Interprocedural Parallelization Analysis in SUIF

MARY W. HALL
USC Information Sciences Institute
SAMAN P. AMARASINGHE
Massachusetts Institute of Technology
BRIAN R. MURPHY and SHIH-WEI LIAO
Intel Corporation
and
MONICA S. LAM
Stanford University

As shared-memory multiprocessor systems become widely available, there is an increasing need for tools to simplify the task of developing parallel programs. This paper describes one such tool, the automatic parallelization system in the Stanford SUIF compiler. This article represents a culmination of a several-year research effort aimed at making parallelizing compilers significantly more effective. We have developed a system that performs *full interprocedural parallelization analyses,* including array privatization analysis, array reduction recognition, and a suite of scalar data-flow analyses including symbolic analysis. These analyses collaborate in an integrated fashion to exploit *coarse-grain parallel loops,* computationally intensive loops that can execute on multiple processors independently with no cross-processor synchronization or communication. The system has successfully parallelized large interprocedural loops over a thousand lines of code completely automatically from sequential applications.

This article provides a comprehensive description of the analyses in the SUIF system. We also present extensive empirical results on four benchmark suites, showing the contribution of individual analysis techniques both in executing more of the computation in parallel, and in increasing the granularity of the parallel computations. These results demonstrate the importance of

This research was supported by the Air Force Material Command and DARPA contracts DABT63-95-C-0118, F30602-95-C-0098 and DABT63-94-C-0054; NSF Contract ACI-9721368; an NSF CISE postdoctoral fellowship; fellowships from Intel Corporation and AT&T Bell Laboratories; and an NSF Young Investigator Award.
Authors' current addresses: M. W. Hall, USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292; email: mhall@ISI.edu; S. P. Amarasinghe, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139; email: saman@lcs.mit.edu; M. S. Lam, Computer Systems Laboratory, Stanford University, Stanford, CA 94305; email: lam@cs.stanford.edu; B. R. Murphy, and S.-W. Liao, Intel Corporation; email: {shih-wei.liao, brian.r.murphy}@intel.com.

interprocedural array data-flow analysis, array privatization and array reduction recognition; a third of the programs spend more than 50% of their execution time in computations that are parallelized with these techniques. Overall, these results indicate that automatic parallelization can be effective on sequential scientific computations, but only if the compiler incorporates all of these analyses.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*array and vector processors*; *parallel processors*; D.3.4 [**Programming Languages**]: compilers; optimization

General Terms: Algorithms, Analysis, Experimentation, Performance

Additional Key Words and Phrases: Data dependence analysis, interprocedural data-flow analysis, parallelization, symbolic analysis

1. INTRODUCTION

Parallel computing is well recognized as a means of achieving the highest performance on compute-intensive applications, but developing an efficient parallel program is substantially more difficult than developing its sequential counterpart. In an effort to simplify the programmer's job, there has been a significant body of compiler research over the last two decades to automatically extract parallel code—both for vector and multiprocessor platforms—from sequential applications. This work has become increasingly important in recent years as multiprocessors become widely available as compute servers.

In the past, automatic parallelization was shown to be successful for vector architectures, for which the compiler's goal is to find *fine-grain* computations (e.g., inner loops in a loop nest) that execute over large amounts of data. When targeting symmetric multiprocessors, however, a series of experiments in the early 90s showed that state-of-the-art commercial parallelizing compilers were frequently not effective at deriving efficient parallel code, even for applications with inherent parallelism [Blume and Eigenmann 1992; Singh and Hennessy 1991].

The compilers' limitations stemmed primarily from their inability to find parallelism beyond the fine-grain innermost loops parallelized by vectorizing compilers. On multiprocessors, fine-grain loops often carry too much synchronization and communication overhead for profitable parallel execution. Efficient parallel code on a multiprocessor must instead exploit *coarse-grain parallelism*, such as that available when large, outer loops are parallelized. Vectorizing compilers relied on *data dependence analysis* as the sole analysis for locating parallelism [Banerjee 1988]. 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. The previously described experiments on commercial parallelizers pointed to the need for additional analyses and transformations, beyond data dependence analysis, for locating coarse-grain parallelism.

A secondary reason for the limitations of these compilers was that even the implementations of their existing analyses were not sufficiently powerful to recognize many constructs that arise in practice. Compilers are very complex software systems. To manage the complexity of a compiler implementation, there is a tendency to simplify the implementation by being conservative in the presence of difficult-to-analyze constructs. Simplification often has little impact on the overall effectiveness of a traditional optimizing compiler, where the goal is to apply hundreds or thousands of small optimizations across the program that in the aggregate produce better performance. However, in a parallelizing compiler, the key to yielding good parallel performance may be parallelizing a single outermost loop in the application; a simplification of one of the analysis components may mean the difference between successful parallelization of the loop and no improvement at all.

Increasing the effectiveness of parallelizing compilers for multiprocessors required a comprehensive approach involving three main components: (1) additional analyses to locate coarse-grain parallelism; (2) a well-integrated and complete suite of parallelization analyses and transformations; and, (3) extensive experimentation to understand the analysis requirements and characterize overall impact of the optimization techniques. This article describes how such an approach was taken in an automatic parallelization system that is part of the Stanford SUIF compiler. This paper represents the culmination of a several-year effort, presenting a thorough description of the compiler's analysis algorithms and implementation, its transformations to parallel code, and extensive experimental results obtained with the system. This article makes three principal contributions, corresponding to the three essential aspects of the approach described above.

The first contribution of this article is its detailed presentation of the analysis and transformation components necessary to successfully exploit coarse-grain parallelism. Coarse-grain outer loops may span multiple procedures; analysis must be as effective across procedure boundaries as within a single procedure. For this reason, the SUIF parallelization system performs *full interprocedural analysis* seamlessly across procedure boundaries, and has successfully parallelized large loops spanning numerous procedures and consisting of up to 1000 lines of code. Locating coarse-grain parallelism involves enhanced array analysis and transformations. In some cases, loops that carry data dependences can be parallelized by transforming array data structures. For example, it is very common for each iteration of a loop to first define and then use the same variable. The compiler must *privatize* the variable, providing each processor with a local copy of the variable, for the loop to be parallelizable. In some cases where that is not possible, a compiler can still parallelize a *reduction* operation (e.g., computation of a sum, product, maximum, or other commutative and associative operation over data elements) by having each processor compute a partial reduction locally, updating a global result only upon completion of the loop. The SUIF system extends previous work on data-flow analysis for array privatization and array reduction recognition, and is designed to be practical and effective in the interprocedural setting.

A second contribution of this article is its description of frameworks that we have used to manage the complexity of building the automatic parallelization system. The interprocedural analysis framework is *region based*, managing the costs of analysis by summarizing data-flow information at program regions such as loops and procedures. Our system also utilizes an *integer linear inequality framework* to simplify the array data-flow analysis implementation.

Through the use of these frameworks, we have developed a comprehensive and well-integrated implementation of parallelization analyses. This article presents a detailed description of the analysis frameworks and individual analyses used to produce these results.

The third contribution of this article is its presentation of extensive experimental results on 39 scientific programs from four different benchmark suites: SPECFP95, the sample NAS programs, SPECFP92, and PERFECT. While a small subset of these performance results have appeared in another article [Hall et al. 1996], this paper gives the first detailed treatment showing contributions of individual analysis components. Overall, the results demonstrate that a comprehensive approach to automatic parallelization can yield effective results on scientific benchmark applications. More than two-thirds of the programs yield parallel speedup on the multiprocessor systems used, the Digital Alphaserver 8400 and the SGI Challenge. Half of the programs owe their speedup to interprocedural analysis, array privatization, and array reduction transformations.

There have been many research compiler systems addressing automatic parallelization, several of which will be discussed later in this article. The two automatic parallelization systems most closely related to SUIF are the Polaris system from University of Illinois [Blume et al. 1996] and the PIPS system from Ecole des Mines [Irigoin et al. 1991; Creusillet and Irigoin 1995]. Like these other systems, SUIF is most successful on scientific array-based applications written in FORTRAN; the analyses for such applications are the focus of this article. Our work on the SUIF system is distinguished from these other systems because it combines a comprehensive, fully interprocedural suite of parallelization analyses with a complete implementation and extensive experimental results.

This article is organized into ten remaining sections. In Section 2, we present the types of advanced analyses required to parallelize full applications. Section 3 presents the region-based interprocedural analysis framework. Section 4 describes the scalar data-flow analyses implemented in the system, while Section 5 presents the integer linear inequality framework and the array data-flow analysis. Section 6 describes the approach to reduction recognition and explains how reductions are parallelized by the compiler. Section 7 puts the analyses together, and Section 8 describes how the compiler uses the results of these analyses to generate parallel code. Related work is the topic of Section 9. Subsequently, Section 10 presents the experimental results, followed by concluding remarks in Section 11.

2. PARALLELIZATION ANALYSIS TECHNIQUES

Parallelizing coarse-grain outer loops requires that a compiler use many techniques beyond the standard analyses available in traditional parallelizing compilers. In this section, we describe briefly the parallelization techniques in our system and give examples extracted from real programs encountered in our experiments to motivate the need for more advanced analysis techniques.

2.1 Analysis of Scalar Variables

2.1.1 *Scalar Parallelization Analysis.* Scalar parallelization analysis locates data dependences on scalar variables. Where there are scalar dependences, this analysis determines whether parallelization may be enabled by two data structure transformations, *privatization* or *reduction*, as defined in Section 1.

For the most part, scalar parallelization analysis involves a straightforward extension of traditional scalar data-flow analysis (live variable, modification, and reference analyses). It also includes an algorithm to locate reduction computations on scalar variables. These analyses are presented in Section 4.1.

2.1.2 *Scalar Symbolic Analysis.* Parallelizing compilers incorporate a host of scalar symbolic analyses, including constant propagation, value numbering, and induction variable recognition. These can eliminate some scalar dependences which cannot be removed by simple privatization. For example, rather than generating an induction variable's value by incrementing the value from the previous iteration, the value can be computed independently for each loop iteration by multiplying the step size by the iteration count and adding this to the initial value of the variable.

The symbolic analyses are even more important for gathering information needed for precise analysis of arrays. Parallelizing compilers typically 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. In such a case, testing for a data dependence has been shown to be equivalent to an integer programming problem and can usually be solved efficiently and exactly [Goff et al. 1991; Maydan et al. 1991]. The symbolic analyses rewrite scalar variable values into a form suitable for integer programming, providing integer coefficients for subscript variables, and deriving affine equality relationships among variables, as discussed in Section 4.2. Some systems also propagate inequality relations and other relational constraints on integer variables imposed by surrounding code constructs (`IF`s and loops) to their uses in array subscripts [Havlak 1994; Irigoin 1992]. SUIF's symbolic analysis is described in Section 4.2.

2.1.3 *Nonlinear Relations on Scalar 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 the result of our symbolic analysis algorithm. Of particular interest is the access to the `XRSIJ` array. To parallelize the outermost loop, we must show that the array index `MRSIJ` has a different value each time it is executed, and thus the write operations to the `XRSIJ` array are independent. Before each iteration of the two innermost loops, `MRSIJ` is set to `MRSIJ0`, which in turn is incremented by `NRS` (= `NUM*(NUM+1)/2`) in each iteration of the outer loop. The value of `MRSIJ` is incremented by one in each of the `MORB*(MORB-1)/2` iterations of the two innermost loops. Now from our interprocedural symbolic analysis, we know that `MORB` has the same value since `NUM` was assigned to `NORB` and `NORB` was passed into

```
DO NUM = 10, 40, 5
  NORB=NUM
  CALL OLDA(...,NORB,...,NUM)

SUBROUTINE OLDA(...,MORB,...,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. (b) Symbolic information.

Fig. 1. Nonlinear induction variable in the Perfect benchmark `trfd`.

`OLDA` as `MORB`. Thus, the assignment `MRSIJ` = `MRSIJ0` only assigns the current value of `MRSIJ` back to itself from the second iteration on. Thus, `MRSIJ` is always incremented by one before it is used as an array index, so all the write accesses are independent, and the outermost loop is parallelizable. Our symbolic analysis implementation recognizes nonlinear subscript expressions of this kind as second-order induction variables. If it can prove certain properties of the variable coefficients of the loop index variable, it rewrites the expressions in an affine form suitable for array analysis.

2.2 Analysis of Array Variables

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

2.2.1 *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 more concisely present these examples, FORTRAN 90 array notation is used in place of the actual loops.) Although the same array locations in `TM` 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 `TM` is accessed by each process. Finding privatizable arrays requires that data-flow analysis previously performed only for scalar variables be applied to individual array elements; this analysis is called *array data-flow analysis*. We describe array data-flow analysis in the SUIF system in Section 5.

2.2.1.1 *Array Privatization with Initialization.* Array privatization is usually only applied to arrays where each iteration of the loop in which the array appears 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

```
DO K = 2, NZ-1
  DO M = 1, 5
    TM(1:5,M) = ...
  DO M = 1, 5
          ... = TM(1:5,M)
```

(a) Array privatization (appbt)

```
DO LAT = 1, 38
  DO K=1, 12
    ZE(2,K) = RELVOR(K)
    CALL UVGLOB(...,ZE(1,K),...)
    ZE(2,K) = ABSVOR(K)
...
SUBROUTINE UVGLOB(...,ZE,...)
REAL ZE(961)

...  = ZE(1:961)
```

(b) Interprocedural array privatization with initialization (spec77).

```
DO LAT = 1, 38
  W(1:2,1:UB)   = ...
  W(3:36,1:UB)  = ...
  W(62:96,1:UB) = ...
  W(37:48,1:UB) = ...
  W(51:61,1:UB) = ...
  W(49:50,1:UB) = ...
    ...         = W(1:2,1:UB)+
                  W(33:34,1:UB)+
                  W(65:66,1:UB)
    ...         = W(3:32,1:UB)+
                  W(35:64,1:UB)+
                  W(67:96,1:UB)
```

(c) Recognizing array summaries across loops (spec77)

Fig. 2. Array data-flow analysis examples.

execution begins. In our system, 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 10.3). Each iteration of the outer loop writes to the second row of array ZE before reading the whole array in the subroutine UVGLOB. The loop is not parallelizable as it is because different iterations are writing to the same second row. However, each iteration only reads the row it writes and therefore the loop can be parallelized if every processor has its own private copy of the array. Since the processor needs to read the original contents of all but the second row of the array, their private copy must be initialized accordingly.

2.2.1.2 *Creating Complicated Array Summaries.* Data-flow analysis on arrays has intrinsically higher space requirements than analysis on scalar variables, as it is necessary to keep track of accesses to individual array elements. Many approaches to array data-flow analysis manage the space costs by summarizing all the accesses to an array in a program region with a single array summary, a convex hull, that represents the conservative approximation of all the accesses. While space efficient, imprecision is often introduced if there are very different accesses to the array in a loop, or if the accesses are spread across multiple loops.

In such cases, the challenge to the compiler is to keep precise enough information for the analysis while maintaining efficiency. Figure 2(c), 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

```
SUM = 0.
DO I = 1, N
  SUM = SUM+A(I)
```

(a) Scalar reduction.

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

(b) Sparse reduction (cgm).

```
DO I = 1, 499
  JBEG = NBINDX(I)
  JEND = NBINDX(I+1)-1
  IF (JBEG .LE. JEND) THEN
    CALL JLOOPU(I,JBEG,JEND,...)

SUBROUTINE JLOOPU(I,JBEG,JEND,...)
  DO JX = JBEG, JEND
    J = NTABL(JX)
    IF ... THEN
      XFORCE(I) = XFORCE(I) + FIJX
      XFORCE(J) = XFORCE(J) - FIJX
```

(c) Multiple reduction statements in an interprocedural reduction (mdljdp2).

Fig. 3. Reduction recognition examples.

from the collection of write operations that W is completely defined before it is read. Had we summarized the effect of the first three array write statements with a convex hull, we would conclude that the program may, but not necessarily, write to the first 96 rows of the array. Our system instead keeps multiple summaries per array for each program region, and only merges the summaries whenever no information is lost by doing so. For example, it is able to merge the summaries of W associated with the first two inner loops (W(1:2,1:UB) and W(3:36,1:UB)), but keeps W(62:96,1:UB) temporarily separate. By doing so, our analysis is able to conclude precisely that the entire array must be overwritten before it is read, thus allowing array privatization be applied.

2.2.2 *Array Reduction Recognition.* Most parallelizing compilers will recognize scalar reductions such as the accumulation into the variable SUM in Figure 3(a). Such reductions can be transformed to a parallel form by creating a private copy of SUM for each processor, initialized to 0. Each processor updates its private copy with the computation for the iterations of the I loop assigned to it, and following execution of the parallel loop, atomically adds the value of its private copy to the global SUM. Reductions on array variables are also common in scientific codes and are a potential source of significant improvements in parallelization results. Section 6 describes reduction recognition in SUIF.

2.2.2.1 *Sparse Array Reductions.* Sparse computations pose what is usually considered a difficult construct for parallelizing compilers. When arrays are part of subscript expressions, a compiler cannot determine 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, so 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 SPECFP92 benchmark `mdljdp2`. The compiler can prove it is safe to parallelize the outer loop (`DO I`) if it performs sparse reduction recognition interprocedurally. This example also demonstrates that reductions in a loop may consist of multiple updates to the same array.

3. INTERPROCEDURAL DATA-FLOW ANALYSIS FRAMEWORK

Automatic parallelization requires the solution of a number of data-flow analysis problems on both scalar and array variables. A distinguishing feature of the SUIF parallelization system is that all of its analyses are performed across procedure boundaries, enabling parallelization of large coarse-grain loops. We derive interprocedural information via interprocedural data-flow analysis, rather than the technique of inline substitution found in some other systems [Blume et al. 1996]. Inline substitution replaces a procedure call with a body of the invoked procedure; thus, with full inlining the compiler analyzes a separate copy of a procedure for every call chain in which it is invoked. For this reason, inline substitution is not practical for large programs because it can lead to unmanageable code expansion (an example of this problem is described in Section 10.3).

Many previous interprocedural parallelization systems perform relatively simple *flow-insensitive* analysis, in which a procedure is analyzed without regard to what control flow paths may be taken through it [Havlak and Kennedy 1991; Li and Yew 1988; Triolet et al. 1986]. Flow-insensitive analysis, while useful for dependence analysis and reduction recognition, is not sufficiently precise for scalar symbolic analysis or array privatization. To capture precise interprocedural information requires a *flow-sensitive* analysis, which distinguishes analysis results along different control flow paths through a procedure. More precision can be achieved if the analysis is also *context sensitive,* analyzing a procedure in light of the different calling contexts in which it is invoked. The interprocedural analysis for parallelization in the SUIF system is both flow and context sensitive.

To manage the software engineering complexity of building interprocedural versions of the suite of analyses required for parallelization, we utilize a common framework, analogous to a traditional data-flow analysis framework [Kam and Ullman 1976]. The common framework, a significant extension of the Fiat [Hall et al. 1993] system, facilitates implementation of interprocedural data-flow analyses by providing parameterized templates for solving data-flow problems. Each analysis problem is implemented by instantiating the templates with functions to compute solutions to data-flow equations. The parameterized template for flow-insensitive analysis has been previously described [Hall et al. 1993]. The flow-sensitive, context-sensitive interprocedural data-flow analysis framework is the topic of this section.

3.1 Overview

3.1.1 *Efficiency vs. Precision Concerns.* Precise and efficient flow- and context-sensitive interprocedural analysis is difficult because information flows into a procedure both from its callers (representing the *calling context* in which the procedure is invoked) and from its callees (representing the side effects of the invocation). For example, in a straightforward interprocedural adaptation of traditional iterative analysis, analysis might be carried out over a program representation called the *supergraph* [Myers 1981], 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 is slow because the number of control flow paths through which information flows increases greatly as compared to intraprocedural analysis, in terms not only of interprocedural loop nesting depth but also of call nesting depth in the call graph. Such an analysis is also imprecise, losing precision by propagating information along *unrealizable paths* [Landi and Ryder 1992]; calling context information from one caller may propagate through a procedure and return to a different caller.

We illustrate these issues by applying interprocedural analysis to the program fragment shown in Figure 4(a). An inlining approach, as in Figure 4(b), results in significant expansion of the compiler's representation of the program. A supergraph-based flow-sensitive analysis, as in Figure 4(c), keeps just one representation of procedure `FOO` but introduces control flow edges from and to each call site and the procedure representation. While saving space, this approach introduces unrealizable paths and complicates the analysis.

A procedure summary approach (or *functional* approach [Sharir and Pnueli 1981]), as in Figure 4(d), avoids this problem to some extent. It computes a summary of a procedure's effect which is used at each call site when analyzing the caller, so that the results at that call site do not mix with those from other call sites, so some context sensitivity is retained. There are two potential problems with the procedure summary approach. One is that for many analysis problems, it is not possible to precisely summarize a procedure's effect. The second problem arises from the fact that such analyses are usually monovariant, computing a single result for each program point. When information is propagated to the called procedures (in a second phase of the analysis), information from different contexts is usually mixed together, since just a single value is retained for each point within the procedure. Thus some context sensitivity is lost.

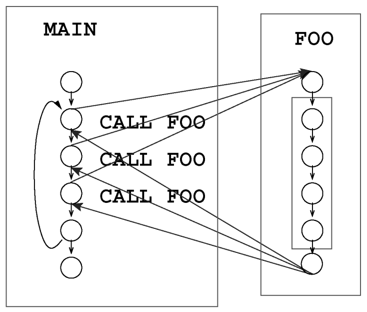
3.1.2 *Region-Based Flow- and Context-Sensitive Analysis.* Our interprocedural framework performs what we call a region-based analysis that avoids the previously described efficiency and precision problems but still provides flow and context sensitivity.

Rather than perform an iterative analysis over an unstructured supergraph, region-based analysis structures the program into a hierarchical representation that allows the aggregation of information at the boundaries of single-entry program regions. Region-based analysis separates side-effect analysis and calling context propagation into two distinct phases. In each phase, analysis information flows in only a single direction in the call graph, from callee to caller in the side-effect analysis and from caller to callee in context propagation. Thus, the system is able to perform an analysis efficiently in just two passes over the procedures in the program, assuming no recursion. The two-phase region-based analysis is similar to what is traditionally called interval analysis, where the intervals of interest in this case are loops and procedure bodies [Allen and Cocke 1976].

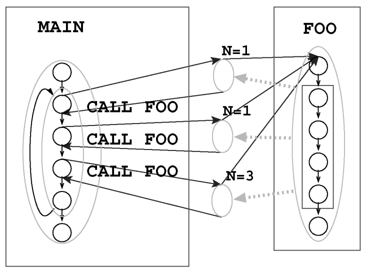
```
PROGRAM MAIN
    DO I=. . .
        CALL FOO(1)
        CALL FOO(1)
        CALL FOO(3)

SUBROUTINE FOO(N)
    ...
```

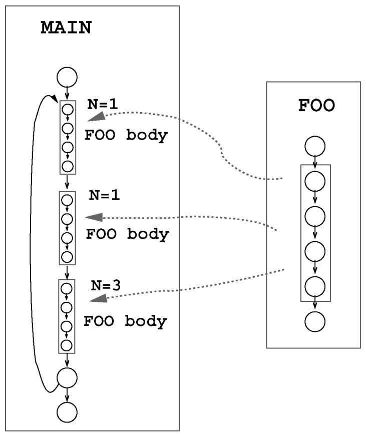
(a) Example Program



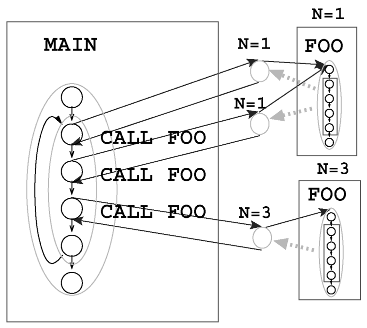
(c) Supergraph Analysis



(d) Procedure Summaries



(b) Inlined Program



(e) Procedure Cloning

Fig. 4. Interprocedural Analysis Techniques.

The analysis framework is immediately applicable to nonrecursive languages without pointers, such as FORTRAN 77. To support recursion would require analysis to determine fixed point summaries for recursive functions; if the called procedure is known for each call site, it would be sufficient to initialize all procedure summaries to an optimistic value and recompute them until they stabilize.

3.1.3 *Selective Procedure Cloning.* With the region-based analysis approach described above, precision may still be lost as compared to intraprocedural analysis of a fully inlined program.

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 leading to the procedure. Such approximations can affect the precision of analysis if a procedure is invoked along paths that contribute very different information. For example, suppose the analysis finds two possible values for variable `N` on entry to `FOO`, it must assume that the initial value of `N` is not a constant in procedure `FOO`.

To avoid this loss of precision, the analysis uses a technique called *selective procedure cloning*, in which the analysis results are replicated for a procedure when the differences in its calling contexts may yield significantly different results [Cooper et al. 1993]. Because the replication is done *selectively* according to the unique data-flow information it exposes, the analysis manages to obtain the same precision as intraprocedural analysis of a fully inlined program at a more reasonable cost. The analysis thus provides a bit of polyvariance (multiple analysis results per program point) where it might be useful. In Figure 4(e), the summarized representation of procedure `FOO` is cloned, yielding two representations corresponding to the two distinct calling contexts, $\mathtt{N} = 1$ and $\mathtt{N} = 3$.

In the SUIF parallelizer, procedure cloning is used to refine the results of symbolic analysis as described in Section 4.2, so that the scalar relations presented to the array analysis are more precise. It is important to note that the analysis does not actually replicate procedure bodies, but rather replicates their data-flow information only—just "virtually" cloning the procedures. Before final code generation, we actually merge results back together so that only a single copy of each procedure remains.

3.2 Background on Data-Flow Analysis

Our interprocedural data-flow analysis framework draws from previous work on single-procedure, global data-flow analysis frameworks [Kam and Ullman 1976], flow-insensitive interprocedural analysis [Ball 1979; Cooper and Kennedy 1984], and interval analysis [Allen and Cocke 1976], which we summarize here. Comprehensive descriptions of global and interprocedural data-flow analysis can be found elsewhere [Marlowe and Ryder 1990; Cooper et al. 1995]. Included in this description are the program representations used by analysis. In this section, we also introduce notation that we will later use in the description of the interprocedural data-flow analysis framework operations in Table I and the algorithm in Figure 5.

3.2.1 *Global Data-Flow Analysis Frameworks.* A data-flow analysis determines properties that are true of all possible execution paths passing through each program point. In a global data-flow analysis framework, analysis is performed only within a single procedure. Execution paths are approximated by a static program representation called a *control flow graph*, a directed multigraph $CFG = \langle CFGN, CFGE, entry, exit \rangle$. $CFGN$ is a set of nodes corresponding to *basic blocks* and *call sites*. A basic block ($b \in BasicBlock$) is a single-entry, single-exit sequence of consecutive statements in the program. A procedure call is represented by a distinct call-site node ($s \in CallSites$). Node $entry \in CFGN$

Table I. Summary of Region-based Analysis Specification by Phase

Direction	Forward or Backward	Flow Direction
	PHASE 1:	
$f \in \mathcal{F}$		transfer function representations
BasicBlockTF	$: BasicBlock \to \mathcal{F}$	provides node transfer function
$\circ$	$: \mathcal{F} \times \mathcal{F} \to \mathcal{F}$	composes two transfer functions
$\bigwedge$	$: \mathcal{F} \times \mathcal{F} \to \mathcal{F}$	performs meet of transfer functions
$\top$	$: \mathcal{F}$	identity for meet operation
Loop	$: \mathcal{F} \times Region \to \mathcal{F}$	summarizes complete behavior of the given loop region, using the given summary of loop body
UpCall	$: CallSites \times \mathcal{F} \to \mathcal{F}$	maps callee transfer function into caller's scope at given call site
	PHASE 2:	
$v \in \mathcal{L}$		data-flow values
v_0	$: \mathcal{L}$	initial program context
$\wedge$	$: \mathcal{L} \times \mathcal{L} \to \mathcal{L}$	performs meet of data-flow values
Partition	$: 2^{\|\mathcal{L}\|} \to 2^{2^{\|\mathcal{L}\|}}$	partitions a set of data-flow values into sets of compatible calling contexts
Enter	$: Region \times \mathcal{L} \to \mathcal{L}$	maps a data-flow value into a region
Iters	$: \mathcal{F} \times Region \to \mathcal{F}$	given the summary of the loop body yields relative transfer function from the entry of the given loop region to the start of an iteration
Apply	$: \mathcal{F} \times \mathcal{L} \to \mathcal{L}$	applies transfer function
DownCall	$: CallSites \times \mathcal{L} \to \mathcal{L}$	maps calling context into callee's scope from given call site

represents the entry to the procedure and has no incoming edges; node $exit \in CFGN$ represents the exit of the procedure and has no outgoing edges. The set of edges is $CFGE \subseteq CFGN \times CFGN$. An edge $\langle n_1, n_2 \rangle$ leaves node n_1 and enters node n_2; n_1 is a predecessor of n_2, and n_2 is a successor of n_1. We denote the set of all predecessors of n_1 in the graph by $Pred(n_1)$.

The *direction* of a data-flow analysis may be either *forward*, describing properties of paths from *entry* to a given program point, or *backward*, describing properties of paths from *exit* to a given program point. A backward analysis is in most ways identical to a forward analysis over the reversed control flow graph. For the remaining discussion, we describe a forward analysis only, highlighting where differences occur.

A global data-flow analysis framework is defined by $(\mathcal{L}, \mathcal{F}, \wedge)$, where $\mathcal{L}$ represents the domain of data-flow values to be propagated over *CFG*, $\mathcal{F} \subseteq \mathcal{L} \to \mathcal{L}$ is a set of transfer functions at nodes, and $\wedge : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ is a binary meet operation on $\mathcal{L}$ at confluence points in *CFG*. The domain of data-flow values $\mathcal{L}$ is characterized as a *semi-lattice*, defined by its idempotent, commutative and associative $\wedge$ operation, and a partial ordering $\leq$ on the values, corresponding to an informational order. A greater value corresponds to more precise knowledge about some program property, a lesser one to more uncertainty about properties. The identity element $\top$ is defined such that $\forall v \in \mathcal{L}, v \leq \top$; it represents an optimistic case where all possible assertions are true.

```
/* PHASE 1: Derive Transfer Functions */
for each procedure $p$ from bottom to top over call graph:
    for each region $r$ from innermost to outermost:
        if $r$ is a basic block $b$,
            $f_r = BasicBlockTF(b)$
        if $r$ is a loop with body $r'$,
            $f_r = Loop(f_{r'}, r)$
        if $r$ is a call at site $s$ to procedure with body $r'$,
            $f_r = UpCall(s, f_{r'})$
        if $r$ is a loop body or procedure body,
            /* without loss of generality, assume a forward data-flow problem */
            iteratively solve for the maximal values of $f_{r,r'}$
                for all $r' \in SubRegions(r)$ satisfying:
                    $f_{r,r'} = \begin{cases} \top & \text{if } r' = entry(r) \\ \bigwedge_{pr \in pred(r')} f_{pr} \circ f_{r,pr} & \text{otherwise} \end{cases}$
            Using this solution, $f_r = f_{exit(r)} \circ f_{r,exit(r)}$

/* PHASE 2: Derive Contextual Values using Transfer Functions and Clone */
$l_{\texttt{Main}} = \{v_0\}$
for each procedure $p$ from top to bottom over call graph $CG = \langle CGN, CGE \rangle$,
    $l_p = \bigcup_{\langle p', p \rangle \in CGE} \{Enter(p, v) \mid v \in l_{\langle p', p \rangle}\}$
    $l_{entry(p)} = \left\{ \bigwedge_{c \in \mathcal{C}} c \;\middle|\; \mathcal{C} \in Partition(l_p) \right\}$
    for each region $r$ in $p$ from outermost to innermost,
        if $r$ is a loop with body $r'$,
            $l_{r'} = \bigcup_{v \in l_r} Apply(Iters(f_r, r), Enter(r, v))$
        if $r$ is a loop body or procedure body,
            for each subregion $r'$ of $r$,
                $l_{r'} = \bigcup_{v \in l_r} Apply(f_{r,r'}, Enter(r, v))$
        if $r$ is a call at site $s$ with corresponding call graph edge $e$,
                $l_e = \bigcup_{v \in l_r} DownCall(s, v)$
```

Fig. 5. Region-based interprocedural analysis algorithm.

The domain of transfer functions $\mathcal{F}$ contains representations f of monotonic functions over $\mathcal{L}$. While the representations might be an executable form of the functions themselves, we still explicitly invoke an operation *Apply* to clarify our later exposition of uses of an f as a function over $\mathcal{L}$.

The data-flow analysis solution v_n at each node n other than *entry* is initialized to $\top$. (The initialization of *entry* depends on the data-flow problem; see Kam and Ullman [1976]). Then, a data-flow analysis solution v_b exiting each basic block b is computed iteratively until convergence by solving the following simultaneous equations:

$$v_b = \bigwedge_{p \in Preds(b)} Apply\ (BasicBlockTF(b), v_p)$$

where *BasicBlockTF*(b) denotes the transfer function for basic block b.

3.2.2 *Interprocedural Flow-Insensitive Analysis.* Flow-insensitive interprocedural data-flow analysis is performed on the call graph, where a single data flow value is computed for each procedure.

A call graph (CG) for a FORTRAN program is a directed, acyclic multigraph $CG = \langle CGN, CGE, \texttt{Main}\rangle$, where CGN is the set of procedures p in the program, and an edge $\langle p1, p2\rangle \in CGE$ uniquely corresponds to a call site s to $p2$ in procedure $p1$. In this article, we sometimes identify the callsite s with the associated call graph edge. `Main` is a special procedure representing the main program body. For FORTRAN, which has limited possibilities for indirect procedure calls, the called procedure at each call site is determined using the analysis described in Hall and Kennedy [1992]. Although FORTRAN programs may not have dynamic recursive calls, they may have a cyclic static call graph; where static recursion is present in the original program, a prepass duplicates procedures as necessary to eliminate the apparent recursion. Thus, we are able to assume the call graph is acyclic. Selective cloning might also be used to duplicate procedures as necessary to gain context sensitivity if desired, as described in Cooper et al. [1993].

In a more general setting where recursion is allowed, flow-insensitive interprocedural analysis could be computed iteratively, similar to the global data-flow analysis framework described above. The analysis is performed on the call graph rather than a control-flow graph. (To maintain the correspondence, and enable the use of standard solution algorithms, if procedure cloning were used, it would have to be applied either in advance or in only a very limited way to allow the appearance of a fixed call graph.) Analogous to the choice of forward or backward flow in data-flow analysis, information may be propagated interprocedurally in a top-down manner, from callers to callees, or in a bottom-up manner, from callees to callers.

The most significant difference between intraprocedural data-flow analysis and interprocedural flow-insensitive analysis is that a variable may be referred to by different names in different procedures, such as when a variable is passed by reference as a parameter at a call site. For this reason, analysis must translate data-flow values across procedure boundaries using the following two operators. For top-down problems, where information is propagated from caller to callee in the call graph, *DownCall* : $CallSites \times \mathcal{L} \rightarrow \mathcal{L}$ takes a data-flow value in the caller's scope and maps it across call site s into a value appropriate for the callee. For bottom-up problems, *UpCall* : $CallSites \times \mathcal{L} \rightarrow \mathcal{L}$ maps a callee's data-flow value across call site s into the caller's scope.

3.2.3 *Interval Analysis.* In the descriptions above, data-flow analysis is computed iteratively. In contrast, *interval analysis*, sometimes referred to as elimination analysis, is a divide-and-conquer approach to data-flow analysis [Allen and Cocke 1976; Ullman 1973; Graham and Wegman 1976; Carroll and Ryder 1987]. Interval analysis has been shown to have a lower asymptotic complexity than iterative analysis [Kennedy 1976].

Interval analysis derives a transfer function for particular subsets of nodes in the program or procedure representation, independent of the rest of the program. In this paper, we refer to such an aggregation of nodes as a *region*. Analysis proceeds by deriving the transfer functions for larger and larger regions until the data flow for the entire program is represented by a single transfer function. Then in a second phase, the aggregation of nodes is reversed. Data-flow information propagates from the rest of program into individual nodes; at each region, the transfer function is applied to the incoming data-flow information. The literature has many different ways of aggregating regions into larger regions. In our approach, regions within a single procedure can be basic blocks, loop bodies and loops.

The first phase of interval analysis operates on transfer functions rather than data-flow values. Each basic block $b \in \mathit{BasicBlock}$ is a region, whose behavior can be defined directly:

$$\mathit{BasicBlockTF}(b) : \mathit{BasicBlock} \rightarrow \mathcal{F}.$$

Subsequently, the transfer functions are derived for larger and larger regions until a transfer function is obtained for the entire program or procedure, as a single region. For interval analysis, the requirements on the lattice $\mathcal{L}$ representation is simplified. We need not have computable lattice operations, other than the application of a transfer function to a value $\mathit{Apply} : \mathcal{F} \times \mathcal{L} \rightarrow \mathcal{L}$. Computation instead is carried out on the transfer function domain $\mathcal{F}$. This domain $\mathcal{F}$ must be a meet semi-lattice, with a meet identity element $\top$. A meet function $\bigwedge : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{F}$ derives a meet of two transfer functions. We compose two transfer functions using $\circ : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{F}$. To derive a valid approximation for a loop's behavior, we provide for a closure operation $* : \mathcal{F} \rightarrow \mathcal{F}$.

To obtain the transfer function for a region, the analysis first finds solutions for subregions by composing transfer functions, starting with the region entry (exit for backward problems), and composing the solution at adjacent regions with the partial solution computed so far. Analysis performs meets of partial solutions at confluence points within the region. The transfer function for a loop is computed by applying the closure operation to the transfer function for the loop body.

The second phase of analysis computes the final data-flow value at each program point. At the beginning of the second phase, the program representation has been reduced to a single region. Assuming a forward data-flow problem, data-flow values on exit from a region are calculated by applying the transfer function of the region to the data-flow values on entry. Within a region, the transfer function of a subregion can be applied to the data-flow value entering that region, to determine the data-flow value entering successor regions. Propagation to subregions proceeds until a data-flow value has been calculated for each basic block.

3.3 Region-Based Analysis

Our flow- and context-sensitive interprocedural data-flow analysis framework is an interval style of analysis, as described above. In our approach, we use a hierarchical program representation called the *region graph* that captures the program call graph and the control flow graphs of the individual procedures. We compute data-flow solutions in a two-phase analysis that derives transfer functions for each region in the first phase and propagates data-flow values in the second phase.

3.3.1 *Representation: The Region Graph.* A *region* is a single-entry body of code corresponding to either a basic block, a procedure call site, a loop body or loop (restricted to those loops corresponding to source program `DO` loops in the implementation), or an entire procedure body. Nodes in the region graph are regions. There are two kinds of edges: *call edges* between a call site and the procedure it invokes, similar to edges in a call graph; and *control flow edges* between any two other kinds of regions—representing control flow between those two regions, within the surrounding region—similar to edges in a control flow graph.

Region graph nodes that represent loops, loop bodies or procedure bodies are hierarchical. Each such region has a corresponding *subregion graph* that represents control flow internal to the region: each node corresponds to a subregion, each edge to a control flow path between them. The subregions of a region r are subsequently referred to with the operation $\mathit{SubRegions}(r)$. The set of control-flow predecessors of a subregion r' within the subregion graph are denoted $\mathit{Pred}(r')$. When selective procedure cloning is employed, each clone of a procedure is represented by a separate region.

Figure 4(e) includes the region graph for the code in Figure 4(a). There is a node for each call to `FOO` in `MAIN`, and the collection of calls and their control flow edges comprise a region for the loop body. An enclosing region represents the whole loop, and another enclosing region represents the body of `MAIN`. There are call edges from the calls to `FOO` to the appropriate clone of `FOO`.

As previously mentioned, each region r has a single entry subregion, which we denote $\mathit{entry}(r)$. We will usually simplify presentation by describing the analysis algorithms as if regions also have a single exit, $\mathit{exit}(r)$. This need not be the case; the necessary modifications to the framework for multiple exits will be discussed in Section 4 as part of the description of the scalar live variable analysis.

3.3.2 *Parameterized Template for Region-Based Analysis.* The algorithm for the interprocedural region-based analysis is shown in Figure 5. To use the framework to solve a data-flow problem requires instantiating the parameterized template, whose domains and operators are summarized in Table I. In the remainder of this article, we will describe specific data-flow analyses by giving definitions for the entries in Table I.

Phase 1: *Calculating Region Transfer Functions.* Phase 1 of the algorithm computes the transfer function f_r for each region r from innermost loop to outermost loop, and from bottom to top in the call graph. In the algorithm and Table I, $\top$, *BasicBlockTF*, and $\bigwedge$ are as defined in Section 3.2.

A few additional definitions are required. To map the transfer function of a procedure to the call site scope, we need *UpCall* : *CallSites* $\times \mathcal{F} \rightarrow \mathcal{F}$, an operation in the transfer-function domain corresponding to the *UpCall* operation in the value domain defined in Section 3.2.

For loop bodies or procedure bodies r, the transfer function is a combination of the transfer functions of the subregions of r. For all subregions r', we compute the *relative transfer function* from the entry of the outer region r to the entry of the subregion r'; we denote this transfer function by $f_{r,r'}$. These functions are computed as the solution to a set of equations—each subregion's transfer function is derived from the transfer functions of predecessors in the subregion graph.

When the subregion graph is acyclic, the solution is obtained in a single pass over the subregions, in reverse postorder within the region, so that a subregion is visited before its successor subregions. If the subregion graph is cyclic, as when there are unstructured loops built with `GOTO`, then analysis is iterative within the subregion. In either case, the final transfer function for the loop body or procedure body is derived from the relative transfer function $f_{r,exit(r)}$ for the exit subregion of the region r.

For call sites, the transfer function is obtained from the called procedure's transfer function by the *UpCall* operator, which performs problem-specific actions such as renaming variables in the representation from formal to actual parameters.

For loops r with body r', the loop transfer function is obtained from the transfer function of the loop body $f_{r'}$ by applying a closure operator $Loop(f_{r'}, r)$, which incorporates loop bounds and eliminates loop-varying variables such as the loop index. This is similar to the $*$ operation more typical in interval analyses, but we provide the loop region as a parameter, to allow use of more specialized information, such as the loop index bounds.

In many cases $Loop(f_{r'}, r)$ will be derived from the value of $Iters(f_{r'}, r)$, the relative transfer function from loop entry to loop body; for example, in symbolic analysis, recurrences are solved by the *Iters* operation, then the loop index variable is projected away to yield the result of *Loop*. The relative transfer function itself, computed by *Iters*, is not needed itself until Phase 2, however, so the definition of *Iters* is associated with Phase 2 rather than Phase 1.

Phase 2: *Deriving Calling Contexts and Computing Final Values.* The second phase of the algorithm derives the data-flow value holding on entry to each region, and applies the transfer function to a region's possible inputs to obtain final data-flow values. The analysis gains context sensitivity by propagating not just a single value to each region but instead a *set* of values. When a procedure has several call sites, distinct values from different contexts can be kept distinct. In this way, *procedure cloning* for the purposes of analysis is accomplished.

In the following, $\top$, $\wedge$, *Apply*, and *DownCall* are as defined in Section 3.2. We also define an initial data-flow value v_0 for the entry of the program. For a loop, we need an iteration-summary operation *Iters* : $\mathcal{F} \times Region \to \mathcal{F}$, which tells how the value changes in going from the entry of the loop to the entry of the body on some iteration (the count of previous loops is given by a loop index associated with the loop *Region*). This transfer function might subsequently replace the index variable with its range or upper bound, depending on the data-flow problem being solved, to derive the behavior of the loop. The *Enter* : $Region \times \mathcal{L} \to \mathcal{L}$ operation transforms a value on entry to a region, eliminating irrelevant information such as properties of global variables not accessed within the region.

One additional definition is specifically related to selective procedure cloning. For a procedure p, the analysis first derives the set of calling contexts $\{l_{\langle p',p\rangle} \mid \langle p', p\rangle \in CGE\}$ contributed by the calls to p from other procedures. In the absence of procedure cloning, analysis would perform a meet using $\wedge$ over all of the calling contexts. With procedure cloning, contexts which are different are not combined, but are kept separate. The operation *Partition* : $2^{|\mathcal{L}|} \to 2^{2^{|\mathcal{L}|}}$ divides the contexts entering a procedure into sets representing equivalence classes; the contexts in the same equivalence class are combined with a meet function.

Analysis proceeds from top to bottom in the call graph, and from outermost to innermost loops within each procedure body. For each procedure p, analysis derives l_p, the set of data-flow values contributed by the predecessors of procedure p. Then, the *Enter* operation is used to filter irrelevant information from the elements of l_p. Subsequently, the data-flow values in l_p are partitioned using the *Partition* operation. Within each partition, the meet of its values is computed using the $\wedge$ operation. The data-flow value resulting from this meet operation is the initial value, representing the partition, that is propagated within the procedure.

Within a region, each subregion is analyzed. For a region representing a call site, the value is mapped across the call using the *DownCall*, to be used in analyzing the called procedure. For propagation of data-flow values within a region, first analysis maps the incoming data-flow value into the region using *Enter*. Then, relative transfer functions $f_{r,r'}$ are used to map the value from a region r to each immediate subregion r'. In the case of a loop body or procedure body, these relative transfer functions were computed in Phase 1. At loops, the *Iters* operation is applied to the transfer function for the loop body, to map the loop entry value to the entry of a loop iteration. This may have been already computed in Phase 1.

3.3.2.1 *Backward Data-Flow Problems.* In the first phase, analyses with backward flow are analogous to the forward version presented here, but performed over a reversed control flow graph. Thus, the *entry* node of a region r appears at the exit, and the control-flow *successors* (rather than predecessors) contribute to a subregion's value. For each immediate subregion r', a transfer function $f_{r,r'}$ describes the effect of backwards flow from the exit of r to the *exit* of r'.

One complication of the backward analysis is that we allow regions to have multiple exits, so a backward analysis must handle the equivalent of multiple entries. For example, a `DO` loop might have `GOTO` statements which exit the loop, or any piece of code might have an early RETURN from the enclosing procedure. There is also the interesting case of a loop without any exits (typically calling an exit procedure somewhere inside), from which the procedure exit is unreachable.

For the parallelization analyses, since loops containing early exits and RETURN statements are not candidates for parallelization, backward analyses whose goal is to find parallelizable loops can handle such cases precisely enough by just marking the enclosing loops not parallel and going on to the next loop. One of the parallelization analyses, scalar live variable analysis, must analyze such loops more precisely to determine if they redefine or reference variables that appear in other parts of the program (e.g., to determine if such a variable is live on exit from another loop).

For parallelization, our experience suggests that arbitrary exits need not be handled exactly. There are just three important cases: normal region exits, early procedure returns, and GOTO statements branching to just after the immediately enclosing loop (like a C break statement). To be able to precisely describe the transfer function for a loop body, therefore, a triple of simple transfer functions are used: one describing region executions that lead to a normal exit, one describing executions leading to a procedure return, and one describing executions leading to an early loop exit (break). In a more general application, as long as a finite number of such exits exist, a similar tuple of transfer functions would suffice. Such is the case in a language with structured control flow and exceptions. Completely unstructured control flow (GOTO into or out of loops) would have to be handled by dismantling the regions. The case of an infinite loop can be handled conservatively by inserting an artificial exit somewhere within each loop, or more precisely by adding a transfer function to program exit to the tuple; this exit transfer function would be empty except for regions containing a call to the exit procedure.

The template for the second phase looks the same regardless of the direction of data flow, although the interpretation is different. For a forward problem, the transfer function $f_{r,r'}$ describes the effects of the portion of region r from its entry through subregion r', so applying the transfer function $f_{r,r'}$ to a value that holds at r gives the desired result. For a backward problem, applying the transfer function $f_{r,r'}$ to a value that holds at r produces an absolute value for r' with respect to the exit of the region. Thus, for a backward problem the calling context of a call site actually refers to the data-flow information that holds at the return point of the call rather than the entry point.

3.3.2.2 *Context-Independent Analyses.* Certain data-flow problems are specified such that the data-flow values are independent of region context. For example, the array data-flow analysis of Section 5 is defined to compute the array reads and writes before the current point and after the most recent region entry. In such a case, the context information is irrelevant, as the privatizability of an array section does not rely on knowing what array is referred to; all information of interest is independent of context, so we are able to omit the second phase of the analysis algorithm for the array analysis. In this case, we need not provide the Phase 2 operators, but just those shown in Phase 1 of Table I.

3.3.2.3 *Region-Based Flow-Insensitive Analyses.* The regions and nesting relationships between them form a tree for each procedure. This can be combined with the call graph to obtain more localized results in flow-insensitive analysis problems. We refer to this combined graph as the interprocedural region nesting graph.

4. SCALAR DATA-FLOW ANALYSIS

The scalar data-flow analysis in the parallelization system involves two separate analyses: scalar parallelization analysis and scalar symbolic analysis.

4.1 Scalar Parallelization Analysis

A number of standard analyses ensure that scalar variables do not limit the parallelism available in a loop; we refer to these analyses collectively as *scalar parallelization analysis*. These analyses locate scalar dependences, privatizable scalars and opportunities for scalar reduction transformations. In our system, all operate interprocedurally.

4.1.1 *Scalar Dependences.* To detect scalar dependences, we use a simple flow-insensitive mod-ref analysis [Banning 1979]. For each region r corresponding to a loop, it computes the sets of scalar variables that may be modified or referenced by the loop, Mod_r and Ref_r, respectively. The variables in $(Mod_r \cap Ref_r)$ induce a scalar dependence. As a useful byproduct, the analysis computes Mod_r and Ref_r for every other region r; these are used to optimize the other scalar analyses. These analyses use the simpler flow-insensitive analysis template in the interprocedural framework, and are described in more detail elsewhere [Hall et al. 1993].

4.1.2 *Scalar Reductions.* With a straightforward extension, flow-insensitive mod-ref analysis also provides the necessary information to locate scalar reductions in the loop. The set $Reduction_r$ contains all scalar variables carrying dependences in loop r which can be removed by a reduction transformation. It is computed by tagging each modification and reference which occurs as part of a commutative and associate update of a variable. If every access to a variable occurs in such an update with the same operation, that variable is included in this set. These analyses are combined with the scalar mod-ref analysis above.

4.1.3 *Scalar Privatization.* A flow-sensitive live-variable analysis is used to detect privatizable scalar variables. At each loop r, the system tries to privatize all scalar variables z in $(Mod_r \cap Ref_r) - Reduction_r$; these variables induce dependences not removable by a reduction transformation. A scalar variable z is privatizable iff z has no *upwards-exposed reads* in the body of loop r; a read reference is upwards exposed to the beginning of r if there exists a definition outside of r that may reach the read. The live variable information indicates whether a privatized variable needs to be *finalized*, where the compiler copies the final value from a private copy into the shared variable. Those variables that are not live on exit from a loop need not be finalized.

Table II. Analysis Specification for Scalar Live Variable Analysis

Direction: Backward		
PHASE 1:		
Transfer function $f \in \mathcal{F}$ is a pair of variable sets $\langle Gen, Kill \rangle$		
$BasicBlockTF(b)$	$=$	$\langle Gen_b, Kill_b \rangle$, where Gen_b have upwards-exposed reads in b and $Kill_b$ are vars with assignments in b
$f_{r1} \circ f_{r2}$	$=$	$\langle (Gen_{r1} - Kill_{r2}) \cup Gen_{r2}, Kill_{r1} \cup Kill_{r2} \rangle$
$f_{r1} \wedge f_{r2}$	$=$	$\langle Gen_{r1} \cup Gen_{r2}, Kill_{r1} \cap Kill_{r2} \rangle$
$\top$	$=$	$\langle \emptyset, \text{Vars} \rangle$
$Loop\,(f_{Body}, r)$	$=$	$\langle Gen_{Body}, \emptyset \rangle$
$UpCall\,(s, f_{p2})$	$=$	$\langle \{g \mid g \in (Gen_{p2} \cap Globals)\} \cup$ {actual | actual passed to formal $\in Gen_{p2}$ at s}, $\{g \mid g \in (Kill_{p2} \cap \text{Globals})\} \cup$ {actual | actual passed to formal $\in Kill_{p2}$ at s}$\rangle$ where s is a call to $p2$
PHASE 2:		
Value $v \in \mathcal{L}$ is a set of live variables V		
v_0	$=$	$\{\}$
$v_1 \wedge v_2$	$=$	$V_1 \cup V_2$
$Partition(l)$	$=$	$\{l\}$
$Enter\,(r, v_r)$	$=$	$v_r \cap (Mod_r \cup Ref_r)$
$Iters\,(f_{Body}, r)$	$=$	$\langle Gen_{Body}, \emptyset \rangle$
$Apply(f, v)$	$=$	$(V_v - Kill_f) \cup Gen_f$
$DownCall\,(s, v)$	$=$	$\{g \mid g \in (V_v \cap Globals)\}$ $\cup$ {formal | formal is returned to actual $\in v$ at s}

Finalization is more difficult to implement for a variable which is not always written on every loop iteration. To finalize such a variable, it is necessary to determine which iteration is the last one to write it, and copy the value written to the privatized variable into the shared variable. This requires either some additional analysis or some synchronization and communication at run time to determine which processor has the latest value. Fortunately, our experiments suggested that it is unnecessary to privatize scalar variables requiring finalization, since not a single loop in the benchmarks can take advantage of such a technique.

The live variable analysis is similar to the standard intraprocedural live variable analysis based on GEN/KILL transfer functions. It is implemented as a two-phase region-based backward analysis, by instantiating the framework with the operations shown in Table II. In Phase 1, the transfer function for each region is computed in the standard way as a pair of sets: *Gen* contains variables with upwards-exposed reads in the region, and *Kill* contains variables written in the region [Aho et al. 1986]. In Phase 2, the set of live variables on entry to a region is determined from the set of variables live on exit from the region. At procedure calls, formal parameters are mapped to actual parameters (and vice versa in Phase 2), global variables (from *Globals*) are retained across the call, and local variables are eliminated.

We do not perform cloning based upon this analysis; the *Partition* operator always merges all contexts into a single equivalence class. Employing cloning could not increase the number of variables found to be privatizable, but

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

$$\texttt{A(J) = A(K)} \longleftarrow \begin{cases} \text{here, } \mathtt{J} = \mathtt{J}_{\text{entry}} + 2\mathtt{i} \\ \qquad \mathtt{K} = \mathtt{J}_{\text{entry}} + 2\mathtt{i} + 1 \\ \text{where } \mathtt{i} = \mathtt{I} - 1 \end{cases}$$

Fig. 6. Simple loop needing symbolic analysis.

potentially could have decreased the number of variables needing finalization for some calling contexts. If finalization had been necessary, then we would have cloned procedures for cases when a variable is live on exit only for some calling contexts.

4.2 Support for Array Analysis: Scalar Symbolic Analysis

The array data-flow dependence and privatization analysis of Section 5 is more effective when the array subscript expressions it encounters are phrased as affine expressions in terms of loop indices and loop invariants. For example, the loop in Figure 6 is parallelizable because `A(J)` and `A(K)` refer to a disjoint set of locations. The array analysis cannot compare the two subscripts, however, unless the references to variables `J` and `K` are rewritten in terms it can understand.

A host of traditional scalar analyses are typically used to simplify array subscripts: constant propagation, induction and loop-invariant variable detection, common subexpression recognition, and value numbering. Some of these analyses, such as induction variable recognition, can also eliminate some scalar dependences. Our system combines the effect of such analyses in a single interprocedural *symbolic analysis*.

This symbolic analysis determines, for each variable appearing in an array access or loop bound, a symbolic value: an expression describing its value in terms of constants, loop-invariant *symbolic constants*, and *loop indices*. In the example in Figure 6, the values of `J` and `K` in an iteration of the body are rephrased in terms of a symbolic constant $\mathtt{J}_{\text{entry}}$, representing the value of `J` on entry to the loop, and loop index `i`, indicating the number of previous iterations.

A value represented in these terms is easily recognized as an induction variable (if it contains loop indices), a loop-invariant value (if it contains variables), a constant (if it is representable), or none of these (if it cannot be represented). Common subexpressions are recognized as shared subterms of symbolic value expressions. By using the symbolic expressions to rewrite the program some true scalar dependences across loop iterations can be eliminated.

The symbolic analysis is able to produce more information than the array data-flow analysis can consume directly. The array analysis is precise only for affine array indices, but the symbolic values resulting from the symbolic analysis may be nonaffine. In one common case of nonaffine array indices—those resulting from a higher-order induction variable such as the example in Figure 1—symbolic analysis recognizes and simplifies the nonaffine symbolic value, providing it in an affine form which is more useful to the array data-flow analysis. Since this information is provided in a slightly different form, we first describe the symbolic analysis without the support for second-order induction variables, and then present the necessary extensions.

4.2.1 *Symbolic Value Analysis.* The symbolic analysis finds, for each program point, a symbolic expression for each live variable's value in terms of constants, loop indices, and symbolic constants. The symbolic data-flow value domain and transfer function domain are both based on structures called *symbolic maps*.

4.2.2 *Representation*

4.2.2.1 *Symbolic Values.* A *symbolic expression* (*symexpr*), as defined in the equation below, is an arithmetic or conditional expression in terms of constants, symbolic constants, variables, and loop indices, where *constant* is a program constant, such as a literal integer or character or real, and *var* is a source program variable. We use idx_r to refer to the normalized count of previous iterations of the body of a loop r. The value of a loop index becomes undefined if it passes out of its loop, following FORTRAN semantics. A *symbolic constant* (*symconst*) refers to the unknown or nonaffine value of a variable at the last execution of a particular static program point within a single execution of some program region; the value is undefined outside of that region, since there may not be an unambiguous "single execution" of that region to which we may refer. A symbolic constant $SC\left(var, r, 0\right)$ is introduced for each region r to refer to the value of each variable *var* whose value is unknown or nonaffine on entry. A symbolic constant $SC(var, r, d)$ is introduced to refer to the value assigned to a variable *var* at definition d whose smallest enclosing region is r.

A *symbolic value* (*sym*) is either a symbolic expression or unknown:

$$\begin{aligned} symexpr :=\ & constant \mid symconst \mid var \mid idx \\ & \mid -symexpr_1 \mid \texttt{not}(symexpr_1) \mid \ldots \\ & \mid symexpr_1 + symexpr_2 \mid symexpr_1 \times symexpr_2 \mid \ldots \\ & \mid (symexpr_1 \rightarrow symexpr_2\ ;\ symexpr_3) \\ sym :=\ & \texttt{unknown} \mid symexpr \end{aligned}$$

The arithmetic and Boolean expressions have the same meaning as in the source language. The conditional expression "$a \rightarrow b\ ;\ c$" has value c if a evaluates to **true**, and b otherwise. The implementation uses a set of simple algebraic identities to reduce the size of the symbolic expression representation and to simplify comparisons. For example, properties such as "$0 + x \equiv x$", "$1 \times x \equiv x$", "$0 \times x \equiv 0$", and "$(\ \mathbf{true} \rightarrow x\ ;\ y) \equiv x$" are exploited. This simplification occurs whenever an expression is created or modified.

4.2.2.2 *Sym bolic Maps.* A *symbolic map* is a representation of a total function from variables to symbolic expressions. The representation of a symbolic map *SM* consists of a set of pairs binding variables var_i to symbolic descriptions of their values sym_i:

$$SM = \{\langle var_1, sym_1\rangle, \langle var_2, sym_2\rangle, \ldots\}$$

A map may contain at most one pair for each distinct *var*. The pairs designate the value of the represented function on each variable. For any variable *var* not in the symbolic map, an absolute map yields the default value unknown, while a relative map yields *var* itself as a symbolic expression:

$$SM(var) = \begin{cases} sym & \textit{if } \langle var, sym\rangle \in SM \\ var & \textit{otherwise, if SM is relative} \\ \texttt{unknown} & \textit{otherwise, if SM is absolute} \end{cases}$$

A symbolic map may be either an *absolute map* or a *relative map*. The function represented by an absolute map may never return an expression containing a program variable. Relative maps are not so constrained. The distinction is made by context. The set of all symbolic maps *SM* we denote SymMaps. Absolute maps are $AM \in$ AbsMaps and relative maps are $RM \in$ RelMaps.

We define several basic operations on all symbolic maps *SM*:

—*Application:* $SM(sym)$ replaces every occurrence of a variable *var* in *sym* with $sym' = SM(var)$, then applies algebraic simplification. If the simplified expression contains any occurrence of unknown, then the application yields unknown; otherwise, it yields the simplified expression. Whenever *sym* is a constant, $SM(sym)$ returns *sym*.

—*Asymmetric Union:* $SM_1 \uplus SM_2 = SM_1 \cup \{\langle var, sym\rangle \mid \langle var, sym\rangle \in SM_2$ *and var is not bound by* $SM_1\}$. This operator includes all explicit bindings from SM_1, and only those bindings from SM_2 for variables not found in SM_1. It is used to combine a precise but incomplete description SM_1 of a loop's fixed-point behavior with a complete but less precise solution SM_2. The result is an absolute map if SM_1 and SM_2 are both absolute.

—*Meet:* $SM_1 \wedge SM_2 = (SM_1 \cap SM_2) \uplus \langle\{\langle var, \texttt{unknown}\rangle \mid \langle var, sym\rangle \in (SM_1 \cup SM_2)\}$. Meet builds a map with all bindings common to both SM_1 and SM_2, mapping variables whose bindings are not identical in both to unknown. If both SM_1 and SM_2 are absolute maps, their meet is an absolute map.

—*Composition:* $SM_1 \circ SM_2 = \{\langle var, SM_2(sym)\rangle \mid \langle var, sym\rangle \in SM_1\} \uplus SM_2$. Composition yields a new map which, when applied, has the effect of applying SM_1 and then SM_2. If SM_2 is absolute, then the result is absolute.

—*Restrict Domain:* $SM \downarrow V = \{\langle var, sym\rangle \mid \langle var, sym\rangle \in SM, var \in V\}$. Selects only bindings for variables in V.

—*Conditional Combination:* $(sym \rightarrow SM_1\ ;\ SM_2) = \{\langle var, (sym \rightarrow sym_1; sym_2)\rangle$ such that *var* is bound in SM_1 or SM_2, $sym_1 = SM_1(var)$, $sym_2 = SM_2(var)$, and $sym_1 \neq sym_2\} \uplus (SM_1 \wedge SM_2)$. Builds a conditional expression for every variable bound to different values in the two maps (for a more precise meet result in the case when the meet predicate *sym* is known exactly). The result is absolute if both arguments are absolute.

To simplify presentation of the symbolic analysis algorithm, we also first define some auxiliary map definitions and operations on maps; these will be used in the data-flow problem definition.

—*Parameter Map:*
$ParamMap_s = \{\langle \text{actual}, \text{formal}\rangle \mid \text{actual } \textit{is passed to formal at call site } s\}$.
$ParamMap_s$ is a relative symbolic map that equates actual parameters to their corresponding formal parameters. Because an actual parameter may be an expression and not just a variable, $ParamMap_s$ may contain additional mappings that relate a variable representing the actual parameter to a symbolic expression that defines its value.

—*Inverse Parameter Map:*
$ParamMap_s^{-1} = \{\langle \text{formal}, \text{actual}\rangle \mid \text{actual } \textit{is passed to formal at call site } s\}$.
This relative symbolic map equates formal parameters to symbolic expressions corresponding to the actual parameters passed at the call.

—*Loop Varying Variables:* ("0th-order" approximation to loop fixed-point.)
$Fix_0(RM, r) = \{\langle var, \texttt{unknown}\rangle \mid \langle var, m\rangle \in RM, m \neq var\}$.
This operator takes a relative symbolic map representing the effect of the body of a given loop r and returns a relative symbolic map that sets to unknown symbolic values of variables that are modified within the loop.

—*Induction Variables:* ("1st-order" approximation to loop fixed-point.)
$Fix_1(RM, r) = \{\langle var, var + c \times idx_r\rangle \mid \langle var, var + c\rangle \in RM, Fix_0(RM, r)(c) \neq \texttt{unknown}\}$. This operator takes a relative symbolic map representing the body of a loop r, recognizes variables that are incremented by a constant c (an expression containing no variables modified in RM) on each iteration of the loop (i.e., induction variables), and returns a new relative symbolic map where the symbolic expression associated with each such variable is a linear function of the normalized loop index variable of r.

4.2.3 *The Symbolic Analysis Specification.* The symbolic analysis is a data-flow problem that computes an approximation to a total function from variables to symbolic values at each program point. We solve this data-flow problem by instantiating the region-based analysis framework as shown in Table III.

The data-flow values are functions from variables to symbolic values, which we represent by an absolute symbolic map. The data-flow value domain is thus the set of absolute maps AbsMaps. A safe program entry value is the absolute map {} (mapping all variables to `unknown`). The meet operation is just the symbolic map meet operation defined above.

Transfer functions over AbsMaps are represented as relative symbolic maps; a transfer function is applied by composing the (transfer function) relative map with the (value) absolute map, yielding an absolute map. The region-based data-flow framework also requires the existence of an identity element for the $\wedge$ operation in the transfer function domain $\mathcal{F}$. The set of relative maps RelMaps is lacking such an element, so we lift the domain to $\text{RelMaps}^{\top} = \text{RelMaps} \cup \{\top\}$, where $\top$ is defined to be the identity for $\wedge$. We extend all other symbolic map operations to $\text{RelMaps}^{\top}$ by making them strict over $\top$ (if a parameter is $\top$, then the result is $\top$).

At a basic block the analysis derives a relative map showing the effect of the block on every variable modified in the block. A modified variable is mapped to some expression in terms of constants and variable values on block entry

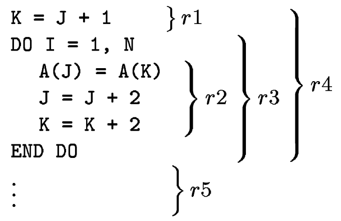
Table III. Analysis Specification for Scalar Symbolic Analysis

Direction: Forward		
PHASE 1:		
Transfer function $f \in \mathcal{F}$ is a relative symbolic map $RM \in \text{RelMaps}^{\top}$		
$BasicBlockTF(b)$	$=$	$\{\langle var, sym\rangle \mid b$ assigns var to a value representable as $sym\} \cup$ $\{\langle var, SC(var, r, d)\rangle \mid b$ assigns unknown value to var at definition d, and r is the nearest enclosing region$\}$
$f_{r1} \circ f_{r2}$	$=$	$RM_{r1} \circ RM_{r2}$
$f_{r1} \wedge f_{r2}$	$=$	$RM_{r1} \wedge RM_{r2}$
$\top$	$=$	$\top$ of $\text{RelMaps}^{\top}$
$Loop(f_{Body}, r)$	$=$	$((ub_{idx_r} \geq 1) \rightarrow (RM_{Body} \circ Iters(f_{Body}, r) \circ \{\langle idx_r, ub_{idx_r} - 1\rangle\})$ $;\ (Iters(f_{Body}, r) \circ \{\langle idx_r, \min(ub_{idx_r}, 0)\rangle\}))$
$UpCall(s, f_{p2})$	$=$	$(ParamMap_s \circ RM_{p2} \circ ParamMap_s^{-1}) \downarrow (Locals_{p1} \cup Globals)$, where s is a call from $p1$ to $p2$
PHASE 2:		
Value $v \in \mathcal{L}$ is an absolute symbolic map AM		
v_0	$=$	$\{\}$
$v_1 \wedge v_2$	$=$	$AM_1 \wedge AM_2$
$Partition(l) =$	$=$	$\{\{AM\} \mid AM \in l\}$
$Enter(r, v)$	$=$	$\{\langle var, sym'\rangle \mid \langle var, sym\rangle \in AM_v \downarrow Gen_r,$ $sym' = \begin{cases} sym & \text{if linear} \\ SC(var, r, 0) & \text{otherwise} \end{cases}$ $\}$
$Iters(f_{Body}, r)$	$=$	$Fix_1(RM_{Body}, r) \uplus Fix_0(RM_{Body}, r)$
$Apply(f, v)$	$=$	$RM_f \circ AM_v$
$DownCall(s, v)$	$=$	$(ParamMap_s^{-1} \circ AM_v) \downarrow (Locals_{p2} \cup Globals)$, where s is a call to procedure $p2$

when possible. If the value assigned to the variable is not representable, a new symbolic constant is introduced to represent it.

At a loop with body transfer function f_{Body}, we compute the transfer function to an iteration, $Iters(f_{Body}, r)$, while computing the loop closure $Loop(f_{Body}, r)$. $Iters(f_{Body}, r)$ is a combination of two approximations of the iterative behavior of the loop using the transfer function representation RM_{Body}. $Fix_1(RM_{Body}, r)$ replaces symbolic expressions for induction variables by expressions in terms of the normalized loop index variable; $Fix_0(RM_{Body}, r)$ replaces symbolic expressions for all variables modified in the loop by `unknown`.

The closure operation $Loop(RM, r)$ composes $Iters(f_{Body}, r)$ with a map setting the final value of the normalized loop index variable for loop r to derive the value of each induction variable on loop exit. The expression for the upper bound ub_{idx_r} on the normalized loop index is derived from the original loop bounds. In cases where the loop is known to execute at least one iteration, we can obtain more precise information than is provided by the iteration operator (which implicitly assumes that the loop might be skipped). To account for these cases we build a conditional combination of maps which separates the case where the loop is known to execute at least once and handles it specially (effectively peeling the last iteration). The implementation of the conditional map operator pushes the conditional into the symbolic value for each variable.



```
K = J + 1
DO I = 1, N
  A(J) = A(K)
  J = J + 2
  K = K + 2
END DO
⋮
```

(a) A simple loop.

$$\begin{aligned}
RM_{r1} &= \{\langle \mathtt{K}, \mathtt{J}+1\rangle\}\\
RM_{r2} &= \{\langle \mathtt{J}, \mathtt{J}+2\rangle, \langle \mathtt{K}, \mathtt{K}+2\rangle\}\\
RM_{r3} &= \mathit{Loop}\,(RM_{r2}, r3)\\
&= \{\langle \mathtt{J}, \mathtt{J}+((\mathtt{N}\geq 1)\rightarrow 2\mathtt{N}\ ;\ 0)\rangle,\\
&\quad\ \langle \mathtt{K}, \mathtt{K}+((\mathtt{N}\geq 1)\rightarrow 2\mathtt{N}\ ;\ 0)\rangle\}\\
RM_{r4} &= \{\langle \mathtt{J}, \mathtt{J}+((\mathtt{N}\geq 1)\rightarrow 2\mathtt{N}\ ;\ 0)\rangle,\\
&\quad\ \langle \mathtt{K}, \mathtt{J}+((\mathtt{N}\geq 1)\rightarrow 2\mathtt{N}+1\ ;\ 1)\rangle\}
\end{aligned}$$

(b) Transfer function map for each region.

$$\begin{aligned}
AM_{r1} &= \{\langle \mathtt{J}, \mathtt{J}_{r4}\rangle, \langle \mathtt{N}, \mathtt{N}_{r4}\rangle\}\\
AM_{r2} &= \{\langle \mathtt{J}, \mathtt{J}_{r4}+2\mathtt{i}\rangle,\\
&\quad\ \langle \mathtt{K}, \mathtt{J}_{r4}+2\mathtt{i}+1\rangle\}\\
AM_{r3} &= \{\langle \mathtt{J}, \mathtt{J}_{r4}\rangle, \langle \mathtt{K}, \mathtt{J}_{r4}+1\rangle, \langle \mathtt{N}, \mathtt{N}_{r4}\rangle\}\\
AM_{r4} &= AM_{r1}\\
AM_{r5} &= \{\langle \mathtt{J}, \mathtt{J}_{r4}+((\mathtt{N}_{r4}\geq 1)\rightarrow 2\mathtt{N}_{r4}\ ;\ 0)\rangle,\\
&\quad\ \langle \mathtt{K}, \mathtt{J}_{r4}+((\mathtt{N}_{r4}\geq 1)\rightarrow 2\mathtt{N}_{r4}+1\ ;\ 1)\rangle\}
\end{aligned}$$

(c) Value maps on entry to each region.

Fig. 7. Example of symbolic maps.

For a given callsite s, the *UpCall* operation maps symbolic expressions in the callee name space to equivalent ones in the caller name space, and removes mappings for variables not visible in the caller. First, this operation composes the actual-formal parameter mapping $\mathit{ParamMap}_s$ with the *RM* representing the procedure body of the callee. This retains return values for formal parameters, rewriting symbolic map entries ⟨formal, *sym*⟩ to entries of the form ⟨actual, *sym*⟩, where formal is a formal parameter and actual is the corresponding actual. The resulting symbolic map is composed with the inverse parameter mapping $\mathit{ParamMap}_s^{-1}$ to rewrite mappings where formal appears in the symbolic expression in terms of the corresponding actual.

The second phase of analysis derives absolute maps. On entry to each region r, the *Enter* operation filters out the values of variables with no upwards-exposed references, and replaces an unknown or nonlinear value of live variable *var* by a symbolic constant $SC\left(var, r, 0\right)$ (whose scope is limited to the region r).

At procedure entries, this filtering identifies similar maps to yield values which should be considered equivalent for procedure cloning; the partitioning operator *Partition* makes a partition for each distinct map. No information is lost by filtering out the unreferenced variables, and we find that this simple filter keeps the amount of replication in analysis information manageable.

The *DownCall* operation rewrites mappings with symbolic expressions involving actual parameters in terms of the corresponding formal parameters at the call s, and removes variables not visible in the called procedure.

Example. To demonstrate how this analysis works on the example in Figure 6, we show the derivation of symbolic analysis information in Figure 7. Figure 7(a) shows a division of the example into regions; the transfer function for each region is shown in Figure 7(b), and the final absolute maps giving a symbolic value for each live variable on entry to each region are shown in Figure 7(c).

Symbolic analysis introduces idx_{r3}, which we abbreviate as `i`, the induction variable `I` normalized to start at 0. Assuming that `J`, `K`, and `N` have `unknown` values on entry to $r4$, the *Enter* operation introduces symbolic constants $SC\left(\texttt{J}, r4, 0\right)$ and $SC\left(\texttt{N}, r4, 0\right)$, abbreviated as $\texttt{J}_{r4}$ and $\texttt{N}_{r4}$, to refer to the initial values of `J` and `N` on entry to region $r4$ (`K` is not live so no value $\texttt{K}_{r4}$ introduced).

The symbolic variable values on entry to the loop body are given by the absolute map AM_{r2}, which indicates that `J`'s value on a particular iteration `i` of the loop is $\texttt{J}_{r4} + 2\texttt{i}$, while `K`'s value is $\texttt{J}_{r4} + 2\texttt{i} + 1$. This information is sufficient for the array analysis to determine that the accesses to array `A` do not overlap.

4.2.4 *Nonlinear Induction Variables.* The iteration operator *Iters* can be extended to recognize higher order induction variables, such as the linearized accesses in Figure 1. Such nonlinear subscript expressions arise often, whenever a loop nest is linearized to access a conceptually two-dimensional array as a one-dimensional array in order to produce long vectors. These linearized subscript expressions can be thought of as 2nd-order induction variables. Symbolic analysis recognizes these by extending the iteration operator with an auxiliary mapping of *Nonlinear Induction Variables*, which we denote $Fix_2\,(RM, r)$, redefining the iteration operation $Iters\,(f_{Body}, r)$ to use the values from this mapping, when available:

$$Iters\,(f_{Body}, r) \;=\; Fix_2\,(RM_{Body}, r) \uplus Fix_1\,(RM_{Body}, r) \uplus Fix_0\,(RM_{Body}, r).$$

The nonlinear induction variables are computed using a variant on the usual closed form of a summation $\sum_{i=0}^{n-1}(c_1 + c_2 i) = c_1 n + c_2 \frac{n(n-1)}{2}$. The computation is complicated a bit because the increment of the higher-order induction variable involves simple induction variables, which must be extracted from $Fix_1\,(RM, r)$.

$$\begin{aligned} Fix_2\,(RM, r) \;=\; \{ & \langle var, var + c_1 \times (i \times (i-1))/2 + c_2 \times i \rangle \mid \\ & \quad \langle var, var + c_1 \times i + c_2 \rangle \\ & \qquad \in (RM \circ Fix_1\,(RM, r)), \\ & \quad var \text{ not bound in } Fix_1\,(RM, r), \\ & \quad \text{and } RM_1(c_1), RM_1(c_2) \neq \texttt{unknown}, \\ & \quad\; \text{where } RM_1 = Fix_1\,(RM, r) \uplus Fix_0\,(RM, r) \} \\ & \text{where } i = idx_r \end{aligned}$$

For example, suppose that RM contains a pair $\langle z, z + y \rangle$ and $Fix_1\,(RM, r)$ contains a mapping $\langle y, y + idx_r \rangle$. Then, z is a second-order induction variable whose value on entry to iteration numbered idx_r is given by $z + (idx_r \times (idx_r - 1))/2 + y \times idx_r \rangle$, where z here denotes the value of z on entry to the loop.

Unfortunately, this resulting closed form of a second-order induction variable is nonaffine and not directly useful to the affine parallelization tests used in the array data-flow analysis. For this reason, the symbolic analysis in this case introduces a special placeholder variable e, whose scope is limited to the loop body, and replaces the non-affine expression $var + c_1 * (idx_r * (idx_r - 1))/2 + c_2 * idx_r$ with the affine expression $var + e$. When the array analysis performs a comparison between accesses containing such a variable e, acting as a stand-in for a non-affine expression, then additional affine difference information about

e is provided. For example, if $c_1 \geq 0$ and $c_2 \geq 0$, then for iteration i_1 of the loop r we have $e = e_1$ and for iteration i_2, we have $e = e_2$ such that if $i_1 < i_2$ then $e_1 \leq e_2 + c_1 + c_2$. Similar useful affine information can be provided under other conditions on c_1 and c_2. This approach enables commonly occurring cases of non-affine symbolic values in array subscripts to be handled without an expensive extension to the array dependence tests, such as that found in some previous work [Blume and Eigenmann 1994; Maslov 1992].

4.2.5 *Inequality Constraints.* The symbolic analysis described thus far can only determine equality constraints between variables. 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 as well as equalities. Equality constraints determined by the symbolic analysis are used first to rephrase each predicate in loop-invariant terms, if possible.

A separate top-down analysis pass constructs and carries this control context of loop and predicate constraints to each relevant array access. The control context is represented by a set of affine inequalities in the form discussed in Section 5; these inequalities will be in terms of constants, symbolic constants, and loop indices. This simple analysis over the loop structure is a region-based flow-insensitive analysis.

5. ARRAY DATA-FLOW ANALYSIS

The array data-flow analysis calculates data-flow information for array elements for each program region, using the analysis framework from Section 3. This analysis provides the information needed to locate loops that carry no data dependences on array elements or that can be safely parallelized after array privatization.

The array data-flow analysis approach is driven by the desire to compute both data dependences (which are location-based) and value-based dependences for array privatization in a framework that is suitable for flow-sensitive interprocedural analysis. An important feature of the array data-flow analysis is the use of *array summaries* to represent the transfer functions for analysis; that is, the indices of all accesses to an array in a program region are combined into a single summary description. The use of summaries eliminates the need to perform $O(n^2)$ pairwise dependence and privatization tests for a loop containing n array accesses. This efficiency consideration may be unimportant within a single procedure but is crucial when analyzing large loops that span multiple procedures and have hundreds of array accesses.

5.1 Representation: Summaries

5.1.1 *Sets of Systems of Inequalities.* We represent array summaries by a set of systems of integer linear inequalities, with each such system describing a multi-dimensional polyhedron. For example, consider the loop nest in Figure 8(a). The access at program region $r1$ can be represented with the system of inequalities in Figure 8(b). This system of inequalities describes possible

```
DO I = 1, N                      ⎫
    Z(1,I)            }r3        ⎪
    DO J = I, M          ⎫    ⎫  ⎪
        Z(J+1, I)     }r1⎬r2  ⎬r4⎬r5
    END DO               ⎭    ⎭  ⎪
END DO                           ⎭
```

(a) A simple loop nest.

$$S_{r1} = \left\{ \left\{ (z_1, z_2) \;\middle|\; \begin{array}{ll} I \le J \le M, & z_1 = J + 1, \\ 1 \le I \le N, & z_2 = I \end{array} \right\} \right\}$$

$$S_{r2} = \mathit{Proj}(r1, \{J\}) = \left\{ \left\{ (z_1, z_2) \;\middle|\; \begin{array}{l} 1 \le I \le N, \;\; z_2 = I \\ I \le z_1 \le M \end{array} \right\} \right\}$$

$$S_{r3} = \left\{ \left\{ (z_1, z_2) \;\middle|\; \begin{array}{c} z_1 = 1, \\ 1 \le I \le N, \;\; z_2 = I \end{array} \right\} \right\}$$

$$S_{r4} = S_{r2} \circ S_{r3} = \left\{ \left\{ (z_1, z_2) \;\middle|\; \begin{array}{c} 1 \le I \le N, \\ z_2 = I \\ I + 1 \le z_1 \le M + 1 \end{array} \right\}, \left\{ (z_1, z_2) \;\middle|\; \begin{array}{c} z_1 = 1 \\ 1 \le I \le N, \;\; z_2 = I \end{array} \right\} \right\}$$

$$S_{r5} = \mathit{Proj}(r4, \{I\}) = \left\{ \left\{ (z_1, z_2) \;\middle|\; \begin{array}{c} 1 \le z_2 \le N \\ z_2 < z_1 \le M + 1 \end{array} \right\}, \left\{ (z_1, z_2) \;\middle|\; \begin{array}{c} z_1 = 1 \\ 1 \le z_2 \le N \end{array} \right\} \right\}$$

(b) Array summary S at program regions $r1$, $r2$, $r3$, $r4$ and $r5$.

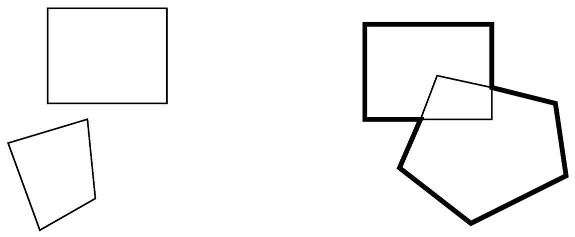
Fig. 8. Example of array summaries.

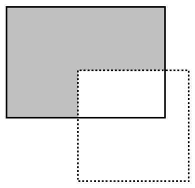
values for the *dimension variables* z_1 and z_2, representing accesses in the first and second dimension of array Z. The system of inequalities is parameterized by the program variables `M` and `N`, and loop index variables `I` and `J`.

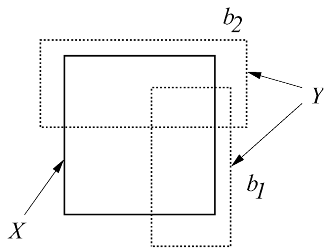
This system of inequalities represents a *parameterized index set*, a set of array indices that are parameterized by variables defining a dynamic instance (e.g., loop index variables). A *projection* function eliminates a parameter by assigning a value range to that parameter. Projection is performed using a single step of Fourier-Motzkin elimination [Dantzig and Eaves 1973; Schrijver 1986]. In Figure 8, we obtain the portion of the array accessed by the J loop at $r2$ by projecting the 2-dimensional polyhedron represented by the system of inequalities in $r1$ onto a 1-dimensional subspace derived by eliminating loop index variable J.

Each array summary contains a *set* of systems of linear inequalities. We represent the array summaries at $r4$ (and subsequently $r5$) in Figure 8(b) each by a set that precisely describes accesses to the first row and the lower triangular portion below the diagonal of array Z. If analysis had attempted to represent these accesses with a single convex region, the resulting system of inequalities would have included the entire array, introducing imprecision into the solution.

Intuitively, a set of systems of linear inequalities is necessary rather than a single system because different accesses to an array may refer to distinctly different portions of the array. Mathematically, many of the operators applied to array summaries result in nonconvex regions, which cannot be precisely described with a single system of inequalities.


(a) Two examples of a union resulting in non-convex section.


(b) Subtraction of two convex sections resulting in non-convex section.



(c) Subtraction $X - Y$ benefits from multiple iterations.

Fig. 9. Examples for summary operators.

5.1.2 *Definitions of Key Operators.* The parallelization tests and instantiation of the array data-flow analysis rely on four operators on array summaries. We provide a brief description of these operators, focusing on their most important features; a more detailed treatment can be found elsewhere [Amarasinghe 1997].

—*Union:* $X \cup Y = \{z \mid z \in X \text{ or } z \in Y\}$. The union of two summaries X and Y combines systems in Y with the set X. As the summary for loop body $r3$ in Figure 8, as well as Figure 9(a), illustrate, the union of two convex sections can result in a nonconvex section. Precise union is realized by maintaining a set of systems of inequalities rather than a single system.

A straightforward method of computing the union of summaries X and Y is to append the list of systems in both X and Y together without eliminating any redundant systems. However, to prevent the resulting explosion in the number of systems, the implementation subsequently eliminates or combines systems in the set, using the *Merge* operator whenever doing so does not result in a loss of precision (see below).

—*Intersection:* $X \cap Y = \{x \cap y \mid x \in X \text{ and } y \in Y \text{ and } x \cap y \neq \emptyset\}$. The intersection of two sets of systems is the set of all nonempty pairwise intersections of the systems from each. The implementation of the intersection operator is very simple; the inequalities of two systems are combined to form a single system. Intersecting two systems with no common array indices results in a combined system with no solution. Such systems are eliminated by checking for empty systems after combining.

—*Subtraction:* $X - Y$ subtracts all systems of set Y from each system in X, using a subtraction operation $Subtract(x, y)$ to subtract each array section y from an array section x. A precise subtraction of two convex array sections might result in a nonconvex section, as shown in Figure 9(b).

Although subtraction resulting in a nonconvex section can be performed within our framework, a simple heuristic solution that creates a convex result while guaranteeing that $x - y \subseteq Subtract(x, y) \subseteq x$ was found to be sufficient in practice. The result of $Subtract(x, y)$ is precisely the empty set if all indices of x are also in y. Otherwise, the implementation looks for an inequality in y that describes the portion of x also contained in y. If a single such inequality is found, the result of the subtraction is precise. Otherwise, subtraction would yield a nonconvex region; the implementation conservatively approximates the result as x.

The subtraction of each system $y \in Y$ from a single system x is attempted repeatedly, since system x may be contained by the combination of systems in Y, as shown in Figure 9(c).

—*Projection:* *Proj*(X, I) eliminates the set of variables I from the constraints of all the systems in set X by applying the Fourier–Motzkin elimination technique to each system.

5.1.3 *Ancillary Operators.* The principal operators rely on a number of ancillary operators to simplify and reduce the number of systems in a summary following the above principal operators.

—*Merge(X)*: Merge is applied following each of the four main operators to simplify the resulting set of systems of inequalities, reducing the number of systems in the set while avoiding loss of precision. Two heuristics are used to guide merging: (1) If there are two systems x_i and x_j in X such that $x_i \subseteq x_j$, then x_i is removed from the set; and, (2) If two systems are rectilinear and adjacent, they are combined to form a single system. In practice, these heuristics keep the sets to a manageable size.

—*IsEmpty?(X)*: $(X = \emptyset) = \forall_{x \in X}\ (x = \emptyset)$. An array summary is empty if and only if every system in the summary is empty; a system is empty if there is no integer solution for the system of inequalities. The existence of an integer solution is tested by using a Fourier-Motzkin elimination algorithm, extended by using a branch-and-bound technique [Dantzig and Eaves 1973; Schrijver 1986]. *Merge*, *Subtraction* and *Intersection* all make use of the emptiness test.

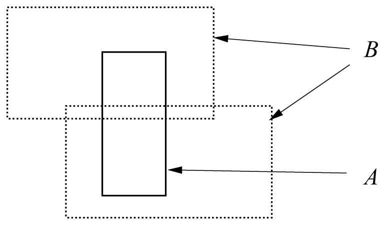


Fig. 10. A single convex region is contained in multiple regions.

```
DO 10 I = 1, N
      DO 20 J = 1, 2*M                  }
          Z(2*J, I)             } r1    } r2  }
      DO 30 J = 1, M                    }     } r5
          Z(4*J, I)             } r3    } r4  }
```

(a) Code segment containing sparse array accesses.

$$S_{r1} = \left\{\left\{(z_1, z_2) \,\middle|\, \begin{matrix} 1 \le J \le 2M, & z_1 = 2J, \\ 1 \le I \le N, & z_2 = I \end{matrix}\right\}\right\}$$

$$S_{r2} = \mathit{Proj}(r1, \{J\}) = \left\{\left\{(z_1, z_2) \,\middle|\, \begin{matrix} 2 \le z_1 \le 4M, & \exists\alpha \textit{ such that } z_1 = 2\alpha \\ 1 \le I \le N, & z_2 = I \end{matrix}\right\}\right\}$$

$$S_{r3} = \left\{\left\{(z_1, z_2) \,\middle|\, \begin{matrix} 1 \le J \le M, & z_1 = 4J, \\ 1 \le I \le N, & z_2 = I \end{matrix}\right\}\right\}$$

$$S_{r4} = \mathit{Proj}(r3, \{J\}) = \left\{\left\{(z_1, z_2) \,\middle|\, \begin{matrix} 4 \le z_1 \le 4M, & \exists\beta \textit{ such that } z_1 = 4\beta \\ 1 \le I \le N, & z_2 = I \end{matrix}\right\}\right\}$$

$$S_{r5} = S_{r2} \circ S_{r4} = \left\{\left\{(z_1, z_2) \,\middle|\, \begin{matrix} 2 \le z_1 \le 4M, & \exists\alpha \textit{ such that } z_1 = 2\alpha \\ 1 \le I \le N, & z_2 = I \end{matrix}\right\}\right\}$$

(b) Array summary S for program regions $r1$, $r2$, $r3$, $r4$ and $r5$.

Fig. 11. Example of sparse array sections.

—*IsContained?(X,Y)*: $X \subseteq Y \Leftrightarrow (x \in X \Rightarrow x \in Y)$. The *Merge* operator relies on the containment test to determine when merging two array sections will not result in a loss of precision. Note that a convex region in X might be contained by multiple convex regions in Y as shown in Figure 10. The *IsContained?* test is conservative in that it will return false in such cases.

5.1.4 *Sparse Array Sections.* The analysis supports convex array sections that are *sparse*, i.e., where not all the integer points within the convex polyhedron are in the index set. For example, the J loop corresponding to program region $r2$ in Figure 11 accesses only the even elements of the array Z.

We represent sparse array sections within the system of inequality framework by introducing *auxiliary variables*, special variables that create additional linear constraints. These variables can be viewed as additional dimensions of the convex polyhedron. For example, when projecting the loop index variable J from the system of inequalities S_{r1} to compute S_{r2}, we introduce an auxiliary α to retain the constraint that z_1 must be even.

Special care must be taken by the operators on array summaries in the presence of auxiliary variables. Each time a sparse pattern arises in an array section, a new auxiliary variable is introduced. As a result, union, intersection and subtraction operations may introduce redundant auxiliary variables with different names but that describe the same sparse pattern. Although operations on systems with redundant auxiliary variables derive correct results, solving systems with redundant variables is unnecessarily inefficient. To eliminate these redundant auxiliary variables and facilitate merging of systems, the *Merge* operator implementation employs two simple heuristics. It merges two systems with auxiliary variables if the following rules apply: (1) we can identify corresponding auxiliary variables such that they have either identical integer coefficients or one coefficient is a multiple of the other coefficient; and, (2) all remaining variables have the same integer coefficients in the two systems.

In Figure 11, program regions $r2$ and $r4$ corresponding to the J loops access sparse array sections. The analysis introduces two auxiliary variables α and β to retain the sparseness of the accesses. When computing the union of the accesses at $r2$ and $r4$ to derive the S summary at $r5$, the *Merge* operator recognizes that, because 4 is a multiple of 2, the auxiliary variable β can be eliminated and the section described by the summary at $r4$ can thus be eliminated.

5.2 Parameter Passing and Array Reshapes

Three features of FORTRAN provide different "views" of the shape of the same array data, specifically array reshaping at procedure boundaries, common block reshapes across different procedures and equivalences. These language features demand that the array analysis, to maintain precision, be able to map array summaries from one view to another. Some compilers that perform inline substitution or interprocedural analysis adopt a simple approach to this mapping, *linearizing* multi-dimensional arrays by converting their accesses to linear offsets of the beginning of the array [Burke and Cytron 1986; Grout 1995]. While linearization eliminates the problem of reshapes, it introduces array accesses with complex and often nonlinear subscript expressions, and therefore unanalyzable for the array analysis. Other systems are able to perform simple mappings of reshapes through pattern matching [Cooper et al. 1991; Havlak and Kennedy 1991; Li and Yew 1988; Triolet et al. 1986].

Instead of using pattern matching, which is limited to a few common cases of simple reshapes, we have developed a general algorithm based on systems of linear inequalities. When the reshape can be described within the affine framework, this algorithm is capable of transforming array summaries between different shapes of an array and identifying the simple regions [Hall et al. 1995a, 1995b, 1995c]. For brevity, we present only array reshaping across procedure boundaries here; common block reshapes and equivalences are handled similarly [Amarasinghe 1997].

The reshape operator rewrites summaries at a callee into equivalent summaries in the caller's name space. The only interesting mapping across the call is mapping formal parameters to corresponding actual parameters. Information on local variables are dropped, and summaries for global arrays are copied over unchanged.

Because the formal parameters may be declared with different dimensions from those of the actual, and that the origin of the formal may not be aligned with the actual, we need to reshape the summaries computed for the formal and write them as accesses of the actual arrays. We assume that the elements of both arrays are of the same type. Our algorithm starts by deriving a system of inequalities to describe the relationship between the elements of the formal and actual arrays. The system consists of a convex array section describing the summary of data accessed in the formal array, an equation showing how the address of the same array element is expressed in terms of actual and formal array indices, and finally inequalities describing the dimensions of the two array. To map the accesses of the formal array into those of the actual array, we then use projection to eliminate the dimension variables of the formal array. To illustrate how this projection works, consider the following system of inequalities,

$$\{(i, j, k) | 100i = 100j + k, 0 \leq k < 100\}.$$

Although there are many real solutions for i, j and k, since the system is constrained to be integral, there is only one solution, $i = j$ and $k = 0$, which can be found by integer programming [Schrijver 1986]. This property of integer systems allows a precise extraction of the simple reshape regions that occur in practice.

We now present a set of definitions used in the reshape algorithm:

—*Declared bounds for formal array F:* $F(lb_1^F : ub_1^F, \ldots, lb_m^F : ub_m^F)$, where F is an m-dimensional array, and dimension i has integer lower and upper bounds lb_i^F and ub_i^F, respectively.
—*Dimension variables for formal array F:* $f_1, \ldots, f_m$, variables to represent each dimension of F.
—*Declared bounds for actual array A:* $A(lb_1^A : ub_1^A, \ldots, lb_n^A : ub_n^A)$, where A is an n-dimensional array, and dimension i has integer lower and upper bounds lb_i^A and ub_i^A, respectively.
—*Dimension variables for actual array A:* $a_1, \ldots, a_n$, variables to represent each dimension of A.
—*Access expression for actual array A at call s:* $A(e_1, \ldots, e_n)$, where the e_i are affine expressions. If the beginning of the array is passed, then $\forall_{i=1,n} e_i = lb_i^A$.

Given an array summary S of formal array F, the corresponding array summary for actual array A at call site s is calculated by the following equation:

$\text{Reshape}_s(S) = \mathit{Proj}\,(S \cap \{\mathit{ReshapeConstraints}\}, \{f_1, \ldots, f_m\})$, where

$$\mathit{ReshapeConstraints} = \left\{ \begin{array}{c} lb_1^A \le a_1 \le ub_1^A \\ \cdots \\ lb_n^A \le a_n \le ub_n^A \\ lb_1^F \le f_1 \le ub_1^F \\ \cdots \\ lb_m^F \le f_m \le ub_m^F \\ \sum_{i=1}^{n} \left((a_i - lb_i^A + e_i) \prod_{j=1}^{i-1} (ub_j^A - lb_j^A + 1) \right) \\ = \sum_{i=1}^{m} \left((f_i - lb_i^F) \prod_{j=1}^{i-1} (ub_j^F - lb_j^F + 1) \right) \end{array} \right\}.$$

The above definition for *ReshapeConstraints* effectively linearizes the dimension variables and subscript expressions for A and the dimension variables for F. Then, the dimension variables for F are projected from the summary to yield a new summary that refers only to the dimension variables of A.

Many reshapes found in practice have common inner dimensions. As an optimization, we can produce a better representation for *ReshapeConstraints* whenever $k-1$ inner dimensions of F and A match. Formally, if $\forall_{i=1,k-1}$, $lb_i^A = lb_i^F$, $ub_i^A = ub_i^F$, and $e_i = lb_i^A$, then we can derive *ReshapeConstraints* with the following equations:

$$\mathit{ReshapeConstraints} = \left\{ \begin{array}{c} a_1 = f_1 \\ \cdots \\ a_{k-1} = f_{k-1} \\ lb_1^A \le a_1 \le ub_1^A \\ \cdots \\ lb_n^A \le a_n \le ub_n^A \\ lb_1^F \le f_1 \le ub_1^F \\ \cdots \\ lb_m^F \le f_m \le ub_m^F \\ \sum_{i=k}^{n} \left((a_i - lb_i^A + e_i) \prod_{j=k}^{i-1} (ub_j^A - lb_j^A + 1) \right) \\ = \sum_{i=k}^{m} \left((f_i - lb_i^F) \prod_{j=k}^{i-1} (ub_j^F - lb_j^F + 1) \right) \end{array} \right\}.$$

The effect of this optimization is to equate a_i to f_i for the first $k-1$ dimensions. Only the remaining outer dimensions starting at k must be linearized. The simplification for the first $k-1$ dimensions of the linearized access functions reduces the complexity of the projection operation and allows for more precise results than if the entire access were linearized [Amarasinghe 1997]. Even with this optimization, in practice some reshapes will yield nonaffine regions, and the results must be approximated as will be described shortly in Section 5.3.

By using this algorithm on the reshape in Figure 12(a) from the SPECFP95 program `turb3d`, we can determine that the result of the reshape is a simple plane of the array `U`. The original array region, given in Figure 12(b), is the convex array section that describes the elements of the array `X` read by the first call to `DCFT`. The special system of inequalities of the reshape problem, given in Figure 12(c), includes the array section of the original shape, bounds on the dimensions, and the equality of the linearized access functions. Note that these

```
                                    SUBROUTINE DCFT(X, INC)
DIMENSION U(66,64,64)                 REAL*8 X(*)
...                                   DO I = 1, 33
DO KN = 1, 64                           DO J = 1, 64
  CALL DCFT(U(1,1,KN),33)                 ... = X((I-1)*2 + (J-1)*2*INC + 1)
...                                       ... = X((I-1)*2 + (J-1)*2*INC + 2)
```

(a) Code extract from turb3d

$$\{(x_1) \mid 1 \leq x_1 \leq 4224\}$$

(b) Convex array region in the formal parameter

$$\begin{array}{rl} 1 \leq x_1 \leq 4224 & 1 \leq u_1 \leq 66 \\ 1 \leq \mathtt{KN} \leq 64 & 1 \leq u_2 \leq 64 \\ & 1 \leq u_3 \leq 64 \end{array}$$
$$x_1 - 1 = ((u_3 - \mathtt{KN})64 + u_2 - 1)66 + u_1 - 1$$

(c) System of inequalities before projection.

$$\left\{ (u_1, u_2, u_3) \;\middle|\; \begin{array}{c} 1 \leq u_1 \leq 66 \\ 1 \leq u_2 \leq 64 \\ u_3 = k \end{array} \right\}$$

(d) After projection, convex array region in the actual parameter.

Fig. 12. Reshape example from turb3d.

equations, while representing a valid index set for the actual array, describe a section with a complex set of inequalities, even though they describe a simple region. The index set of the actual array, with an equality of linearized access functions, will not directly describe these simple regions. By using projection to eliminate the dimension variable x_1, the integer solver finds that the only solution for u_1, u_2 and u_3 is a plane in the first two dimensions of the array U. Thus, we are able to find the convex array region of U with the simple region description as shown in the Figure 12(d).

5.3 Region-Based Array Analysis

The array data-flow analysis calculates the following four array summaries for each array:

write set W. array indices possibly written within the program region.
must write set M. array indices always written within the program region.
read set R. array indices possibly read within the program region.

Table IV. Specification for Array Summary Analysis

Direction: Forward		
PHASE 1:		
$f \in \mathcal{F}$ is a 4-tuple $\langle W, M, R, E\rangle$ for each array		
BasicBlockTF(b)	$=$	Compose reads and writes in block Read access: $\langle \emptyset, \emptyset, \{a\}, \{a\}\rangle$, Write access: $\langle \{a\}, \{a\}, \emptyset, \emptyset\rangle$ where a is a system of inequalities representing the access
$f_{r1} \circ f_{r2}$	$=$	$\langle W_{r1} \cup W_{r2}, M_{r1} \cup M_{r2}, R_{r1} \cup R_{r2}, E_{r1} \cup (E_{r2} - M_{r1})\rangle$
$f_{r1} \wedge f_{r2}$	$=$	$\langle W_{r1} \cup W_{r2}, M_{r1} \cap M_{r2}, R_{r1} \cup R_{r2}, E_{r1} \cup E_{r2}\rangle$
$\top$	$=$	$\langle \emptyset, U, \emptyset, \emptyset\rangle$ where U is an unconstrained access
Loop(f_{Body}, r)	$=$	$\langle Proj(W_{Body}, I), Proj(M_{Body}, I),$ $Proj(R_{Body}, I), Proj(E_{Body}, I)\rangle$, where I contains the normalized loop index idx_r and other variables modified in *Body*.
UpCall(s, f_{p2})	$=$	$\langle \bigcup_{a \in W_{p2}} \text{Reshape}_s(a), \bigcup_{a \in M_{p2}} \text{Reshape}_s(a),$ $\bigcup_{a \in R_{p2}} \text{Reshape}_s(a), \bigcup_{a \in E_{p2}} \text{Reshape}_s(a)\rangle$ where s is a call to procedure $p2$

exposed read set E. array indices whose reads are possibly upwards exposed to the beginning of the program region.

The array data-flow values are computed as a transfer function using a region-based analysis as described in Section 3. The second phase of the analysis framework can be omitted since the dependence and privatization tests do not require the absolute names of the arrays being accessed.

We have defined these four sets such that their values do not necessarily need to be exact. It is not always possible to find the exact set of indices that are accessed when the corresponding code is executed since that information may be undecidable at compile time, or not representable in the array summary framework. We require that the set values W, M, R, and E be valid approximations of W^{Exact}, M^{Exact}, R^{Exact}, and E^{Exact}, respectively, corresponding to the exact set of indices realized at run time. The sets W, R and E must be supersets of their exact counterparts, while M must be a subset. That is, $W \supseteq W^{Exact}$, $R \supseteq R^{Exact}$, $E \supseteq E^{Exact}$, and $M \subseteq M^{Exact}$.

The array data-flow analysis transfer function for a program region r on a particular array is represented as the 4-tuple

$$f_r = < W_r, M_r, R_r, E_r >,$$

where the elements are the sets informally defined previously. The operations defined on f tuples are presented in Table IV.

5.4 Dependence and Array Privatization Tests

At each program region r, the data-flow value for the array data-flow analysis consists of a 4-tuple $\langle W_r, M_r, R_r, E_r\rangle$ of summaries for each array. Operators on array summaries to derive this 4-tuple must ensure that M_r describes a subset of the indices of the accessed portion of the array, while W_r, R_r and E_r each describe a superset of the indices of the accessed portion.

The system examines these data-flow values at loop bodies to determine if parallelization of the loop is safe. At a given loop body denoted by program region r with normalized loop index i, the system first performs dependence analysis. Because data dependence testing is a conservative location-based test, independent of data flow, the system simply examines the W and R components of the 4-tuple. A loop may be safely parallelized if there are no loop-carried true, anti or output dependences. In the following, the notation $W_r|_y^x$ refers to replacing the variable x with the expression y in W_r. (The index variable and any other loop-varying variables that are functions of the index variable are replaced.)

(1) There is no loop-carried *true dependence* if *IsEmpty?*$(W_r|_i^{i_1} \cap R_r|_i^{i_2} \cap \{i_1 < i_2\})$
(2) There is no loop-carried *anti-dependence* if *IsEmpty?*$(W_r|_i^{i_1} \cap R_r|_i^{i_2} \cap \{i_1 > i_2\})$
(3) There is no loop-carried *output dependence* if *Is Empty?*$(W_r|_i^{i_1} \cap W_r|_i^{i_2} \cap \{i_1 < i_2\})$

For some of the remaining arrays involved in dependences, array privatization may be able to eliminate these dependences and enable safe parallelization.

Our formulation of array privatization is an extension of Tu and Padua's algorithm Tu and Padua [1993]. Tu and Padua recognize an array as privatizable only if there are *no* upwards-exposed reads within the loop. As illustrated by the example in Figure 2(b), our dependence algorithm allows upwards-exposed reads to an array as long as they do not overlap writes in other iterations of the same loop:

—*Array privatization* is possible if *IsEmpty?*$(W_r|_i^{i_1} \cap E_r|_i^{i_2} \cap \{i_1 < i_2\})$

It is straightforward to generate parallelized code for loops for which there are no dependences, but in the presence of array privatization, the system must ensure that initial and final values of the array are copied to and from the private copies. If an array has upwards-exposed read regions, the compiler must copy these regions into the private copy prior to execution of the parallel loop. If an array is live on exit of the loop, then after a parallel execution of the loop the array must contain the same values as those obtained had the loop been executed sequentially. We do not test whether arrays are live on exit, so we limit privatization to those cases where every iteration in the loop writes to exactly the same region of data; this is the case if $W = M|_i^{ub}$, where the loop index i has upper bound ub. The generated parallel code uses private copies of the array on all processors except the one which executes the final loop iteration. For the programs tested, this simple finalization transformation was sufficient; it was not necessary to use the combination of static and dynamic "last value assignment" techniques described by Tu and Padua [1993].

6. REDUCTION RECOGNITION

As defined previously, a reduction occurs when a location is updated on each loop iteration, where a commutative and associative operation is applied to that location's previous contents and some data value. We have implemented a simple, yet powerful approach to recognizing reductions, in response to the common cases we have encountered in experimenting with the compiler. The reduction recognition algorithm for both scalar and array variables is similar; this section focuses on array reduction recognition, which is integrated with the array data-flow analysis described in the previous section.

6.1 Locating Reductions

The reduction recognition algorithm searches for computations that meet the following criteria:

(1) The computation is a commutative update to a single memory location A of the form, $A = A$ `op`..., where `op` is one of the commutative operations recognized by the compiler. Currently, the set of such operations includes $+$, $*$, `MIN`, and `MAX`. `MIN` (and, equivalently, `MAX`) reductions of the form `if (a(i) < tmin) tmin = a(i)` are also supported.

(2) In the loop, all reads and writes to the location referenced by A are also commutative updates of the same type described by `op`.

(3) There are no remaining dependences in the loop that cannot be eliminated either by a privatization or reduction transformation.

This approach allows multiple commutative updates to an array to be recognized as a reduction, even without information about the array indices. This point is illustrated by the sparse reductions in `cgm` from Figure 3(b) in Section 2. The reduction recognition correctly determines that updates to `Y` are reductions on the outer loop, even though `Y` is indexed by another array `ROWIDX` and so the array access functions for `Y` are not affine expressions.

6.2 Analysis

In terms of the data-flow analysis framework, interprocedural reduction recognition requires only a flow-insensitive examination of each loop and procedure body. Scalar reduction recognition is therefore integrated with the flow-insensitive mod-ref analysis. For convenience, array reduction recognition is integrated with the array data-flow analysis from the previous section. Whenever an array element is involved in a commutative update, the array analysis derives summaries for the read and written subarrays and marks the system of inequalities as a reduction of the given reduction operator `op`. When meeting two systems of inequalities during the interval analysis, the reduction types are also met. The resulting system of inequalities will only be marked as a reduction if both reduction types are identical.

6.3 Code Transformation

For each variable involved in a reduction, the compiler makes a private copy of the variable for each processor. The executable code for the loop containing the reduction manipulates the private copy of the reduction variable in three separate parts. First, the private copy is initialized prior to executing the loop with the identity element for `op` (e.g., `0` for $+$). Second, the reduction operation is applied to the private copy within the parallel loop. Finally, the program performs a global accumulation following the loop execution whereby all nonidentity elements of the local copies of the variable are accumulated into the original variable. Synchronization locks are used to guard accesses to the original variable to guarantee that the updates are atomic.

6.4 Avoiding Unnecessary Overhead and Improving Parallelism

As the results in Section 10 clearly demonstrate, reduction operations commonly occur in scientific codes; failure to parallelize them significantly limits the effectiveness of a parallelizing compiler. However, reduction transformations can introduce substantial overhead as compared to parallel loops without reductions. In particular, for reduction computations performed only on subarrays (and not on the entire array), the cost of initialization and global accumulation over the entire array may be significant relative to the work performed within the parallel loop, and the overhead may overwhelm the benefits of parallelization.

The reduction transformation recognizes certain opportunities for reducing the overhead of initialization and global accumulation. In particular, if the reduction computation is performed on only a single location of an array, the transformation promotes the reduction to be performed on a temporary scalar variable instead. Other simplifications are possible with array data-flow analysis but are beyond the scope of our implementation. For example, the array data-flow analysis could pinpoint the subregions of an array involved in the reduction computation for the current loop, and create a temporary variable the size and dimensionality of the subarray rather than the size of the entire array.

The reduction transformation implementation also performs optimizations to reduce contention for accessing the same memory locations during the global accumulation phase. Global accumulation requires that each processor access a lock prior to updating a global array with its local partial values. The run-time system provides a *rotating lock* mechanism, rather than having each processor lock the entire array before doing its updates, which could cause significant contention among the processors. The array is divided into portions, with different locks controlling each portion. Initially, each processor accesses a different lock and updates different portions of the array. Subsequently, each processor accesses the next lock from the one it just released, wrapping around to the first lock when it finishes updating the end of the array.

It is sometimes advantageous to further parallelize the global accumulation when executing on larger numbers of processors than the 4- and 8-processor systems used for this experiment. Rather than each processor accumulating directly into the global copy of the array as the SUIF run-time system does, the transformed code could instead perform updates of local copies on pairs of processors in *binary combining trees* [Blelloch 1990].

7. PUTTING IT ALL TOGETHER

In Figure 13, we put together the analysis phases, demonstrating that the entire analysis system could execute in just four passes over the program's call

(1) Flow-insensitive pass: scalar analysis
—*Scalar parallelization analysis:* Find modified and referenced variables, and scalar reductions
(2) Bottom-up pass: scalar analysis
—*Live variable analysis:* Find privatizable scalars
—*Symbolic analysis:* Derive relative symbolic maps
(3) Top-down pass: scalar analysis
—*Live variable analysis:* Determine which privatizable variables need finalization
—*Symbolic analysis:* Apply calling context to relative maps to derive absolute maps; perform selective cloning based on this analysis.
—*Inequality constraints:* Propagate loop and control-flow constraints; perform selective cloning based on this analysis.
(4) Bottom-up pass: array analysis
—*Array data-flow analysis:* Summarize W, M, R, E
—*Parallelization testing:* Find data dependences (intersect W and R), privatizable arrays (intersect W and E), and array reductions

Fig. 13. Phases of interprocedural parallelization analysis.

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. The next pass performs the bottom-up portion of live variable analysis for scalar privatization, as was described in Section 4.1, and the subsequent top-down pass completes live variable analysis by determining which privatizable variables are live on exit from the loop in which they are referenced and therefore require finalization. In this article, we refer to these scalar analyses—modifications, references, reductions and live variables—collectively as scalar parallelization analysis.

In the same two passes that compute scalar live variables, we can perform scalar symbolic analysis, as described in Section 4.2. During the top-down phase of symbolic analysis, we can also collect inequality constraints on loop bounds and control flow as described in Section 4.2.5. In this same top-down pass, selective procedure cloning is performed based on these two analyses.

The final bottom-up pass performs the array data-flow and parallelization testing described in Section 5.

8. GENERATING PARALLELIZED CODE IN SUIF

SUIF is a fully functional compiler for both FORTRAN and C, but for this experiment, we consider FORTRAN programs only. The compiler takes as input sequential programs and outputs the parallelized code as an SPMD (Single Program Multiple Data) parallel C program that can then 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 (SGI Challenge and Power Challenge, and Digital 8400 multiprocessors) and scalable shared-memory architectures (Stanford DASH and SGI Origin 2000). The SUIF run-time system provides standard SPMD functionality, and is built from ANL macros for thread creation, barriers, and locks.

The full SUIF system incorporates the analysis to locate coarse-grain parallelism presented in this article, as well as additional optimizations to improve locality of compiler-parallelized applications. Other optimizations include locality and parallelism loop transformations [Wolf 1992], compiler-inserted prefetching [Mowry et al. 1992], data and computation colocation [Anderson et al. 1995], data transformations [Anderson et al. 1995], synchronization elimination [Tseng 1995] and compiler-directed page coloring [Bugnion et al. 1996]. The parallelization analysis presented in this paper in conjunction with these other optimizations can achieve significant performance improvements for sequential scientific applications on multiprocessors. As evidence, the full SUIF system has obtained higher SPECFP92 and SPECFP95 ratios than previously reported results on an 8-processor Digital Alphaserver 8400. The SPECFP95 result was over 50% higher than the best reported SPECFP95 ratios, in December 1996, when the experiment was performed [Amarasinghe et al. 1996; Hall et al. 1996].

However, in this article, we focus on the ability of the compiler to detect coarse-grain parallelism. Thus, to obtain these results, we omit the locality optimizations and perform a straightforward translation of the loops found to be parallelizable by our analysis. The compiler parallelizes only the outermost loop that the analysis has proven to be parallelizable, and no loop transformations are applied. The iterations of a parallel loop are evenly divided between the processors into consecutive blocks at the time the parallel loop is spawned. This simple strategy of blocking the iteration space for processors works well for loops where the amount of work in each iteration of the outermost loop is roughly equivalent, but may lead to load imbalance for loop nests containing triangular loops. Load imbalance can also occur if the parallelized loop has a small number of iterations relative to the number of processors. While beyond the scope of this project, the literature presents many solutions to these potential load imbalance problems, including block-cyclic distribution of iterations, dynamic scheduling of iterations to processors [Polychronopoulos 1986; Hummel et al. 1992] and exploiting multiple levels of parallelism in a loop nest.

In a few cases, our system chooses not to parallelize the outermost loop. This occurs if a loop is too fine-grained, or contains reductions, resulting in too much overhead to profitably execute in parallel. For example, an outer loop that can be parallelized only with an array reduction transformation is sometimes left as a sequential loop if it contains an parallel inner loop.

We have so far discussed compile-time decisions about which loops to parallelize, but the compiler also interacts with the run-time system to make dynamic decisions about whether to suppress parallelization. Using the knowledge of the iteration count at run time, the run-time system executes the loop sequentially if it is considered too fine-grained to have any parallelism benefit. The decision as to how much work is sufficient for profitable parallel execution is machine dependent; therefore, this limit on work is configurable at run time.

9. RELATED WORK

We consider work related to this system both from the system perspective, comparing other parallelization systems and experimental results, and from the perspective of each of the individual analysis components.

9.1 Array Data-Flow Analysis

Data dependence analysis computes location-based dependence information, determining whether two accesses possibly point to the same memory location. An analysis which tracks value-based dependence determines whether the source of a value used in one iteration of a loop may have been produced by an assignment outside the loop. Such an analysis, when applied to the elements of arrays, has come to be called an *array data-flow analysis*, whether or not data-flow analysis techniques are used. There have been three major approaches to finding this data-flow information for array elements.

The first approach, pioneered by Feautrier [1988a, 1988b, 1991] extends the data dependence analysis framework. Data-dependence analyses first targeted small loops, and it was sufficient to carry out pairwise comparisons between the access expressions, making use of methods which can be efficient in the affine domain [Maydan et al. 1991, 1992; Pugh and Wonnacott 1992; Wolfe and Banerjee 1987]. Array data-flow dependence computations simply extend these comparisons to determine relative ordering of array reads and writes, by performing pairwise comparisons that are precise in the affine domain [Feautrier 1988a, 1988b, 1991; Maydan et al. 1993; Pugh and Wonnacott 1992; Ribas 1990]. While precise, these approaches are severely limited in the presence of general control flow, and further, pairwise testing of accesses is not practical if applied across large bodies of code. In addition, the presence of multiple writes in a code segment significantly increases the complexity of such exact solutions.

A second approach performs intraprocedural analysis in the same pairwise fashion, but aggregates all array accesses occurring in a called procedure into a single descriptor at the call site [Hind et al. 1994; Li and Yew 1988]. The precision of this descriptor may be traded off for efficiency, but there is still a pairwise comparison within each procedure, so while it may be more efficient than the exact solutions, $O(n^2)$ dependence tests can be prohibitively expensive when applied to large bodies of code.

Because of these practicality considerations, the SUIF system uses the third and more common approach to array data-flow analysis, which extends scalar data-flow analysis techniques. Instead of representing accesses to an array with a single bit, these techniques describe array accesses with an array index set [Balasundaram and Kennedy 1989; Creusillet and Irigoin 1995; Granston and Veidenbaum 1991; Gross and Steenkiste 1990; Triolet et al. 1986; Tu 1995]. Array data-flow analysis efficiently supports complex control flow and multiple accesses by conservatively merging these index sets.

The accuracy of these array data-flow analysis approaches is governed by the precision of their representations, which varies in three ways: (1) the precision with which they represent a single access; (2) how precisely they support multiple accesses, and (3) how they maintain precision in the presence of array reshapes at procedure boundaries. First, a number of approaches exactly represent only a limited domain of rectilinear, triangular or diagonal array sections [Havlak and Kennedy 1991], with more complex spaces represented using multiple such sections [Tu 1995]. Our approach instead uses a representation based on linear inequalities, which is exact for more general array sections, convex polyhedrons. Triolet et al. [1986] first proposed using a system of inequalities representation for interprocedural array data dependence analysis in the PIPS system, but did not precisely represent all convex regions, particularly sparse access patterns, instead approximating with a convex hull of all the indices.

Second, some approaches merge multiple accesses to a single array section, with the goal of avoiding an explosion of array sections [Creusillet and Irigoin 1995; Havlak and Kennedy 1991; Triolet et al. 1986]. In our experiments, we have found that a single array section is not sufficient to precisely represent arrays with multiple accesses. Our analysis merges only if no information is lost, and we have found that this merging is sufficient to keep the number of array sections manageable. A few other approaches retain separate sections for multiple accesses [Tu 1995; Li and Yew 1988].

Third, precise interprocedural array analysis must support the reshaping of array parameters across procedure boundaries. Most commonly, previous approaches to this problem have only been precise for simple reshapes, such as when the formal parameter is declared identically to the lower dimensions of its corresponding actual [Cooper et al. 1991; Havlak and Kennedy 1991; Li and Yew 1988; Triolet et al. 1986]. Other approaches represent multidimensional arrays with equivalent one-dimensional linearized arrays [Burke and Cytron 1986; Grout 1995]. However, linearized arrays are often much more difficult to analyze than their multi-dimensional counterparts in many cases, so while the linearized representation is precise, precision may be lost during analysis. Our system incorporates the first algorithm, capable of handling many complex reshape patterns that occur in practice, using integer projections. A similar approach has recently been adopted by Creusillet and Irigoin for the PIPS system [Creusillet and Irigoin 1995; Creusillet 1996]; it extends our earlier algorithm which did not eliminate lower dimensions [Hall et al. 1995b].

9.2 Array Reduction Recognition

Reductions have been an integral component in the study of vectorizing and parallelizing compilers for many years [Padua and Wolfe 1986]. More recently, reduction recognition approaches have been proposed that rely on symbolic analysis or abstract interpretation to locate many kinds of complex reductions [Ammarguellat and Harrison 1990; Haghighat and Polychronopolous 1996]. It is unclear whether the significant additional expense of these approaches is justified by the types of reductions that appear in practice.

SUIF's reduction recognition is most closely related to recent research by Pottenger and Eigenmann in the Polaris system [Pottenger and Eigenmann 1995]. Our reduction recognition, in conjunction with the scalar symbolic analysis, is capable of locating the recurrences described by Pottenger and Eigenmann [1995]. However, our work is distinguished by its ability to parallelize interprocedural and sparse reductions.

9.3 Interprocedural Symbolic Analysis

Some other approaches to symbolic analysis incorporate the additional features of demand-based propagation [Blume 1995; Tu 1995] and sparseness [Havlak 1994; Tu 1995], within a procedure to reduce the expense of the analysis. While they may improve analysis performance, due to the complexity of the techniques, interprocedural analysis is handled differently than intraprocedural analysis. Since the cost of the parallelization analysis is dominated by the array analysis, we have not found a need to incorporate these special techniques, and are able to provide uniformly precise information both intra- and interprocedurally.

Also of note in our approach is the recognition of second-order induction variables, which supports analysis of some nonlinear array subscripts. This simple approach captures the information required for common nonlinear subscripts that arise in practice, namely triangular and linearized arrays, without the need to adopt more sophisticated nonlinear dependence tests [Blume and Eigenmann 1994; Maslov 1992]. In addition, the unique use of selective procedure cloning to obtain context-sensitive information has been critical to parallelizing the largest loops in the benchmark suites.

Haghighat derives, in some cases, stronger symbolic information, focusing on highly developed methods for rewriting expressions, but it is unclear how many of the unique techniques he describes have a significant applicability in practice [Haghighat and Polychronopolous 1996].

Like the inequality constraint propagation from loops and conditionals in the SUIF system, some other interprocedural systems propagate inequality and other relational constraints on integer variables imposed by surrounding code constructs to their uses in array subscripts [Havlak 1994; Irigoin 1992].

9.4 Flow-Sensitive and Context-Sensitive Interprocedural Analysis

Precise flow-sensitive interprocedural analysis was historically thought to be too expensive for practical use in a compiler since Myers [1981] proved that it was Co-NP-Complete in the presence of aliasing.

Earlier techniques for avoiding propagation along unrealizable paths and deriving context-sensitive interprocedural information have relied upon either inline substitution or tagging data-flow sets with a path history through the call graph. Tagging with a full path history incurs a data-flow set expansion problem corresponding to the code explosion problem of inlining. As a result, most tagging approaches limit the length of a tag, thus sacrificing precision [Harrison 1989; Myers 1981; Sharir and Pnueli 1981; Shivers 1991]. Instead of tagging which increases space regardless of whether context-sensitivity improves analysis results, our system utilizes context-sensitive information only when it provides unique data-flow information through the use of selective procedure cloning.

The region-based flow-sensitive analysis is similar to analysis on the *hierarchical structured control-flow graph* in PIPS [Irigoin 1992; Irigoin et al. 1991]. The most significant difference between the two approaches is that we have defined a common, general interprocedural framework that is used for all the flow-sensitive data-flow problems in the parallelizer. As such, the analysis solves both backward and forward data-flow problems, and it is capable of more precise analysis of unstructured code constructs. In addition, our analysis framework incorporates additional context-sensitive information through the use of selective procedure cloning.

9.5 Interprocedural Analysis Frameworks

There have been many general-purpose frameworks to support the construction of program analyses, both intra- and interprocedural. We divide them into summary-based and iterative frameworks, although some do not provide solution algorithms, so could conceivably be used with either type of solver.

9.5.1 *Summary-Based Frameworks.* The systems Sharlit, PIPS, and our work, provide for more efficient analysis by user definition of methods for combining transfer functions. The procedure summary approach provides some context sensitivity to the PIPS and FIAT systems; the FIAT system also gains precision from procedure cloning.

Sharlit [Tjiang and Hennessy 1992] supported the construction of intraprocedural data-flow analyses. Given user definitions (in a special-purpose language) of data-flow values, transfer functions, and monoid operations on the transfer functions, it derived efficient analysis passes using an elimination technique based on Tarjan's path-compression method [Tarjan 1981a, 1981b].

The PIPS parallelizer was built on a general-purpose analysis framework, which supported interprocedural analysis based on procedure summaries over the hierarchical structured control flow graph mentioned above. In addition to program representation, the system provides a program called `pipsmake` for dependence-driven dynamic recomputation of analysis data as needed. The program call graph is implicit in an analysis solver. There is no procedure cloning.

Our work is based upon FIAT, which in its original form supported the construction of flow-insensitive interprocedural analyses of FORTRAN programs, with selective procedure cloning [Hall et al. 1993]. The work described in this paper extends FIAT to also support flow-sensitive interprocedural analysis. Call graph construction, local analysis, interprocedural analysis drivers, cloning and Mod-Ref analysis have been provided by the original FIAT system.

9.5.2 *Iterative Frameworks.* A number of frameworks seem aimed not at supporting the construction of complex analyses, but rather at making efficient analyses. These frameworks are based on more traditional abstract interpretation methods, or iterative program supergraph approaches, rather than elimination or summary-based techniques. The interprocedural frameworks support context-sensitivity only by duplicating procedure representations.

Dwyer and Clarke [1999] presented an intraprocedural analysis framework in Ada to support not just construction of a single analysis, but combinations of analyses (such as Cartesian products) and parameterized analyses (e.g., find variable values in terms of a value lattice which can be plugged in later).

The Vortex compiler [Chambers et al. 1999] includes a framework which combines transformation and iterative program analysis. Like the system of Dwyer and Clarke, the Vortex framework supports composition of analyses; it can also derive an interprocedural analysis automatically from a given intraprocedural analysis. Context sensitivity is obtained by explicit representations of procedure instances from distinct contexts. There is no support for procedure summaries, other than memoization of input/output pairs.

The PAG system [Alt and Martin 1995] bases analysis construction upon an abstract interpretation model of analysis, providing a special language to describe program representation, analysis lattice, and flow functions, and has an analysis solver. Interprocedural analyses are based on an *extended supergraph*, in which first each procedure is cloned for some fixed number of contexts, and then an iterative analysis is done over the resulting program supergraph. No procedure summaries are used.

System Z [Yi and Harrison 1993] is an analyzer generator which produces an intra- and interprocedural analysis written in C for a collecting abstract interpreter of a target language. The user provides nonrecursive set and lattice definitions in a specialized, lisp-like language; the analysis lattice must be finite. There seems to be no provision for context sensitivity.

The McCat system, by Hendren et al. [1993], is an interprocedural analysis framework for C, which does not provide analysis solvers but a program representation over which solution can be reached. The analyses built on the program representation will be context-sensitive, in that the call graph representation is expanded by a given call graph builder to an *instantiation graph,* in which every nonrecursive call chain is distinguished; the result is similar to a program that is fully inlined except for recursive calls.

9.6 Experiments with Automatic Parallelization Systems

In the late 1980s, experiments using flow-insensitive array analysis for interprocedural dependence testing demonstrated that these techniques were effective in parallelizing linear algebra libraries [Havlak and Kennedy 1991; Li and Yew 1988; Triolet et al. 1986]. More recently, the FIDA system was developed at IBM to obtain more precise array sections through dependence testing of partially inlined array accesses, demonstrating comparable results [Hind et al. 1994].

Concurrently with the development of the SUIF automatic parallelization system, the Polaris system at University of Illinois was also pushing the state-of-the-art in parallelization technology [Blume et al. 1996]. The Polaris system and ours have many similar features and a few complementary ones individually noted above. The most fundamental difference between the two systems is that Polaris performs no interprocedural analysis other than interprocedural constant propagation, instead relying on inlining of programs to obtain interprocedural array and symbolic information. Both systems have produced results demonstrating significant improvements over previously reported parallelization results. Direct comparisons between results from the two systems are difficult. For example, optimizations such as elimination of unused procedures and loops with zero trip counts, which eliminate some loops, make comparisons of static loop counts impossible. Further, speedup measurements cannot be directly compared as they are relative to different baselines and are machine dependent. The latest results from the Polaris compiler can be found in Blume et al. [1996].

A few additional automatic parallelization systems incorporate interprocedural analysis, although none have published recent results on the effectiveness of their system. The PIPS system is the most similar to ours, but, as noted previously, lacks several features that lead to more precise results, such as selective procedure cloning, extensive interprocedural scalar parallelization and symbolic analysis, interprocedural reduction recognition and stronger array reshape analysis. The PFC and ParaScope systems at Rice University included implementations of flow-insensitive array data-flow analysis and interprocedural symbolic analysis [Havlak and Kennedy 1991; Havlak 1994]; these projects were later replaced by the D System, which focused on compiling for distributed-memory machines [Adve et al. 1994]. The Parafrase-2 system includes interprocedural symbolic analysis, but apparently not interprocedural array analysis [Haghighat and Polychronopolous 1996].

10. EMPIRICAL EVALUATION

We have fully implemented the interprocedural parallelization analysis described in this paper in an experimental version of the Stanford SUIF compiler based on SUIF version 1.0. We have performed extensive evaluations of its effectiveness. This section presents results on a collection of programs from four benchmark suites, consisting of more than 115,000 lines of FORTRAN and 39 programs.

Previous evaluations of interprocedural parallelization systems have provided static measurements of the number of additional loops parallelized as a result of interprocedural dependence analysis [Havlak and Kennedy 1991; Hind et al. 1994; Li and Yew 1988; Triolet et al. 1986]. We have compared the most recent of these empirical studies, which examines the SPEC89 and PERFECT benchmark suites using the FIDA system [Hind et al. 1994]. 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 giving some indication of the importance of interprocedural array privatization and reduction recognition (a comparison with FIDA is presented in detail in Hall et al. [1995a]).

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, we also present a series of results gathered from executing the programs on the Silicon Graphics Challenge and Digital AlphaServer 8400. 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 interprocedural array analysis, array privatization and reduction recognition.

10.1 Multiprocessor Systems

We obtain dynamic results by executing the programs parallelized by our compiler on two multiprocessor machines, the Silicon Graphics Challenge and the Digital AlphaServer 8400. The Challenge is a 4-processor machine with 768 MBytes main memory, consisting of 200 MHZ R4400 processors. The R4400 is a single-issue superpipelined processor, although each floating point unit is not fully pipelined. The 8-processor AlphaServer has 4 GBytes of main memory and 300 MHz Digital 21164 Alpha processors. The Digital 21164 Alpha processor is a quad-issue superscalar microprocessor with two 64-bit integer and two 64-bit floating point pipelines. We divide the benchmark suites across the two architectures, executing the programs with short execution times and small data sets on the Challenge, and the programs with much longer execution times and very large data sets on the faster and larger capacity AlphaServer. As mentioned in Section 8, the SUIF run-time system suppresses parallelization at run time those loops with insufficient granularity to execute profitably in parallel. Because of the wide difference in performance of these two architectures, we use two different limits on the work within the loop required to execute in parallel. Across all applications executed on a system, we use the same limit.

10.2 Benchmark Programs

To evaluate our parallelization analysis, we measured its success at parallelizing four standard benchmark suites described by Table V: the FORTRAN programs from SPECFP95 and SPECFP92, the sample NAS benchmarks, and PERFECT [Bailey et al. 1991; Uniejewski 1989; Reilly 1995; Cybenko et al. 1990]. In a few cases, a version of the same program appears in two benchmark suites: there are four programs that appear in both SPECFP95 and SPECFP92, `mgrid` and `applu` appear in both SPECFP95 and NAS, and the programs `apsi` in SPECFP95 and `adm` in PERFECT are the same. To show the differences in performance between the two multiprocessor systems and between two inputs to a program, the table gives sequential execution time for the programs on the Challenge and the AlphaServer.

SPECFP95 is a set of 10 floating-point programs created by an industry-wide consortium and is currently the industry standard in benchmarking uniprocessor architectures and compilers. In our analysis, we omit `fpppp` because it has significant parameter-passing type errors; it is considered to have very little loop-level parallelism.

SPECFP92 is a set of 14 floating-point programs from the 1992 version of SPEC. The six programs `fpppp`, `hydro2d`, `su2cor`, `swm256`, `tomcatv` and `wave5` are nearly identical to their SPECFP95 versions, but with smaller data sets. We omit `alvinn` and `ear`, the two C programs, and `spice`, a program of mixed FORTRAN and C code. We also omit `fpppp` for the same reasons given above. (The programs are presented in alphabetical order of their program names.)

Table V. Benchmark Programs

	Length			Seq. Run Time	
	(lines)	Description		Challenge	AlphaServer
SPECFP95					
`tomcatv`	190	mesh generation			314.4s
`swim`	429	shallow water model			282.1s
`su2cor`	2332	quantum physics			202.9s
`hydro2d`	4292	Navier-Stokes			350.1s
`mgrid`	484	multigrid solver			367.3s
`applu`	3868	parabolic/elliptic PDEs			393.4s
`turb3d`	2100	isotropic, homogeneous turbulence			347.7s
`apsi`	7361	mesoscale hydrodynamic model			193.3s
`wave5`	7764	2-D particle simulation			217.4s
NAS					
`appbt`	4457	PDEs (block	$12^3 \times 5^2$ grid	10.0s	2.3s
		tridiagonal)	$64^3 \times 5^2$ grid		3039.3s
`applu`	3285	PDEs (parabolic/	$12^3 \times 5^2$ grid	4.6s	1.2s
		elliptic)	$64^3 \times 5^2$ grid		2509.2s
`appsp`	3516	PDEs (scalar	$12^3 \times 5^2$ grid	7.7s	2.2s
		pentadiagonal)	$64^3 \times 5^2$ grid		4409.0s
`buk`	305	integer bucket sort	65,536 elts	0.6s	0.3s
			8,388,608 elts		45.7s
`cgm`	855	sparse conjugate	1,400 elts	5.4s	2.0s
		gradient	14,000 elts		93.2s
`embar`	135	random number	256 iters	4.6s	1.4s
		generator	65,536 iters		367.4s
`fftpde`	773	3-D FFT PDE	64^3 grid	26.3s	6.2s
			256^3 grid		385.0s
`mgrid`	676	multigrid solver	32^3 grid	0.6s	0.2s
			256^3 grid		127.8s
SPECFP92					
`doduc`	5334	Monte Carlo simulation		20.0s	4.8s
`mdljdp2`	4316	equations of motion		45.5s	19.4s
`wave5`	7628	2-D particle simulation		42.9s	12.6s
`tomcatv`	195	mesh generation		19.8s	9.2s
`ora`	373	optical ray tracing		89.6s	21.5s
`mdljsp2`	3885	equations of motion, single precision		40.5s	19.5s
`swm256`	487	shallow water model		129.0s	42.6s
`su2cor`	2514	quantum physics		156.1s	20.1s
`hydro2d`	4461	Navier-Stokes		110.0s	31.6s
`nasa7`	1105	NASA Ames FORTRAN kernels		143.7s	59.0s
PERFECT					
`adm`	6105	pseudospectral air pollution model		20.2s	6.4s
`arc2d`	3965	2-D fluid flow solver		185.0s	46.4s
`bdna`	3980	molecular dynamics of DNA		63.7s	12.4s
`dyfesm`	7608	structural dynamics		18.3s	3.8s
`flo52`	1986	transonic inviscid flow		24.1s	7.2s
`mdg`	1238	molecular dynamics of water		194.5s	62.1s
`mg3d`	2812	depth migration		410.9s	250.7s
`ocean`	4343	2-D ocean simulation		71.8s	23.6s
`qcd`	2327	quantum chromodynamics		9.6s	3.1s
`spec77`	3889	spectral analysis weather simulation		124.6s	20.7s
`track`	3735	missile tracking		6.2s	1.8s
`trfd`	485	2-electron integral transform		21.1s	5.5s

NAS is a 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 Applied Parallel Research that separates the first call to a function, which initializes static data, from the other calls. We present results for both small and large data set sizes.

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 FORTRAN 77 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, 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.[1] All the programs produce valid results when executed in parallel.

10.3 Case Studies: Examples of Coarse-Grain Parallelism

The analysis presented in this article has successfully parallelized some very large loops in these benchmark programs. The largest loop SUIF parallelizes is from the `gloop` subroutine of `spec77`, consisting of 1002 lines of code from the original loop and its invoked procedures, graphically displayed in Figure 14. In the figure, the boxes represent procedures and the edges connecting them procedure calls. The outer parallelizable loop is shaded in grey. This 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.

Another example of a large coarse-grain parallel loop parallelized by the compiler is one of a series of loops performing Fourier and Inverse-Fourier Transforms in the main subroutine of the SPECFP95 program `turb3d`, illustrated in Figure 15. Its four main computation loops compute a series of 3-dimensional FFTs. While these loops are parallelizable, they all have a complex control structure and complicated array reshapes (described in Section 5.2). Each parallel loop consists of over 500 lines of code spanning nine procedures and containing 42 procedure calls.

[1]The modified PERFECT benchmarks can be found at the following web site, http://www.isi.edu/asd/compiler/project.html.

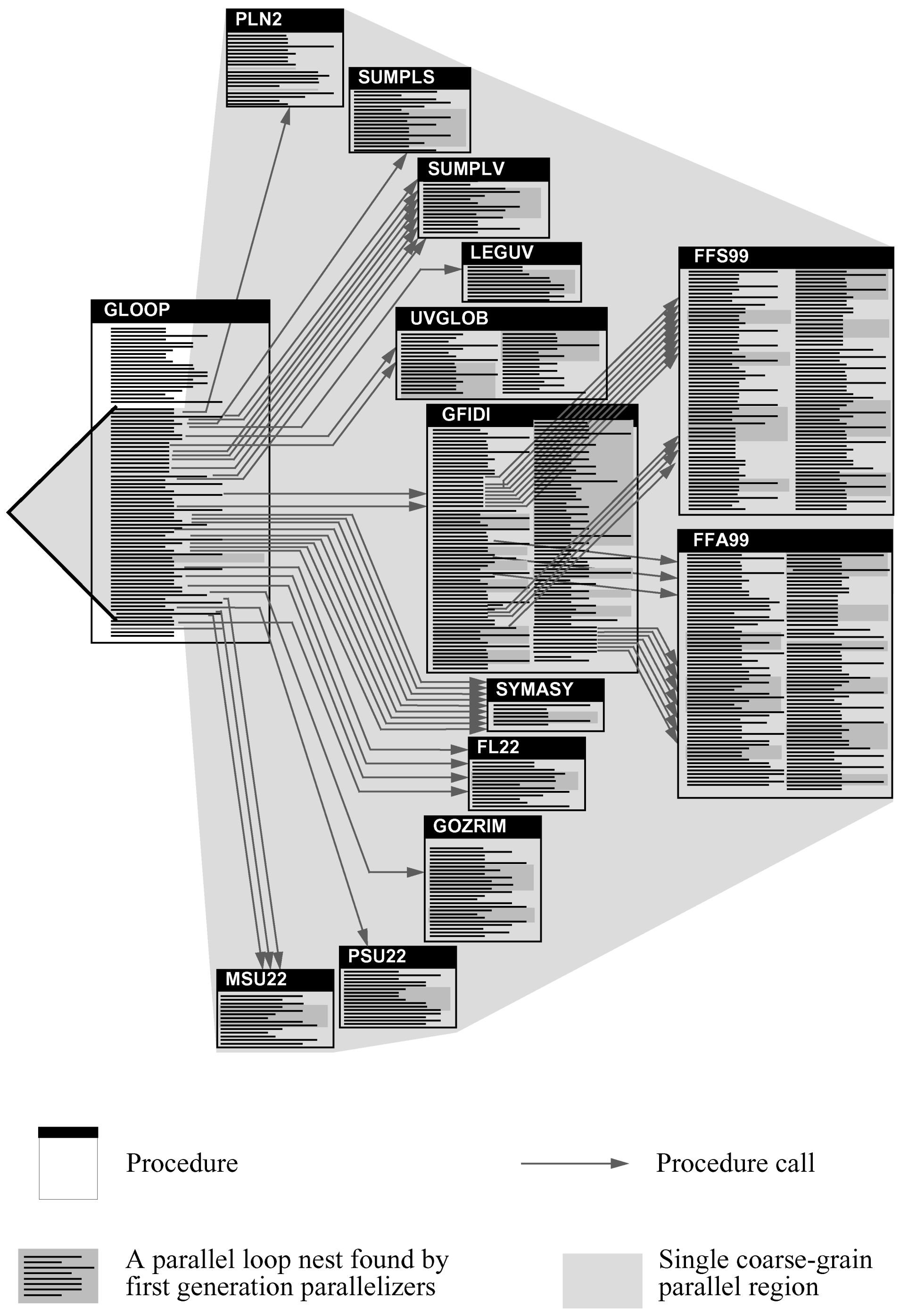


Fig. 14. A large parallel loop from Perfect benchmark spec77.

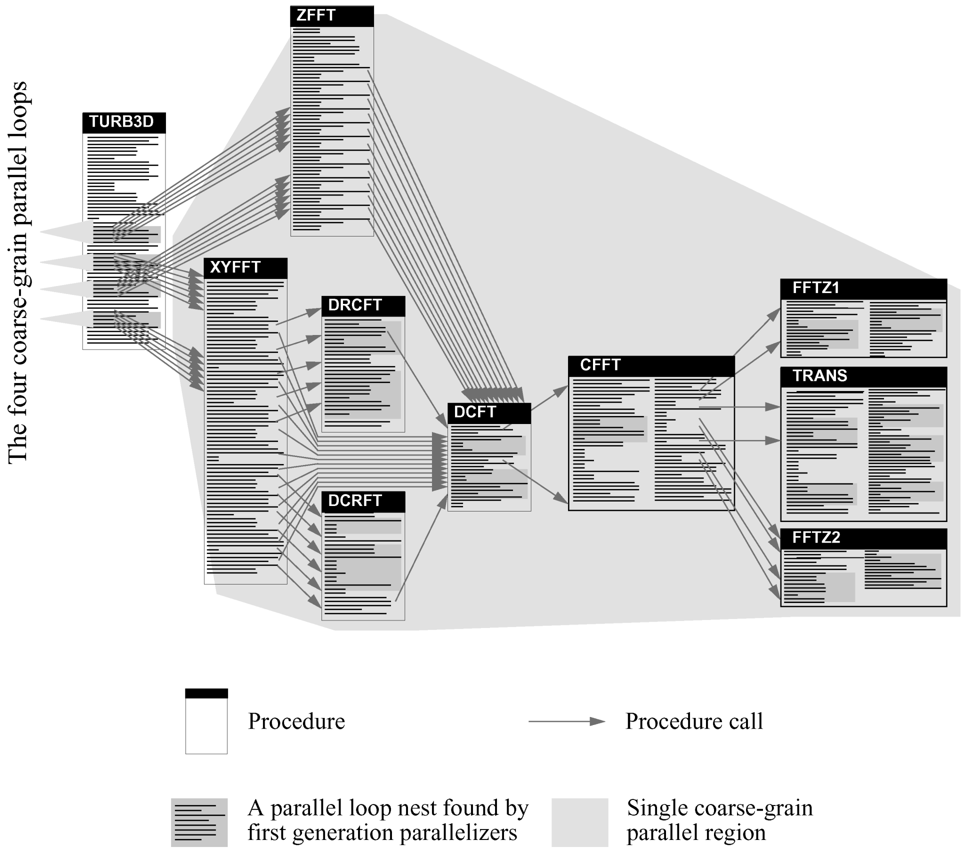

The four coarse-grain parallel loops
TURB3D
ZFFT
XYFFT
DRCFT
DCRFT
DCFT
CFFT
FFTZ1
TRANS
FFTZ2
Procedure
Procedure call
A parallel loop nest found by first generation parallelizers
Single coarse-grain parallel region

Fig. 15. A large parallel loop from SPECFP95 benchmark `turb3d`.

10.4 Applicability of Advanced Analyses

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 many existing parallelizing compilers as it contains all the *interprocedural scalar analysis* discussed in Section 4.

We focus presentation of results on the impact of three array analysis components: interprocedural analysis, array privatization and array reduction recognition. The experimental framework does not support isolation of the contributions of the interprocedural scalar analyses, but we have seen numerous examples where these analyses are essential. For example, the two largest loops parallelized by our compiler (see Section 14) require selective procedure cloning based on constant parameter values. Other performance-critical loops in the programs `embar`, `mdljdp2` and `ora` would not have been parallelized without interprocedural scalar privatization and scalar reduction recognition.

10.4.1 *Static Measurements.* Table VI gives counts of the number of loops in the SUIF-parallelized program that require a particular technique to be parallelizable. The first column gives the total number of loops in the program. The second column provides the number of loops that are parallelizable in the baseline system. The last column presents the number of loops parallelized by the SUIF system. (The number of parallelizable loops includes those nested within other parallel loops which would consequently not be executed in parallel under the parallelization strategy.) The remaining columns show the combinations of analysis techniques required to parallelize additional loops beyond the baseline results.

The third through seventh columns present measurements of the applicability of the intraprocedural versions of advanced array analyses. We present separately the effects of reduction recognition and array privatization, and then show loops that require both techniques. The next set of four columns all have interprocedural data dependence analysis. The seventh to ninth columns measure the effect of adding interprocedural reduction recognition, privatization and the combination of the three.

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.

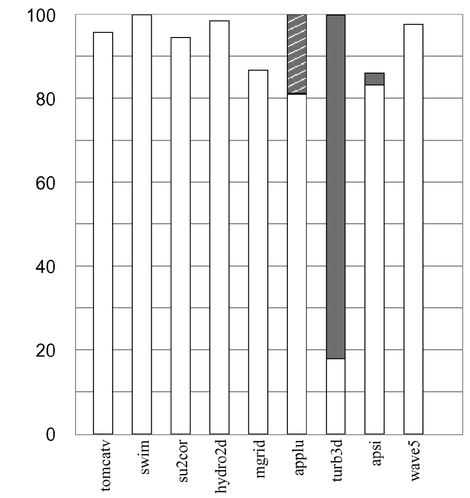
10.4.2 *Dynamic Measurements.* We also measure the dynamic impact of each of the advanced array analyses. Figure 16 and Figure 17 present results for two benchmark suites on the Digital AlphaServer, SPECFP95 and NAS with large data sets. Figures 18, 19, and 20 present these results for three benchmark suites on the SGI Challenge, NAS with small data sets, SPECFP92, and PERFECT.

10.4.2.1 *Performance Metrics.* 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 article. For example, techniques to improve data locality and minimize synchronization can potentially improve the speedups obtained significantly. Thus, to more precisely capture how well the parallelization analysis performs, we use the two following metrics:

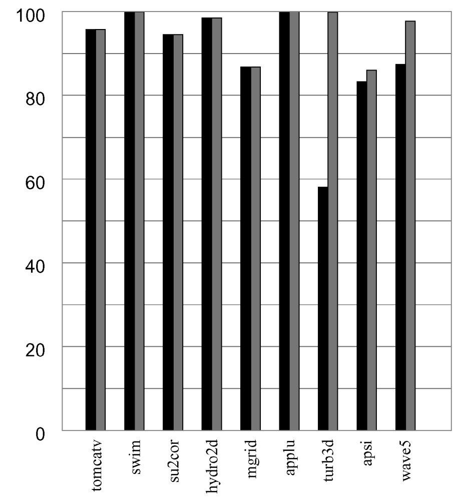
—*Parallelism Coverage:* The overall percentage of time spent in parallelized regions is the parallelism coverage. Coverage is an important metric for measuring the effectiveness of parallelization analysis. By Amdahl's law, programs with low coverage will not get good parallel speedup. For example,

Table VI. Static Measurements: Number of Parallel Loops Using Each Technique

		Parallel Loops								
	# loops	*Intraprocedural*				*Interprocedural*				*Total*
Array Reduction			√		√		√		√	
Array Privatization				√	√			√	√	
SPECFP95										
`tomcatv`	16	10								10
`swim`	24	22								22
`su2cor`	117	89								89
`hydro2d`	163	155								155
`mgrid`	46	35								35
`applu`	168	127	10	6		6				149
`turb3d`	70	55		3		4				62
`apsi`	298	169				2				171
TOTAL	5262	2635	95	66	0	68	5	10	11	2890
NAS										
`appbt`	192	139	3	18		6			3	169
`applu`	168	117	4	6		6			3	136
`appsp`	198	142	3	12		6			3	166
`buk`	10	4								4
`cgm`	31	17	2							19
`embar`	8	3	1						1	5
`fftpde`	50	25								25
`mgrid`	56	38								38
SPECFP92										
`doduc`	280	230				7				237
`mdljdp2`	33	10	2	1			2			15
`wave5`	364	198								198
`tomcatv`	18	10								10
`ora`	8	5				3				8
`mdljsp2`	32	10	2	1			2			15
`swm256`	24	24								24
`su2cor`	128	65	3			1				69
`hydro2d`	159	147								147
`nasa7`	133	59	1	6						66
PERFECT										
`adm`	267	172				2		2		176
`arc2d`	227	190								190
`bdna`	217	111	28					1		140
`dyfesm`	203	122	5	2			1	5		135
`flo52`	186	148		1		7				156
`mdg`	52	35		1				2		38
`mg3d`	155	104	2							106
`ocean`	135	102	1	6						109
`qcd`	157	92	7							99
`spec77`	378	281	13	2		17			1	314
`track`	91	51	3			1				55
`trfd`	38	15	5	1						21



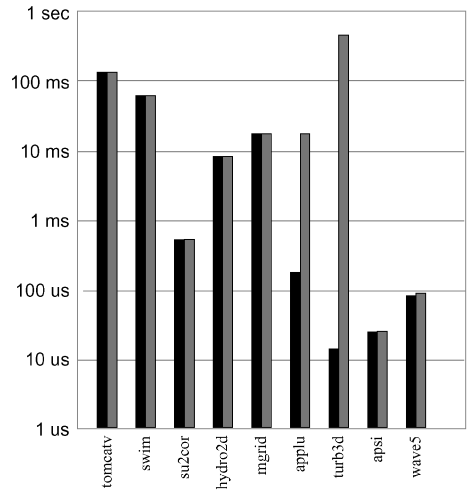
(A) Applicable % of Computation



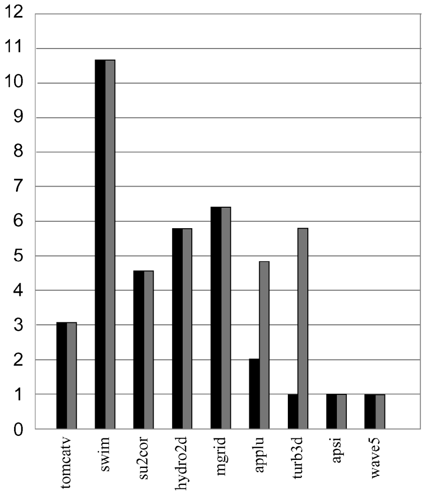
(B) Parallelism Coverage (%)

Intra–	Inter– procedural	Techniques
☐	■	Data Dependence Analysis
▧	▧	+ Array Reduction
▨	▨	+ Array Privatization
▦	▦	+ Array Reduction + Array Privatization

■	Baseline: Interprocedural Scalar Analysis Intraprocedural Data Dependence Analysis
▩	SUIF: Interprocedural Scalar Analysis Data Dependence Analysis Array Privatization Array Reduction



(C) Granularity of Parallelism



(D) Speedup on 8 Processors

Fig. 16. SPECFP95 dynamic measurements on 8-processor Digital AlphaServer.

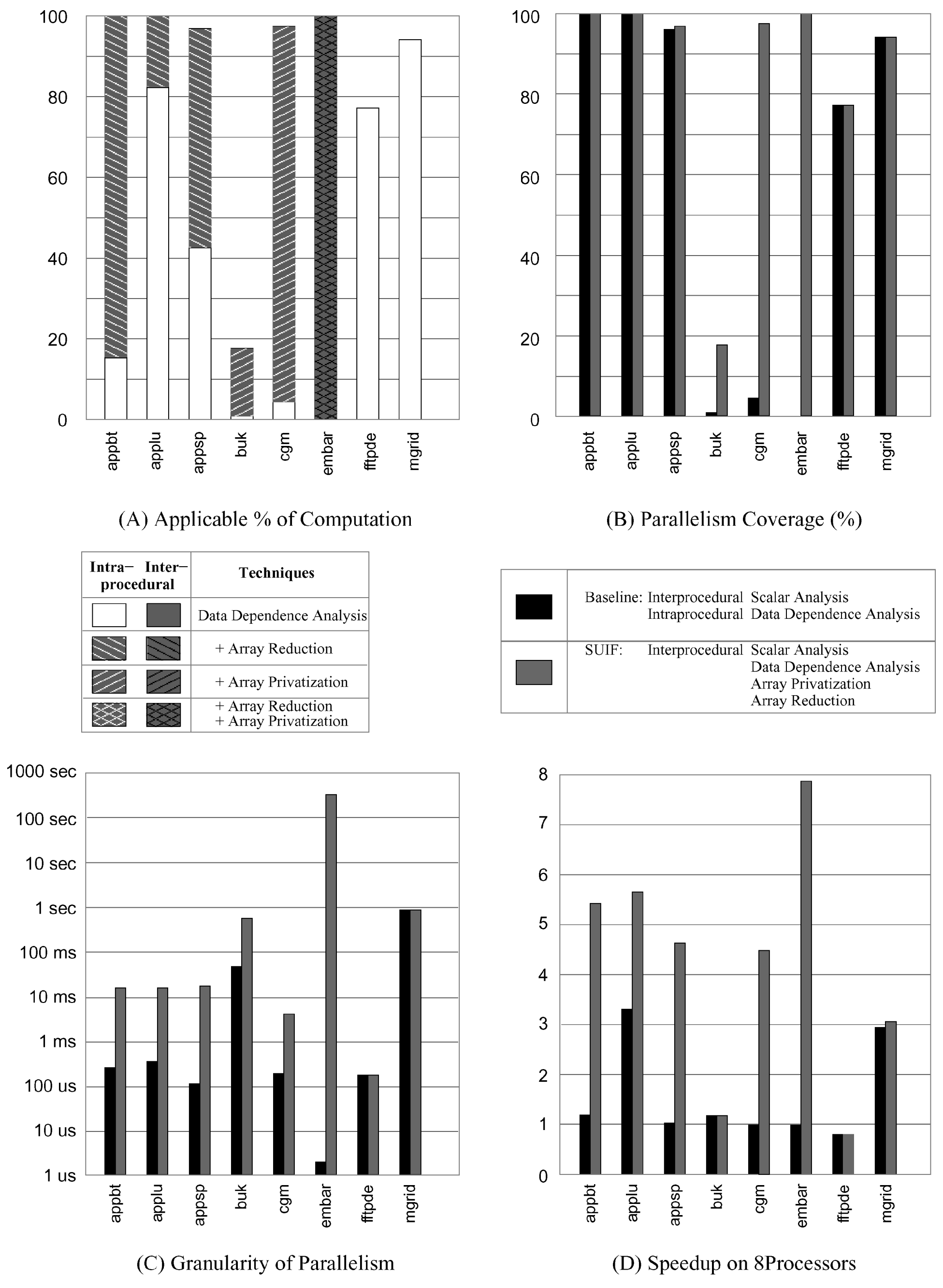


Fig. 17. NAS dynamic measurements with large data sets on 8-processor Digital AlphaServer.

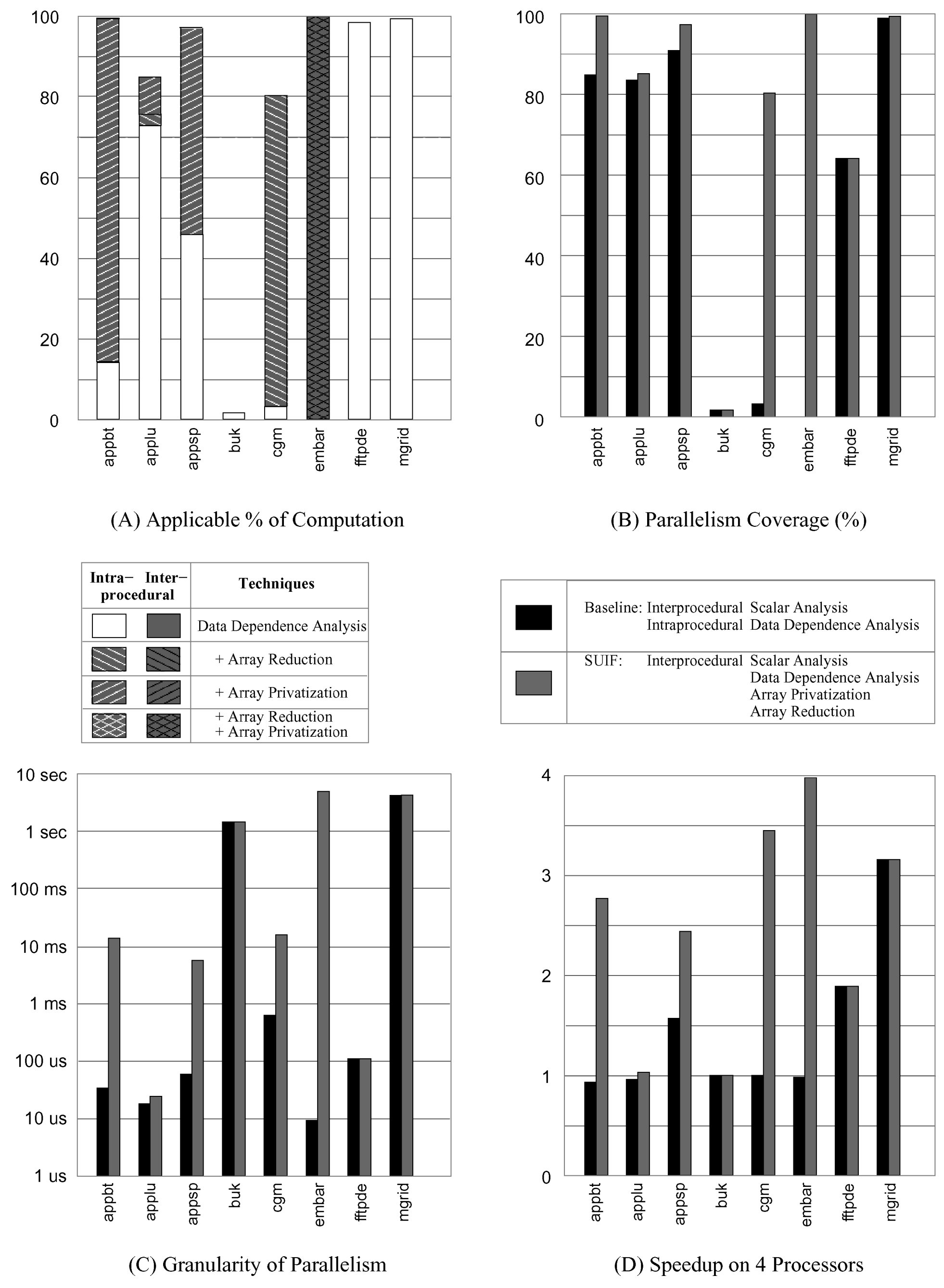


Fig. 18. NAS dynamic measurements with small data sets on 4-processor SGI Challenge.

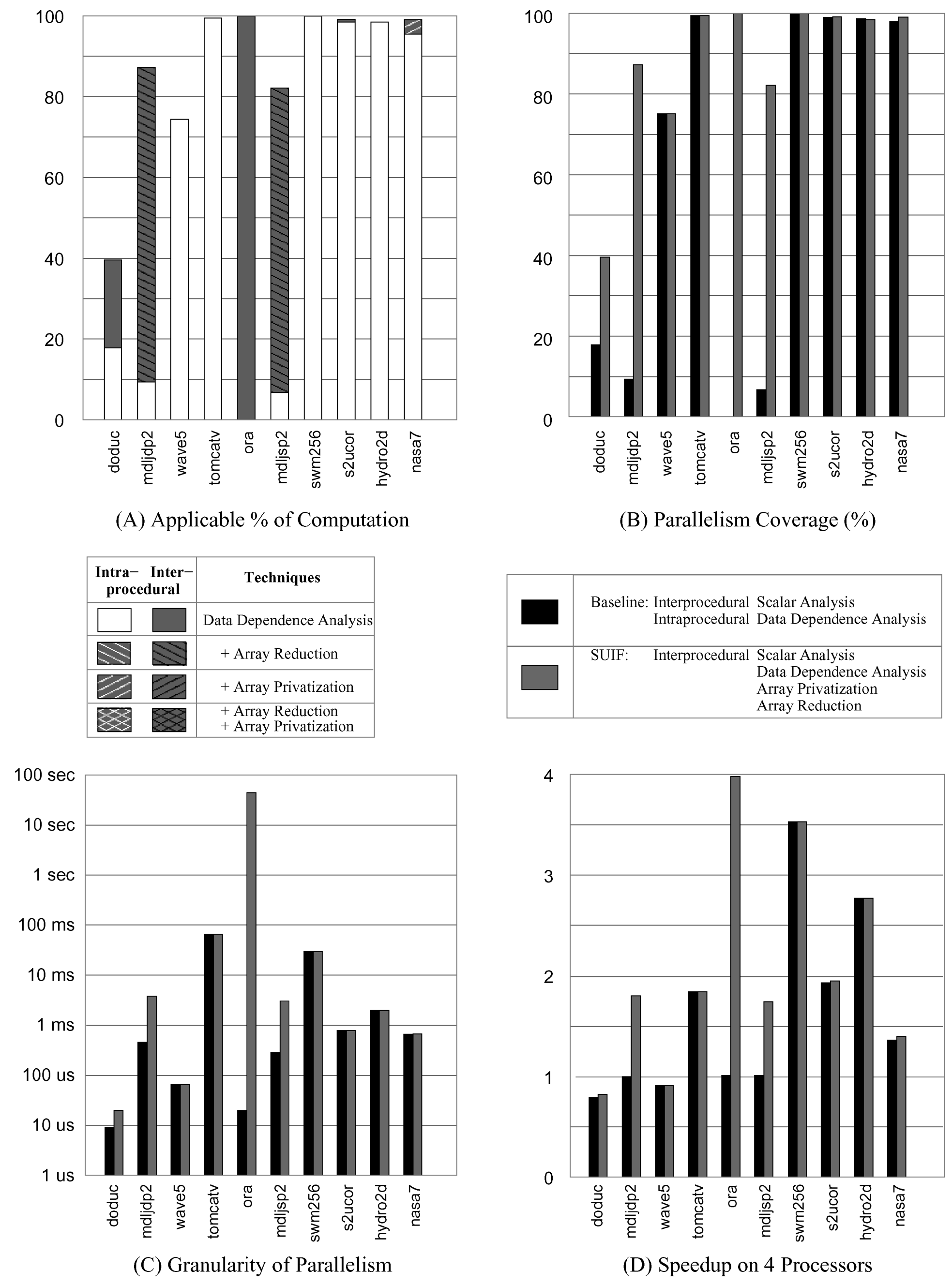


Fig. 19. SPECFP92 dynamic measurements on 4-processor SGI Challenge.

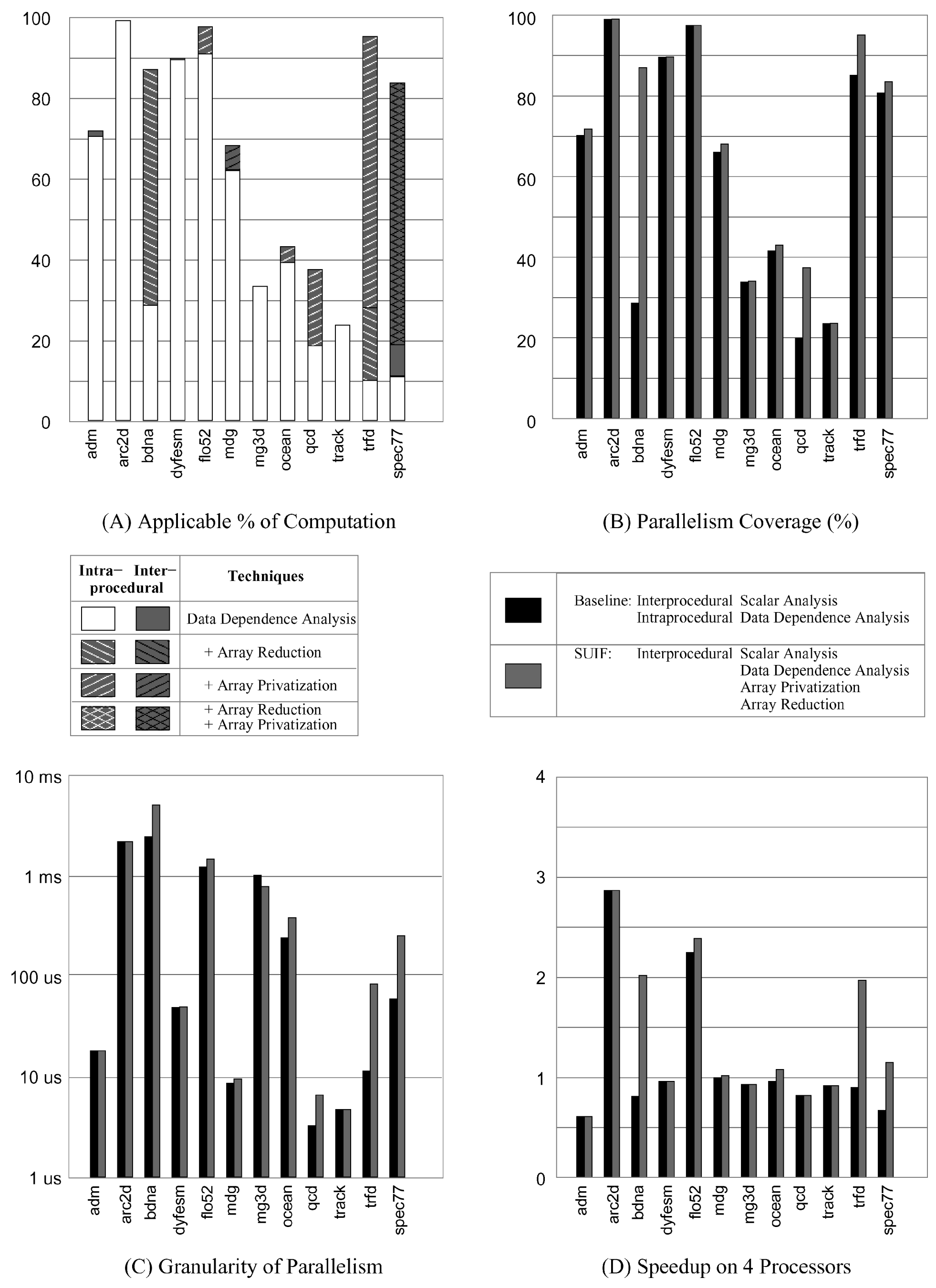


Fig. 20. PERFECT dynamic measurements on 4-processor SGI Challenge.

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.
Coverage measurements were taken by running the programs on a single processor of the target multiprocessor and measuring both overall execution time and time spent in parallelized program regions.

—*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 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.

In Figures 16–20, column (A) presents overall coverage, depicting how much of it is spent in regions parallelized by the advanced array analyses. The contribution of each analysis component is measured by recording the specific array analyses that apply to each parallelized loop while obtaining coverage measurements. This graph demonstrates how important parallelizing a single loop requiring one of the advanced analysis techniques can be. For example, the program `mdljdp2` in Figure 19 contains just two loops requiring interprocedural reduction, but those two loops are where the program spends 78% of its time. Note that even when interprocedural analysis is used to parallelize say, 100% of the computation, it does not mean that a noninterprocedural parallelizer will find no parallelism at all, as it may parallelize an inner loop.

We also present coverage measurements in column (B) of Figures 16–20 that compare the SUIF-parallelized programs with baseline results. Column (C) of the figures shows a comparison of granularity between SUIF and the baseline compiler.

For the sake of completeness, we also present a set of speedup measurements in column (D) of the figures. The programs in Figures 16–17 are sufficiently coarse-grained and have large enough data set sizes for the 8-processor Digital AlphaServer. The programs in Figures 18–20 have relatively short execution times as well as fine granularities of parallelism. Most of these programs cannot utilize a large number of processors effectively so we present results for the 4-processor SGI Challenge only. Speedups are calculated as ratios between the execution time of the original sequential program and the parallel execution time.

10.4.2.2 SPECFP95 *Benchmarks.* Results for the SPECFP95 programs appear in Figure 16. The overall coverage of the SPECFP95 programs is high, above 80%. The advanced analyses significantly affect two of the programs—interprocedural array analysis in `turb3d` and array privatization in `applu`. The dramatically improved speedups for these two programs illustrate the two ways in which the advanced analyses can enhance performance behavior. For `turb3d`, the interprocedural array analysis leads to a substantial increase in parallelism coverage, while for `applu`, only granularity of parallelism is greatly improved.

These results also illustrate that high coverage is a necessary but not sufficient condition for good speedup. The programs apsi and wave5 are too fine-grained to yield any parallel speedup on the chosen target machines.

Memory effects also contribute to the performance of these programs. The program swim shows superlinear speedup because its working set fits into the multiprocessor's aggregate cache. Additionally, performance of tomcatv and swim can be further improved with optimizations to improve locality and minimize synchronization [Anderson et al. 1995; Bugnion et al. 1996; Tseng 1995].

10.4.2.3 NAS *Benchmarks.* The advanced array analyses in SUIF are essential to the successful parallelization of the NAS benchmarks, as can be seen in the Figure 17, which presents measurements using the large data sets on the Digital AlphaServer, and Figure 18, which presents measurements using the small data sets on the SGI Challenge. While there are some variations in results for the two sets of runs, the relative behavior of the programs is similar.

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 applu, appbt, 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% for the small data set sizes—is significant. With 85% of the program parallelized, by Amdahl's Law, the best that can be achieved is a speedup of 2.75, instead of 4, on a machines with 4 processors.

The improvements in coverage and granularity in NAS translate to good speedup results. Six of the eight programs on the AlphaServer and seven of the eight programs on the Challenge yield a speedup. Of the two programs that do not yield a speedup, buk's low coverage is not surprising as it implements a bucket sort algorithm, which is not amenable to the parallelization techniques described in this paper. Fftpde, although it has high coverage, is too fine-grained to yield any speedup on the Alpha. Overall, the advanced array analyses are important for NAS; five of the eight programs would not speed up without these techniques.

10.4.2.4 SPECFP92 *Benchmarks.* Figure 19(B) shows that the advanced array analyses dramatically increase parallelism coverage on 3 of the 10 programs. These new parallel loops are also rather coarse grained, as can be observed from Figure 19(C). Overall the compiler, achieves good results parallelizing SPECFP92. Coverage is above 80% for 8 of the 10 programs, and a speedup is achieved on all of these eight.

Again, we see that programs must have both high coverage and large granularity to yield good speedups. Programs with fine granularity of parallelism, even those with high coverage such as su2cor, tomcatv and nasa7, tend to have lower speedups. Locality and synchronization optimizations can further improve the results of tomcatv and nasa7 [Anderson et al. 1995; Tseng 1995].

10.4.2.5 PERFECT *Benchmarks.* As displayed in Figure 20, 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 three benchmark suites. As Figure 20 suggests, 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 fine-grained loops in PERFECT as discussed in Section 8, 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 SPECFP92 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 nonlinear 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 loops from subroutine run in adm (and apsi) is parallelizable only if the problem size, which is unknown at compile time, is even. (For the given inputs, the loops are parallelizable.) A promising solution is to have the program check if the loop is parallelizable at run time, 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, our compiler finds in the program qcd a 617-line interprocedural loop in subroutine linkbr that would be parallelizable if not for a small procedure. Examination of that

Table VII. Summary of Experimental Results

	SPEC95FP	NAS	SPEC92FP	PERFECT
Number of Programs	9	8	10	12
Improved Coverage (>80%)	1	2	2	1
Increased Granularity	1	3	0	2
Improved Speedup (>50% of ideal)	2	5	1	2

procedure reveals that it is a random number generator, which a user can potentially modify to run in parallel. With a little help from the user, the compiler can parallelize the loop and perform all the tedious privatization and reduction transformations automatically.

10.4.2.6 *Summary.* Overall, we observe rather good coverage (above 80%) for all of the 9 programs in SPECFP95, 7 of the 8 NAS programs, 8 of the 10 programs in SPECFP92, 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.

Table VII summarizes the impact of the improvements from the advanced array analyses on coverage, granularity and speedup in the four benchmark suites. The first row contains the number of programs reported from each benchmark suite. The second row shows how many programs have their coverage increased to be above 80% after adding the advanced array analyses. The third row gives the number of programs that have increased granularity (but similar coverage) as a result of the advanced array analyses. The fourth row shows how these significant improvements affect overall performance. For those with either improved coverage or increased granularity, all but 3 have a speedup better than 50% of ideal.

11. CONCLUSIONS

This article has described the analyses in a fully interprocedural automatic parallelization system. We have presented extensive experimental results demonstrating that interprocedural array data-flow analysis, array privatization, and reduction recognition are key technologies that greatly improve the success of automatic parallelization for shared-memory multiprocessors. By finding coarse-grain parallelism, the compiler increases parallelization coverage, lowers synchronization and communication costs and, as a result, 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 provided a good starting point to investigate these techniques.

More information on the SUIF compiler project can be found on the SUIF web site at `http://suif.stanford.edu`. The parallelized SPECFP95 benchmarks can be obtained from the Spec committee at their web site, under the directory `http://www.specbench.org/osg/cpu95/par-research/`.

ACKNOWLEDGMENTS

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

REFERENCES

ADVE, V., CARLE, A., GRANSTON, E., HIRANANDANI, S., KENNEDY, K., KOELBEL, C., U. KREMER, J. M.-C., TSENG, C.-W., AND WARREN, S. 1994. Requirements for data-parallel programming environments. *IEEE Trans. Paral. Distrib. Tech. 2*, 3, 48–58.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*, Second ed. Addison-Wesley, Reading, Mass.

ALLEN, F. AND COCKE, J. 1976. A program data flow analysis procedure. *Commun. ACM 19*, 3 (Mar.), 137–147.

AMARASINGHE, S. 1997. Parallelizing compiler techniques based on linear inequalities. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ, Stanford, Calif.

AMARASINGHE, S., ANDERSON, J., WILSON, C., LIAO, S., MURPHY, B., FRENCH, R., LAM, M., AND HALL, M. 1996. Multiprocessors from a software perspective. *IEEE Micro 16*, 3 (June), 52–61.

AMMARGUELLAT, Z. AND HARRISON, W. 1990. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, N.Y). ACM, New York.

ANDERSON, J. M., AMARASINGHE, S. P., AND LAM, M. S. 1995. Data and computation transformations for multiprocessors. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, Calif.). ACM, New York, 166–178.

BAILEY, D., BARTON, J., LASINSKI, T., AND SIMON, H. 1991. The NAS parallel benchmarks. *Int. J. Supercomput. Appl. 5*, 3 (Fall), 63–73.

BALASUNDARAM, V. AND KENNEDY, K. 1989. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, Ore.). ACM, New York.

BALL, J.E. 1979. Predicting the effects of optimization on a procedure body. *ACM SIGPLAN Notices 14*, 8, 214–220.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Mass.

BANNING, J. P. 1979. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th Annual Symposium on Principles of Programming Languages*. ACM, New York.

BLELLOCH, G. E. 1990. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa. Nov. (Also published in *Synthesis of Parallel Algorithms*, ed. J. E. Reif.)

BLUME, W. 1995. Symbolic analysis techniques for effective automatic parallelization. Ph.D. dissertation.

BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAEK, Y., POTTENGER, B., RAUCHWERGER, L., AND TU, P. 1996. Parallel programming with Polaris. *IEEE Comput. 29*, 12 (Dec.), 78–82.

BLUME, W. AND EIGENMANN, R. 1992. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Trans. Paral. Distrib. Syste. 3*, 6 (Nov.), 643–656.

BLUME, W. AND EIGENMANN, R. 1994. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of Supercomputing '94*. IEEE Computer Society Press, New York.

BUGNION, E., ANDERSON, J. M., MOWRY, T. C., ROSENBLUM, M., AND LAM, M. S. 1996. Compiler-directed page coloring for multiprocessors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (Cambridge, Mass.).

BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (Palo Alto, Calif.). ACM, New York.

CARROLL, M. AND RYDER, B. 1987. An incremental algorithm for software analysis. In *Proceedings of the SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Notices 22*, 1, 171–179.

COOPER, K., HALL, M., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Comput. Lang. 19*, 2 (Apr.).

COOPER, K., HALL, M. W., AND TORCZON, L. 1991. An experiment with inline substitution. *Softw.—Pract. Exper. 21*, 6 (June), 581–601.

COOPER, K. AND KENNEDY, K. 1984. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction. SIGPLAN Notices 19*, 6, 247–258.

COOPER, K. D., HALL, M. W., KENNEDY, K., AND TORCZON, L. 1995. Interprocedural analysis and optimization. *Commun. Pure Appl. Math. 48*.

CREUSILLET, B. 1996. Array region analyses and applications. Ph.D. dissertaion. Ecole des Mines de Paris.

CREUSILLET, B. AND IRIGOIN, F. 1995. Interprocedural array region analyses. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, New York.

CYBENKO, G., KIPP, L., POINTER, L., AND KUCK, D. 1990. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing* (Amsterdam, The Netherlands). ACM, New York.

DANTZIG, G. AND EAVES, B. 1973. Fourier-Motzkin elimination and its dual. *J. Combinat. Theory (A) 14*, 288–297.

FEAUTRIER, P. 1988a. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing* (St. Malo, France).

FEAUTRIER, P. 1988b. Parametric integer programming. *Recherche Operationnelle/Oper. Res. 22*, 3 (Sept.), 243–268.

FEAUTRIER, P. 1991. Dataflow analysis of scalar and array references. *Int. J. Paral. Prog. 20*, 1 (Feb.), 23–52.

GOFF, G., KENNEDY, K., AND TSENG, C.-W. 1991. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ont., Canada). ACM, New York.

GRAHAM, S. L. AND WEGMAN, M. 1976. A fast and usually linear algorithm for global data flow analysis. *J. ACM 23*, 1, 172–202.

GRANSTON, E. D. AND VEIDENBAUM, A. 1991. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91* (Albuquerque, N.M.) IEEE Computer Society Press, Los Alamitos, Calif.

GROSS, T. AND STEENKISTE, P. 1990. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Soft.—Pract. Exper. 20*, 2 (Feb.), 133–155.

GROUT, J. 1995. Inline expansion for the Polaris research compiler. M.S. thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

HAGHIGHAT, M. AND POLYCHRONOPOLOUS, C. 1996. Symbolic analysis for parallelizing compilers. *ACM Trans. Prog. Lang. Syst. 18*, 4 (July).

HALL, M., ANDERSON, J., AMARASINGHE, S., MURPHY, B., LIAO, S., BUGNION, E., AND LAM, M. 1996. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Comput. 29*, 12 (Dec.), 84–89.

HALL, M. AND KENNEDY, K. 1992. Efficient call graph analysis. *ACM Lett. Prog. Lang. Syst. 1*, 3 (Sept.), 227–242.

HALL, M. W., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., AND LAM, M. S. 1995a. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95* (San Diego, Calif.). IEEE Computer Society Press, Los Alamitos, Calif.

HALL, M. W., MELLOR-CRUMMEY, J., CARLE, A., AND RODRIGUEZ, R. 1993. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing* (Portland, Ore.).

HALL, M. W., MURPHY, B. R., AND AMARASINGHE, S. P. 1995b. Interprocedural analysis for parallelization: Design and experience. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, Calif.). SIAM, Philadelphia, Pa, 650–655.

HALL, M. W., MURPHY, B. R., AMARASINGHE, S. P., LIAO, S.-W., AND LAM, M. S. 1995c. Interprocedural analysis for parallelization. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, New York.

HARRISON, W. 1989. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation 2*, 3/4 (Oct.), 179–396.

HAVLAK, P. 1994. Interprocedural symbolic analysis. Ph.D. dissertation. Dept. of Computer Science, Rice University.

HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Tran. Paral. Distrib. Syst. 2*, 3 (July), 350–360.

HIND, M., BURKE, M., CARINI, P., AND MIDKIFF, S. 1994. An empirical study of precise interprocedural array analysis. *Sci. Prog. 3*, 3, 255–271.

HUMMEL, S. F., SCHONBERG, E., AND FLYNN, L. E. 1992. Factoring: A method for scheduling parallel loops. *Commun. ACM 35*, 8 (Aug.), 90–101.

IRIGOIN, F. 1992. Interprocedural analyses for programming environments. In *Proceedings of the NSF-CNRS Workshop on Evironments and Tools for Parallel Scientific Programming*. North-Holland, Amsterdam, The Netherlands.

IRIGOIN, F., JOUVELOT, P., AND TRIOLET, R. 1991. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing* (Cologne, Germany). ACM, New York.

KAM, J. AND ULLMAN, J. 1976. Global data flow analysis and iterative algorithms. *J. ACM 23*, 1 (Jan.), 159–171.

KENNEDY, K. 1976. A comparison of two algorithms for global data flow analysis. *SIAM J. Computing 5*, 1, 158–180.

LANDI, W. AND RYDER, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices 27*(7), 235–248.

LI, Z. AND YEW, P. 1988. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)* (New Haven, Conn.). ACM, New York.

MARLOWE, T. J. AND RYDER, B. G. 1990. Properties of data flow frameworks: A unified model. *Acta Inf. 28*, 121–163.

MASLOV, V. 1992. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*. ACM, New York.

MAYDAN, D. E., AMARASINGHE, S. P., AND LAM, M. S. 1993. Array data-flow analysis and its use in array privatization. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages* (Charleston, S.C.). ACM, New York, 2–15.

MAYDAN, D. E., HENNESSY, J. L., AND LAM, M. S. 1991. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ont., Canada). ACM, New York.

MAYDAN, D. E., HENNESSY, J. L., AND LAM, M. S. 1992. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*.

MOWRY, T., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)* (Boston, Mass.). 62–73.

MYERS, E. 1981. A precise inter-procedural data flow algorithm. In *Conference Record of the 8th Annual Symposium on Principles of Programming Languages*. ACM, New York.

PADUA, D. A. AND WOLFE, M. J. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (Dec.), 1184–1201.

POLYCHRONOPOULOS, C. 1986. On program restructuring, scheduling, and communication for parallel processor systems. Ph.D. dissertation. Dept. Computer Science. Univ. Illinois at Urbana-Champaign.

POTTENGER, B. AND EIGENMANN, R. 1995. Parallelization in the presence of generalized induction and reduction variables. In *Proceedings of the 1995 ACM International Conference on Supercomputing*. ACM, New York.

PUGH, W. AND WONNACOTT, D. 1992. Eliminating false data dependences using the Omega test. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, Calif.). ACM, New York.

REILLY, J. 1995. SPEC describes SPEC95 products and benchmarks. Spec newsletter, SPEC. September.

RIBAS, H. 1990. Obtaining dependence vectors for nested-loop computations. In *Proceedings of the 1990 International Conference on Parallel Processing* (St. Charles, Ill.).

SCHRIJVER, A. 1986. *Theory of Linear and Integer Programming*. Wiley, Chichester, Great Britain.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs., N.J.

SHIVERS, O. 1991. Control-flow analysis of higher-order languages. Ph.D. dissertation, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

SINGH, J. P. AND HENNESSY, J. L. 1991. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors* (Tokyo, Japan).

TARJAN, R. 1981a. A unified approach to path problems. *J. ACM 28*, 3 (July), 577–593.

TARJAN, R. E. 1981b. Fast algorithms for solving path problems. *J. ACM 28*, 3 (July), 594–614.

TJIANG, S. AND HENNESSY, J. 1992. Sharlit—A tool for building optimizers. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation. SIGPLAN Notices 27*, 7, 82–93.

TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (Palo Alto, Calif.). ACM, New York.

TSENG, C.-W. 1995. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, Calif.). ACM, New York, 144–155.

TU, P. 1995. Automatic array privatization and demand-driven symbolic analysis. Ph.D. dissertation, Dept. Computer Science, Univ. Illinois at Urbana-Champaign.

TU, P. AND PADUA, D. 1993. Automatic array privatization. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing* (Portland, Ore.).

ULLMAN, J. 1973. Fast algorithms for the elimination of common subexpressions. *Acta Inf. 2*, 191–213.

UNIEJEWSKI, J. 1989. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC. Fall.

WOLF, M. E. 1992. Improving locality and parallelism in nested loops. Ph.D. dissertation. Dept. of Computer Science, Stanford Univ., Stanford, Calif.

WOLFE, M. J. AND BANERJEE, U. 1987. Data dependence and its application to parallel processing. *Int. J. Parall. Prog. 16*, 2 (Apr.), 137–178.

Received May 2003; revised December 2003; accepted April 2004