford SUIF compiler to constitute our interprocedural parallelization system. FIAT is an *interprocedural framework*, analogous to traditional data-flow analysis frameworks [11]. A framework is even more important for interprocedural optimization because of the complexity of collecting and managing information about all the procedures in a program.

Although the FIAT system has been described previously [6], we have extended the system to obtain precise flow-sensitive interprocedural results through the combination of two techniques which we now describe. We have also added to the system a mathematical formulation of array reshapes (see Section 5.5) in order to support interprocedural array analysis.

Interprocedural Interval Analysis. Interprocedural analysis is classified as either *flow sensitive* or *flow insensitive*. A flow-sensitive analysis derives effects common to all control flow paths, whereas flow-insensitive analysis represents a conservative approximation of all paths. Whereas a flow-insensitive analysis can ignore control flow within each procedure, flow-sensitive analysis requires that a representation of each procedure's control flow be available during interprocedural analysis. Flow-insensitive analysis is well-understood, can be done efficiently, and is beginning to appear in commercial compilers.

Flow-sensitive analysis is more complex because values flow both within a procedure and between a procedure and its callers. Consider a straightforward interprocedural adaptation of iterative data-flow analysis: analysis is carried out over a program representation called the *supergraph* [16], where individual control flow graphs for the procedures in the program are linked together at procedure call and return points. Iterative analysis over this structure may be slow because information must be propagated through call chains to determine the local effects of a procedure call on its caller. In addition, such analysis may lose precision by propagating information along *unrealizable paths* [17]; the analysis may propagate information from one caller through a procedure and return the information to a different caller.

In our system, we use an interval-based approach which solves the problems of unrealizable paths and slow convergence. By separating a procedure's calling environment from its behavior, we can perform analysis efficiently in two passes through the program. A bottom-up analysis over the call graph (*i.e.,* procedures are analyzed before

---

[1] The reader may have seen speedup results from Applied Parallel Research on a few of the programs presented here. Note that these programs were parallelized with programmer directives that instruct the compiler to ignore dependences it finds on a particular loop.

their callers) produces descriptions of the *behavior* of program regions (regions are basic blocks, loop bodies and loops), in the form of a transfer function. In a top-down analysis over the call graph, data-flow information from the calling environment is applied to these transfer functions to derive the final analysis results for a procedure[2]. Within each procedure, we aggregate information at regions, the basic blocks, loop bodies and loops; the bottom-up phase proceeds from innermost loops to outermost loops, while the top-down phase operates in the opposite way. This corresponds to what is traditionally called interval-based analysis. For reducible flow graphs, a region's behavior is a simple combination of its nested subregion behaviors; irreducible subgraphs are handled by iterative solution of a dataflow problem in the enclosing single-entry region. An application of this interval-based analysis is described in detail in Section 4. (To simplify presentation, we will only describe the analysis for reducible flow graphs in the remainder of the paper; the actual implementation also handles irreducible graphs.)

Selective Procedure Cloning. For procedures invoked on multiple distinct paths through a program, traditional interprocedural program analysis forms a conservative approximation of the information entering the procedure that is correct for all paths. Such approximations can affect the precision of analysis if a procedure is invoked along paths that contribute very different information.

To illustrate the effects of path-specific information on optimization, consider the following example taken from the SPEC89 benchmark `matrix300`.

```
SUBROUTINE SAXPY(...,X,IX,Y,IY)
     REAL X(IX,1), Y(IY,1)
     DO I = 1, N
        Y(1,I) = Y(1,I) + A*X(1,I)
```

The arrays `X` and `Y` have symbolic dimension sizes, inhibiting any optimizations that rely on precise knowledge of array accesses. In calls to `SAXPY`, the value passed to `IX` is either 1 or 100, depending on whether we are accessing `X` or its transpose. `IY` is similarly either 1 or 102. Because their values vary across invocations, traditional techniques assume no knowledge of the values of these dimension variables, resulting in lost precision.

Path-specific interprocedural information has previously been obtained either by inline substitution or by tagging data-flow sets with a path history through the call graph, incurring a data-flow set expansion problem corresponding to the code explosion problem of inlining [19, 16, 18, 7]. To avoid such excessive space usage, we utilize path-specific information only when it may provide opportunities for improved optimization. Our system incorporates *selective procedure cloning*, a program restructuring in which the compiler replicates the analysis results for a procedure to analyze it in the context of distinct calling environments [4]. By applying cloning *selectively* according to the unique data-flow information it exposes, we can obtain the same precision as full inlining without unnecessary replication. The cloning phase corresponds to the partitioning of the calling contexts according to the information they contain before applying the context to the transfer function for a procedure. Cloning is reduced through the use of smart filters that eliminate from the data-flow sets information not relevant to the current optimization, further avoiding unnecessary replication. In final code generation, multiple versions of the "cloned" procedure are only produced if they have different parallelization. Otherwise we merge the cloned information before generating code.

Combining Flow-Sensitive Interval Analysis and Selective Procedure Cloning. The combination of the above two techniques allows us to produce analysis results with the same level of precision as if the program were fully inlined, without resulting in unmanageable code expansion. The precision of this approach depends on the ability to derive transfer functions $\mathcal{T}$ that do not approximate information. For the analyses in our syve accomplished this goal.

4 Scalar Data-Flow Analysis

Scalar data-flow analysis is crucial for parallelizing loops. Analyses of scalar variables in a loop are necessary both to support precise analysis of array accesses and to detect and eliminate scalar dependences.

---

[2] This approach can work on recursive programs as well, by collapsing cycles in the call graph into a single node and iterating within the cycle. [13]

4.1 Support for Array Analysis: Interprocedural Symbolic Analysis

Analyzing array accesses precisely requires analysis of integer variable values—including constant propagation, induction and loop-invariant variable detection, copy propagation, and common subexpression recognition. Our system provides such analysis interprocedurally, through a flow-sensitive symbolic analysis.

For each variable of interest to the array analysis within a loop, the symbolic analysis determines a value for that variable in terms of loop iteration numbers and loop invariants. Once variables used in an array index are expressed in such terms, the locations accessed by one or more iterations of the loop are easily determined. For example, at the array accesses in

```
      K = J + 1
      DO 10 I=1,N
          A(J) = A(K)
          J = J + 2
          K = K + 2
   10 CONTINUE
```

`J` is found to have a value $\mathtt{J}_0 + 2i$, for some loop-invariant $\mathtt{J}_0$ and base-0 iteration count i, and `K` has a value $\mathtt{J}_0 + 2i + 1$. Using these symbolic values, an array analysis can determine that there is no dependence between the two accesses.

The symbolic analysis works by computing a *value map* for each program point, associating with each variable a symbolic description of its value: either *Unknown* or an arbitrary expression in terms of constants, variables, loop indices, and symbolic constants. During the analysis *relative* value maps are used to describe the values on exit from a region in terms of variables on entry to the region. The final result of the analysis is an *absolute* value map, which contains only variable-free symbolic values.

We obtain absolute value maps describing variable values at every program point in two passes, as an interval-based data-flow problem.

Computing Transfer Functions. A bottom-up traversal through the call graph determines the behavior of each region in the program. The behavior of a basic block is described by a relative value map, whose bound values give each variable a value on exit in terms of constants, symbolic constants, and variable values on entry. A symbolic constant is introduced when a variable is assigned an unknown value; the value is considered unknown but invariant within the nearest enclosing loop body or procedure.

The behavior of a loop body or procedure is found by composing and meeting basic block, sub-region, and called procedure behavior maps; for a region contain irreducible loops, an iterative dataflow problem is solved to obtain a conservative region behavior. As with basic blocks, a symbolic constant is introduced to denote an unknown variable value on exit from a subregion. Procedure effects are determined without considering calling context; for reducible programs, procedure behavior can be exactly represented, so no information is lost in the process.

Computing Calling Contexts. In the second phase of the scalar analysis, we compute the context of each region, as an absolute map giving each live variable's value in terms of constants, symbolic constants, and indices of enclosing loops. The context of the outermost region is a map binding all variables to *Unknown*. We proceed top-down over the call graph, determining the context of each subregion or called procedure from the context of the enclosing region, by applying and combining subregion behaviors.

We employ selective procedure cloning based on the values in the map. Currently, the filter function eliminates from the map relations for variables that are no longer live, which we have found significantly reduces the amount of replication in the analysis.

Inequality Constraints The symbolic analysis described thus far can only determine equality constraints between variables. Since array analysis also benefits from knowledge of loop bounds and other control-based contextual constraints on variables (e.g., `if` predicates), which may contain inequalities, a separate top-down pass carries loop and predicate constraints to relevant array accesses. Equality constraints determined by the symbolic analysis are used to rephrase each predicate in terms of loop-invariant terms, if possible. The control context is represented by a set of inequalities in the form discussed in Section 5. This pass also may selectively clone.

4.2 Scalar dependence and privatization

A number of standard analyses ensure that scalar variables do not limit the parallelism available in a loop. The basic techniques are well known in an intraprocedural setting. We apply them interprocedurally. A simple flow-insensitive mod-ref analysis[1] detects scalar dependences. A flow-sensitive live-variable analysis, discussed below, allows detection of privatizable scalar variables. The flow-sensitive symbolic analysis of Section 4.1 also finds induction and loop-invariant integer variables, which can then be privatized.

Live Variable Analysis. We solve the interprocedural scalar live-variable problem through a interval-based approach like the one used for symbolic analysis. Since liveness is a backwards data-flow problem, the solution is not so straightforward. While all program regions have a single entry, they may have multiple exits, since loops can contain returns and breaks. A single transfer function is not sufficient to describe the behavior of a loop in a backwards data-flow problem. Instead, we summarize the behavior of a loop body by three transfer functions—from *loop body* exit, from *loop* exit, and from *enclosing procedure* exit. A loop is described by just two transfer functions—from *loop* exit and from *procedure* exit. A single transfer function still suffices to describe a procedure. In other respects the analysis is straightforward.

5 Array Data-Flow Analysis

Our system performs array dependence, privatization, and reduction recognition analyses in a single bottom-up pass over the call graph, using context information provided by the scalar analyses of Section 4. For simplicity, we describe each analysis separately, after first giving a brief overview of the common representation—based upon systems of linear inequalities—used for array regions.

5.1 Representation of Array Regions

The read or written portion of an array is represented as a set of convex polyhedra, whose enclosed integral points denote index tuples of accessed elements. Each polyhedron is represented by a system of linear inequalities. The array analyses are based upon these sets of systems of linear inequalities. This framework is more powerful than previous work based on regular sections [3, 12] and less expensive than using convex hulls [20].

We have developed a library that implements basic operations on these representations: union, intersection, difference, emptiness and containment tests, and projection to a lower-dimensional space. The representation of data accesses as a set of polyhedra allows us to trade off precision for efficiency where necessary. For example, our union operator combines two polyhedra into one only if it is computationally inexpensive. The projection operation uses a Fourier-Motzkin elimination algorithm [5] that has been enhanced to work for the integer domain. Our system introduces auxiliary variables and constraints on these variables where necessary to ensure no information is lost due to projection.

5.2 Data Dependence Analysis

Our analysis formulates a test for array data dependence as an integer programming problem: if the locations possibly written on one iteration intersect the locations accessed on another iteration there is a dependence. For accesses with affine subscripts the data dependence problems encountered in practice can be solved efficiently and exactly [14]. We summarize the regions of data accessed in a loop to eliminate the need to perform n^2 dependence tests for a loop containing n array accesses.

Our data dependence analyzer is fairly conventional. In a bottom-up manner, the analyzer summarizes the data that have been read and data that have been written within each loop and procedure. We create for each array access a *data descriptor*, which can accurately represent all affine array index expressions. A non-affine index in a dimension is replaced by a conservative approximation: the entire dimension may be accessed. In addition, we add to the descriptor all the relationships in the scalar context that involve any of the variables used in the array index calculation.

We compute the union of the array descriptors to represent the Read and Write array regions accessed in a sequence of statements, with or without conditional flow. At loop boundaries, we derive a loop summary by projecting away the loop index variables in the array descriptors. At procedure boundaries, we perform parameter mapping using the method discussed in Section 5.5. At each loop level, we apply a data dependence test to the Read and Write data summaries.

5.3 Array Privatization Analysis

To find array dependences which yield to privatization of the arrays, we determine for each program region the array regions that are definitely written (MustWrite), that may be written (MayWrite), and that are upwards-exposed read accesses (ExposedRead). A written array region considered a MayWrite if any of its array indices is not an affine expression; otherwise it is a MustWrite. The meet (at a control-flow merge point) of two MustWrite regions yields a MustWrite of their intersection and a MayWrite of their disjunction; the meet of any other two Write regions yields a MayWrite region. We calculate the ExposedRead of a statement within a loop by subtracting from the Read region of the statement the MustWrite region for preceding statements within the loop.

Our formulation of array privatization is an extension of Tu and Padua's algorithm[21]. Tu and Padua recognize an array as privatizable only if there are *no* upwards-exposed reads within the loop. Our algorithm is more general in that upwards-exposed reads are acceptable as long as they do not overlap writes in other iterations of the same loop.

Consider the following excerpt from the `spec77` program:

```
      DO 1000 LAT = 1, 38
         ...
         DO 9 K=1, 12
            ZE(2,K) = RELVOR(K)
            CALL UVGLOB(...,ZE(1,K),...)
            ZE(2,K) = ABSVOR(K)
9        CONTINUE
         ...
1000  CONTINUE
```

Each call to `UVGLOB` reads a column of the array `ZE`. The `LAT` loop writes the second row of the matrix, and none of the elements in the second row are read before they are written in the same iteration. Although there is an upwards-exposed read to all but the second row in the array, we can privatize the outer loop by giving each processor a private copy of `ZE` and initializing the upwards-exposed region with the contents of the original array.

If an array is live on exit of the loop, it must contain the same values as those obtained had the loop been executed sequentially; we limit privatization to those cases where every iteration in the loop writes to exactly the same region of data. To do so the analysis checks that all the writes are in MustWrite, and the region written is not parameterized by the index of the loop to be parallelized. The processor executing the last iteration of the loop uses the original array as its private copy, and is not allowed to alter the array until all other processors have completed their initialization.

5.4 Reductions

A reduction occurs when a location is updated on each loop iteration with the result of a commutative, associative operation applied to its previous contents and some data value. The loop can be parallelized if each processor updates a private location instead, and the original location is finalized by combining it with the private ones. We currently recognize reductions on scalar variables and array locations involving the operations $+$, $*$, `MIN`, and `MAX`.

The design of our reduction analyzer is a response to the common cases we have encountered experimenting with the compiler. By combining basic reduction recognition and array summary techniques, our array reduction analyzer is simple yet powerful. We illustrate our technique with the following sparse matrix-vector code found in the `cgm` program among the NAS benchmarks:

```
      DO 200 J = 1, N
         XJ = X(J)
         DO 100 K = COLSTR(J) , COLSTR(J+1)-1
            Y(ROWIDX(K)) = Y(ROWIDX(K)) + A(K) * XJ
 100     CONTINUE
 200  CONTINUE
```

Starting with the statement in the innermost loop, the analysis recognizes that the add operation is an accumulation to the data element `y(rowidx(k))`. Since the compiler cannot tell the value of the index, it has to conservatively assume that any location in the array could be updated. When considering each loop, it finds that the array `y` is written only by accumulation operations within each loop, and is thus a potential reduction target;

for each loop the potentially updated region is the entire array. The compiler parallelizes the outermost loop by having each processor accumulate to its private copy of the `y` vector and summing up all the vectors at the end of the computation. This simple technique allows the compiler to find parallelism even among sparse computations.

The ability to recognize reductions to array variables is particularly instrumental to parallelizing large loops. Consider the following example from the PERFECT benchmark `spec77`:

```
...                               SUBROUTINE FL22(...,FLN,...)
DO LAT = 1, 38                    DO LL = 1, 31
   ...                               DO I = 1, 31, 2
   DO K = 1, 12                         FLN(I,LL) = FLN(I,LL) + ...
      CALL FL22(...,Y(1,K),...)      DO I = 2, 30, 2
   ...                                  FLN(I,LL) = FLN(I,LL) + ...
```

Even though the compiler can parallelize both loops in `FL22` via standard data dependence analysis, our analysis also notes that all the accesses to `FLN` within `FL22` are accumulation operations. Mapping `FLN` to `Y` at the procedure boundary, the compiler retains the reduction information across the procedure call. When the analysis considers the `LAT` loop, it determines that the loop carries a dependence on `Y`. It examines whether all accesses corresponding to dependences on the loop are marked as potential reductions. They are, so the loop can be parallelized, by generating specialized reduction code.

5.5 Array Reshaping

Array reshapes occur commonly in Fortran programs, as when a slice of an array is passed into a procedure, or when a multi-dimensional array in one procedure is treated as a linear array in another. Linearization is particularly common as it provides long vectors for vector machines.

We formulate the array reshape problem formally as rewriting a system of linear inequalities in terms of different variables. We illustrate our algorithm with the following example:

```
SUBROUTINE FOO                   SUBROUTINE BAR(B)
INTEGER A(10000, 10)             INTEGER B(100, 100)
 ....                                DO 9 I= 1,100
CALL BAR(A(1, K))                       DO 9 J= 1,50
                                           B(I,J) = ...
                                 9   CONTINUE
```

In this example, `FOO` passes `BAR` the `K`th column of the array `A`. This 10000-element vector, on the other hand, is manipulated as a 100×100 array in `BAR`. For a location `B[`b_1, b_2`]` in `BAR` to coincide with a location `A[`a_1, a_2`]` in `FOO`, the indices must be related by:

$$100 * (b_2 - 1) + (b_1 - 1) = 10000 * (a_2 - K) + (a_1 - 1) \tag{1}$$

Array bounds for arrays `A` and `B` define additional constraints:

$$\begin{array}{rr} 1 \le a_1 \le 10000 & 1 \le b_1 \le 100 \\ 1 \le a_2 \le 10 & 1 \le b_2 \le 100 \\ 1 \le K \le 10 & \end{array} \tag{2}$$

To map the summary of array `B` across the procedure call, we add the constraints from (1) and (2) to a system representing the region written by `BAR` and apply our integer projection algorithm to eliminate b_1 and b_2:

$$\left\{ B[b_1, b_2] \,\middle|\, \begin{array}{c} 1 \le b_1 \le 100 \\ 1 \le b_2 \le 50 \end{array} \right\} \quad \leadsto \quad \left\{ A[a_1, a_2] \,\middle|\, \begin{array}{c} 1 \le a_1 \le 5000 \\ a_2 = K \\ 1 \le K \le 10 \end{array} \right\}$$

This technique is simple and general. For example, it easily handles cases where complex numbers in one procedure are treated as real numbers in another by modeling a complex number as an array with two elements. If array dimensions are unknown, there is no linear relationship between the indices of the actual and formal parameters. If the unknown dimensions are identical, we can handle this case exactly, but otherwise we must approximate.

	adm	arc2d	bdna	dyfesm	flo52q	mdg	mg3d	ocean	qcd2
Loops w/ calls	35	1	9	21	9	7	12	12	40
Parallel (Fida)	*	0		0 0	7	0	0	0	0
Parallel (us)	2	0	0	6	8	2	0	0	2

	spec77	track	trfd	doduc	matrix300	dnasa7	**TOTAL**
Loops w/ calls	35	18	6	19	11	8	234
Parallel (Fida)	*	1	0	2	0	0	10
Parallel (us)	17	1	0	8	8	0	54

Figure 1: Comparison of our system with Fida.

6 Empirical Evaluation

We present a series of empirical results to quantify the effectiveness of our parallelization system. We are the first to present any dynamic results on the effectiveness of interprocedural analysis, reduction or array privatization. Every program presented in this section was parallelized by our system completely automatically. We start with the original Fortran77 version of the program. The compiler generates an SPMD (Single Program Multiple Data) parallel C version of the program that is compiled by native C compilers and linked to our own ANL-macro-based thread package; the thread package supports parallel execution on the Stanford DASH and Kendall Square Research distributed-shared-memory architectures and the Silicon Graphics Challenge series of bus-based shared-memory architectures.

To evaluate our parallelization analysis, we measured its success at parallelizing three standard benchmark suites, Spec, Perfect and the sample Nas benchmarks. Nas is a highly parallel suite used for benchmarking parallel computers. Perfect is a set of originally sequential codes used to benchmark parallelizing compilers. Spec is used to benchmark uni-processor architectures and compilers. In the following, we present complete results for 23 of these programs. These programs each consist of 500 to 5,500 lines of source code— 41,000 lines of code in all. Of the remaining 10 programs in the suites, 8 of them have less than 40% of their execution parallelized, so we omitted them from the presentation to concentrate on those that are improved by parallelization.

The results reported below were generated without using any special flags to the compiler, and we did not rely on user directives to assist in the parallelization. We have made no modifications to the original programs except to correct a few type declarations in `spec77` that violated Fortran 77 semantics.

6.1 Comparison with Previous Interprocedural Systems

We first compare against the Fida system (Full Interprocedural Data-Flow Analysis), an interprocedural system that performs precise flow-insensitive array analysis [9] (see Section 2). Earlier work on interprocedural array analysis had demonstrated that such systems were effective at locating parallel loops in the Linpack library functions, which are cleanly written and have very regular array access patterns, but had not evaluated their effectiveness on real applications [20, 12, 8]. The Fida system was the first to measure how changes in precision of the analysis affect the number of parallel loops that the system can recognize. They reported results for the Linpack libraries, observing results comparable to those from the earlier studies. More importantly, they expanded the scope of their experiment to consider full applications from the Perfect and Spec89 benchmark suites.

The table in Figure 1 examines programs from the Perfect and Spec89 benchmark suites, comparing the number of loops found parallel by Fida and by our system. The first row provides a count of the number of loops in each program that contain calls; other loops in the program are ignored. The second row shows the number of these loops that Fida discovered were parallel. The third row gives the number of these loops that our system found to be parallel.

The Fida system found only 10 loops containing calls to be parallel, appearing in 3 of the 13 programs for which they reported results. Moreover, the called procedures contained no accesses to arrays in their caller's scope, so to detect that these loops are parallel requires only scalar Mod/Ref information. No additional parallel loops were uncovered as a result of the interprocedural array dependence analysis.

On the same set of programs, our system located 54 parallel loops, appearing in 9 of the 15 programs for which we reported results. (The asterisk in the FIDA entries for `adm` and `spec77` indicate that they did not report results on these programs.) The marked difference between our results and theirs can be attributed to the additional techniques employed in our system.

6.2 Robustness of Analysis

The compiler has been able to find parallel loops spanning a large number of lines of code. The largest loop it has parallelized is from spec77; it consists of 1002 lines of code from the original loop and its invoked procedures. This loop stresses all of our analyses, including interprocedural reduction, privatization, and complex array reshapes; many of these provided us with examples used earlier in the paper. Within this loop, there are 48 interprocedural privatizable arrays, 5 interprocedural reduction arrays and 27 other arrays accessed independently.

For the purposes of comparison, we attempted full inlining of this program with KAP, a commercial source-to-source parallelizing compiler. KAP failed to inline the entire loop because of the difficult reshapes. The partially inlined loop contained 9839 lines (almost a factor of 10 increase in size) and still had 29 calls to 8 different functions. Such a loop illustrates the problems with inlining for parallelizing large programs; the expansion would have been even more significant if full inlining of the loop had occurred!

6.3 Contribution of analysis components

This section presents measurements on the impact of individual analysis techniques on parallelization, both in the intraprocedural and interprocedural setting. We have observed that interprocedural scalar privatization is a key factor in parallelizing programs such as SPEC benchmarks ora and doduc. Cloning based on symbolic analysis provided constant bounds on dimension sizes in spec77 that enabled array reshaping to obtain precise results. In the remainder of this section, we focus on the contributions of array analysis. In all our comparisons, *Traditional* includes a large collection of parallelization analyses, including *intraprocedural* dependence analysis, scalar privatization, scalar reduction, constant propagation, and induction variable recognition, corresponding to the techniques in most currently available automatic parallelization systems. *SUIF* results extend these analyses to include interprocedural dependence analysis and both intra- and interprocedural array privatization and array reduction recognition.

6.3.1 Static measurements

We measured the number of loops in the 23 programs, and identified whether the loop was parallelized and what analysis techniques were required. Interprocedural scalar data-flow analysis was applied in all cases.

There were 3294 loops in all the programs, of which 2362 were found to be parallelizable. Of the 2362 parallelizable loops, 2080 were simple DOALL loops. Of the remaining, 151 required array privatization, 50 required reduction recognition, 59 were interprocedural DOALL loops, 5 required interprocedural privatization, 7 required interprocedural reduction and 10 required both interprocedural privatization and reduction.

6.3.2 Dynamic measurements

We use two metrics to quantify the dynamic contributions of the array analyses. First, *coverage* refers to the percentage of sequential execution time spent in parallelized regions of the program. The coverage percentage must be high for parallelization to yield much benefit. Second, DOALL invocations refers to the number of times a parallel region is entered, signifying the costs of synchronization. Synchronization costs are a second order effect; they add overhead which diminishes the improvements possible from parallelization. For each program, we collected timings and counted barrier invocations during program execution. Because these measurements were obtained at run time, they are subject to minor variations.

Figure 2 shows coverage measurements for the array analysis. Note that, in the following, the programs are grouped according to application suite and are presented in order of impact due to the enhanced array analyses in our system. From the graph, we observe improved coverage for 13 of the programs presented here when all the analyses were applied. Six of the programs, ora, mdljdp2, mdljsp2, cgm, embar and bdna, have very low coverage in the traditional case which is increased dramatically by the enhanced array analyses in the SUIF system. In fact, the coverage for ora and embar is essentially 0%, so the "traditional" bar for these programs does not appear on the graph.

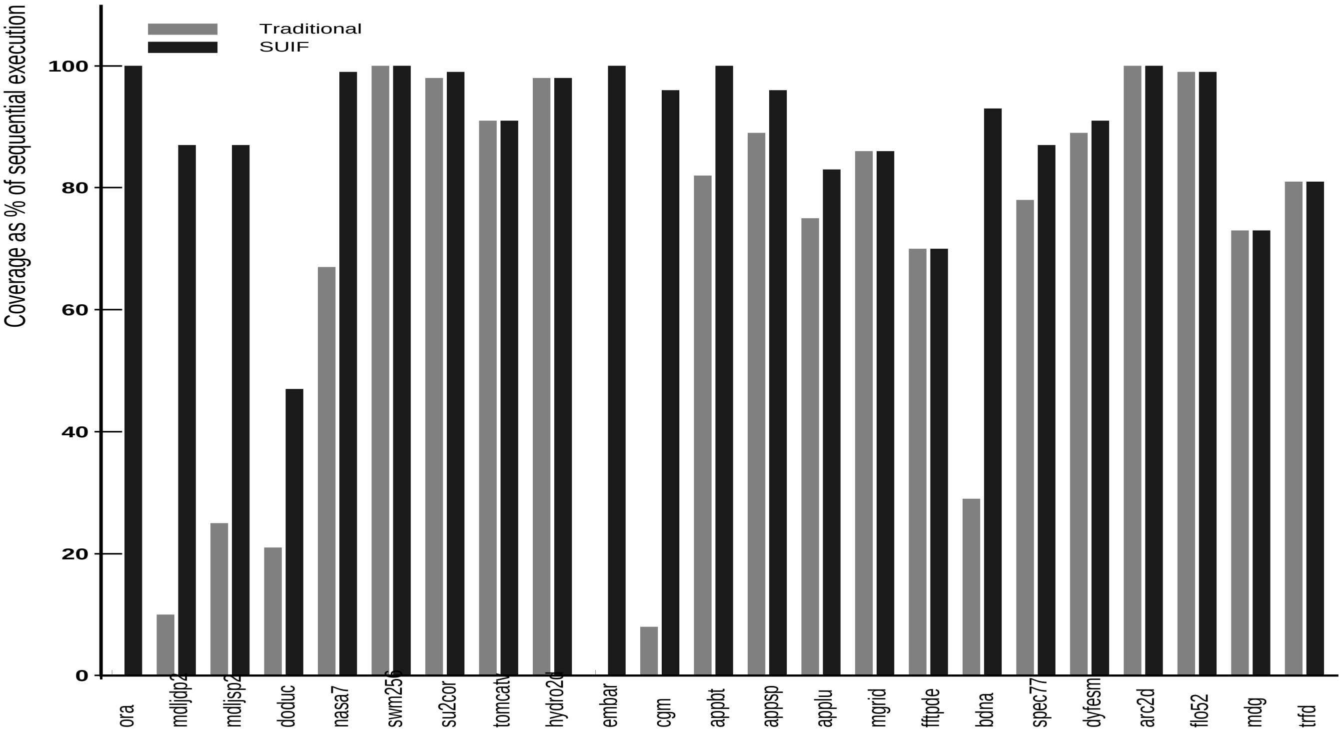


Figure 2: Coverage with and without interprocedural array analysis, array privatization and array reduction.

Figure 3 shows synchronization invocations for the traditional vs. the full SUIF system. The synchronization invocations are normalized with respect to the results for intraprocedural array analysis with no privatization or reduction. We observe that synchronization is dramatically reduced for appbt, appsp, embar, and spec77. For almost all of the programs, synchronization is less than 1, signifying an overall decrease in synchronization costs. We might expect synchronization to go up for the 13 programs with higher coverage, but in fact, synchronization increases for only five of the programs, and it remains manageable.

Table 1 identifies what analysis component was required for each result. When examined in conjunction with the coverage and synchronization figures, we can discover which components are important for each program. In the table, a check (√) in a column indicates that a particular analysis was required to parallelize the program. *Priv* refers to intra-procedural array privatization, *Red* refers to intra-procedural reduction, *Inter* describes an interprocedural DOALL loop, *Inter+Priv* requires interprocedural privatization, *Inter+Red* requires an interprocedural reduction and *Inter+Priv+Red* requires both interprocedural privatization and interprocedural reduction.

In Figure 4 we show speedups on the 8 programs most improved as a result of the interprocedural array analysis, array privatization and reduction recognition. These were all programs that exhibited no speedup whatsoever without these techniques, and continue to get speedups up to 8 processors after being run through our compiler. We compare these speedups with the *Traditional* version of the program, to demonstrate the improvements over what one would obtain using a currently available commercial parallelizer. Timings were gathered on 1, 2, 4 and 8 processors of an SGI Challenge, a bus-based multiprocessor, and compared against the sequential execution time of the program (without any of the parallelization overhead). The following is a brief description of these eight programs and what was required to parallelize them:

bdna (Perfect, *3979 lines*) Bdna is a molecular dynamics code. The program spends more than 50% of its time in a doubly-nested parallel loop, consisting of 75 lines of code. The loop computes reductions on 3 2000-element vectors and a scalar. There are 8 other parallel loops requiring reduction.

mdljdp2, mdljsp2 (Spec, *4172, 3885 lines*) These programs solve the equations of motion for a model of 500 atoms; they are the same model in double and single-precision, respectively. Successful parallelization of this program relies on interprocedural reduction recognition in an outer loop where the program spends approximation 80% of its time. If

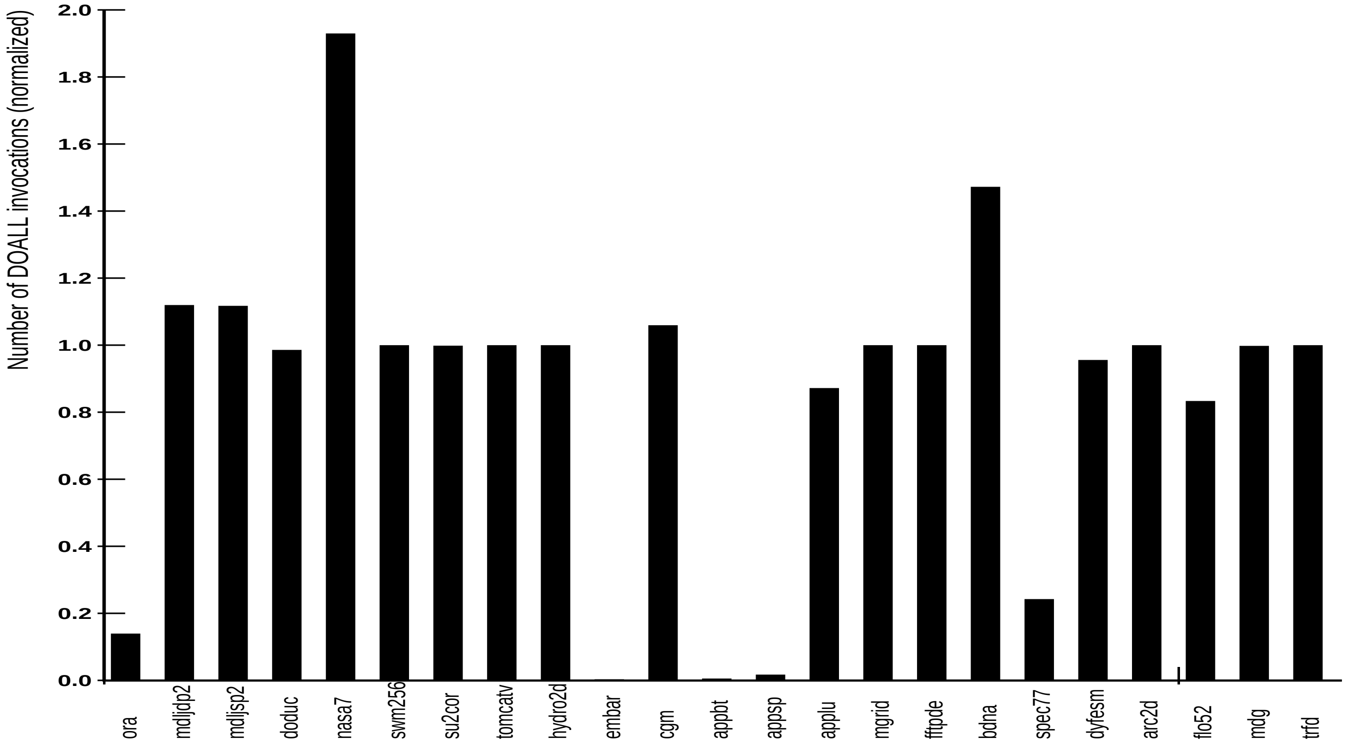


Figure 3: Normalized synchronization costs with and without interprocedural array analysis, array privatization and array reduction.

the compiler computed the reduction in the inner loop, which would not have required interprocedural analysis, the program would have executed more than 60 times more barrier synchronizations, with the overhead exceeding the benefits of parallelization.

appsp (Nas, *3991 lines*) Appsp computes 5 scalar-pentadiagonal PDEs. The bulk of the computation is spent in a single loop consisting of over a hundred lines of code, parallelized via array privatization. There are several interprocedural parallel loops in this program as well, that further reduce the synchronization overhead.

appbt (Nas, *4442 lines*) Appbt computes 5 coupled, block-tridiagonal partial differential equations. It has a similar structure to appsp, with a large outer loop parallelized by array privatization and several interprocedural parallel loops.

cgm (Nas, *855 lines*) Cgm is a sparse conjugate gradient code. The main computation is the sparse matrix-vector multiply, which was used as an example in Section 5.4. Although sparse codes are typically considered beyond the scope of automatic parallelization, our compiler was successful on this code because of the generality of the reduction recognition technique.

embar (Nas, *265 lines*) Embar builds a table for a random number generator. Here, we have used a version of this code obtained from Applied Parallel Research.[3]

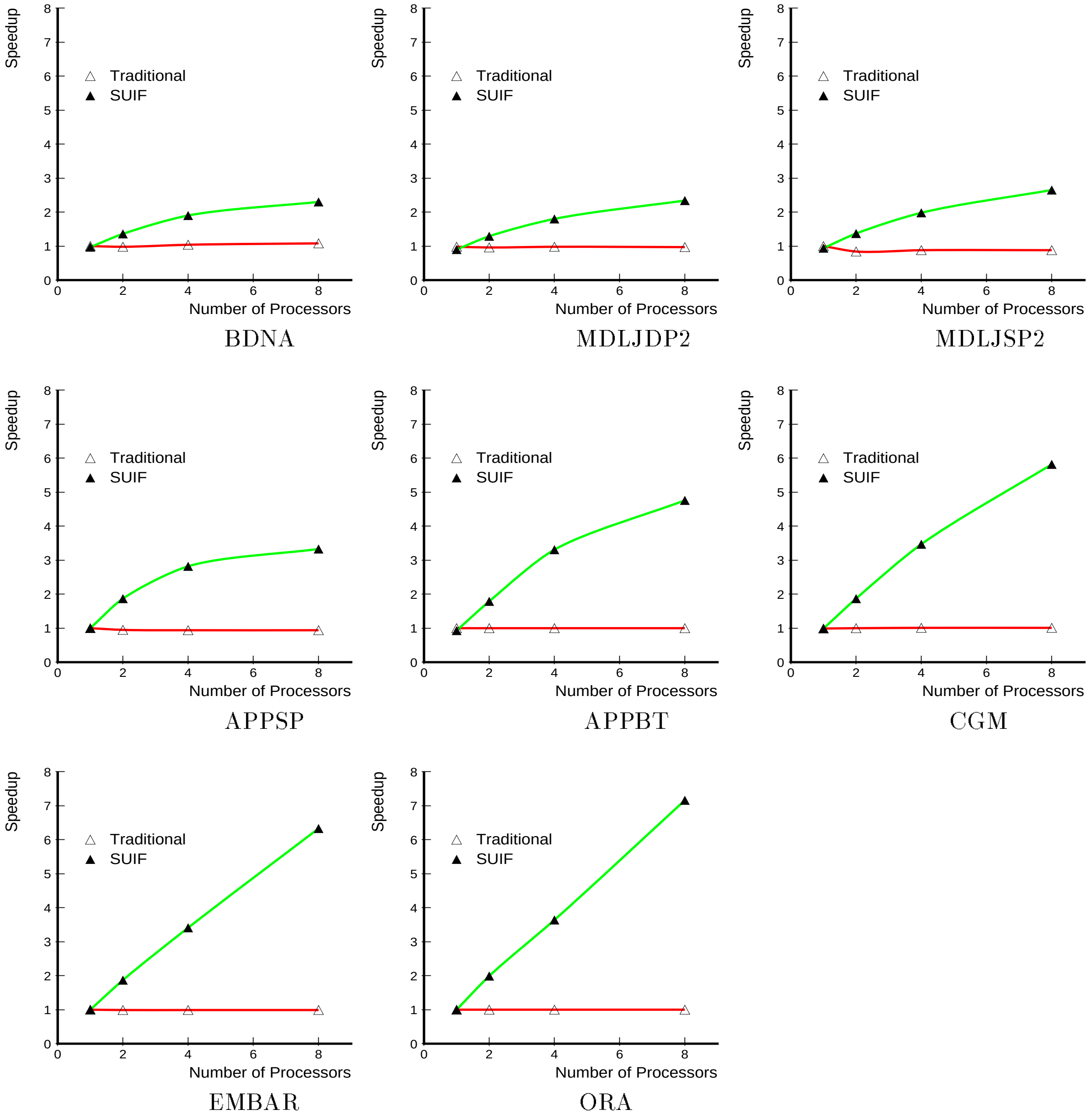
ora (Spec, *373 lines*) Ora is an optical tracing code. Nearly 100% of its execution time is spent in a single loop, which contains a procedure call. In addition to interprocedural analysis, the compiler is also required to buffer I/O inside this loop and print it following the exit. This loop also contains an exit; the compiler applies constant propagation and control flow simplification to prove this jump is never taken.

7 Conclusions

This paper has presented extensive experimental results using a fully interprocedural automatic parallelization system. We have demonstrated that interprocedural array data-flow analysis, array privatization, and reduction

---

[3] This slightly different version separates the first call to a function, which initializes static data, from the other calls.



BDNA MDLJDP2 MDLJSP2

APPSP APPBT CGM

EMBAR ORA

Figure 4: Comparison of speedups with and without interprocedural analysis, array privatization and array reduction.

	Priv	*Red*	*Inter*	*Inter+Priv*	*Inter+Red*	*Inter+Priv+Red*
Spec92						
ora			√			
mdljdp2		√			√	
mdljsp2		√			√	
doduc			√		√	
nasa7	√	√				
swm256						
su2cor		√	√			
tomcatv						
hydro2d						
Nas						
embar			√			
cgm		√				
appbt	√		√			√
appsp	√		√			√
applu	√		√			√
mgrid						
fftpde						
Perfect						
bdna		√				
spec77	√	√	√			√
dyfesm	√	√		√	√	
arc2d						
flo52	√		√			
mdg	√	√		√		
trfd						

Table 1: Contributions of array analysis components.

recognition are key technologies that greatly improve the success of the compiler at finding parallel code segments that translate into increased parallelization coverage, lower synchronization costs and improved speedups. Through our work, we discovered that the effectiveness of an interprocedural parallelization system depends on the strength of all the individual analyses, and their ability to work together in an integrated fashion. This comprehensive approach to parallelization analysis is why our system has been so much more effective at automatic parallelization than previous interprocedural systems and commercially available compilers.

There are two remaining analysis techniques that we feel are important to incorporate into this system. First, the array analysis deals only with affine subscript expressions. A common idiom in vectorized codes is to linearize multi-dimensional arrays to create long vectors; the resulting subscript expressions are non-affine. A simple analysis approach can de-linearize these