Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler

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

Stanford University, Stanford, CA 94305
Computer Systems Laboratory
†California Institute of Technology
Computer Science Dept. 256-80
Pasadena, CA 91125

Abstract

This paper presents an extensive empirical evaluation of an interprocedural parallelizing compiler, developed as part of the Stanford SUIF compiler system. The system incorporates a comprehensive and integrated collection of analyses, including privatization and reduction recognition for both array and scalar variables, and symbolic analysis of array subscripts. The interprocedural analysis framework is designed to provide analysis results nearly as precise as full inlining but without its associated costs. Experimentation with this system shows that it is capable of detecting coarser granularity of parallelism than previously possible. Specifically, it can parallelize loops that span numerous procedures and hundreds of lines of codes, frequently requiring modifications to array data structures such as privatization and reduction transformations. Measurements from several standard benchmark suites demonstrate that an integrated combination of interprocedural analyses can substantially advance the capability of automatic parallelization technology.

1 Introduction

Symmetric shared-memory multiprocessors, built out of the latest microprocessors, are now a widely available class of computationally pow-
erful machines. As hardware technology advances make pervasive parallel computing a possibility, it is ever more important that tools be developed to simplify parallel programming. A parallelizing compiler that automatically locates parallel computations in sequential programs is a particularly attractive programming tool, as it frees programmers from the difficult task of explicitly managing parallelism in their programs.

Unfortunately, today's commercially available parallelizing compilers are not effective at getting good performance on multiprocessors [3, 23]. As these parallelizers were developed from vectorizing compiler technology, they tend to be successful in parallelizing only innermost loops. Parallelizing just inner loops is not adequate for multiprocessors for two reasons. First, inner loops may not make up a significant portion of the sequential computation, thus limiting the parallel speedup by limiting the amount of parallelism. Second, synchronizing processors at the end of the inner loops leaves little computation occurring in parallel between synchronization points. The cost of frequent synchronization and load imbalance can potentially overwhelm the benefits of parallelization.

Multiprocessors are more powerful than vector machines in that they can execute different threads of control simultaneously, and can thus exploit a coarser granularity of parallelism. Thus, for a parallelizing compiler to target a multiprocessor effectively, it must identify outer-loop parallelism to extract coarse-grain parallelism. This requires two major improvements over standard parallelization techniques:

**Advanced Array Analyses.** A loop is often not parallelizable unless the compiler modifies the data structures it accesses. For example, it is very common for each iteration of a loop to define and use the same variable. The compiler must give each processor a *private* copy of the variable for the loop to be parallelizable [3, 23]. As another example, a compiler can parallelize a *reduction* (e.g., computation of a sum, product, or maximum over data elements) by having each processor compute a partial reduction locally and update the global result at the end. Compilers traditionally only perform privatization or reduction transformations to scalar variables. To find outer parallel loops, the compiler must be able to perform these transformations on array variables as well.

**Interprocedural Analysis.** If programs are written in a modular style, it is natural that coarse-grain parallel loops will span multiple procedures. For this reason, procedure boundaries must not pose a barrier to analysis [3]. 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, as it is inefficient in both time and space.
Interprocedural analysis, which applies data-flow analysis techniques across procedure boundaries, can be much more efficient as it analyzes only a single copy of each procedure. Although much research has been devoted to interprocedural analysis for parallelization [8, 12, 13, 15, 18], it 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. Moreover, there is an inherent tradeoff between performing analysis efficiently and obtaining precise results. To be successful, an interprocedural compiler must tackle the complexity of the compilation process, while maintaining reasonable efficiency without sacrificing too much precision.

We have developed an automatic parallelization system that is fully interprocedural. The system incorporates all the standard analyses included in today’s automatic parallelizers, such as data dependence analysis, analyses of scalar variables including constant propagation, value numbering, induction variable recognition and scalar dependence and reduction recognition. In addition, the system employs analyses for array privatization and array reduction recognition. The implementation of these techniques extends previous work to meet the demands of parallelizing real programs. The interprocedural analysis is designed to be practical while providing nearly the same quality of analysis as if the program were fully inlined.

This paper presents a comprehensive evaluation of the effectiveness of this system at locating coarse-grain parallel computations in a large collection of programs from the Spec92fp, Nas and Perfect benchmark suites. We demonstrate how the techniques in our system significantly improve the performance of automatically parallelized codes both by increasing the portion of the program that is executed in parallel and by reducing the synchronization overhead as a result of parallelizing outer loops rather than inner ones.

The remainder of the paper is organized into seven sections. In Sections 2, we present the types of advanced analyses required to parallelize full applications, and in Section 3, we overview the requirements for precise and efficient interprocedural analysis. Section 4 overviews the parallelization analysis in our system. Section 5 compares our work with other automatic parallelization systems. Section 6 is devoted to the empirical evaluation of this system, followed by a conclusion.

2 Parallelization Analysis Techniques

Parallelizing coarse-grain outer loops requires that compilers incorporate many techniques beyond the standard analyses currently avail-
able in commercial parallelizing compilers. In this section, we briefly describe all the parallelization analysis techniques giving examples extracted from real programs encountered in our experiments to motivate the need for advanced analysis techniques.

2.1 Analysis of Scalar Variables

Scalar Parallelization Analysis. Scalar parallelization analysis locates data dependences on scalar variables. A data dependence occurs when a memory location written on one iteration of a loop might be accessed (read or written) on a different iteration; in this case, we say the loop carries a dependence and cannot be safely parallelized. Where there are scalar dependences, this analysis determines whether parallelization may be enabled by privatization or reduction transformations.

Scalar Symbolic Analysis. Parallelizing compilers perform data dependence analysis on arrays to check for loop-carried dependences on individual array elements. Array data dependence analysis is most effective when all subscript expressions are affine functions of loop indices and loop invariants. Within this domain, data dependence analysis has been shown to be equivalent to integer programming. While integer programming is potentially expensive, the data dependence problems found in practice are simple, and efficient algorithms have been developed that usually solve these problems exactly [7, 20]. For this reason, parallelizing compilers incorporate a host of scalar symbolic analyses to put array indices in affine form, including constant propagation, value numbering, and induction variable recognition. These analyses provide integer coefficients for subscript variables and derive affine equality relationships among variables. Some systems also propagate inequality relations and other relational constraints on integer variables imposed by surrounding code constructs (IFs and loops) to their uses in array subscripts [12, 15].

Relations on Non-Linear Variables. Propagating only affine relations among scalars is not sufficient to parallelize some loops. For example, scientific codes often linearize accesses to (conceptually) multidimensional arrays, resulting in subscript expressions that cannot be expressed as an affine function of the enclosing loop indices. The loop nest in Figure 1(a), from the Perfect benchmark trfd, illustrates such a situation. Figure 1(b) shows information that can be determined by a symbolic analysis of the loop. The access to array \( \text{XRSIJ} \) does not induce a dependence between iterations of the outer loop, since the index \( \text{MRSIJ} \) never has the same value on two different iterations.
DO NUM = 10, 40, 5
   NORB=NUM
   CALL OLDA(., NORB, ..., NUM)
SUBROUTINE OLDA(., NORB, ..., NUM)
   NRS=(NUM*(NUM+1))/2          NRS = (NUM * (NUM + 1))/2
   MRSIJ0=0
   DO MRS = 1, NRS                1 ≤ MRS ≤ NRS
      MRSIJ = MRSIJ0
      MRSIJ0 = (MRS - 1) * NRS
      DO MI = 1, MORB
         1 ≤ MI ≤ MORB, MORB = NUM
         DO MJ = 1, MI
            1 ≤ MJ ≤ MI
            MRSIJ = MRSIJ + 1
            XRSIJ(MRSIJ) = XIJ(MJ)
            MRSIJ = MRSIJ0 + (MI * (MI - 1))/2 + MJ
   MRSIJ0 = MRSIJ0 + NRS
(a) nonlinear induction variable (trfd)   (b) symbolic information

Figure 1: Non-linear analysis example.

2.2 Analysis of Array Variables

The scalar parallelization analyses described above (dependence, privatization and reduction) must be generalized to apply to array variables.

Array Data-Flow Analysis and Array Privatization. A simple example motivating the need for array privatization is the K loop in Figure 2(a), a 160-line loop taken from the NAS sample benchmark appbt. (To concisely present these examples, we use Fortran 90 array notation in place of the actual loops.) Although the same array locations in T Mk are defined and used on different iterations of the outer loop, no value flows across iterations. Consequently, it is safe to parallelize this loop if a private copy of T Mk is accessed by each process. Finding privatizable arrays requires that data-flow analysis previously only performed for scalar variables be applied to individual array elements; this analysis is called array data-flow analysis.

Array privatization with initialization. Array privatization is usually applied to the case where each iteration first defines the values in the array before they are used. However, it is also applicable to loops whose iterations use values computed outside the loop; the private copies must be initialized with these values before parallel execution begins. In other words, array privatization is illegal only when iterations refer to values generated by preceding iterations in the loop. An example of array privatization with initialization is shown in Figure 2(b). The figure shows a portion of a 1002-line loop in the Perfect benchmark spec77 (see Section 6.3.2). Here, part of array ZE, the second row, is modified before referenced; the remainder of the array is not modified.
DO K = 2, NZ-1
   DO M = 1, 5
      TM(1:5,M) = ...
   END DO
   DO M = 1, 5
      ... = TM(1:5,M)
   END DO
   DO LAT = 1, 38
      DO K = 1, 12
         ZE(2,K) = RELVOR(K)
         IF UVGLOB reads the entire
            W(1:2,1:UB) = ...
         // Kth column of array ZE
            W(33:34,1:UB) = ...
         CALL UVGLOB(...,ZE(1,K),...)
            W(35:64,1:UB) = ...
         ZE(2,K) = ABSVOR(K)
            W(65:66,1:UB)
      END DO
   END DO
   W(49:50,1:UB) = ...
   W(37:48,1:UB) = ...
   W(51:61,1:UB) = ...
(a) array privatization (appbt)
(b) interprocedural array privatization
(c) recognizing regions across loops
with initialization (spec77) (spec77)

Figure 2: Array analysis examples.

at all in the loop. Array ZE is privatizable in the outer loop by giving
each processor a private copy with all but the second row initialized
with the original values.

Recognizing complicated regions. Data flow analysis on arrays is
intrinsically more difficult than analysis on scalar variables, as it is
necessary to keep track of accesses to individual array elements. In
some cases, the reads and writes to an array may appear in multiple
loops, potentially interleaving reads and writes of disjoint regions. The
compiler must keep precise enough information for the analysis while
maintaining efficiency. Figure 2(b), which is also part of the 1002-line
loop from spec77, illustrates the complexity of the problem. Each
statement in array notation here corresponds to a doubly nested loop.
The compiler can determine that the array W is privatizable by inferring
from the collection of write operations that W is completely defined
before it is read.

Array Reduction Recognition. Most compilers will recognize sim-
ple reductions such as the accumulation into the variable SUM in Fig-
ure 3(a). Reductions on array variables are also common in scientific
codes and are a potential source of significant improvements in parallel-
ization results.

Sparse array reductions. Sparse computations pose what is usually
considered an insurmountable problem for parallelizing compilers.
When arrays are part of subscript expressions, a compiler cannot de-
terminate the locations of the array being read or written. In some cases,
loops containing sparse computations can still be parallelized if the computation is recognized as a reduction. For example, the loop in Figure 3(b) constitutes the main computation in the NAS sample benchmark cgm. We observe that the only accesses to sparse vector \( Y \) are commutative and associative updates to the same location. Thus, it is safe to transform this reduction to a parallelizable form. Figure 3(c) is an excerpt of a 148-line loop that makes up the main computation in the SPEC92 benchmark mdjdp2. The outer loop \( \text{DO } I \ldots \) is parallelizable if sparse reduction is performed interprocedurally. This example also demonstrates reductions in a loop may consist of multiple updates to the same array.

### 3 Interprocedural Analysis Issues

Interprocedural data-flow analysis can support interprocedural optimization in a much more space-efficient manner than inlining by analyzing only a single copy of each procedure. To capture precise interprocedural information requires a flow-sensitive approach, which derives analysis results along each possible control flow path through the program. Precise and efficient flow-sensitive interprocedural analysis is difficult because information flows into a procedure both from its callers (representing the calling context in which a procedure is invoked) and from its callees (representing the side effects of the invocations). For example, in a straightforward interprocedural adaptation of traditional iterative analysis, analysis might be carried out over a program representation called the supergraph [22], where individual control flow graphs for the procedures in the program are linked together at pro-
cEDURE call and return points. Iterative analysis over this structure will be slow because the number of control paths through which information flows can be very large. Such analysis also loses precision by propagating information along unrealizable paths [17]; the analysis may propagate calling context information from one caller through a procedure and return the side-effect information to a different caller.

Region-Based Flow-Sensitive Analysis. In our system, we use a region-based analysis that solves the problems of unrealizable paths and slow convergence. We perform analysis efficiently in two separate passes over the program. Proceeding bottom-up through the call graph, the first pass analyzes each procedure to obtain a description of its side-effect behavior in the form of a transfer function; the transfer function is in turn inserted at call sites when computing transfer functions of callers. In a second, top-down pass over the call graph, data-flow information from the calling context is applied to these transfer functions to derive the final analysis results for each procedure. We use a similar approach within each procedure, summarizing code region (e.g., loop) behavior during the bottom-up pass and propagating context into each region during the top-down pass.

Selective Procedure Cloning. With the region-based analysis approach described above, precision may still be lost as compared to full inlining. In the second pass when deriving the calling context for a procedure, the analysis must represent the conservative approximation of information contributed by all program paths to the procedure. Such approximations can affect the precision of analysis if a procedure is invoked along paths that contribute very different information. To avoid this loss of precision, we incorporate into the calling context analysis a technique called selective procedure cloning, where the compiler replicates analysis results for a procedure to determine its data-flow information along paths in the program that contribute significantly different calling contexts [5]. Because the replication is done selectively according to the unique data-flow information it exposes, we manage the analysis costs and can usually obtain the same precision as full inlining.

Interprocedural Framework. As different data-flow problems share many commonalities (for example, support for parameter passing), it is useful to have an interprocedural framework to manage the complexity of the implementation and allow code reuse. The region-based analysis and selective procedure cloning techniques are encapsulated in a common interprocedural framework as part of FIAT, a tool for developing interprocedural analysis systems [11]. FIAT facilitates adding
new interprocedural analyses by providing parameterized templates to drive flow-sensitive analysis and cloning; each analysis problem is implemented by instantiating the templates with functions to compute solutions to data-flow equations. For the interprocedural parallelization system in SUIF, we have extended FIAT significantly to support array data-flow analysis and flow-sensitive analysis.

4 Parallelization Analysis Algorithms

This section overviews the parallelization analysis algorithms and describes how the different phases of the analysis fit together. Further description can be found elsewhere [9, 10].

4.1 Scalar Analysis

Our system has interprocedural scalar analysis that encompasses both scalar parallelization analysis and scalar symbolic analysis. For the scalar parallelization analysis, simple flow-insensitive analysis—interprocedural analysis that does not consider control flow within a procedure—provides the information to locate scalar dependences and scalar reductions. Recognizing privatizable scalars is more complex, requiring a bottom-up flow-sensitive region-based analysis to find upwards-exposed reads at loop boundaries.

An interprocedural symbolic analysis combines constant propagation, value numbering, and induction variable recognition. The analysis is performed, through a region-based analysis, in two passes over the call graph. Procedures and regions are summarized by scalar value maps, which describe variable values on exit as an arbitrary expression of variable values on entry. The top-down pass determines symbolic values for integer variables in terms of loop indices and loop invariants, and selectively clones based on symbolic values used in the procedure. This analysis also provides support for analyzing some non-linear array subscripts. The analysis recognizes higher-order induction variables (such as MRSIJ in the MI loop of Figure 1(a)) and, in place of such variables, presents to the array analysis additional linear inequality constraints describing the difference in value of the variable from one iteration to the next. This simple approach captures what is required to analyze many non-linear subscript expressions without the need to adopt sophisticated non-linear dependence tests [4, 19].

A separate top-down interprocedural context analysis propagates contextual relations (in the loop-relative terms obtained from the symbolic analysis) from enclosing IF predicates and loop bounds; a context of relations is represented as a set of systems of linear equalities similar to the array summary descriptors described below.
4.2 Array Analysis

Summaries. Traditional data dependence analysis solves an integer programming problem for every pair of array accesses in a loop of interest. This $O(n^2)$ analysis becomes prohibitively expensive for very large loops, particularly in the interprocedural setting. One way to improve efficiency is to summarize the array accesses in a region of code; a data dependence analysis is then applied to a small number of summaries. The summaries also provide the representation used by the array data-flow analysis, described below.

A set of array accesses is represented by a list of systems of linear inequalities: the array indices are equated to affine expressions of outer loop indices and loop-invariant values, constrained further by inequalities derived from the loop bounds. The representation of data accesses as a set of systems allows us to trade off precision for efficiency where necessary. For example, union of two summaries combines two systems into one only if it is computationally inexpensive and no loss of information results. Further, representing multiple regions for an array is essential to capture the examples from Figure 2(c)-(d); any approximations such as capturing the regions as convex hulls would not have the necessary precision.

We create a summary of an access outside an enclosing loop by projecting away the loop index variable, using a Fourier-Motzkin projection that has been enhanced for the integer domain. We similarly transform an access summary across a procedure call by equating array subscript variables in the formal parameter to the subscript variables in the actual parameter, further constrained by the declared types for both formal and actual. Projection eliminates the formal parameter subscripts and replaces them with the actual parameter subscripts. This strategy for transforming summaries across procedure boundaries provides a general mechanism for analyzing array reshapes, where the number or size of array dimensions are altered at a call. A similar approach to array reshapes has also recently been adopted by Creussile [6].

Array Data-Flow Analysis. A single array data-flow analysis is used to determine arrays involved in data dependences, to locate privatizable arrays and to recognize reductions. Array data-flow analysis is a bottom-up interprocedural analysis on the loops and procedures of the program, using the region-based analysis framework described above. The analysis computes the following four sets of summaries for each program region: the array sections that are definitely written (MustWrite), that may be written (Write), that may be read (Read), and that may be read before they are written (ExposedRead).

Data dependence testing at a loop is a comparison of the Read and Write sets to determine if they are disjoint for different iterations of the
loop. Some arrays involved in data dependences may yield to privatization. An array can be privatized if the Write set and the ExposedRead set are disjoint. Our array privatization analysis is an extension of Tu and Padua’s approach [26]. Their algorithm requires that a privatizable array have no read locations upwards-exposed to the beginning of a loop iteration. Our approach is more general, capturing cases such as the one in Figure 2(c).

**Array Reductions.** Array reduction recognition is performed by a simple algorithm that is integrated with the array data-flow analysis. Recognizing reductions begins with locating commutative and associative $+$, $*$, MIN or MAX operations to the same memory location. We mark the corresponding representations of these accesses in the summary with the reduction type (according to the operator). During the bottom-up interprocedural propagation for array data-flow analysis, we ensure that all accesses to the potential reduction array within the current loop are reduction operations of the same type. At each loop, we evaluate whether variables that carry a dependence and cannot be privatized are involved in a reduction computation. The variable’s computation can only be reduced if every region described in the summary has the same reduction type. This simple algorithm is sufficiently powerful to recognize and parallelize the reductions in Figure 3(a)-(c).

### 4.3 Putting It All Together

The analysis techniques are built using the region-based analysis framework, with selective cloning. In Figure 4, we put together the analysis phases, demonstrating that the entire analysis system could execute in just four passes over the program’s call graph. Scalar modifications, references and reductions are performed in an initial flow-insensitive pass; these analyses could fold into the next pass, but a flow-insensitive implementation can be performed more efficiently.

### 5 Related Work

In the late 1980s, a series of papers presented results on interprocedural parallelization analysis [13, 18, 24]. 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 [14] (see
1. Flow-insensitive pass:
   - Find modified and referenced variables
   - Find scalar reductions

2. Bottom-up pass: scalar analysis
   - Find privatizable scalars
   - Summarize symbolic behaviors (side-effects)

3. Top-down pass: scalar analysis
   - Apply calling context to symbolic value maps for symbolic analysis
   - Extract and propagate program control-flow constraints (inequality relations)
   - Selectively clone based on the above two analyses.

4. Bottom-up pass: array analysis
   - Summarize MustWrite, Write, Read, ExposedRead
   - Find data dependences (intersect Write and Read)
   - Find privatizable arrays (intersect Write and ExposedRead)
   - Recognize array reductions and record reduction operator type

Figure 4: Phases of Interprocedural Parallelization Analysis.

Section 6).

Irigoin et al. have developed the PIPS system, an interprocedural analysis system that is part of an environment for parallel programming [16]. More recently, PIPS has been extended to incorporate interprocedural array privatization [15, 6]. 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.

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.

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 [21]. Applied Parallel Research has demonstrated good speedup results on some of the programs presented here; these programs were parallelized with programmer directives that instruct the compiler to ignore dependences and to privatize certain variables. We know of no commercial system that currently employs any flow-sensitive array analysis, particularly interprocedural array privatization.

6 Empirical Evaluation

The interprocedural parallelization analysis described in the previous sections is implemented as part of the Stanford SUIF compiler. This section provides an empirical evaluation of the results of the parallelization analysis on a large collection of benchmark programs.

Previous evaluations of interprocedural parallelization systems have provided static measurements of the number of additional loops parallelized as a result of interprocedural analysis [13, 14, 18, 24]. We have compared our results with the most recent of these empirical studies, which examines the SPEC99 and PERFECT benchmark suites [14]. When considering only those loops containing calls for this set of 16 programs, the SUIF system is able to parallelize greater than five times more of these loops [9]. The key difference between the two systems is that SUIF contains full interprocedural array analysis, including array privatization and reduction recognition (see Section 5).

Static loop counts, however, are not good indicators of whether parallelization will be successful. Specifically, parallelizing just one outermost loop can have a profound impact on a program’s performance. Dynamic measurements provide much more insight into whether a program may benefit from parallelization. Thus, in addition to static measurements on the benchmark suites, we also present a series of results gathered from executing the programs on a parallel machine. We present overall speedup results, as well as other measurements on some of the factors that determine the speedup. We also provide results that identify the contributions of the analysis components of our system, focusing on the advanced array analyses.

6.1 Benchmark Programs

To evaluate our parallelization analysis, we measured its success at parallelizing three standard benchmark suites described by Table 1: the Fortran programs from SPEC92FP, the sample NAS benchmarks, and PERFECT.
**Spec92fp** is a set of 14 floating-point programs used to benchmark uniprocessor architectures and compilers. We omit four in this study. Because the parallelization analysis currently is only available for Fortran, we omit *alvinn* and *ear*, the two C programs, and *spice*, a program of mixed Fortran and C code. We also omit *fppp* because it contains type errors in the original Fortran source; this program is considered to contain very little loop-level parallelism. (The programs are presented in alphabetical order of their program names).

**NAS** is a highly parallel suite of eight programs used for benchmarking parallel computers. NASA provides sample sequential programs plus application information, with the intention that they can be rewritten to suit different machines. We use all the NASA sample programs except for *embar*. We substitute for *embar* a version from APR that separates the first call to a function, which initializes static data, from the other calls.

Lastly, *Perfect* is a set of originally sequential codes used to benchmark parallelizing compilers. We present results on 12 of 13 programs here. *Spice* contains pervasive type conflicts and parameter mismatches in the original Fortran source that violate the Fortran77 standard, and that the interprocedural analysis flags as errors. This program is considered to have very little loop-level parallelism.

The programs have been parallelized completely automatically by our system without relying on any user directives to assist in the parallelization. We have made no modifications to the original programs.\(^1\) All the programs produce valid results when executed in parallel.

### 6.2 SUIF Compiler System

SUIF is a fully functional compiler that takes both Fortran and C as input languages. (For this experiment, we consider Fortran programs only.) The parallelized code is output as an SPMD (Single Program Multiple Data) parallel C version of the program that can be compiled by native C compilers on a variety of architectures. The resulting C program is linked to a parallel run-time system that currently runs on several bus-based shared memory architectures (the SGI Challenge and Power Challenge, and the Digital 8400 multiprocessors) and scalable shared-memory architectures (Stanford DASH and Kendall Square KSR-1).

There are two major components to automatic parallelization in SUIF. First, the *analysis* component locates the available parallelism in the code. This component encompasses all the interprocedural parallelization analyses presented in this paper. (In addition, SUIF includes

\(^1\)except to correct a few type declarations and parameter passing in *arc2d*, *bdna*, *dyfesm*, *mgrid*, *mdg* and *spec77*, all of which violated Fortran 77 semantics.
<table>
<thead>
<tr>
<th>Program</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doduc</td>
<td>5334 lines</td>
<td>Monte Carlo simulation</td>
</tr>
<tr>
<td>mljdp2</td>
<td>4316 lines</td>
<td>equations of motion</td>
</tr>
<tr>
<td>wave6</td>
<td>7628 lines</td>
<td>2-D particle simulation</td>
</tr>
<tr>
<td>tomcatv</td>
<td>195 lines</td>
<td>mesh generation</td>
</tr>
<tr>
<td>ora</td>
<td>373 lines</td>
<td>optical ray tracing</td>
</tr>
<tr>
<td>mljsp2</td>
<td>3885 lines</td>
<td>equations of motion, single precision</td>
</tr>
<tr>
<td>swm256</td>
<td>487 lines</td>
<td>shallow water model</td>
</tr>
<tr>
<td>su2cor</td>
<td>2514 lines</td>
<td>quantum physics</td>
</tr>
<tr>
<td>hydro2d</td>
<td>4461 lines</td>
<td>Navier-Stokes</td>
</tr>
<tr>
<td>nasa7</td>
<td>1105 lines</td>
<td>NASA Ames Fortran kernels</td>
</tr>
<tr>
<td>appbt</td>
<td>4457 lines</td>
<td>block tridiagonal PDEs</td>
</tr>
<tr>
<td>applu</td>
<td>3285 lines</td>
<td>parabolic/elliptic PDEs</td>
</tr>
<tr>
<td>appsp</td>
<td>3516 lines</td>
<td>scalar pentadiagonal PDEs</td>
</tr>
<tr>
<td>buk</td>
<td>305 lines</td>
<td>integer bucket sort</td>
</tr>
<tr>
<td>cgm</td>
<td>855 lines</td>
<td>sparse conjugate gradient</td>
</tr>
<tr>
<td>embar</td>
<td>135 lines</td>
<td>random number generator</td>
</tr>
<tr>
<td>fftpde</td>
<td>773 lines</td>
<td>3-D FFT PDE</td>
</tr>
<tr>
<td>mgrid</td>
<td>676 lines</td>
<td>multigrid solver</td>
</tr>
<tr>
<td>adm</td>
<td>6105 lines</td>
<td>pseudospectral air pollution model</td>
</tr>
<tr>
<td>arc2d</td>
<td>3965 lines</td>
<td>2-D fluid flow solver</td>
</tr>
<tr>
<td>bdna</td>
<td>3980 lines</td>
<td>molecular dynamics of DNA</td>
</tr>
<tr>
<td>dyfesm</td>
<td>7608 lines</td>
<td>structural dynamics</td>
</tr>
<tr>
<td>flo52</td>
<td>1986 lines</td>
<td>transonic inviscid flow</td>
</tr>
<tr>
<td>mdg</td>
<td>1238 lines</td>
<td>molecular dynamics of water</td>
</tr>
<tr>
<td>mg3d</td>
<td>2812 lines</td>
<td>depth migration</td>
</tr>
<tr>
<td>ocean</td>
<td>4343 lines</td>
<td>2-D ocean simulation</td>
</tr>
<tr>
<td>qcd</td>
<td>2327 lines</td>
<td>quantum chromodynamics</td>
</tr>
<tr>
<td>spec77</td>
<td>3889 lines</td>
<td>spectral analysis weather simulation</td>
</tr>
<tr>
<td>track</td>
<td>3735 lines</td>
<td>missile tracking</td>
</tr>
<tr>
<td>trfd</td>
<td>485 lines</td>
<td>2-electron integral transform</td>
</tr>
</tbody>
</table>

Table 1: Benchmark programs.
C pointer analysis to support parallelization of C programs, but this is outside the scope of this paper.) The second major component is *parallel code generation*, which includes a number of optimizations to improve performance once the parallelism has been found. Specifically, the full SUIF system incorporates data and loop transformations to increase the granularity of parallelism and to improve the memory behavior of the programs [1, 27] and optimizations to eliminate unnecessary synchronization [25].

In this paper, however, we adopt a very simple parallel code generation strategy that does not include these optimizations in order to focus on the effects of the parallelization analysis. The compiler parallelizes the outermost loop that the analysis has proven to be parallelizable. Our compiler suppresses parallelization of array reductions if the overheads involved are expected to overwhelm the benefits. In addition, the run-time system estimates the amount of computation in each parallelizable loop using the knowledge of the iteration count at run time, and runs the loop sequentially if it is considered too fine-grained to have any parallelism benefit. The iterations of a parallel loop are evenly divided between the processors at the time the parallel loop is spawned.

### 6.3 Applicability of Advanced Analyses

The experimental framework currently does not support isolation of the contributions of the interprocedural scalar analyses, but we know that these analyses are important. For example, performance-critical loops in the programs *embar*, *mdjdp2*, *ora* and *spec77* would not have been parallelized without one of interprocedural scalar privatization, scalar reduction recognition, or selective procedure cloning based on interprocedural constants.

Here we present static and dynamic measurements to assess the impact of the array analysis components. We define a *baseline* system that serves as a basis of comparison throughout this section. Baseline refers to our system without any of the advanced array analyses. It performs intraprocedural data dependence, and does not have any capability to privatize arrays or recognize reductions. Note that the baseline system is *much* more powerful than existing parallelizing compilers as it contains all the *interprocedural scalar analysis* discussed in Section 4.1.

#### 6.3.1 Static Measurements

Table 2 gives counts of the number of loops in the SUIF-parallelized program that require a particular technique to be parallelizable. In this table, we count all parallelizable loops, including those nested within other parallel loops which would consequently not actually be executed in parallel under our parallelization strategy. The first column gives the
number of loops that are parallelizable in the baseline system. The next three columns measure the applicability of the intraprocedural versions of advanced array analyses. They measure the effect of including reduction recognition, privatization, and both reduction recognition and privatization, respectively. The next set of four columns all have interprocedural data dependence analysis. Similarly, the sixth to eighth columns measure the effect of adding interprocedural reduction recognition, privatization, and both reduction recognition and privatization, respectively.

We see from this table that the advanced array analyses are applicable to a majority of the programs in the benchmark suite, and several programs can take advantage of all the interprocedural array analyses. Although the techniques do not apply uniformly to all the programs, the frequency in which they are applicable for this relatively small set of programs demonstrates that the techniques are general and useful. We observe that there are many more loops that do not require any new array techniques. However, loops parallelized with advanced array analyses often involve more computation and, as shown below, can make a substantial difference in overall performance.

6.3.2 Dynamic Measurements

We also measure the dynamic impact of each of the advanced array analyses. The contribution of each analysis component is measured by recording the specific array analyses that apply to each parallelized loop, and instrumenting the sequential code to determine the execution time of each of the loops. We present the execution times as percentages of the total computation times in Figure 5(C). The measurements were taken by running the programs on a single processor in a 200Mhz SGI Challenge; as the results are reported in relative terms, they are applicable to a large class of microprocessors. Note that when we say that interprocedural reduction recognition is applicable to, say, 100% of the computation, it does not mean that a non-interprocedural parallelizer will find no parallelism in the code, as it may parallelize an inner loop.

We term the overall percentage of time spent in parallelized regions as the parallelism coverage. Overall, we observe rather good coverage (above 80%) for 8 of the 10 programs in Spec92fp, 7 of the 8 NAS programs and 6 of the 12 Perfect benchmarks. A third of the programs spend more than 50% of their execution time in loops requiring advanced array analysis techniques.

This graph also demonstrates how important parallelizing a single loop requiring one of the advanced analysis techniques can be. For example, the program mdijdp2 contains just two loops requiring interprocedural reduction, but those two loops are where the program
<table>
<thead>
<tr>
<th></th>
<th>Intraprocedural</th>
<th>Interprocedural</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array Reduction</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Array Privatization</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>SPEC92FP</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>doduc</td>
<td>230</td>
<td>7</td>
</tr>
<tr>
<td>mdjdp2</td>
<td>10 2 1</td>
<td>2</td>
</tr>
<tr>
<td>wave5</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>tomcatv</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>ora</td>
<td>5 3</td>
<td></td>
</tr>
<tr>
<td>mdjsp2</td>
<td>10 2 1</td>
<td>2</td>
</tr>
<tr>
<td>swm256</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>su2cor</td>
<td>65 3</td>
<td>1</td>
</tr>
<tr>
<td>hydro2d</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>nasa1</td>
<td>59 1 6</td>
<td></td>
</tr>
<tr>
<td><strong>NAS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>appbt</td>
<td>139 3 18</td>
<td>6</td>
</tr>
<tr>
<td>applu</td>
<td>117 4 6</td>
<td>6</td>
</tr>
<tr>
<td>appsp</td>
<td>142 12</td>
<td>6</td>
</tr>
<tr>
<td>buck</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>cgms</td>
<td>17 2</td>
<td></td>
</tr>
<tr>
<td>embar</td>
<td>3 1</td>
<td></td>
</tr>
<tr>
<td>fftpde</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>mgrid</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td><strong>PERFECT</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adm</td>
<td>172 2</td>
<td>2</td>
</tr>
<tr>
<td>arc2d</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>bdna</td>
<td>111 28</td>
<td>1</td>
</tr>
<tr>
<td>dyfesm</td>
<td>122 5 2</td>
<td>1</td>
</tr>
<tr>
<td>flo62</td>
<td>148 1</td>
<td>7</td>
</tr>
<tr>
<td>mgd</td>
<td>35 1</td>
<td></td>
</tr>
<tr>
<td>mg3d</td>
<td>104 2</td>
<td></td>
</tr>
<tr>
<td>ocean</td>
<td>102 1 6</td>
<td></td>
</tr>
<tr>
<td>qfd</td>
<td>92 7</td>
<td></td>
</tr>
<tr>
<td>track</td>
<td>51 3</td>
<td>1</td>
</tr>
<tr>
<td>trfd</td>
<td>15 5 1</td>
<td></td>
</tr>
<tr>
<td>spec77</td>
<td>281 13 2</td>
<td>17</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>2666 85 57 0</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 2: Static Measurements: Number of Loops Using Each Technique
spends 78% of its time.

Not only do some of these SUIF-parallelized loops execute for a long time, they can also be very large. The largest loop SUIF parallelizes is from spec77, consisting of 1002 lines of code from the original loop and its invoked procedures. The loop contains 60 subroutine calls to 13 different procedures. Within this loop, there are 48 interprocedural privatizable arrays, 5 interprocedural reduction arrays and 27 other arrays accessed independently. Such a loop illustrates the advantage of interprocedural analysis over inlining for parallelizing large programs. If instead this loop had been fully inlined, it would have contained nearly 11,000 lines of code.

6.4 Effectiveness of Advanced Analyses

Section 6.3 establishes that the advanced techniques are applicable to many programs; this section addresses the effectiveness of the techniques. We provide quantitative data to show that these techniques are more effective than previous techniques in parallelizing the programs in the benchmark suites.

6.4.1 Metrics and Results

While parallel speedups measure the overall effectiveness of a parallel system, they are also highly machine dependent. Not only do speedups depend on the number of processors, they are sensitive to many aspects of the architecture, such as the cost of synchronization, the interconnect bandwidth and the memory subsystem. Furthermore, speedups measure the effectiveness of the entire compiler system and not just the parallelization analysis, which is the focus of the paper. For example, techniques to improve data locality and minimize synchronization can greatly improve the speedups obtained. Thus, to more precisely capture how well the parallelization analysis performs, we use the two following metrics:

Parallelism Coverage. Coverage, as introduced in Section 6.3.2, is an important metric for measuring the effectiveness of parallelization analysis. By Amdahl's law, programs with low coverage will not get good parallel speedup. For example, even for a program with 80% coverage, its ideal speedup is only 2.5 on 4 processors. High coverage is indicative that the compiler analysis is locating significant amounts of parallelism in the computation.

Granularity of parallelism. A program with high coverage is not guaranteed to achieve parallel speedup due to a number of factors. The granularity of parallelism extracted is a particularly important factor, as frequent synchronizations can slow down, rather
Figure 5: Dynamic Measurements of SUIF and the Baseline Compiler
than speed up, a fine-grain parallel computation. To quantify this property, we define a program's granularity as the average execution time of its parallel regions.

Figures 5(B) and (C) show a comparison of the parallelism coverage and granularity achieved by the SUIF and the baseline compiler.

For the sake of completeness, we also present a set of speedup measurements. The programs in the benchmark suite have relatively short execution times as well as fine granularities of parallelism, as shown in Figure 5(C). Most of these programs cannot utilize a large number of processors effectively. For our experiment, we run all the programs on a 4-processor 200MHz SGI Challenge. Speedups are calculated as ratios between the execution time of the original sequential program and the parallel execution time. The results are shown in Figure 5(D).

6.4.2 Discussion

SPEC92FP Benchmarks. Figure 5(B1) shows that the advanced array analyses dramatically increase parallelism coverage on 3 of the 10 programs. In other words, all the major loops that require sophisticated array analyses do not contain many loops that can be parallelized using conventional techniques. These new parallel loops are also rather coarse grained, as can be observed from Figure 5(C1). Overall the compiler achieves good results parallelizing SPEC92FP. Coverage is above 80% for 8 of the 10 programs, and a speedup is achieved on all of these 8.

The results also show that coverage is necessary but not sufficient for high speedups. Programs with fine granularity of parallelism, even those with high coverage such as su2cor, tomatc and nasa7, tend to have lower speedups. Another important factor that affects speedups is data locality. Two of these programs, tomatc and nasa7, have poor memory behavior. The performance of these programs can be improved significantly via data and loop transformations to improve cache locality[1] and techniques to minimize synchronization[25].

NAS Benchmarks. The advanced array analyses in SUIF are important to the successful parallelization of the NAS benchmarks, as can be seen in Figure 5(B2-D2). Comparing SUIF with the baseline system, we observe that the array analyses have two important effects. They enable the compiler to locate significantly more parallelism in two of the programs, cgm and embar. They also increase the granularity of parallelism in appbt and appsp by parallelizing an outer loop instead of inner loops nested inside it. Observe that what seems like a moderate improvement of coverage in appbt—from 85% to nearly 100%—is significant. This difference corresponds to a change in ideal speedup from 2.75 to 4 on 4 processors.
The improvements in coverage and granularity in NAS translate to good speedup results. Six of the eight programs yield a speedup. Of the other two, buk’s low coverage is not surprising as it implements a bucket sort algorithm. Applu, although it has high coverage, is too fine-grained to yield any speedup. Overall, the advanced array analyses are important for NAS; half of the benchmark suite would not be sped up without these techniques.

Perfect Benchmarks. As displayed in Figure 5(B3-D3), the advanced array analyses significantly improve the parallelism coverage of bdna and qcd. For bdna, the additional parallel loops provide a reasonable granularity that leads to speedup. Granularity is increased for spec77 and trfd, and speedup is achieved in the case of trfd. Although little parallel speedup is observed on spec77, the improvement over the baseline system confirms the validity of our preference for outer loop parallelism. As a whole, SUIF doubles the number of programs that achieve a speedup from 2 to 4.

The overall parallelization of Perfect was not as successful as for the other two benchmark suites. As Figure 5 indicates, there are two basic problems. Half of the programs have coverage below 80%. Furthermore, the parallelism found is rather fine-grained, with most of the parallelizable loops taking less than 100 μs on a uniprocessor. In fact, had the run-time system not suppressed the parallelization of many fine-grained loops in Perfect, the results would have been much worse. Thus, not only is the coverage low, the system can only exploit a fraction of the parallelism extracted.

We now examine the difficulties in parallelizing Perfect to determine the feasibility of automatic parallelization, and to identify possible future research directions. We found that some of these programs are simply not parallelizable as implemented. Some of these programs contain a lot of input and output (e.g. mg3d and spec77); their speedup depends on the success of parallelizing I/O. Further, “dusty deck” features of these programs, such as the use of equivalence constructs in ocean, obscure information from analysis. In contrast, most of the SPEC92FP and NAS programs are cleanly implemented, and are thus more amenable to automatic parallelization.

For many of these programs, particularly ocean, adm, and mdg, there are key computational loops that are safe to parallelize, but they are beyond the scope of the techniques implemented in SUIF. Ocean and adm contain non-linear array subscripts involving multiplicative induction variables that are beyond the scope of the higher-order induction variable recognition. There will always be extensions to an automatic parallelization system that can improve its effectiveness for some programs; nonetheless, there is a fundamental limitation to static parallelization.
Some programs cannot be parallelized with only compile-time information. For example, the main loop in `adm` is parallelizable only if the problem size, which is unknown at compile time, is even. A promising solution is to have the program check if the loop is parallelizable at runtime, using dynamic information. Interprocedural analysis and optimization can play an important part in such an approach by improving the efficiency of the run-time tests. It can derive highly optimized run-time tests and hoist them to less frequently executed portions of the program, possibly even across procedure boundaries. The interprocedural analysis in our system provides an excellent starting point for work in this area.

The advanced analysis can also form the basis for a useful interactive parallelization system. Even when the analyses are not strong enough to determine that a loop is parallelizable, the results can be used to isolate the problematic areas and focus the users' attention on them. For example, when we ran our compiler on the program `qcd`, it finds a 617-line interprocedural loop that is parallelizable except for the static variables in one procedure. Examination of that procedure reveals that the function is a random number generator, which a user can easily modify to run in parallel. By requesting very little help from the user, the compiler can parallelize the loop and perform all the tedious privatization and reduction transformations automatically.

### 6.4.3 Summary

Table 3 summarizes the impact of the improvements from the advanced array analyses on coverage, granularity and speedup in the three benchmark suites. The first row contains the number of programs reported on from each benchmark suite. The second row shows how many programs had their coverage increased to be above 80% after adding the advanced array analyses. The third row gives the number of programs that had increased granularity (but similar coverage) as a result of the advanced array analyses. The fourth row shows how these significant improvements impacted overall performance. For those with either improved coverage or increased granularity, all but 3 have a 2-fold speedup.

<table>
<thead>
<tr>
<th></th>
<th>Spec92FP</th>
<th>NAS</th>
<th>Perfect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Programs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improved Coverage (&gt; 80%)</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Increased Granularity</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Improved Speedup (&gt; 2.0)</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Summary of Experimental Results
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 recognition are key technologies that greatly improve the success of automatic parallelization. By finding coarse-grain parallelism, the compiler increases parallelization coverage, lowers synchronization costs and improves 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.

For some programs, our analysis is sufficient to find the available parallelism. For other programs, it seems impossible or unlikely that a purely static analysis could discover parallelism—either because correct parallelization requires dynamic information not available at compile time or because it is too difficult to analyze. In such cases, we can benefit from some support for run-time parallelization or user interaction. The aggressive static parallelizer we have built will provide a good starting point to investigate these techniques.

Acknowledgements. The authors wish to thank Alex Seibulescu and Patrick Sathyanathan for their contributions to the design and implementation of this system, and the rest of the SUIF group, particularly Chris Wilson and Jennifer Anderson, for providing support and infrastructure upon which this system is built.

References


