Interprocedural Analysis for Parallelization

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

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

†*Computer Science Dept.*
California Institute of Technology
Pasadena, CA 91125

Abstract. This paper presents an extensive empirical evaluation of an interprocedural parallelizing compiler, developed as part of the Stanford SUIF compiler system. The system incorporates a comprehensive and integrated collection of analyses, including privatization and reduction recognition for both array and scalar variables, and symbolic analysis of array subscripts. The interprocedural analysis framework is designed to provide analysis results nearly as precise as full inlining but without its associated costs. Experimentation with this system on programs from standard benchmark suites demonstrate that an integrated combination of interprocedural analyses can substantially advance the capability of automatic parallelization technology.

1 Introduction

Symmetric shared-memory multiprocessors, built out of the latest microprocessors, are now a widely available class of powerful machines. As hardware technology advances make pervasive parallel computing a possibility, compilers which can extract parallelism from sequential codes become important tools to simplify parallel programming. Unfortunately, today's commercially available parallelizing compilers are not effective at getting good performance on multiprocessors [3, 19]. These compilers tend to be successful in parallelizing only innermost loops. Parallelizing just inner loops is not adequate for multiprocessors for two reasons. First, inner loops may not make up a significant portion of the sequential computation, thus limiting the parallel speedup by limiting the amount of parallelism. Second, synchronizing processors at the end of inner loops leaves little computation occurring in parallel between synchronization points. The cost of frequent synchronization and its associated load imbalance can potentially overwhelm the benefits of parallelization.

If compilers are to successfully locate outer, coarse-grain parallel loops, two improvements are needed. First, parallelizing compilers must incorporate ad-

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

vanced array analyses, generalizing techniques currently only applied to scalar variables. For example, the compiler must recognize opportunities for *array privatization*, whereby storage-related dependences on array variables are eliminated by making a private copy of the array for each processor. As another example, the compiler must recognize opportunities to parallelize *array reductions*, such as computations of a sum, product, or maximum over array elements.

A second essential requirement for recognizing coarse-grain parallel loops is that procedures must not pose a barrier to analysis. One way to eliminate procedure boundaries is to perform inline substitution—replacing each procedure call by a copy of the called procedure—and perform analysis in the usual way. This is not a practical solution for large programs, as it is inefficient in both time and space. Interprocedural analysis, which applies data-flow analysis techniques across procedure boundaries, can be much more efficient as it analyzes only a single copy of each procedure. However, progress in interprocedural analysis has been inhibited by the complexity of interprocedural systems and the inherent tradeoff between performing analysis efficiently and obtaining precise results.

We have developed an automatic parallelization system that is fully interprocedural, and incorporates all standard analyses included in today's parallelizers, such as data dependence analysis, analyses of scalar values such as induction variable recognition, and scalar dependence and reduction recognition. In addition, the system employs analyses for array privatization and array reduction recognition. This system has allowed extensive empirical evaluation of automatic parallelization of three standard benchmark suites, demonstrating significant improvements over previous interprocedural parallelization systems and the technology available in commercial systems.

This paper describes the components of this system, and the interprocedural analysis framework in which they were developed. The key distinguishing features of this system are as follows. First, the interprocedural analysis is designed to be practical while providing nearly the same quality of analysis as if the program were fully inlined. Second, the array analysis incorporates a mathematical formulation of array reshapes at procedure boundaries, supporting changes in dimension between actual and corresponding formal parameters. Third, the system recognizes interprocedural array reductions. Finally, because the system has been used in an extensive empirical evaluation, the implementations of all the analysis techniques extend previous work to meet the demands of parallelizing real programs.

The remainder of the paper is organized into seven sections. Section 2 compares our work with other automatic parallelization systems. In Section 3, we present the interprocedural analysis framework and algorithm. Sections 4 and 5, describe the analysis of scalar variables and array variables, presented as instantiations of the analysis framework from Section 3. Section 6 describes how the interprocedural array analysis is extended to recognize array reductions. The final two sections discuss experiences with this system and conclude.

2 Related Work

In the late 1980s, a series of papers presented results on interprocedural parallelization analysis [9, 15, 20]. Their common approach was to determine the sections of arrays that are modified or referenced by each procedure call, enabling parallelization of some loops containing calls whenever each invocation modifies array elements distinct from those that are referenced or modified in other invocations. These techniques were shown to be effective in parallelizing linear algebra libraries. More recently, the FIDA system was developed at IBM to obtain more precise array sections through partial inlining of array accesses [10] (see Section 7).

Irigoin et al. developed an interprocedural analysis system, called PIPS, that is part of an environment for parallel programming [12]. More recently, PIPS has been extended to incorporate interprocedural array privatization [11, 5]. PIPS is most similar to our work, but lacks three important features: (1) path-specific interprocedural information such as obtained through selective procedure cloning, (2) interprocedural reductions, and (3) extensive interprocedural scalar data-flow analysis such as scalar privatization.

The Polaris system at University of Illinois is also pushing the state of the art in parallelization technology [2]. The most fundamental difference between our system and Polaris is that Polaris performs no interprocedural analysis, instead relying on full inlining of the programs to obtain interprocedural information. The Polaris group has demonstrated that good coverage (fraction of the program parallelized) can be obtained automatically. Although they report that full inlining is feasible on eight medium-sized programs, this approach will likely have difficulty parallelizing large loops containing thousands of lines of code.

3 Interprocedural Framework

Parallelization depends upon the solution of a large number of data-flow analysis problems, which share many commonalities. Traditional data-flow analysis frameworks help reduce development time and improve correctness by capturing these common features in a single module [13]. In an interprocedural setting, a framework is even more important because of the complexity of collecting and managing information about all the procedures in a program.

We use FIAT [6], a tool which encapsulates the common features of interprocedural analysis, in combination with the Stanford SUIF compiler to constitute our interprocedural parallelization system. The FIAT system has been described previously, but we have extended the system to obtain precise flow-sensitive interprocedural results through the combination of two techniques which we now describe. We have also added to the system a mathematical formulation of array reshapes (see Section 5.2) in order to support interprocedural array analysis. This section describes FIAT's parameterized templates that drive the parallelization analysis.

Region-Based Flow-Sensitive Analysis. To capture precise interprocedural information requires a *flow-sensitive* analysis approach, which derives analysis results along each possible control flow path through the program. Precise and efficient flow-sensitive interprocedural analysis is difficult because information flows into a procedure both from its callers (representing the *calling context* in which the procedure is invoked) and from its callees (representing the 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* [16], where individual control flow graphs for the procedures in the program are linked together at procedure call and return points. Iterative analysis over this structure is slow because the number of control flow paths through which information flows increases greatly. Such analysis also loses precision by propagating information along *unrealizable paths* [14]; the analysis may propagate calling context information from one caller through a procedure and return the side-effect information to a different caller. In our system, we use a region-based analysis that solves the problems of unrealizable paths and slow convergence. We perform analysis efficiently in two passes over the program.

Selective Procedure Cloning. For procedures invoked on multiple distinct paths through a program, traditional interprocedural analysis forms a conservative approximation of the information entering the procedure that is correct for all paths. Such approximations can affect the precision of analysis if a procedure is invoked along paths that contribute very different information. Path-specific interprocedural information has previously been obtained either by inline substitution or by tagging data-flow sets with a path history through the call graph, incurring a data-flow set expansion problem corresponding to the code explosion problem of inlining [8, 16, 17, 18]. To avoid such excessive space usage, we utilize path-specific information only when it may provide opportunities for improved optimization. Our system incorporates *selective procedure cloning*, a program restructuring in which the compiler replicates the analysis results for a procedure to analyze it in the context of distinct calling environments [4]. By applying cloning *selectively* according to the unique data-flow information it exposes, we can obtain the same precision as full inlining without unnecessary replication.

3.1 The Region Graph

The region-based analysis aggregates information at the boundaries of program regions: basic blocks, loop bodies and loops (restricted to DO loops), procedure calls, procedure bodies, and procedures. We use a program representation called the *region graph* to represent the loop nesting and procedure nesting of the program. The region graph is a directed graph whose nodes represent regions and whose edges represent nesting relationships. With each region is associated an *immediate subregions graph*, a directed flow graph consisting of immediately nested regions and control flow edges between them.

Each region has a single entry node. To simplify presentation, we primarily describe analyses with regions that also have a single exit. (The actual analysis framework implementation is more general. Irreducible graphs are supported in the scalar data-flow analysis described in Section 4, although the array analysis approximates when graphs are irreducible or when loops contain multiple exits.)

3.2 Data-Flow Functions

The first phase of any program analysis using this framework yields a transfer function for each region in a problem-specific form. For each analysis, a representation for transfer functions $\mathcal{T}$ with the following operations must be provided:

- Extract basic block transfer function ($\mathcal{T} = \mathrm{BasicBlockTF}(b)$)
- Composition ($\circ$)
- Meet ($\bigwedge$), with identity value ($\top$)
- Iteration ($\mathcal{T}^i$): yield effect after i iterations, where i is the loop's normalized index variable.
- Closure ($\mathcal{T}^*$): eliminate the most recent loop index variable to describe the effect of the entire loop.
- RetMap($\mathcal{T}$, *callsite*): map procedure transfer function into caller space

The first phase computes relative information that summarizes the behavior of each region. To compute absolute information, a second phase may optionally be performed. The second phase determines absolute information on entry to each procedure and region, using the transfer functions found in the first phase to propagate a problem-specific data-flow value. A representation for this value must be provided, along with the following operations:

- Context of program (input value $-$)
- Meet ($\bigwedge$), with identity value ($\top$)
- Apply transfer function to a data-flow value to yield another
- CallMap(*val*, *callsite*): map call context into procedure space
- Filter(*val*, *Proc*): remove information not relevant to *Proc*
- Partition(val_1, val_2): equivalence relation on procedure contexts

3.3 Algorithm

A region-based analysis, as shown in Figure 1, proceeds in one or two phases. In the first phase, we analyze each procedure independent of its calling environment to obtain a transfer function $\mathcal{T}_p$; this transfer function is used (with appropriate parameter mapping) at call sites when analyzing callers. The second phase propagates data-flow values, applying them to the transfer functions from the previous phase, to yield the data-flow input to each region. The two-phase region-based analysis is similar to what is traditionally called interval-based analysis, where the intervals of interest are loops and procedure bodies.

```
/* 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, compute $\mathcal{T}_R = \text{BasicBlockTF}(R)$
        if $R$ is a loop with body $R'$,
            $\mathcal{T}_{R,R'} = \mathcal{T}^i_{R'}$ for this loop's normalized index variable $i$
            $\mathcal{T}_R = \mathcal{T}^*_{R,R'}$
        if $R$ is a call at site $cs$ to procedure with body $R'$,
            $\mathcal{T}_R = \text{RetMap}(\mathcal{T}_{R'}, cs)$ /* map parameters */
        if $R$ is a loop body or procedure body,
            for a forward data-flow problem,
                for each immediate subregion $R'$,
                    compute $\mathcal{T}_{R,R'}$ = transfer function from entry of $R$ to entry of $R'$
                by finding least solution to (for all $R'$):
                    $\mathcal{T}_{R,R'} = \bigwedge_{p \in pred(R')} \mathcal{T}_p \circ \mathcal{T}_{R,p}$
                $\mathcal{T}_R = \mathcal{T}_{\text{Exit}(R)} \circ \mathcal{T}_{R,\text{Exit}(R)}$
            for a backward data-flow problem,
                for each immediate subregion $R'$,
                    compute $\mathcal{T}_{R,R'}$ = transfer function from exit of $R$ to exit of $R'$
                by finding least solution to (for all $R'$):
                    $\mathcal{T}_{R,R'} = \bigwedge_{p \in succ(R')} \mathcal{T}_p \circ \mathcal{T}_{R,p}$
                $\mathcal{T}_R = \mathcal{T}_{\text{Entry}(R)} \circ \mathcal{T}_{R,\text{Entry}(R)}$

/* PHASE 2: Derive Procedure Contexts using Transfer Functions */
/*          and propagate data flow information to regions */
$\mathcal{C}_{\text{Entry}(Program)} = \{\perp\}$
for each procedure $P$ from top to bottom over call graph,
    let $\mathcal{C}$ be the union of calling contexts on incoming edges
    let $\mathcal{C}' = \{\text{Filter}(c, P) \mid c \in C\}$
    let $\mathcal{P}$ be the equivalence classes of $C'$ with respect to Partition
    for each partition $p \in \mathcal{P}$
        $\mathcal{V}_{p,P} = \bigwedge_{c \in p} c$
        for each region $R$ in $P$ from outermost to innermost,
            for each subregion $R'$ of $R$,
                $\mathcal{V}_{p,R'} = \mathcal{T}_{R,R'}(\mathcal{V}_{p,R})$
            if $R$ is a call at site $cs$ with corresponding call graph edge $e$,
                add context $\text{CallMap}(\mathcal{V}_{p,R}, cs)$ to edge $e$ /* map parameters */
```

Fig. 1. Region-Based Interprocedural Analysis Framework

Phase 1: Calculating Region Transfer Functions For each region R from innermost loop to outermost loop, and from bottom to top in the call graph, we compute its transfer function $\mathcal{T}_R$. A basic block's transfer function is derived directly (using the BasicBlockTF function). The transfer function of a procedure call takes the procedure body's transfer function and maps it to the caller space, renaming variables in its representation (using the RetMap operation).

The transfer function for a loop applies the Iteration operation to the transfer function of its loop body $\mathcal{T}_{R'}$ to obtain a transfer function $\mathcal{T}_{R,R'}$ representing the effect of i iterations of the body, where i is the loop's normalized index variable. The final transfer function showing the total effect of the loop is obtained by using the Closure operation to eliminate the iteration counter i.

For loop bodies and procedure bodies, deriving the transfer function involves the transfer functions of its immediate subregions. In a forward data-flow problem, for each subregion R', we compute $\mathcal{T}_{R,R'}$, the transfer function from the entry of R to the entry of R'. This calculation results from a meet over the predecessors of R'. If the immediate subregions graph is cyclic, then an iterative solution may be required to find the transfer function. Otherwise, the subregions are simply visited in the appropriate (reverse postorder) order within the region. The final transfer function for the loop body or procedure body is derived by composing the transfer function $\mathcal{T}_{R,\text{Exit}(R)}$ for the subregion that represents the exit from region R, with the transfer function $\mathcal{T}_{\text{Exit}(R)}$ of that region.

Data-flow problems that require a backward propagation within the intervals are analogous. For an acyclic subregion graph, a postorder traversal over the subregions derives transfer functions $\mathcal{T}_{R,R'}$ to describe the effects from the exit of R up to the exit of R'.

Phase 2: Deriving Calling Contexts and Computing Final Values. For a two-phase problem, the second phase of the algorithm derives the data-flow input to each procedure and its subregions. This phase of the analysis is performed top-down over the call graph and from outermost to innermost loops within each procedure body. For a procedure, the analysis derives the set of calling contexts $\mathcal{C}$ contributed by calls to the procedure. Instead of performing a meet operation over all of the calling contexts, the analysis partitions these contexts into equivalence classes under the Partition relation according to their data-flow information before meeting only the contexts within each equivalence class.

The number of partitions is reduced by first using a Filter to eliminate from the data-flow values information not relevant to the called procedure. (We describe an example of this filtering in Section 4.)

Each partition defines a data-flow input value $\mathcal{V}_{p,P}$, the meet of the calling contexts in that partition. For each partition, the analysis applies the transfer functions to this data-flow value to propagate information from outermost to innermost to yield the input to inner regions. For a region R representing a procedure call, the analysis adds to the corresponding call graph edge the calling context $\mathcal{V}_{p,R}$. It is important to note that our analysis does not actually generate cloned procedure bodies, but merely replicates their data-flow information for the purposes of analysis.

4 Scalar Data-Flow Analysis

Scalar data-flow analysis is crucial for parallelizing loops. Analyses of scalar variables in a loop are necessary both to detect and eliminate scalar dependences and to support precise analysis of array accesses. Array analysis support is provided by an interprocedural symbolic analysis and a separate inequality constraint propagation.

4.1 Support for Array Analysis: Interprocedural Symbolic Analysis

To precisely represent the array accesses in a loop (using an analysis such as the one to be described in Section 5) requires that array indices be rephrased in terms which are valid throughout the loop. Using traditional program analyses, a set of analyses of integer variable values is needed: constant propagation, induction and loop-invariant variable detection, and common subexpression recognition.

Our system provides the effect of such analyses through a single symbolic analysis, which is performed interprocedurally. For example, to parallelize the following loop:

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

the array index expressions in terms of loop-varying variables (`J` and `K`) are mapped into expressions in terms of normalized (base 0) loop indices and loop invariants. In this particular loop, a new loop-invariant variable $\mathtt{J}_0$ is introduced to refer to the value of `J` on entry to the loop and a base-0 iteration count variable i is introduced, local to the loop body. `J` is found to have a value $\mathtt{J}_0 + 2i$ and `K` a value $\mathtt{J}_0 + 2i + 1$. Substituting these values into the array indices allows comparison of the portions of array `A` read and written by the loop.

The symbolic analysis determines for each variable appearing in an array access a symbolic value: an arbitrary expression describing its value in terms of constants, loop-invariant variables, and normalized loop indices, if possible.

Array dependence analysis typically only handles affine array indices precisely; nevertheless, the symbolic values resulting from the symbolic analysis may be non-affine. In some cases our system is currently unable to make use of this non-affine information. In one common case of non-affine array indices—those resulting from a higher-order induction variable—we extract additional information which can be provided in an affine form, as discussed below.

Representation: Symbolic Maps More formally, a symbolic value expression *sym* is either *Unknown* or an arbitrary arithmetic/conditional expression in terms of constants, variables, and loop indices. A *symbolic map*

$$SM = \{< var_1, sym_1 >, \ldots\}$$

binds variables var_i to symbolic descriptions of their values sym_i. A symbolic map associated with a region R may be either *relative* or *absolute*. In a *relative* map, variables within bound values refer to their values on entry to R; in an *absolute* map, no bound value may contain a variable modified within R.

For convenience below, we define an operation $SM(sym)$ on symbolic map SM and symbolic value sym, which yields *Unknown* if sym contains a var_i not bound in SM, and otherwise yields sym with every occurrence of a var_i bound by SM replaced by the bound value sym_i.

Region-Based Analysis We obtain absolute value maps describing variable values at every program point in two passes, as a region-based data-flow analysis. A bottom-up pass through the program derives the transfer function for each region, as a relative value map that describes variable values at each immediate subregion in terms of entry variable values. A subsequent top-down pass through the program propagates to each region a symbolic context, an absolute map describing actual variable values on region entry in terms of enclosing loop indices and invariants.

Phase 1: Transfer Functions The symbolic behavior of a region R is a relative map SM_R describing every variable's value on exit in terms of enclosing loop indices and variable values on entry to R. The following operations are defined:

- BasicBlockTF(b): forms a map showing the effect of the block on every program variable: unmodified variables are mapped to themselves, modified variables are mapped to a symbolic value expression representing the value on exit in terms of the values on entry. New variables are introduced to represent the values of certain operations with unknown results (e.g., load from memory, I/O read). These variables are limited in scope to the nearest enclosing loop or procedure body.
- Composition ($\circ$): apply SM_2 to every bound value in SM_1:
$$SM_1 \circ SM_2 = \{< var_i, SM_2(sym_i) > \mid < var_i, sym_i > \in SM_1\}$$
- Meet ($\bigwedge$), with identity element $\top_{SM}$:
$$SM_1 \wedge SM_2 = \{< var, sym > \mid sym = (SM_1 \cap SM_2)(var)\}$$
- Iteration: SM^i finds loop invariants and induction variables and rephrases them in terms of the given index variable i. Auxiliary maps SM_0^i (loop invariants) and SM_1^i (induction variables) are used to compute SM^i, as follows:
$$SM_0^i = \{< var, var > \mid < var, var > \in SM\}$$
$$SM_1^i = \{< var, var + i * c > \mid < var, var + c > \in SM, c = SM(c)\}$$
$$SM^i = \{< var, sym > \mid sym = (SM_0^i \cup SM_1^i)(var)\}$$
gives the net change after i iterations of the loop, and includes loop invariant and induction variable recognition.
- Closure: SM^* substitutes an expression *if* $lb \leq ub$ *then* $\lceil (ub - lb)/step \rceil + 1$ *else* 0 for the most recent loop index variable i throughout SM.
- RetMap(SM, *callsite*): maps formals to actuals everywhere in SM.

Phase 2: Symbolic Calling Contexts. The symbolic context of a region R is an absolute map SM_R describing each live variable's value on entry to R in terms of loop invariants and loop indices of enclosing loops.

- Context on entry to program: the initial symbolic context – maps all variables to *Unknown*.
- Meet ($\bigwedge$), identity ($\top$): as for relative maps.
- Apply transfer function: Relative map SM_1 is applied to an absolute map SM_2 to derive a new absolute map: $SM_1\,(SM_2) = SM_1 \circ SM_2$.
- CallMap(SM, *callsite*): map actuals to formals everywhere in SM.
- Filter(SM, $Proc$): Eliminate from the map all bindings of variables with no upwards-exposed reads in $Proc$.
- Partition(SM_1, SM_2): Only identical maps are equivalent.

Cloning. We employ selective procedure cloning based on the values in the map. Currently, the filter function eliminates from the map relations on variables that have no upwards-exposed reads in the called procedure; this significantly reduces the amount of replication in the analysis.

Higher-order Induction Variable Support The closure operation can be extended to recognize higher order induction variables, such as a variable incremented inside a triangular loop. Such variables are not uncommon in scientific codes as linearized array subscripts. To handle 2nd-order induction variables, we extend the iteration operator with an auxiliary SM_2^i map, as follows:

$$\begin{aligned} SM_2^i = \{ &< var, var + c_1 * (i * (i-1))/2 + c_2 * i > \mid \\ &\quad < var, var + c_1 * i + c_2 > \in SM' \circ (SM_0^i \cup SM_1^{i-1}), \\ &\quad c_1 = SM(c_1),\ c_2 = SM(c_2)\} \\ SM^i = \{ &< var, sym > \mid sym = (SM_0^i \cup SM_1^i \cup SM_2^i)(var)\} \end{aligned}$$

Unfortunately, the resulting closed form of a second-order induction variable which is thus introduced is non-affine and not directly useful to the affine parallelization tests used in array analysis. For this reason, the analysis in this case introduces a new variable x, whose scope is limited to the loop body, and in place of the non-affine expression $var + c_1 * (i * (i-1))/2 + c_2 * i$, we use the affine expression $var + x$.

When the array analysis performs a comparison between two accesses containing x, the additional affine information is provided that if, for example, $c_1 \geq 0$ and $c_2 \geq 0$, then for iteration $i = i'$ we have $x = x'$ and for iteration $i = i''$ we have $x = x''$ such that if $i' < i''$ then $x' \leq x'' + c_1 + c_2$. Similar useful affine information can be provided under other conditions on c_1 and c_2. This approach enables one commonly occurring case of non-affine symbolic values in array subscripts to be handled without an expensive extension to the array analysis.

4.2 Inequality Constraints

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

4.3 Scalar Parallelization Analysis

A number of standard analyses ensure that scalar variables do not limit the parallelism available in a loop. These analyses locate scalar dependences, locate opportunities for scalar reduction transformations and determine privatizable scalars. We apply these analyses interprocedurally. A simple flow-insensitive mod-ref analysis[1] detects scalar dependences and, with a straightforward extension, provides the necessary information to locate scalar reductions. A flow-sensitive live-variable analysis, discussed below, allows detection of privatizable scalar variables. The flow-sensitive symbolic analysis of Section 4.1 also finds induction and loop-invariant integer variables, which can then be privatized.

Live Variable Analysis. We solve a standard live-variable problem interprocedurally through a two-phase region-based backward analysis. In Phase 1, the transfer function for each region is computed as a pair of sets: *Gen* set, containing variables with upwards exposed reads in the region, and *Kill* set, containing variables written in the region. In Phase 2, the set of live variables on entry to a region is determined from the set of live variables on exit of the region: $Live_{entry} = (Live_{exit} - Kill) \cup Gen$.

For loops containing returns and breaks, the situation is somewhat complicated, since there is not just a single exit. A single transfer function is not sufficient to describe the behavior of a region with multiple exits in a backward data-flow problem. Instead, we summarize the behavior of a loop body by three transfer functions—from *loop body* exit, from *loop* exit, and from *enclosing procedure* exit. A loop is described by just two transfer functions—from *loop* exit and from *procedure* exit. A single transfer function still suffices to describe a procedure. In other respects the analysis is straightforward.

5 Analysis of Array Variables

The array analysis locates loops that carry no data dependences on array elements or that can be safely parallelized after array privatization. The system, when integrated with the reduction recognition and scalar data-flow analysis, performs an array data-flow analysis based on systems of linear inequalities to analyze affine array access functions. This approach is driven by the need to compute both data dependences 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 summaries, which describe subarrays accessed by a region of the code; summaries eliminate the need to perform $\mathcal{O}(n^2)$ pairwise dependence 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 may span multiple procedures and have hundreds of array accesses.

5.1 Representation: Summaries

We represent each array access by a system of integer linear inequalities. An *array summary* is a set of such systems. For example, consider the following loop nest.

$$\left.\begin{array}{l}\mathtt{DO\ 10\ I\ =\ 1,\ N}\\ \quad\left.\begin{array}{l}\mathtt{DO\ 10\ J\ =\ 1,\ M}\\ \quad\mathtt{A(J+1,\ 2*I)\ =\ \ldots}\ \}\,W_1\end{array}\right\}W_2\end{array}\right\}W_3$$

The region of array `A` written by a single execution of the statement is represented by set containing one system of inequalities, parameterized by the program variables `M` and `N`, and normalized loop index variables `i` and `j`:

$$W_1 = \left\{ (w_1, w_2) \,\middle|\, \left\{ \begin{array}{l} 0 \le \mathtt{j} \le \mathtt{M}-1,\ w_1 = \mathtt{j}+2, \\ 0 \le \mathtt{i} \le \mathtt{N}-1,\ w_2 = 2\mathtt{i}+2 \end{array} \right\} \right\}$$

The included contextual constraints on program variables and loop indices are provided by the scalar context analysis.

Intuitively, a set is necessary because different accesses to an array may refer to distinctly different regions of the array. Mathematically, many of the operators applied to array summaries result in non-convex regions, which cannot be precisely described with a single system of inequalities. To maintain efficiency, we merge systems of inequalities whenever we can guarantee no loss of information will result. The following basic operations are defined on array summaries. Operations marked $*$ are not exact.

– **Empty?** $(A = \emptyset) = \forall_{a \in A}\ (a = \emptyset)$. A set of systems is empty iff all systems in the set are empty. A system of inequalities is empty if there are no integer solutions that satisfy the system. We use a Fourier-Motzkin pair-wise elimination technique with branch-and-bound to check for the existence of an integer solution to a system of inequalities. If no solution exists, the system is empty.

$*$ **Contained?** $A \underset{\sim}{\subseteq} B = \forall_{a \in A} \exists_{b \in B} (a \subseteq b)$. A set of systems is contained in another, iff each system in the first set is contained in a single system in the other set. This is conservative as it may return a false negative. A system of inequalities a is contained in a system of inequalities b if and only if a combined with the negation of any single inequality of b is empty.

- **Union** $A \cup B = \{c \mid c \in A \text{ or } c \in B\}$. The union of two sets of systems simply unions the two sets, then simplifies the set using the following two heuristics:
  - If there are two systems a and b in the set such that $a \subseteq b$, then a is removed from the set.
  - If two systems are rectilinear and adjacent, they are combined to form a single system.

  In practice, these heuristics keep the sets a manageable size and increase the precision of the *Contained?* operator. Since the union of two convex regions can result in a non-convex region, a set is necessary to maintain the precision of the union operator.
- **Intersection** $A \cap B = \{a \cap b \mid a \in A \text{ and } b \in B \text{ and } a \cap b \neq \emptyset\}$. The intersection of two sets of systems is the set of all non-empty pairwise intersections of their elements. Intersection of two systems of inequalities simply concatenates the inequalities of the two systems.

* **Subtraction** $A \sim B = \{a \sim b_1 \sim \ldots \sim b_n \mid a \in A \text{ and } B = \{b_1 \ldots b_n\}\}$. The subtraction of two sets of systems subtracts all systems of the second set from each system in the first. Two systems are subtracted using a heuristic: $a \sim b$ is exact when $a \cap b = \emptyset$ or $a \subseteq b$ or both are simple rectilinear systems; otherwise it is approximated as a.

- **Projections** $\mathit{Proj}(A, v)$ eliminates the variable v from the constraints of all the systems in set A by applying the Fourier-Motzkin elimination technique to each system. Each system $a \in A$ can be viewed as the integer points inside a n-dimensional polytope whose dimensions are the variables of a and whose bounds are given by the inequalities of a; this polytope is projected into a lower-dimensional $(n-1)$ space where the integer solutions of all remaining dimensions remain unchanged. One use of projection is to summarize the effects of array accesses within a loop. For example, for the system of inequalities representing the access to array `A` shown above, projections are used to generate systems of inequalities representing the array accesses for each loop in the nest. In some cases, eliminating a variable may result in a larger region than the actual region. In the example, eliminating the constraint $w_2 = 2\mathtt{i} + 2$ will lose the information that w_2 must be even. For this reason, analysis introduces an auxiliary x in W_3 to retain this constraint.

$$W_2 = \mathit{Proj}(W_1, \mathtt{j}) = \left\{ (\mathtt{w_1}, \mathtt{w_2}) \middle| \left\{ \begin{array}{l} 0 \leq \mathtt{i} \leq \mathtt{N-1},\ w_2 = 2\mathtt{i} + 2 \\ 2 \leq w_1 \leq \mathtt{M+1} \end{array} \right\} \right\}$$

$$W_3 = \mathit{Proj}(W_2, \mathtt{i}) = \left\{ (\mathtt{w_1}, \mathtt{w_2}) \middle| \left\{ \begin{array}{l} 2 \leq w_2 \leq 2\mathtt{N},\quad w_2 = 2x \\ 2 \leq w_1 \leq \mathtt{M+1} \end{array} \right\} \right\}$$

5.2 Array Reshapes

Interprocedural array analysis must provide precise results in the presence of *array reshapes* at procedure boundaries, as when a slice of an array is passed into a procedure, and as in *linearization,* when a multi-dimensional array in one procedure is treated as a linear array in another. In the following example, `FOO` passes `BAR` the `K`th column of the array `X`. This 10000-element vector from `FOO`, is manipulated as a 100×100 array in `BAR`.

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

Mapping an array summary from callee to caller is not a simple rename operation. We perform this mapping by deriving inequalities for the indices of the actual parameter in terms of the indices of the formal parameter and use the projection operation to eliminate the formal parameter's indices.

We formalize the mapping of summary S for an n-dimensional formal array parameter `F`, where $\mathtt{A}(a_1, \ldots, a_m)$ is passed at the call site and actual `A` is an m-dimensional array, using the mapping function:
$\mathcal{M}(S, \mathtt{F}, \mathtt{A}, a_1, \ldots, a_m) = \{(j_1, \ldots, j_m) \mid \mathit{Proj}\,(\{b_\mathtt{F} \cap b_\mathtt{A} \cap r_\mathtt{FA}\} \cap S, \{i_1, \ldots, i_n\})\}$
Where

- $i_1, \ldots, i_n$ are variables representing the indices of accesses to array `F`.
- $j_1, \ldots, j_m$ are variables representing the indices of accesses to array `A`.
- $b_\mathtt{F}$ is the set of bounds for the array `F` given by its type declaration. (Note that the exact bounds of the outermost dimension are not required.)
- $b_\mathtt{A}$ is the set of bounds for the array `A` given by its type declaration.
- $r_\mathtt{FA}$ describes the conditions under which an access $\mathtt{F}(i_1, \ldots, i_n)$ in the procedure and an access $\mathtt{A}(j_1, \ldots, j_m)$ in the callee refer to the same location. This occurs when the memory offset of $\mathtt{F}(i_1, \ldots, i_n)$ is equal to the memory offset of $\mathtt{A}(j_1, \ldots, j_m)$ minus the memory offset of $\mathtt{A}(a_1, \ldots, a_m)$. This relationship between memory offsets is represented as an equality relation; other known facts about variables used in the equality may be included in the system.

For the `FOO`, `BAR` example the mapping function is calculated using:

$$b_\mathtt{F} = \left\{ \begin{array}{c} 1 \le j_1 \le 100 \\ 1 \le j_2 \le 100 \end{array} \right\}, \qquad b_\mathtt{A} = \left\{ \begin{array}{c} 1 \le i_1 \le 10000 \\ 1 \le i_2 \le 10 \end{array} \right\},$$

$$r_\mathtt{FA} = \left\{ \begin{array}{c} 100 * (j_2 - 1) + (j_1 - 1) = 10000 * (i_2 - \mathtt{K}) + (i_1 - 1) \\ 1 \le \mathtt{K} \le 10 \end{array} \right\}$$

Thus when when using the mapping function on the summary of the array `Y` at the start of subroutine `BAR`:

$$\mathcal{M}\left(\left\{(j_1, j_2) \middle| \left\{ \begin{array}{c} 1 \le j_1 \le 100 \\ 1 \le j_2 \le 50 \end{array} \right\}\right\}, \mathtt{Y}, \mathtt{X}, 1, \mathtt{K}\right) = \left\{(i_1, i_2) \middle| \left\{ \begin{array}{c} 1 \le i_1 \le 5000 \\ i_2 = \mathtt{K} \\ 1 \le \mathtt{K} \le 10 \end{array} \right\}\right\}$$

This approach handles precisely cases where complex numbers in one procedure are treated as real numbers in another by modeling a complex number as an array with two elements. Some reshapes are not handled precisely. If array dimensions are unknown, for example, there is no linear relationship between the indices of the actual and formal parameters, and unless the unknown dimensions are identical, we must approximate.

5.3 Region-Based Analysis

The analysis of an array variables computes four distinct sets for each program region. These sets are used by both the dependence and privatization tests to determine the safety of parallelization. The data-flow sets for a given region R are informally defined as follows:

W_R –	*Write*: portions of arrays possibly written within region R
M_R –	*Must Write*: portions of arrays always written within region R
R_R –	*Read*: portions of arrays possibly read within region R
E_R –	*Exposed Read*: portions of arrays whose reads are possibly upwards exposed to the beginning of R

These sets are together computed as a 4-tuple transfer function using a backwards region-based analysis as described in Section 3. Because just the transfer function itself is needed, the second phase of the region-based framework can be omitted.

Transfer Functions. The side-effect transfer function of a region R on a particular array is represented as the 4-tuple

$$S_R =< W_R, M_R, R_R, E_R >$$

where the elements are the sets informally defined as described above. The following operations are defined on S tuples:

- BasicBlockTF(b): result of composing read and write accesses in block b. Read access: $S =< \emptyset, \emptyset, \{a\}, \{a\} >$. Write access: $S =< \{a\}, \{a\}, \emptyset, \emptyset >$. where a is a system describing the access indices, rewritten in loop-relative terms (from symbolic analysis), with relevant inequality constraints added.
- Composition: $S_1 \circ S_2 =< W_1 \cup W_2, M_1 \cup M_2, R_1 \cup R_2, E_1 \cup (E_2 \sim M_1) >$.
- Meet: $S_1 \wedge S_2 =< W_1 \cup W_2, M_1 \cap M_2, R_1 \cup R_2, E_1 \cup E_2 >$.
- Identity element: $\top_S =< \emptyset, \emptyset, \emptyset, \emptyset >$.
- Iteration: $S^i = S$. The given loop index variable i is used to perform dependence and privatization tests, as in the following section.
- Closure: $S^* =< \mathit{Proj}(W, L), \mathit{Proj}(M, L), \mathit{Proj}(R, L), \mathit{Proj}(E, L) >$, where L contains the loop index i and other loop-modified variables.
- RetMap(S, *callsite*) $=< \mathcal{M}(W, \ldots), \mathcal{M}(M, \ldots), \mathcal{M}(R, \ldots), \mathcal{M}(E, \ldots) >$, as discussed in Section 5.2.

5.4 Dependence and Array Privatization Tests

To determine if array accesses allow the parallelization of a loop with index i, dependence and privatization tests are performed on the summary sets for the loop body R:

- There is no loop-carried:
  - *True Dependence* iff $W_R|_i^{i_1} \cap R_R|_i^{i_2} \cap \{i_1 < i_2\} = \phi$
  - *Anti Dependence* iff $W_R|_i^{i_1} \cap R_R|_i^{i_2} \cap \{i_1 > i_2\} = \phi$

    - *Output Dependence* iff $W_R|_i^{i_1} \cap W_R|_i^{i_2} \cap \{i_1 < i_2\} = \phi$
- *Array Privatization* is possible iff $W_R|_i^{i_1} \cap E_R|_i^{i_2} \cap \{i_1 < i_2\} = \phi$

A loop may be safely parallelized if there are no loop-carried true, anti or output dependences. The array privatization test is applied only to the variables that are involved in dependences to determine if privatization will eliminate these dependences.

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

5.5 Generating Executable Code With Array Privatization

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 array liveness on exit, so we limit privatization to those cases where every iteration in the loop writes to exactly the same region of data. To do so the analysis performs the following test to finalize a loop whose index i has upper bound ub: If $W = M|_i^{ub}$, the last loop iteration is peeled, and this final iteration writes to the original array. Earlier iterations write to a private copy of the array. No peeling is necessary if the compiler can guarantee that the last processor executes the last iteration. Then the compiler can generate code which simply writes to private copies in all the processors except the last one.

6 Array Reduction Recognition

A reduction occurs when a location is updated on each loop iteration with the result of a commutative and associative operation applied to its previous contents and some data value. A loop containing a reduction may be safely parallelized since the ordering of the commutative updates need not be preserved.

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, which is integrated with the array analysis described in the previous section, finds reductions involving general commutative updates to array elements, possibly spanning multiple procedures.

6.1 Reduction Recognition

We currently recognize reductions on scalar variables and array locations involving the operations $+$, $*$, `MIN`, and `MAX`. `MIN` (and, equivalently, `MAX`) reductions of the form `if (a(i) < tmin) tmin = a(i)` are also supported.

The system looks for commutative updates to a single location A of the form $A = A$ `op` ..., where A is either a scalar variable or an array location and `op` is one of the operations listed above. This approach allows any commutative update to a single array location to be recognized as a reduction, even without information about the array indices. We illustrate this point with an example sparse matrix-vector multiply found in the NAS sample benchmark `cgm`:

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

Our system 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.

The reduction recognition analysis first locates commutative updates in a loop body; it verifies that the only other reads and writes in the loop to the same location are also commutative updates of the same type described by `op`. A loop is parallelized if all dependences involve variables whose only accesses are reduction operations of identical type.

In terms of our data-flow analysis algorithm, reduction recognition is initialized by examining the code for commutative updates to the same array location. 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 type described by `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.

6.2 Generating Executable Code With Reductions

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 non-identity 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.

Programs	Loops w/ calls	Parallel (Fida)	Parallel (SUIF)
SPEC89:			
doduc	19	2	7
matrix300	11	0	8
nasa7	8	0	0
tomcatv	0	0	0
PERFECT:			
adm	35	*	4
arc2d	1	0	0
bdna	9	0	1
dyfesm	21	0	6
flo52q	9	7	7
mdg	7	0	2
mg3d	12	0	0
ocean	12	0	0
qcd	40	0	0
spec77	35	*	18
track	18	1	1
trfd	6	0	0
Total	234	10	54

Fig. 2. Static loop count comparison of our system with FIDA.

7 Experience with this System

This system has been used as an experimental platform in an extensive empirical evaluation of the effectiveness of automatic parallelization technology. The full results are presented elsewhere [7], but we present a few highlights in this section.

We have compared the results of our interpocedural analysis with the FIDA system (Full Interprocedural Data-Flow Analysis), an interprocedural system that performs precise flow-insensitive array analysis [10] (see Section 2). The FIDA system was the first to measure how interprocedural analysis on full applications (from the PERFECT and SPEC89 benchmark suites) affects the number of parallel loops that the system can automatically recognize. We compare how many loops containing procedure calls are parallelized using the two systems in Figure 2. The SUIF system is able to locate greater than 5 times more parallel loops than FIDA. This marked difference is due to the additional array analysis techniques employed in our system, and the tight integration with comprehensive interprocedural scalar analysis.

As part of our evaluation, we have measured the importance of the individual techniques employed in this system but not available in current commercial systems. In particular, we have measured how much the advanced array analyses for privatization and reduction recognition and the interprocedural array analysis on advanced array analyses impact the results of parallelization. We have found that these techniques are essential to achieving any speedup on three of the twelve SPEC92FP programs, four of the eight NAS sample benchmarks and two of the thirteen PERFECT benchmarks.

8 Conclusions

This paper has described the analyses in a fully interprocedural automatic parallelization system. This system has been used in an extensive experiment that has demonstrated that interprocedural data-flow analysis, array privatization and reduction recognition are key technologies that greatly improve a parallelizing compiler's ability to locate coarse-grain parallel loops. 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 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 might benefit from some support for run-time parallelization or user interaction. The aggressive static parallelizer we have built will provide a good starting point to investigate these techniques.

Acknowledgements. The authors wish to thank 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

1. J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*. ACM, January 1979.
2. B. Blume, R. Eigenmann, K. Faigin, J. Grout, Jay Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.
3. W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.
4. K. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.
5. B. Creusillet and F. Irigoin. Interprocedural array region analyses. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1995.

6. M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
7. M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, December 1995.
8. W.L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.
9. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
10. M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.
11. F. Irigoin. Interprocedural analyses for programming environments. In *NSF-CNRS Workshop on Evironments and Tools for Parallel Scientific Programming*, September 1992.
12. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
13. J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):159–171, January 1976.
14. W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 27(7), pages 235–248, July 1992.
15. Z. Li and P. Yew. 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, CT, July 1988.
16. E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth Annual Symposium on Principles of Programming Languages.* ACM, January 1981.
17. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice Hall Inc, 1981.
18. O. Shivers. *Control-Flow Analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, May 1991.
19. J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, April 1991.
20. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pages 176–185. ACM, July 1986.
21. P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.