1

The ParaScope Parallel Programming Environment

Keith D. Cooper Mary W. Hall Robert T. Hood Ken Kennedy
Kathryn S. McKinley John M. Mellor-Crummey Linda Torczon
Scott K. Warren

September 11, 1992

*This research was supported by the CRPC (Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center), DARPA through ONR grant N0014-91-7-1989, the State of Texas, IBM corporation, and a DARPA/NASA Research Assistantship in Parallel Processing, administered by the Institute for Advanced Computer Studies, University of Maryland. Keith D. Cooper, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, and Linda Torczon are with the Department of Computer Science, Rice University, Houston, TX 77251-1892. Mary W. Hall is with the Center for Integrated Systems, Stanford University, Stanford, CA 94305. Robert T. Hood is with Kubota Pacific Computer, Inc., 2630 Walsh Avenue, Santa Clara, CA 95051-0905. Scott K. Warren is with Rosetta, Inc., 2502 Robinhood, Houston, Texas 77005.

Abstract

The ParaScope parallel programming environment, developed to support scientific programming of shared-memory multiprocessors, includes a collection of tools that use global program analysis to help users develop and debug parallel programs. This paper focuses on ParaScope's compilation system, its parallel program editor, and its parallel debugger. The compilation system extends the traditional single-procedure compiler by providing a mechanism for managing the compilation of complete programs. Thus ParaScope can support both traditional single-procedure optimization and optimization across procedure boundaries. The ParaScope editor brings both compiler analysis and user expertise to bear on program parallelization. It assists the knowledgeable user by displaying and managing analysis and by providing a variety of interactive program transformations that are effective in exposing parallelism. The debugging system detects and reports timing-dependent errors, called data races, in parallel program executions. The system combines static analysis, program instrumentation, and run-time reporting to provide a mechanical system for isolating errors in parallel program executions. Finally, we describe a new project to extend ParaScope to support programming in Fortran D, a machine-independent parallel programming language intended for use with both distributed-memory and shared-memory parallel computers.

1 Introduction

Computation is assuming a key role in the practice of science and engineering. Mathematical models running on today's most powerful computers are replacing physical experiments. Wind tunnels have yielded to aerodynamic simulation. Automotive design relies on computer graphics rather than on clay prototypes. Computer-based models are at the heart of current work in climate modeling, oil recovery, gene sequencing, and bio-remediation.

In large part, this computational revolution is the result of the dramatic increases in available computer power over the last fifteen years. Increased computing power helps in two ways. First, it makes new problems amenable to computer modeling. Second, it allows better solutions to existing problems. For example, two-dimensional models can be replaced with three-dimensional versions. Between these two trends, the demand for ever faster computers to support scientific applications will continue unabated into the next century.

The technology used to fabricate the processing elements used in high-performance machines is approaching physical device limits. To get significant improvements in computing speed, system designers are turning to parallelism. Already, the most powerful scientific computers, like the Thinking Machines CM-5 and the Intel Paragon XP/S, rely on aggregating together many hundreds or thousands of processors. At the same time, parallelism is appearing on the desk-top. Several companies have introduced multiprocessor UNIX workstations; more will appear. Within several years, we expect to see small-scale multiprocessors on a single chip, perhaps four processors and a shared cache.

Parallel machines have been around for more than two decades. In the last ten years, commercial multiprocessors have become common. Yet, surprisingly few applications have been implemented on these machines. Why? Because parallel machines are difficult to program. As with a classical sequential computer, the programmer must specify precisely how to perform the task. Additionally, the programmer must consider the impact of concurrent execution on the program's correctness. Finally, the efficiency of the resulting program depends on many complex and machine-specific facts.

If parallel machines are to be successful, we must find better ways to program them. Ideally, no additional programming would be needed—the compiler would automatically translate programs written in a conventional language like Fortran 77 so that they execute efficiently on a parallel machine. In pursuit of this goal, several research projects and prototypes have been built to investigate *automatic parallelization*. These include Parafrase at the University of Illinois [1], PFC at Rice University [2], PTRAN at IBM Research [3] and SUIF at Stanford University [4]. In addition, several commercial systems, such as Vast by Pacific Sierra, KAP by Kuck and Associates, and compilers by Alliant, Convex, and Cray, perform automatic parallelization. While these systems achieve impressive results for some programs, they fail to find parallelism that exists in others.

The parallelizing compiler fails because it must take a conservative approach. If, for some loop nest, the compiler cannot prove that parallel execution produces the same results as sequential execution, it must generate the sequential version. Often, the compiler is unable to prove the safety of parallel execution because it cannot determine the values of variables used in subscript expressions. For example, the variables might be passed in as parameters or read in as data. Frequently, the programmer knows the values of these variables, either exactly or approximately. Thus, the compiler often fails to find parallelism in loops that the programmer knows to be parallelizable. For this reason, it seems unlikely that automatic techniques by themselves will provide a comprehensive solution to the parallelization problem. It will be necessary for programmers to be involved in the specification of parallelism.

Given the need for programmer involvement, we need a language for explicit specification of parallelism. Usually, these take the form of extensions to standard Fortran. For shared-memory parallel computers, a typical extension permits the programmer to specify one or more loops that should execute in parallel without synchronization between iterations. Parallel Computer Forum (PCF) Fortran is an example of this kind of extension [5]. It lets the programmer asserts that a loop is to run in parallel, even if the compiler's analysis does not support this decision.

Unfortunately, programmer-specified parallelism can be incorrect. The programmer can specify parallel execution for a loop that does not have independent iterations. For example, different iterations of the loop can share a variable, with at least one iteration writing to the variable. The result of executing this loop depends on the run-time scheduling of the iterations. With one schedule, it may get the same results as sequential execution; another schedule might produce different values. Such schedule-dependent problems, or *data races*, are extremely difficult to find. This phenomenon makes debugging parallel programs hard.

Because it is easy to make mistakes in parallel programs and hard to find them, programmers will need substantial help to write and maintain correct parallel programs. One way to provide this help is to make the deep program analysis of parallelizing compilers available to the programmer in an understandable form. With such information, the programmer can concentrate on those program areas that the compiler left sequential but the programmer knows are parallel. The tool can provide precise information about the facts that prevented it from generating parallel code. Automatic vectorization systems rely on this idea. They provide the user with comments about which loops can not be vectorized and suggestions on how to overcome the problems. This frees users from analyzing every detail and allows them to focus on the specific problems beyond the compiler's abilities.

Taken together, these forces are creating a marketplace for parallel applications and the tools to create them. The ParaScope programming environment is designed to present the results of deep compiler analysis to the programmer. It provides three kinds of assistance, It helps determine whether a parallelization strategy preserves the meaning of a program. It helps correctly carry out a parallelization strategy. It helps find

errors introduced during parallelization. These functions are embedded in the three major components of the ParaScope environment.

(1) The *ParaScope compilation system* is designed to permit compiler analysis of whole programs rather than single modules. It provides more precise information about program variables and carries out cross-procedural optimizations without sacrificing all of the benefits of separate compilation.

(2) The *ParaScope editor* is an intelligent Fortran editor designed to help the user interactively parallelize a program. It presents the user with the results of sophisticated compiler analysis and offers a variety of useful correctness-preserving transformations. Since it permits the user to consider alternative parallelization strategies and override the results of analysis, it can be thought of as a tool for exploratory parallel programming.

(3) The *ParaScope debugging system* is a complete source-level debugger. It includes a mechanical method for locating all data races in a program with respect to a given data set. It can be used for programs written in a useful subset of PCF Fortran.

Although several research and production systems provide some features found in the ParaScope environment [6, 7, 8], ParaScope is unique for its integration of these functions into a unified system.

The next three sections of this paper provide an overview of the issues, problems, and methods that have arisen during the implementation of the three ParaScope components described above. The final section describes several directions for further work, much of it aimed at providing adequate support for Fortran D, a language for machine-independent parallel programming.

2 The ParaScope Compilation System

The overall goal of the ParaScope project is to support the development and compilation of parallel programs. The system provides an environment where each tool has access to the results of sophisticated program analysis and transformation techniques. In particular, the tools have been designed for efficient analysis and optimization of whole programs.

Traditionally, compiler-based analyzers have limited their attention to single procedures. This kind of analysis, called *global* or *intraprocedural* analysis, yields substantial improvement on scalar machines. As the architecture of high-performance machines has shifted toward large-scale parallelism, the compiler's task has become more complex. To mitigate the overhead of startup and synchronization of processes on modern parallel machines, the compiler must find large program regions that can be run in parallel. Striving for large granularity parallelism has a natural consequence — the compiler must look for parallelism in regions of the program that span multiple procedures. This kind of analysis is called *whole program* or *interprocedural* analysis.

2.1 Dependence Analysis

From a compiler's perspective, the key to understanding the behavior of parallel programs lies in understanding the patterns of reference to data. For this reason, the compiler performs *dependence analysis* to locate the potential *data and control dependences* within a program [9, 10, 11, 12]. A *data dependence* exists between two memory references if they can access the same location. In the following code fragment, dependence analysis would show that the value stored into `a` on iteration i is used in both iteration i and iteration $i + 2$.

```
do i = lb to ub by 1
   a[i] = ···
      ···
   ··· = a[i] + a[i-2]
end
```

For scalar variable references, the results of data-dependence analysis are equivalent to those produced by classical data-flow analysis techniques. For subscripted variable references, dependence analysis provides much sharper information.

Control dependences provide a concrete representation for the relationship between control flow tests in a program and the execution of individual statements. For example, in the following code fragment, statement *S1* is control dependent on the `if` statement, while *S2* is control dependent only on the loop header.

```
      do i = lb to ub by 1
         if b[i] ≠ 0 then
S1:         c[i] = c[i] / b[i]
S2:      c[i] = 0.5 * c[i] + c[i+1]
      end
```

Control dependences are a compact and precise way of representing the relationship between the values of control-flow expressions and the execution of individual statements.

To provide a concrete representation, compilers build a *dependence graph*. Nodes in the graph represent individual references in the code. Edges correspond to individual dependences. Edges are directed. Thus, a value produced at the *source* is used at its *sink*.

Dependences play a critical role in the construction, compilation, and debugging of parallel programs. The dependences in a program define a partial order on the execution of its statements; a correct execution of the parallel program must preserve that order. If the endpoints of a dependence appear in concurrently executing threads, then the corresponding memory accesses may occur out of order, resulting in incorrect program execution. Compilers use control and data dependences to prove the safety of transformations that change the control-flow in a program.

To perform dependence analysis, pairs of references are tested to determine if they can access the same memory location. ParaScope's dependence analysis applies a hierarchy of tests on each reference [13]. It

starts with inexpensive tests that can analyze the most commonly occurring subscript references. More expensive tests are applied when the simpler tests fail to prove or disprove dependence.

Automatic parallelizers try to transform the program so that the iterations of a loop execute in parallel. Loop-level parallelism is attractive for two reasons. First, in many loops, the amount of work per iteration is approximately fixed; this characteristic produces a roughly balanced workload inside a parallel region. Second, loop-based parallelism is often *scalable*; that is, as the number of processors increases we can increase the program's problem size proportionately and expect roughly the same processor utilization.

The compiler uses the computed dependence information to determine if a loop's iterations can safely execute in parallel. A dependence is *loop-carried* if its endpoints lie in different iterations of the loop. Loop-carried dependences inhibit parallelization of the loop. Dependences that are not loop-carried are termed *loop-independent*; such dependences would exist without the surrounding loop nest. In the first example, the dependence between `a[i]` and `a[i-2]` is loop-carried, while the dependence between the two references to `a[i]` is loop-independent.

Dependence analysis was originally developed to help compilers automatically discover opportunities for vectorization. These early dependence-based compilers were limited to intraprocedural analysis. For vectorization, intraprocedural techniques were usually sufficient; good vectorizing compilers produce substantial improvements by discovering a single parallel loop and making it the innermost loop in a loop nest.

Unfortunately, profitable parallelism is harder to find. To offset the increased startup and synchronization costs of parallel execution requires larger and deeper loop nests that contain more computation. This complicates program analysis in several ways.

1. The compiler may need to combine the bodies of several procedures to get the critical size required to amortize overhead.
2. The likelihood that one or more loops in a loop nest contain a procedure call rises with the size of the region being considered.
3. As the region grows larger, so do the chances for discovering one or more dependences that would rule out parallel execution.

The answer to each of these problems is the same – the compiler needs analytical techniques that enable it to reason about the behavior of regions larger than a single procedure. A combination of whole program analysis and transformation can supply additional information and context about the program that allows the dependence analyzer to compute sharper information. For example, loop bounds are often passed as parameters, particularly in libraries of numerical code. Knowing the constant value of the loop bound may lead to more precise dependence information.

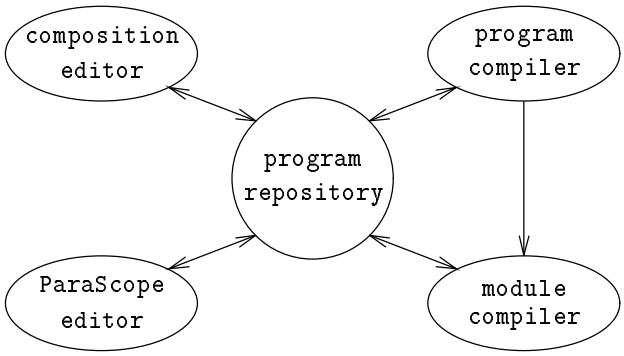

Figure 1 The ParaScope compilation system.

2.2 The Program Compiler

The design of the ParaScope compilation system has been shaped by the decision to perform whole program analysis and optimization. The compiler has been divided into two distinct components: the program compiler and the module compiler. The program compiler deals with issues that are interprocedural in their scope; in contrast, the module compiler handles the detailed job of tailoring individual procedures to the target machine. Through ParaScope's central repository, the program compiler has access to a description of the entire program being compiled, to the abstract syntax tree (AST) for any of its constituent procedures, and to annotations associated with either the program or its procedures. Example annotations are the procedure's symbol table, the set of call sites that occur within a procedure, and a representation of the loop nesting structure and its relationship to call sites. More complex annotations include a data-dependence graph for each procedure and information about interprocedural constants.

Figure 1 shows the relationships between the various components of the ParaScope compilation system. Cooperation between the tools, coordinated through a central repository where the tools store derived information, leads to a more efficient implementation of interprocedural techniques [14]. The ParaScope editor derives information about individual procedures. The composition editor lets the user specify the individual procedures to be included in a program; it produces information that describes the set of possible procedure invocations. Using this information as input, the program compiler

1. computes interprocedural data-flow information,
2. decides where to perform interprocedural optimizations (see Section 2.4), and
3. determines which procedures must be recompiled in response to an editing change.

It employs two rules in the recompilation analysis:

1. If some procedure p has been changed since its last compilation, it will be recompiled.

2. If the interprocedural environment in which p was last compiled has changed in a way that could invalidate an optimization performed, p will be recompiled.

Finally, the module compiler is invoked to produce optimized object code for modules that must be recompiled.

The analysis performed by the program compiler produces information that relates the entire program to each of its component procedures. Thus, the code and some of the annotations are a function of the program that provided the context for the compilation. In the database, program-specific objects must be associated with the program. Only objects and annotations that are independent of any context program are associated with the procedure.

Building on this framework, we plan to build a prototype source-to-source parallelizer for shared-memory multiprocessors [15]. Using the information produced by the program compiler and a performance estimator, it will apply a combination of interprocedural transformations and parallelism-enhancing transformations to provide an initial parallel program [16, 17, 18].

The next two subsections describe how the program compiler computes interprocedural data-flow information and how it applies interprocedural transformations. Recompilation analysis is handled using methods described by Burke and Torczon [19].

2.3 Interprocedural Analysis

Interprocedural data-flow analysis addresses several distinct problems. These include discovering the program's dynamic structure (*call graph construction*), summarizing side-effects of executing a procedure call (*summary analysis*, *regular section side-effect analysis*, and *kill analysis*), understanding the interplay between call-by-reference parameter passing and the mapping of names to memory locations (*alias analysis*), and discovering when a variable has a value that can be derived at compile time (*constant propagation* and *symbolic value analysis*). This section presents a high-level overview of the methods used in ParaScope to address these problems.

Call graph construction. The fundamental structure used in interprocedural data-flow analysis is the program's static call graph. Its nodes represent procedures; it has an edge from x to y for every call site in procedure x that invokes procedure y. In the absence of procedure-valued variables, the call graph can be built in a single pass over the program. However, most programming languages include some form of procedure-valued variables; this feature complicates the construction process.

The program compiler uses an algorithm due to Hall and Kennedy to construct an approximate call graph for languages with procedure-valued parameters [20]. For such languages, the procedure variables may only receive their values through the parameter passing mechanism. Building the call graph involves locating the possible bindings for the procedure parameters, a problem similar to constant propagation (see below).

To provide information about loop nests that span procedure calls, it augments the call graph with nodes and edges that represent loops in a program. If x calls y from within some loop l, there will be an edge from the node for x to a node for l and from there to the node for y.

Summary analysis. The program compiler annotates each edge in the call graph with *summary sets* that describe the possible side effects of executing that call. In particular, each edge e has two associated sets:

1. MOD(e) contains those variables that might be modified if the call is taken, and
2. REF(e) contains those variables that might be referenced if the call is taken.

ParaScope computes *flow-insensitive* summary information; that is, the MOD set computed for a call contains all variables modified along some path from the call. (In contrast, a *flow-sensitive* version of MOD would contain only those variables modified along every path from the call.) The program compiler derives MOD and USE sets by solving a set of simultaneous equations posed over the call graph or one of its derivatives [21].

Regular section side-effect analysis. Classical summary analysis techniques treat array elements naively — they deal only with the entire array. For example, `a` $\in$ MOD(p) asserts that one or more elements of the array `a` may be modified by a call to p. It might be that every element of `a` is always assigned by p. Alternatively, p might assign only to `a[1,1]`. Classical side-effect analysis does not distinguish between these cases. Because of this lack of precision, classical MOD and REF sets produce only limited improvement in the precision of dependence information for array subscripts.

To provide better information about side effects to arrays, the program compiler can compute a more complex form of summary information known as regular sections [22]. The basic idea is to replace the single-bit representation of array side effects with a richer lattice that includes the representation of entire subregions of arrays. A regular section is simply a subregion that has an exact representation in the given lattice.

The following example illustrates this idea. Figure 2 displays a lattice of reference patterns for an array `a`. Here, the regular sections are single elements, whole rows, whole columns, and the entire array. Note that `i`, `j`, and `k` are arbitrary symbolic input parameters to the call. Thus, for a procedure that modified column i of `a`, the regular section MOD set would contain a descriptor `a[*,i]`.

We are implementing regular section side-effect analysis in the program compiler. It has been implemented in the PFC system since 1989; our experience with that implementation shows that the analysis is both efficient and effective [23].

Kill analysis. An important part of determining whether a loop can be run in parallel is the identification of variables that can be made private to the loop body. In the following example, the index variable `i` can be made private in the `do` loop, while `a` must be shared.

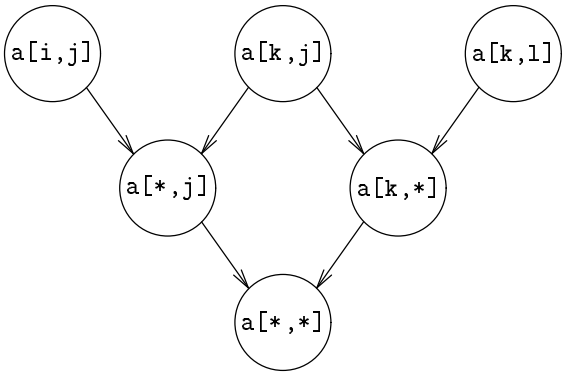

Figure 2 A simple regular section side-effect lattice.

```
do i=1,n
   a[i]=f(a[i]-1)
end
```

Modifications to private variables during one iteration of the loop do not affect other iterations of the loop. For this reason, dependences involving private variables do not inhibit parallelism.

To locate private variables, the compiler can perform KILL analysis. We say that a value is *killed* if it is redefined along *every* path that leads to a subsequent reference. For scalar variables, intraprocedural KILL analysis is well understood. The interprocedural problem is intractable in general [24]; methods that compute approximate solutions have been proposed and implemented [25].

As with summary analysis, computing KILL information is more complex for arrays than for scalar variables [26, 27, 28]. In practice, some simple cases arise in loops. These are easier to detect than the general case. For example, many initialization loops have the property that they define every element of some array `a`, and no element is referenced before its definition. In this case, the compiler can conclude that the loop kills `a`. If the loop contains a procedure call, interprocedural propagation of KILL information may be necessary.

Experience suggests that KILL information on arrays, both interprocedural and intraprocedural, is important in parallelizing existing applications [29, 30]. We plan to explore methods for approximating KILL sets for both arrays and scalars. The program compiler will be our implementation vehicle.

Alias analysis. To give the module compiler more information about the run-time environment in which a procedure will execute, the program compiler annotates each node in the call graph with an ALIAS set [31]. For a procedure p, ALIAS(p) contains pairs of names. Each pair (x, y), where x and y are either formal parameters or global variables, is an assertion that x and y may occupy the same storage location on some

invocation of p. As with the summary sets, the program compiler computes a conservative approximation to the actual set of aliases that might hold at run-time. Because it uses a formulation of the problem that ignores control flow within a procedure, the analysis may include some alias pairs that do not really occur at run-time.

The ALIAS computation is formulated as a set of simultaneous equations posed over the call graph and its derivatives [32].

Constant propagation. Interprocedural constant propagation tags each procedure p in the call graph with a set CONSTANTS that contains (*name*, *value*) pairs. Each pair asserts that *name* contains *value* on entry to the procedure, along all paths leading to an invocation of p. Additionally, it tags each edge e with a CONSTANTS set that contains (*name*, *value*) pairs. These pairs represent variables that are known to be set to a constant value as a result of executing the call.

The algorithm used by the program compiler to derive CONSTANTS works by iterating over the program's call graph. To model the way that values pass through procedures, it uses a set of *jump functions*. These functions describe the relationship between the values of parameters that p passes out at a call site and the values that p itself inherits from the routines that call it. Similarly, *reverse jump functions* are constructed for each variable visible in the caller whose value the callee sets. Using the jump functions for each procedure to model the transmission of values along edges in the call graph, the algorithm efficiently constructs a conservative CONSTANTS set for each node and each edge. The complexity of the jump function implementation affects the type of constants actually found by the algorithm [33].

Symbolic value analysis. Experience with dependence analyzers embedded in both research and commercial compilers has shown the value of symbolic analysis as a way of improving the precision of dependence analysis. In practice, the analyzer can often prove independence for two array references without knowing the value of some variables that arise in the dependence test. Cases that arise include expressions like `(y - y)/x` where the system knows a value for `x` but not `y`. Here, the value of the expression is independent of `y`'s specific value.

ParaScope contains an analyzer that performs intraprocedural symbolic analysis. An analyzer that performs interprocedural symbolic analysis is nearing completion. The analysis extends traditional interprocedural constant propagation to include propagation of symbolic values around the call graph. It uses *gated static single assignment graphs* to find values that are provably equal — a notion similar to the classic *value numbering* technique [34, 35].

When symbolic analysis is complete, the program compiler will discover variables that have consistent symbolic values. This happens during the interprocedural analysis phase; the results are recorded with the

program in the repository. When the dependence analyzer is invoked to examine a procedure, it will check the database for symbolic information and use that information as a natural part of its tests.

2.4 Interprocedural Transformations

Future versions of the program compiler will automatically employ both *interprocedural code motion* and *procedure cloning* to improve the program. Interprocedural code motion is a general term used to describe transformations that move code across a procedure boundary. Names are translated to model the effects of parameter binding and to merge the name spaces. *Inline substitution* is a simple form of interprocedural code motion [36]; it replaces a procedure call with a copy of the code for the called procedure. *Loop extraction* and *loop embedding* are two other forms of interprocedural code motion that the program compiler will use [16]. Loop extraction pulls an outermost enclosing loop from a procedure body into a calling procedure; it is a form of partial inlining. Loop embedding is the inverse operation; it pushes a loop surrounding a call site into the called procedure. Procedure cloning lets the compiler produce multiple versions of a single procedure [37, 38]. Each call site of the procedure is assigned to a specific version of the procedure; that version can be tailored to the calling environment. Call sites with similar calling environments can share a single version.

The program compiler will use these transformations to create additional opportunities for profitable parallel execution. In particular, it will target loop nests that either span procedures or are hidden in a called procedure. In keeping with ParaScope's focus on transforming loop nests, it will try to create more loops with sufficient granularity and the right dependence structure for profitable parallel execution. This can be accomplished as follows:

1. *increasing granularity* – granularity can be increased by optimizations such as loop interchange and loop fusion (see Section 3.1.4). Unfortunately, it is problematic to apply these transformations when the loops lie in different procedures. Thus, the program compiler uses interprocedural code motion to relocate these loops.
2. *improving dependence information* – experience shows that interprocedural information can improve the precision of dependence analysis for loops that contain procedure calls [23, 39, 40, 41, 42]. Procedure cloning can improve the precision of some kinds of interprocedural data-flow information, particularly interprocedural CONSTANTS sets [37].

Interprocedural code motion and procedure cloning can expose additional parallelism and improve the run-time behavior of the program [16]. Unfortunately, these interprocedural techniques can have negative side effects.

- A study by Cooper, Hall, and Torczon using commercial Fortran compilers showed that inline substitution often resulted in slower running code [43, 44]. When this happened, secondary effects in the optimizer outweighed any improvements from inlining. Inline substitution introduces recompilation dependences and increases average procedure size. It can increase the overall source code size and compilation time.

- Loop embedding and extraction are limited forms of inlining designed to avoid some of the secondary effects cited above. However, these transformations still introduce recompilation dependences between the involved procedures. Furthermore, procedure cloning is often required to create the right conditions for their application.
- Procedure cloning does not move code across a call site, avoiding some of the secondary effects of inline substitution. However, it still introduces recompilation dependences, increases source text size, and increases overall compile time.

Despite these negative effects, the program compiler should include these techniques in its repertoire because they can improve code that defies other techniques. In applying them, the program compiler should consider both the positive and negative effects. It should only apply interprocedural transformations when the potential profits outweigh the potential problems [45]. To implement this notion, the program compiler should:

1. Locate focus points in the program. A focus point must have two properties:
   (a) a target intraprocedural optimization would be profitable, if only it could be applied, and
   (b) supplying more context from other procedures would make application possible.
2. Use a combination of interprocedural data-flow information, interprocedural code motion, and procedure cloning to transform the focus point in a way that makes it possible to apply the target optimization.
3. Apply the target optimization.

Thus, the interprocedural transformations will be used exclusively to enable application of profitable target optimizations — in this case, parallelizing transformations. We call this strategy *goal-directed interprocedural optimization* [45]. The effectiveness of this strategy in the context of parallelizing transformations was tested in an experiment with the PERFECT benchmark suite. A goal-directed approach to parallel code generation was employed to introduce parallelism and increase granularity of parallel loops [16]. Because the approach proved to be effective, we plan to use this strategy in the planned prototype source-to-source parallelizer mentioned in Section 2.2.

2.5 Application of Interprocedural Techniques

Several of the techniques described are also implemented in the Convex Applications Compiler™. It performs regular section analysis and constant propagation, as well as the classical summary and alias analysis. It clones procedures to improve the precision of its CONSTANTS sets, and performs limited inline substitution. Metzger and Smith report average improvements of twenty percent, with occasional speedups by a factor of four or more [41].

3 ParaScope Editor

Upon completion, the compilation system described in Section 2 will be capable of automatically converting a sequential Fortran program into a parallel version. A substantial amount of work has been conducted on

the viability of this approach by researchers at Rice and elsewhere [1, 2, 3, 15, 46]. Ideally, automatically parallelized programs will execute efficiently on the target architecture and users will not need to intervene. Although such systems can effectively parallelize many interesting programs, they have not established an acceptable level of success. Consequently, advanced programming tools such as the ParaScope editor are required to assist programmers in developing parallel programs.

At the core of automatic parallelization is dependence analysis (see Section 2.1), which identifies a conservative set of potential data race conditions in the code that make parallelization illegal. In general, the compiler cannot parallelize a loop or make transformations if doing so would violate a dependence and potentially change the sequential semantics of a program. Dependence analysis is necessarily conservative – it reports a dependence if there is a possibility that one exists, *i.e.* is *feasible*. In some cases, it is quite obvious to a programmer that a reported dependence is *infeasible* and will not ever occur. In a completely automatic tool, the user is never given an opportunity to make this determination. Automatic parallelizing systems must also make difficult decisions about how and where to introduce parallelism; evaluating performance tradeoffs can be highly dependent on run-time information such as symbolic loop upper bounds. Again, programmers might be able to provide such information if only there were a mechanism to communicate it to the compiler.

The ParaScope editor, PED, serves this purpose by enabling interaction between the compiler and the programmer. The compilation system, described in Section 2, computes a wealth of information about the program. The sheer volume of information dominates the program size; there is more information than the programmer can reasonably study. PED acts as a sophisticated filter; it provides a meaningful display of the program dependences calculated by the compiler. Furthermore, the tool assists the programmer with parallelization and the application of complex code transformations. The programmer refines conservative assumptions by determining which dependences are valid and then selecting transformations to be applied. The editor updates dependence information and source code to reflect any such programmer actions. The compiler may then use these modifications when generating code.

Research on PED began in the late 1980s. Its interface, design and implementation have been discussed in detail in other papers [47, 30, 48, 49]. This section describes a new version of the ParaScope editor from the user's point of view and discusses new research directions for PED. We sketch also the design issues involved in constructing such a tool and relate how it integrates with the compilation system.

3.1 Ped Functionality

PED supports a paradigm called *exploratory parallel programming*, in which the user converts an existing sequential Fortran program into an efficient parallel one by repeatedly finding opportunities for parallelism and exploiting them via appropriate changes to the program [48, 49]. This process involves a collaboration

in which the system performs deep analysis of the program, the user interprets analysis results and the system helps the user carry out changes. Specifically, the user selects a particular loop to parallelize and PED provides a list of dependences that may prevent direct parallelization. The user may override PED's conservative dependence analysis. Otherwise, the user may consider applying a program transformation to satisfy dependences or introduce partial parallelism of the loop. On request, PED gives the user advice and information on the application of transformations. If the user directs, PED performs transformations automatically and updates the dependence information.

PED uses three user interface techniques to manage and present the voluminous and detailed information computed by the program compiler: (1) a book metaphor with *progressive disclosure* of details, (2) user-controlled *view filtering*, and (3) *power steering* for complex actions. The book metaphor portrays a Fortran program as an *electronic book* in which analysis results are treated as annotations of the main source text analogous to a book's marginal notes, footnotes, appendices, and indices [50]. Progressive disclosure presents details incrementally as they become relevant rather than overwhelming the user with all details at once [51]. View filtering acts as an electronic highlighter to emphasize or conceal parts of the book as specified by the user [52]. The user can choose predefined filters or define new ones with boolean expressions over predefined predicates. Power steering automates repetitive or error-prone tasks, providing unobtrusive assistance while leaving the user in control.

The layout of a PED window is shown in Figure 3. The large area at the top is the *source pane* displaying the main Fortran text. Beneath it are two footnotes displaying extra information, the *dependence pane* and the *variable pane.* All three panes operate in basically the same way. Scroll bars can be used to bring other portions of the display into view, predefined or user-defined view filters may be applied to customize the display, the mouse may be used to make a selection, and menu choices or keyboard input may be used to edit the displayed information. The Fortran text, the dependence information, and the variable information are each described below in terms of the information displayed, the view filtering, and the type of edits that may be performed. Section 3.1.4 describes PED's *transformations,* which provide power steering for complex source edits commonly encountered in parallelization. Section 3.1.5 presents PED's approach to updating dependence information.

3.1.1 Fortran Source Code

Fortran source code is displayed in pretty-printed form by the source pane. At the left are marginal annotations showing the ordinal number of each source line, the start of each loop, and the extent of the *current* loop (see Section 3.1.2). In the program text, placeholders used in structure editing are shown in italics and syntax errors are indicated with boldface messages. Manually controlled ellipsis enables the user to conceal and later reveal any range of source lines.

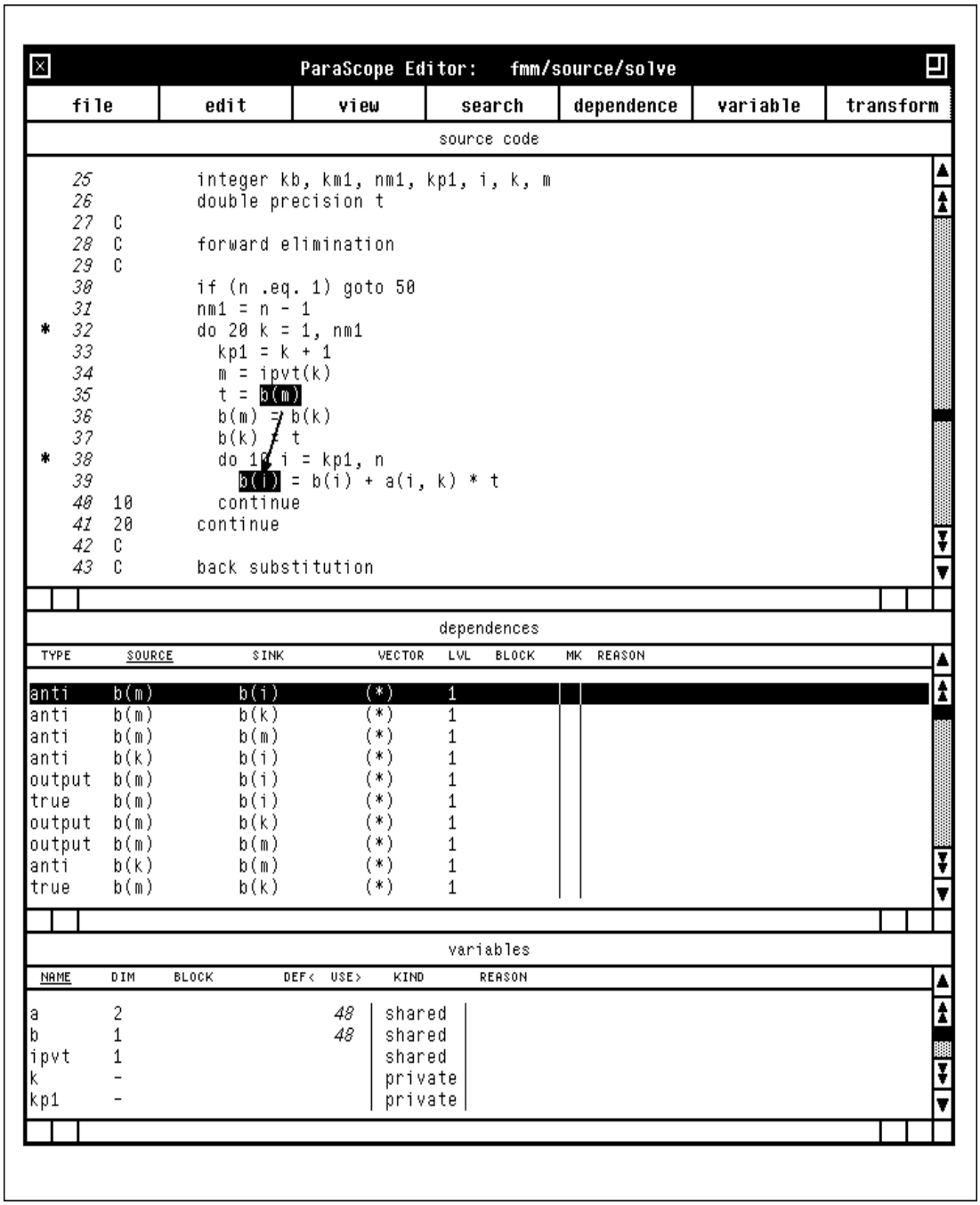

Figure 3 The ParaScope editor.

Source view filtering can be based on either the text or the underlying semantics of each source line. For example, source view filter predicates can test whether a line contains certain text, whether the line contains a syntax or semantic error, whether the line is a specification statement, an executable statement, a subprogram header, or a loop header.

The source pane allows arbitrary editing of the Fortran program using mixed text and structure editing techniques [53]. The user has the full freedom to edit character by character, or at any time to use the power steering afforded by template-based structure editing. During editing, PED directly constructs an abstract syntax tree (AST) representation of the procedure; this internal form for procedures is used throughout the environment. The AST representation of source means that the code is always parsed. This feature has several important benefits. Analysis operates on the parsed trees, which maintain all appropriate semantic information needed by the analysis algorithms. Similarly, program transformations are straightforward manipulations of the tree. Incremental parsing occurs in response to editing changes, and the user is informed of any syntactic or semantic errors.

3.1.2 Dependence Information

The extensive compiler analysis described in Section 2 produces program dependence information that is revealed in the dependence pane. The display contains a list of dependences in tabular form, showing each dependence's source and sink variable references as well as some characterizations of the dependence useful for program transformations. Because Fortran programs may contain an unwieldy number of dependences, PED uses progressive disclosure of dependences based on a *current loop*. When the user expresses interest in a particular loop by clicking on its loop marker in the source pane, that loop's dependences immediately appear in the dependence pane. In addition to the dependence pane's tabular view, the source pane offers a graphical view of dependences. Each dependence is shown as a red arrow from its source variable reference to its sink variable reference, as illustrated in Figure 3. Showing arrows for all of a loop's dependences can produce an unintelligible jumble of red, so PED provides a second level of progressive disclosure. For example, the user can choose whether to see arrows for all listed dependences, for only the dependences affecting a selected range of source code, or for just one dependence selected in the dependence pane. In Figure 3, the user has selected a single dependence which is reflected in the source and dependence panes.

Dependence view filter predicates can test the attributes of a dependence, such as its type, source and sink variable references, loop nesting level, and its source and sink line numbers in the Fortran text. In addition, predicates can test the user-controlled *mark* and *reason* attributes described in the next paragraph. By filtering the dependence list, the user may examine specific dependence relationships more clearly.

Naturally the dependence pane forbids modification of a dependence's computed attributes such as type, source, and sink, because these are implied by the Fortran source code. But the dependence pane does permit

an important form of editing known as *dependence marking*. Because dependence analysis is conservative, the list of dependences may include ones that are infeasible or that do not affect the program's meaning. Users may sharpen PED's dependence analysis by classifying each potential dependence as *accepted* or *rejected*. Each dependence starts out as *proven* or *pending*.

If PED proves a dependence exists with exact dependence tests [13], the dependence is marked as *proven*; otherwise it is marked *pending*. When users mark dependences as rejected, they are asserting that the dependences are not feasible and that the system should ignore them. Therefore, rejected dependences are disregarded when PED considers the safety of a parallelizing transformation. However, rejected dependences are not actually deleted because the user may wish to reconsider these at a later time. Instead, the classification of each dependence is stored in a special *mark* attribute which the user can edit. If desired, the user can use the *reason* attribute to attach a comment explaining the classification decision. A *Mark Dependences* dialog box provides power steering for dependence marking by allowing the user to classify in one step an entire set of dependences that satisfy a chosen filter predicate.

3.1.3 Variable Information

In Section 2.3, we highlighted the importance of locating variables that can be made private to the loop body. Briefly, private variables do not inhibit parallelism. The program compiler locates many of these for scalar variables using local KILL analysis. However, this classification is conservative and may report a variable as shared even if it may legally be made private. The variable pane in PED enables users to peruse the results of this analysis and to correct overly conservative variable classifications for array and scalar variables.

The variable pane displays in tabular form a list of variables in the current loop, showing each variable's name, dimension, common block if any, and shared or private status. As with dependences, progressive disclosure is based on the current loop, filter predicates can test any of the attributes of a variable, and modification of a variable's computed attributes is forbidden. However, the variable pane also provides an important form of editing known as *variable classification*.

Variable classification enables the user to edit a variable's *shared/private* attribute and attach a comment in its *reason* attribute. A *Classify Variables* dialog box (analogous to *Mark Dependences*) provides power steering for variable classification. In this way, the user may sharpen variable analysis by asserting a variable is private when appropriate. This reclassification eliminates dependences from a loop and may enable parallelization.

3.1.4 Transformations

PED provides a number of interactive transformations that can be applied to programs to enhance or expose parallelism. Transformations are applied according to the power steering paradigm: the user specifies the

transformations to be made, and the system provides advice and carries out the mechanical details. The system advises whether the transformation is *applicable* (*i.e.*, make semantic sense), *safe* (*i.e.*, preserves the semantics of the program) and *profitable* (*i.e.*, contributes to parallelization). The complexity of these transformations makes their correct application difficult and tedious. Thus, power steering provides safe, correct and profitable application of transformations. After each transformation, the dependence graph is directly updated to reflect the effects of the transformation.

PED supports a large set of transformations that have proven useful for introducing, discovering, and exploiting parallelism. PED also supports transformations for enhancing the use of the memory hierarchy. These transformations are described in detail in the literature [36, 54, 55, 56, 49, 57, 58, 59]. Figure 4 shows a taxonomy of the transformations supported in PED.

Reordering transformations change the order in which statements are executed, either within or across loop iterations. They are safe if all dependences in the original program are preserved. Reordering transformations are used to expose or enhance loop-level parallelism. They are often performed in concert with other transformations to introduce useful parallelism.

Dependence breaking transformations are used to satisfy specific dependences that inhibit parallelism. They may introduce new storage to eliminate storage-related dependences, or convert loop-carried dependences to loop-independent dependences, often enabling the safe application of other transformations. If all the dependences carried by a loop are eliminated, the loop may be executed in parallel.

Figure 4 Transformation taxonomy in PED.

Reordering

Loop Distribution	Loop Interchange
Loop Skewing	Loop Reversal
Statement Interchange	Loop Fusion

Dependence Breaking

Privatization	Scalar Expansion
Array Renaming	Loop Peeling
Loop Splitting	Alignment

Memory Optimizing

Strip Mining	Scalar Replacement
Loop Unrolling	Unroll and Jam

Miscellaneous

Sequential $\leftrightarrow$ Parallel	Loop Bounds Adjusting
Statement Addition	Statement Deletion

Memory optimizing transformations adjust a loop's balance between computation and memory access to make better use of the memory hierarchy and functional pipelines. These transformations have proven to be extremely effective for both scalar and parallel machines.

3.1.5 Incremental Reanalysis

Because PED is an interactive tool, efficient updates to the program representations are essential to the usability of the tool. We have taken a simple approach to reanalysis in PED that is based on the effect and scope of a change. After a change, PED performs reanalysis as follows.

1. If the effect of a change is known, PED directly updates the effected dependence and program representations.
2. If PED can determine the scope of a change, PED applies batch analysis to the effected region.
3. If PED cannot determine the scope of a change, PED applies batch analysis to the entire procedure.

For example, most of the structure program transformations in PED have the advantage that their effects are predictable and well-understood. Thus, after a transformation, PED directly updates the dependence graph and AST. These updates are very efficient and fast in practice. However, they are specific to each transformation.

Updates to the dependence graph after some simple edits, such as addition and deletion of assignment statements, are performed incrementally. In addition, if PED can determine the scope of a change is limited to a loop nest, then it performs batch analysis on just that nest. These strategies are incremental, but only in a very limited sense.

After a complicated edit or large number of edits, reanalysis is difficult because the effects can be extensive. As described in Section 2.1, the precision of dependence analysis relies on sophisticated data-flow information. These updates may be very expensive in an interactive setting and in some cases, incremental analysis may be more expensive than performing batch analysis for the effected procedures [28]. Therefore, after arbitrary program edits, PED relies on efficient batch algorithms rather than on a general incremental algorithm.

Updates following edits must not only recalculate intraprocedural data-flow and dependence information, but must also recompute interprocedural data-flow information. Changes in interprocedural facts due to an edit in one procedure may cause the analysis of other procedures to be invalid. The program compiler discovers this type of inconsistency when it is invoked to analyze and compile a program (see Section 2.2).

3.2 Ongoing Research

The ParaScope editor was originally inspired by a user workshop on PTOOL, a dependence and parallel program browser also developed at Rice [60, 61, 62]. PED has influenced and been influenced by other

parallel programming tools and environments, such as MIMDizer and its predecessor Forge by Pacific Sierra [63, 64], SIGMACS in the FAUST programming environment [65, 6], and SUPERB [8]. In addition, PED has been continually evaluated internally and externally by users, compiler writers, and user interface designers [66, 47, 30, 67].

In the summer of 1991, we held a workshop on ParaScope to obtain more feedback from users and to further refine our tool [30]. Participants from industry, government laboratories, and supercomputing centers used ParaScope to parallelize programs they brought with them. For the most part, these scientific programmers, tool designers, and compiler writers reported that the user interface and functionality were appropriate to the task of parallelization. Their suggestions and experiences led us to target several areas for future development.

In the following three sections, we describe revisions to the ParaScope editor to further assist scientific programmers. We first present a method to assist users with the technology available from an automatic parallelizing system. Second, we discuss an alternative to the dependence deletion mechanism described above. This mechanism improves system response after editing changes. Finally, we describe improvements to system response when interprocedural information changes.

3.2.1 Combining Interactive and Automatic Parallelization

The purpose of interactive tools like PED is to make up for the shortcomings of automatic parallelization. However, the automatic parallelizer mentioned in Section 2.2 will contribute to knowledge about the effectiveness of transformation and parallelization choices that could be useful to the programmer. We plan to combine the machine-dependent knowledge of the parallelizer and deep analysis of the program compiler with the program-specific knowledge of the user.

We propose the following approach for ParaScope [15]. The program is first transformed using the automatic parallelizer. This source-to-source translation produces a machine-specific source version of the program. If the user is satisfied with the resultant program's performance, then the user need not intervene at all. Otherwise, the user attempts to improve parallelization with PED, incorporating information collected during parallelization. The parallelizer will mark the loops in the original version that it was unable to effectively parallelize. Using performance estimation information, it will also indicate to the user the loops in the program where most of the time is spent, indicating to the user where intervention is most important. The user then makes assertions and applies transformations on these loops. In addition, the user may invoke the parallelizer's loop-based algorithms to find the best combinations of transformations to apply, providing interactive access to the parallelizer's decision process.

3.2.2 User Assertions

In PED, the dependence and variable displays assist users in perusing analysis and overriding conservative analysis. For example, when a user reclassifies a variable as private to a loop, subsequent transformations to that loop will ignore any loop-carried dependences. In addition, any dependences that the user classifies as *rejected* are ignored by the transformations. These mechanisms enable users to make assertions about specific dependences and variables. Unfortunately, the current interface is not rich enough. Consider the following example:

```
read *, k
do i = 1, m
   do j = 1, n
      a[i,j] = a[i,k] + 1
      b[i,j] = b[k,j] * a[k,j]
   enddo
enddo
```

The value of `k` is unknown at compile time. Therefore, PED conservatively reports dependences between `a[i,j]` and `a[k,j]` and between `b[i,j]` and `b[k,j]`, reflecting the possibility that `i` could equal `k` on some iteration of the `i` loop. Similarly, dependences between `a[i,j]` and `a[i,k]` are reported, in case `j` could equal `k` on some iteration of the `j` loop. If the user knew that `k` was not in the range of 1 to `m` and also not in the range 1 to `n`, then all dependences could be deleted. There is no convenient way to express this with the existing dependence filtering mechanism. The best approximation is to delete all dependences on `a` and `b`, but the user would still have to examine each dependence to justify that it was infeasible.

Another problem with explicit dependence deletion is that it fails to capture the reasons behind the user's belief that a dependence does not exist. While the user can add a comment accompanying the dependence for future reference, this information is unavailable to the system. After source code editing, it is very difficult to map dependence deletions based on the original source and program analysis to the newly modified and analyzed program. As a result, some dependences may reappear and programmers will be given an opportunity to reconsider them in their new program context.

For these two reasons, we intend to include a mechanism in ParaScope that will enable users to make assertions that can be incorporated into dependence testing. The assertion language will include information about variables, such as value ranges. All the dependences in the code above could be eliminated with the appropriate assertion about the range of the variable `k`. The programmer could determine the correctness and need for this assertion by examining at most two dependences rather than all of them. Furthermore, by making the assertions part of the program, the system can derive the necessary information to recover dependence deletions following edits. The user will be responsible for maintaining the validity of the assertions.

3.2.3 Reanalysis for Global Changes

PED's current incremental analysis is designed to support updates in response to changes during a single editing session. This approach does not address the more difficult issue of updating after global program changes. More precisely, a programmer might parallelize a procedure p using interprocedural information, put p away, and edit other modules. Later, changes to procedures in the program may affect the validity of information used to parallelize p.

Consider some ways in which the user might make decisions based on interprocedural information: the user may (1) designate a loop as parallel; (2) apply transformations to the program; and, (3) add assertions to be used by dependence testing. Below, we describe the consequences of these changes.

Parallel loops. When the user specifies a loop as parallel in PED, it is either because the system reports no dependences are carried by that loop, or because the user believes reported loop-carried dependences to be infeasible. Later, a change elsewhere in the program may cause the system to reanalyze the procedure containing this loop. If this analysis reports no loop-carried dependences for the same loop, the user's previous specification is still correct. However, if new loop-carried dependences have been introduced, the programmer will need to be warned that the change may have resulted in an erroneous parallel loop.

Structure transformations. The correctness of transformations may also depend on interprocedural information. Safe transformations are guaranteed to preserve the semantics of the program. However, edits are not. To determine if an edit that follows a transformation preserves the information that was needed to safely perform the transformation is very difficult. It requires maintaining editing histories and a complex mapping between a procedure and its edited versions.

Rather than addressing this general problem, we will provide a compromise solution. When a transformation is applied, the system will record the interprocedural information used to prove the transformation safe. After reanalysis due to editing, the system can compare the new interprocedural information with that stored previously. The system will warn of any changes to interprocedural information that might have invalidated transformations. The programmer will be responsible for determining the effects of these changes on the meaning of their program.

Assertions and arbitrary edits. The programmer may also rely on interprocedural facts in deriving assertions or in making arbitrary edits. To allow the system to track the information that the programmer used, we could include assumptions about interprocedural information in the assertion language. Then, the system could warn the user when this information changes.

4 Parallel Debugging in ParaScope

If during the development of a shared-memory parallel Fortran program, the programmer directs the compiler to ignore a feasible data dependence carried by a parallel loop or parallel section construct, a *data race* will occur during execution of the program for some input dataset. A data race occurs when two or more logically concurrent threads access the same shared variable and at least one of the accesses is a write[1]. Data races are undesirable since they can cause transient errors in program executions.

Ignoring feasible dependences during program development is a real concern. Since dependence analysis is inherently conservative, often the dependence analyzer will report parallelism inhibiting dependences that can never be realized at run time. Using PED, programmers can examine dependences that limit opportunities for parallel execution and manually reclassify those thought to be infeasible. Since errors in this classification process are evident only at run-time, programmers need run-time support for helping them determine when they have mistakenly classified a dependence as infeasible.

Traditional debugging tools designed to support state-based examination of executing programs are ill-equipped to help locate data races. Since transient behavior caused by data races is only evident in program executions on shared-memory multiprocessors, using tools for state-based examination to detect data races is only appropriate in this context. Unfortunately, inspecting live executions of programs on parallel machines can perturb the relative timing of operations and change the set of execution interleavings that occur. Thus, the act of trying to isolate a data race causing erroneous behavior by halting an executing program and examining its state can make evidence of the race vanish.

A promising approach for pinpointing data races is to instrument programs to monitor the logical concurrency of accesses during execution [68, 69, 70, 71, 72, 73]. With such instrumentation, a program can detect and report data races in its own execution. The ParaScope debugging system uses such a strategy to detect data races at run time and supports automatic instrumentation of programs for this purpose.

Using the ParaScope debugging system to detect data races requires little effort from a programmer. The programmer simply directs the compilation system to add data race instrumentation when it compiles the program. If a data race occurs during any execution of an instrumented program, diagnostic output from the run-time library reports the pair of references involved in the race and the parallel construct carrying the dependence that caused the race. If an instrumented program is run under control of the ParaScope source-level debugger, each race detected is treated as a breakpoint at which the programmer may inspect the state of the program to determine the conditions that caused the race to occur.

For a restricted (but common) class of programs, the ParaScope debugging system guarantees that if any data race can arise during some execution with a particular input data set, then at least one race

[1] We use the term *thread* to refer to an indivisible unit of sequential work.

will be detected for *any* execution using that input. This is an important property since it makes parallel debugging tractable and frees programmers from needing to consider all possible execution interleavings for any particular input.

The support for parallel debugging in ParaScope consists of several components:

- a run-time library of routines that can detect if a variable access is involved in a data race during program execution,
- an instrumentation system that adds calls to the run-time library routines where necessary so that data races will be detected during execution, and
- a graphical user interface that decodes data race reports from the run-time library and presents them in the context of an interactive source-language debugger.

In the following sections, we describe each of these components of in more detail.

4.1 Run-time Detection of Data Races

Run-time techniques for detecting data races fall into two classes: summary methods [74, 72, 73] that report the presence of a data race with incomplete information about the references that caused it, and access history methods [69, 70, 71] that can precisely identify each of a pair of accesses involved in a data race. In ParaScope, we use an access history method for detecting data races at run time since precise race reports are more helpful to programmers.

To pinpoint accesses involved in data races, access history methods maintain a list of the threads that have accessed each shared variable and information that enables determination of whether any two threads are logically concurrent. When a thread t accesses a shared variable V, t determines if any thread in V's access history previously performed an access that conflicts with t's current access, reports a data race for each such thread that is logically concurrent with t, removes from the history list information about any threads that are no longer of interest, and adds a description of t's access to the access history. In the most general case, the length of a variable's access history may grow as large as the maximum logical concurrency in the program execution.

An open question in the development of run-time techniques for detecting data races is whether the time and space overhead they add to program executions may be too great for the techniques to be useful in practice. For programs with nested parallel constructs that use event-based synchronization to coordinate concurrent threads, the worst case time and space overhead is proportional to the product of the number of shared variables and the number of logically concurrent threads. Such high overhead would often be unacceptable to users. For our initial investigation into the feasibility of using run-time techniques for automatically detecting data races, we have chosen to focus our efforts on developing a run-time library

to support a common but restricted class of programs for which data race detection can be accomplished particularly efficiently. Specifically, we limit our focus to Fortran programs that

- use nested parallel loops and sections to explicitly denote parallelism, and
- contain no synchronization other than that implied by the parallel loop and section constructs.

Such programs are known as *fork-join* programs. In the next two sections, we describe a model of concurrency for such programs and the properties of an efficient protocol for detecting data races in their executions at run-time.

4.1.1 Modeling Run-time Concurrency in Fork-join Programs

Upon encountering a parallel loop or section construct, a thread terminates and spawns a set of logically concurrent threads (i.e., one for each loop iteration or section in the parallel construct). We call this operation a *fork*. Each fork operation has a corresponding *join* operation; when all of the threads descended from a fork terminate, the corresponding join succeeds and spawns a single thread. A thread participates in no synchronization operations other than the fork that spawned it and the join that will terminate it.

The structure of concurrency in an execution of a fork-join program can be modeled by a concurrency graph which is directed and acyclic. Each vertex in a concurrency graph represents a unique thread executing a (possibly empty) sequence of instructions. Each graph edge is induced by synchronization implied by a fork or join construct. A directed path from vertex t_1 to vertex t_2 indicates that thread t_1 terminated execution before thread t_2 began execution.

Figure 5 shows a fragment of parallel Fortran and the corresponding graph that models concurrency during its execution. Each vertex in the graph is labeled with the sequence of code blocks that are sequentially executed by the thread corresponding to the vertex. In Figure 5, the concurrency graph for each parallel loop is formed by linking in parallel the concurrency graph for each loop iteration. The concurrency graph for iteration `I=2` of the outermost parallel loop is formed by linking in series of the concurrency graphs for the two loops nested inside that iteration.

Two vertices in a concurrency graph represent logically concurrent threads in an execution of a program if and only if the vertices are distinct, and there is no directed path between them. To detect if a pair of conflicting accesses is involved in a data race, the debugging run-time system must determine if the threads that performed accesses are logically concurrent. Rather than explicitly building a concurrency graph representation at run time to make this determination, the run-time system assigns each thread a label that implicitly represents its position in the concurrency graph. For this purpose, the system uses *offset-span labeling*, an on-line scheme for assigning a label to each thread in an execution of a program with fork-join parallelism [70]. By comparing the offset-span labels of two threads, their concurrency relationship can be deduced.

```
[code block A]
parallel do i=2,4
  [code block B]
  if (i.eq.2) then
    parallel do j = 1,2
      [code block C]
    enddo
  endif
  [code block D]
  parallel do j=1,i
    [code block E]
  enddo
  [code block F]
enddo
[code block G]
```

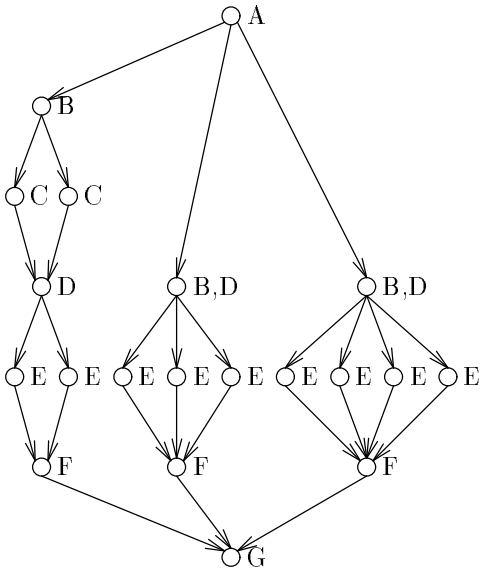

Figure 5 A fragment of parallel Fortran and its corresponding concurrency graph.

4.1.2 A Protocol for Detecting Data Races

In contrast to other access history protocols described in the literature [69, 71], the run-time protocol used by ParaScope to detect data races bounds the length of each variable's access history by a small constant that is program independent [70]. Bounding the length of access histories has two advantages. First, it reduces the asymptotic worst-case space requirements to $O(VN)$, where V is the number of monitored shared variables, and N is the maximum dynamic nesting depth of parallel constructs. Second, it reduces to $O(N)$ the asymptotic worst-case number of operations necessary to determine whether a thread's access is logically concurrent with any prior conflicting accesses. However, since the protocol only maintains information about a bounded number of accesses to each shared variable, information about some accesses to shared variables will have to be discarded so as not to exceed the space bound. A concern is that discarding this information may cause a data race to escape detection.

To guarantee that the ParaScope debugging system always reports a data race if any is present in an execution, the race detection protocol used must be insensitive to the interleaving of accesses in an execution. In a previous paper [70], the following assertion is proven about the run-time protocol used by ParaScope:

> *If an execution of a program with nested fork-join parallelism contains one or more data races, at least one will be reported.*

Proof of this assertion establishes that a bounded access history is sufficient for detecting data races in programs with fork-join parallelism regardless of the actual temporal interleaving of accesses in an execution.

With this condition, an execution will never be erroneously certified as race free. Furthermore, if no race is reported during a monitored execution for a given input, then the program is guaranteed to be deterministic for that input.

Because the ParaScope debugging system uses an access history protocol with this property, it can support the following effective debugging strategy for eliminating data races from a program execution for a given input:

1. Run the program on the given input using the monitoring protocol to check for data races involving accesses to shared variables.
2. Each time a data race is reported (the access history protocol precisely reports both endpoints of the race), fix the cause of the data race, and re-execute the program with the same input.

Since the access history protocol used by ParaScope reports data races (if any exist) regardless of the interleaving order, the protocol can be used to check for races in a program that is executed in a canonical serial order. Executing programs serially while debugging is convenient as it provides the user with deterministic behavior which simplifies the task of determining the origin of variable values that indirectly caused a data race to occur.

4.2 Compile-time Support for Data Race Detection

During an execution, the access history protocol used by the debugging run-time system determines if a data race exists between any two monitored accesses of a shared variable. The task of the compile-time instrumentation system is to guarantee that for each access potentially involved in a data race, there is a corresponding call to the proper run-time support routine to test for the presence of a race. Rather than instrumenting each access to a shared variable, the debugging instrumentation system is tightly integrated with the ParaScope compilation system so that it can exploit compile-time analysis to significantly reduce the number of run-time checks. Here, we first describe the task of the debugging instrumentation system, then we describe how the instrumentation system is integrated into the ParaScope compilation system.

The instrumentation system has two main tasks: to allocate storage for access history variables and to insert calls to run-time support routines that perform access checks or concurrency bookkeeping. To maximize the portability of instrumented code, we have chosen to perform data race instrumentation at the source code level. For any variable that may be involved in a data race at run time, the instrumentation system must allocate storage for an access history.

For references that need to be checked at run time, the instrumentation system inserts a call to the appropriate run-time support routine which will determine if the access is involved in a race. Locating the accesses that need to be monitored is conceptually easy. For variables local to a subprogram, only those accesses that are the endpoints of dependences carried by parallel loops in that subprogram need run-time

checks. Using dependence information computed under control of the ParaScope compilation system (see Section 2.1) considerably reduces the amount of instrumentation necessary.

The instrumentation process becomes somewhat more complicated in the presence of procedure calls. If a procedure is called from within a parallel construct, the instrumentation system must ensure that any side-effects of the called procedure are appropriately checked for participation in data races. Adding this instrumentation requires two steps. First, the system expands the parameter list for each procedure to include a history variable for each formal parameter and adds a run-time check inside the procedure for each access to a variable that appears as one of its original formal parameters. Next, the system expands any common block declarations inside each procedure to include a history variable for each common variable, and adds a run-time check for each access to a common variable.

The instrumentation process as described thus far for programs with procedure calls in parallel loops derives little or no benefit from compile-time analysis to prune the amount of instrumentation needed. To remedy this situation, we are in the process of incorporating interprocedural analysis into the instrumentation system. As currently envisioned, the interprocedural analysis to support data race instrumentation consists of three passes. The first pass determines which accesses inside each procedure require run-time checks. The second pass allocates access history storage for variables as needed. A third pass is necessary to ensure that the set of access history variables added to each declaration of a common block is globally consistent. The ParaScope program compiler directs the collection of this interprocedural information and coordinates instrumentation of the program.

4.3 Reporting Data Races

There are several ways that an instrumented program can be debugged. Simply running the program will produce a collection of error messages that indicate all of the data races detected during execution. If the program is run under the control of a traditional interactive debugger such as *dbx* or *gdb* and a breakpoint is inserted at the appropriate place in the run-time support library, execution will halt as each race is detected enabling the user to investigate the program state.

One drawback of debugging this way is that the user is exposed to the instrumented code. The ParaScope debugging system composes a map created by the instrumentation system with the standard compiler-provided symbol table to create a map between uninstrumented code and machine code. When debugging instrumented code using the ParaScope source-level debugger (a window-based source level debugger with functionality similar to *xgdb*), a user sees the original code displayed on the screen. Debugging in terms of the original code uncluttered by instrumentation is usually preferable.

The ParaScope debugger display shown in Figure 6 illustrates several points. The debugger window shown is divided into three main panes. The top pane contains the source being debugged. The small pane

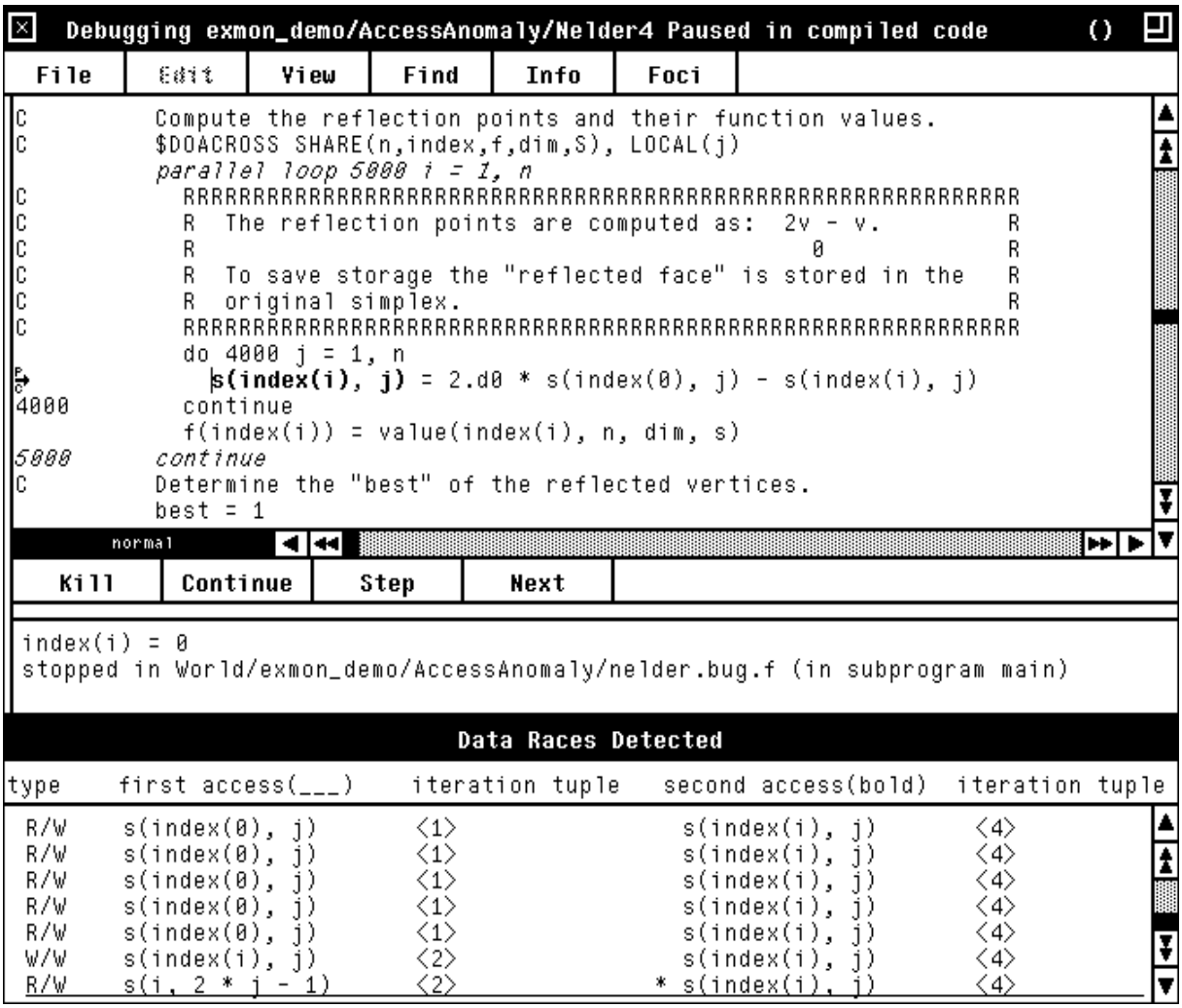

Figure 6 The ParaScope debugger: isolating a race condition.

below that has messages from the debugger, such as values of expressions. The pane at the bottom contains a list of all data races detected during execution.

In the display shown, the underlining in the bottom pane indicates that the user has selected the last data race reported during the debugging session. This selection is reflected in the source pane by underlining the (temporally) first access of the data race and emboldening the second access. The loop carrying the race is italicized. Examination of the source pane indicates that the first access is not currently being displayed. In fact, it is in the function `value`. Clicking the mouse on the data race in the bottom pane will cause the display to shift so that the other endpoint of the selected race is visible.

The ParaScope debugger supports view filtering (see Section 3.1) which is useful in the debugging process. For example, it provides a view filter predicate that matches source lines containing definitions (or uses) of a particular variable. It also defines a predicate that matches lines found to have been involved in a data

race. Thus, it is possible to tailor the display so that only those lines and their surrounding parallel loops are shown. This can often simplify the process of viewing both ends of a race condition whose endpoints are widely separated.

4.4 Preliminary Evaluation

Our experiences thus far with the data race detection support in ParaScope have been positive. The user interface provides a convenient display of race reports that facilitates understanding how races have arisen. Although the implementation of the interprocedural components of the instrumentation system is not yet complete, we have produced a manual simulation of the analysis for a single test program and used it to prune conservative instrumentation generated using strictly local analysis. For our test program, the run-time overhead of the data race instrumentation was slightly less than 100%. This measurement provides evidence that by using an efficient run-time protocol and aggressive compile-time analysis, the overhead of automatic data race instrumentation may well be tolerable in practice during a debugging and testing phase.

5 Future Directions for ParaScope

The current version of ParaScope is designed to assist in the development of programs for shared-memory multiprocessors, the class of parallel architectures most widely used at the commencement of the project in the mid 1980's. The principal challenge faced by scientific users of such machines was finding enough loop-based parallelism to make effective use of the available computation power. Over the past few years, a paradigm shift has occurred and distributed-memory parallel computers, such as the Intel Paragon and the Thinking Machines CM-5, have become the dominant machines for large-scale parallel computation. In these machines, part of the memory is packaged with each of the individual processors and each processor can access directly only its own portion of the memory. To access a remote data location requires that the owning processor send the value to the processor that wishes to use it, an expensive operation. The most important optimization for these machines is minimizing the cost of communication. Data placement plays a central role in determining the extent to which communication can be optimized—good data placement is a prerequisite for optimum performance.

Unfortunately, the shared-memory programming model does not provide the programmer with a mechanism for specifying data placement. Unless the compiler can choose the right data placement automatically—a formidable task—this model will not be very useful on distributed-memory machines. To address this problem, we have developed an extended version of Fortran, called Fortran D [75], which enables the programmer to explicitly specify data distribution and alignment on a multiprocessor system. Furthermore, the specification can be expressed in a machine-independent form, making it possible for the programmer to write a single program image that can be compiled onto different parallel machines with high efficiency.

Machine-independent parallel programming is particularly important if application software developed for the current generation of parallel machines is to be usable on the next generation of machines. We are currently developing compilers that translate Fortran D to run efficiently on two very different target architectures: the Intel iPSC/860 and the Thinking Machines CM-2.

Once the data distribution is known, parallelism can automatically be derived by the Fortran D compiler through the use of the "owner computes" rule, which specifies that the owner of a datum computes its value [76]. Assignment to all locations in an array distributed across the processors can automatically be performed in parallel by having each processor assign values for its own portion of the array, as long as there are no dependences among the assignments. The compiler is also responsible for rewriting a shared-memory Fortran D program, which is simply a Fortran 77 or Fortran 90 program with distribution specifications, as a distributed-memory program that can be run on the target machine. For the Intel iPSC/860, this means strip mining all of the loops to run on a single node of the processor array and inserting message passing calls for communication. The resulting message-passing Fortran program is quite different from the original source.

Since the appearance of the original technical report on Fortran D in December 1990, the language has gained widespread attention. There is currently underway an effort to standardize many of the features of Fortran D into an extended version of Fortran 90, called "High Performance Fortran" (HPF). However, with a language and compiler technology as complex as those required by Fortran D or HPF, the user will need a significant amount of program development assistance to use the machine effectively. To address this need, we have embarked on a project to build a programming environment for Fortran D. We will base it, in part, on ParaScope. The paragraphs below describe the extensions to ParaScope that will be found in the new environment.

Compilation System. Fortran D semantics dictate that distribution specifications for a variable can be inherited from a calling procedure. In addition, the Fortran D distributions are executable, so a called procedure may redistribute a variable. To generate correct and efficient code, the Fortran D compiler must determine which distributions may hold at each point in the program and it must track these distributions interprocedurally. Hence the ParaScope program compiler must be significantly enhanced to accommodate the compilation of Fortran D [77, 78].

Intelligent Editor. Most of the functionality in the current ParaScope editor will be useful in constructing Fortran D programs. For example, users will still want to find loop nests that can be run in parallel. However, an important additional challenge to the user is the selection of distribution specifications that will yield good performance. The new ParaScope editor will permit the user to enter and modify distribution

specifications. However, by itself, this will not be enough. Experience suggests that users will sometimes find the relationship between distributions and performance puzzling. The new ParaScope editor will make this relationship clearer by statically predicting the performance of loop nests under specific collections of Fortran D distributions. To achieve sufficient precision while remaining machine-independent, we will employ a "training set" strategy to produce a performance predictor for each new target machine [79].

Debugger. Since Fortran D programs undergo radical transformations during compilation, source-level debugging tools for Fortran D will require significant support to relate the run-time behavior of a transformed program back to its original source, which is considerably more abstract. Also, the definition of Fortran D currently includes a parallel loop construct, the FORALL statement, that permits a limited form of nondeterminism. We expect that programmers using this construct will need some assistance of the kind provided by the shared-memory debugging system for pinpointing data races.

Automatic data distribution tool. Once completed, the Fortran D compiler will provide a good platform for research on automatic data distribution. We intend to build an experimental system that automatically generates Fortran D distributions. To evaluate this system, we will compile and run the resulting programs, comparing their effectiveness to programs written by hand in Fortran D. If the automatic system produces reasonable results, it will be incorporated into the new ParaScope editor and used to advise the programmer on initial distribution specifications for data.

Performance visualization. The final step in parallel programming is performance tuning. It is imperative that the system provide effective tools to help the programmer understand program performance. We plan to adapt the existing ParaScope analysis tools to provide input to performance visualization systems under development elsewhere, such as the Pablo system at the University of Illinois [80]. An important component of performance visualization for Fortran D will be relating performance data for a transformed program back to the more abstract original program source.

The final result of this effort will be a collection of tools to support the development of machine-independent data-parallel programs. In addition, the environment will be generalized to handle a substantial part of Fortran 90, particularly the array sub-language. At that point, it will be easy to adapt it for use with the emerging High Performance Fortran standard.

6 Acknowledgements

The ParaScope system is the product of many years of labor. A large collection of students and staff have worked on its design and implementation. Researchers and users from outside the project have provided

critical feedback and direction. We are deeply grateful to all these people.

References

[1] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe, "Analysis and transformation of programs for parallel computation," in *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, (Chicago, IL), pp. 709–715, Oct. 1980.

[2] J. R. Allen and K. Kennedy, "PFC: A program to convert Fortran to parallel form," in *Supercomputers: Design and Applications* (K. Hwang, ed.), pp. 186–203, Silver Spring, MD: IEEE Computer Society Press, 1984.

[3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the PTRAN analysis system for multiprocessing," in *Proceedings of the First International Conference on Supercomputing*, Athens, Greece: Springer-Verlag, June 1987.

[4] S. Tjiang, M. E. Wolf, M. S. Lam, K. L. Pieper, and J. L. Hennessy, "Integrating scalar optimization and parallelization," in *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Aug. 1991.

[5] B. Leasure, ed., *PCF Fortran: Language Definition, version 3.1.* Champaign, IL: The Parallel Computing Forum, Aug. 1990.

[6] B. Shei and D. Gannon, "SIGMACS: A programmable programming environment," in *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, (Irvine, CA), Aug. 1990.

[7] K. Smith and W. Appelbe, "PAT - an interactive Fortran parallelizing assistant tool," in *Proceedings of the 1988 International Conference on Parallel Processing*, (St. Charles, IL), Aug. 1988.

[8] H. Zima, H.-J. Bast, and M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, vol. 6, pp. 1–18, 1988.

[9] U. Banerjee, *Dependence Analysis for Supercomputing.* Boston, MA: Kluwer Academic Publishers, 1988.

[10] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[11] D. J. Kuck, *The Structure of Computers and Computations*, vol. 1. New York: John Wiley and Sons, 1978.

[12] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers.* Cambridge, MA: The MIT Press, 1989.

[13] G. Goff, K. Kennedy, and C. Tseng, "Practical dependence testing," in *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[14] K. D. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ programming environment," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 491–523, Oct. 1986.

[15] K. S. McKinley, *Automatic and Interactive Parallelization.* PhD thesis, Rice University, Houston, TX, Apr. 1992.

[16] M. W. Hall, K. Kennedy, and K. S. McKinley, "Interprocedural transformations for parallel code generation," in *Proceedings of Supercomputing '91*, Nov. 1991.

[17] K. Kennedy, N. McIntosh, and K. S. McKinley, "Static performance estimation in a parallelizing compiler," Tech. Rep. TR91-174, Dept. of Computer Science, Rice University, Dec. 1991.

[18] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and memory hierarchy," in *Proceedings of the 1992 ACM International Conference on Supercomputing*, (Washington, DC), July 1992.

[19] M. Burke and L. Torczon, "Interprocedural optimization: Eliminating unnecessary recompilation." To appear in *ACM Transactions on Programming Languages and Systems.*

[20] M. W. Hall, *Managing Interprocedural Optimization.* PhD thesis, Rice University, Houston, TX, Apr. 1991.

[21] K. D. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Languages Design and Implementation,* SIGPLAN Notices 23(7), pp. 57–66, June 1988.

[22] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 517–550, 1988.

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

[24] E. Myers, "A precise inter-procedural data flow algorithm," in *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981.

[25] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pp. 47–56, July 1988.

[26] E. Granston and A. Veidenbaum, "Detecting redundant accesses to array data," in *Proceedings of Supercomputing '91*, (Albuquerque, NM), Nov. 1991.

[27] Z. Li, "Array privatization for parallel execution of loops," in *Proceedings of the 1992 ACM International Conference on Supercomputing*, (Washington, DC), July 1992.

[28] C. Rosene, *Incremental Dependence Analysis*. PhD thesis, Rice University, Houston, TX, Mar. 1990.

[29] R. Eigenmann and W. Blume, "An effectiveness study of parallelizing compiler techniques," in *Proceedings of the 1991 International Conference on Parallel Processing*, (St. Charles, IL), Aug. 1991.

[30] M. W. Hall, T. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. Paleczny, and G. Roth, "Experiences using the ParaScope Editor," Tech. Rep. CRPC-TR91173, Center for Research on Parallel Computation, Rice University, Sept. 1991.

[31] 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 ACM Symposium on Principles of Programming Languages,* San Antonio, TX, pp. 29–41, Jan. 1979.

[32] K. D. Cooper and K. Kennedy, "Fast interprocedural alias analysis," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1989.

[33] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pp. 152–161, July 1986.

[34] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[35] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, Jan. 1988.

[36] F. E. Allen and J. Cocke, "A catalogue of optimizing transformations," in *Design and Optimization of Compilers* (J. Rustin, ed.), Englewood Cliffs, NJ: Prentice-Hall, 1972.

[37] K. D. Cooper, M. W. Hall, and K. Kennedy, "Procedure cloning," in *Proceedings of the IEEE International Conference on Computer Languages*, pp. 96–105, Apr. 1992.

[38] K. D. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 491–523, Oct. 1986.

[39] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pp. 163–275, July 1986.

[40] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program restructuring for parallel programs," in *Proceedings of the SIGPLAN Symposium on Parallel Programs: Experience with Applications, Languages and Systems*, July 1988.

[41] R. Metzger and P. Smith, "The CONVEX application compiler," *Fortran Journal*, vol. 3, no. 1, pp. 8–10, 1991.

[42] R. Triolet, F. Irigoin, and P. Feautrier, "Direct parallelization of call statements," in *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pp. 176–185, July 1986.

[43] K. D. Cooper, M. W. Hall, and L. Torczon, "An experiment with inline substitution," *Software – Practice and Experience*, vol. 21, pp. 581–601, June 1991.

[44] K. D. Cooper, M. W. Hall, and L. Torczon, "Unexpected side effects of inline substitution: a case study." *to appear in Letters on Programming Languages and Systems*, Mar. 1992.

[45] P. Briggs, K. D. Cooper, M. W. Hall, and L. Torczon, "Goal-directed interprocedural optimization," Technical Report TR90-148, Dept. of Computer Science, Rice University, Nov. 1990.

[46] J. Singh and J. 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), Apr. 1991.

[47] K. Fletcher, K. Kennedy, K. S. McKinley, and S. Warren, "The ParaScope Editor: User interface goals," Tech. Rep. TR90-113, Dept. of Computer Science, Rice University, May 1990.

[48] K. Kennedy, K. McKinley, and C. Tseng, "Analysis and transformation in the ParaScope Editor," in *Proceedings of the 1991 ACM International Conference on Supercomputing*, (Cologne, Germany), June 1991.

[49] K. Kennedy, K. McKinley, and C. Tseng, "Interactive parallel programming using the ParaScope Editor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 329–341, July 1991.

[50] N. Yankelovitch, N. Meyrowitz, and A. van Dam, "Reading and writing the electronic book," *IEEE Computer*, vol. 18, pp. 15–29, Oct. 1985.

[51] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem, "Designing the star user interface," *BYTE*, vol. 7, pp. 242–282, Apr. 1982.

[52] D. C. Engelbart and W. K. English, "A research center for augmenting human intellect," in *Proceedings of AFIPS 1968 Fall Joint Computer Conference*, pp. 395–410, Dec. 1968.

[53] R. C. Waters, "Program editors should not abandon text oriented commands," *ACM SIGPLAN Notices*, vol. 17, pp. 39–46, July 1982.

[54] J. R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 491–542, Oct. 1987.

[55] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," in *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 25(6), pp. 53–65, June 1990.

[56] K. Kennedy and K. S. McKinley, "Loop distribution with arbitrary control flow," in *Proceedings of Supercomputing '90*, (New York, NY), Nov. 1990.

[57] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe, "The structure of an advanced retargetable vectorizer," in *Supercomputers: Design and Applications*, pp. 163–178, Silver Spring, MD: IEEE Computer Society Press, 1984.

[58] D. Loveman, "Program improvement by source-to-source transformations," *Journal of the ACM*, vol. 17, pp. 121–145, Jan. 1977.

[59] M. J. Wolfe, "Loop skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, pp. 279–293, Aug. 1986.

[60] J. R. Allen, D. Bäumgartner, K. Kennedy, and A. Porterfield, "PTOOL: A semi-automatic parallel programming assistant," in *Proceedings of the 1986 International Conference on Parallel Processing*, (St. Charles, IL), IEEE Computer Society Press, Aug. 1986.

[61] V. Balasundaram, D. Bäumgartner, D. Callahan, K. Kennedy, and J. Subhlok, "PTOOL: A system for static analysis of parallelism in programs," Technical Report TR88-71, Dept. of Computer Science, Rice University, 1988.

[62] L. Henderson, R. Hiromoto, O. Lubeck, and M. Simmons, "On the use of diagnostic dependency-analysis tools in parallel programming: Experiences using PTOOL," *The Journal of Supercomputing*, vol. 4, pp. 83–96, 1990.

[63] "The MIMDizer: A new parallelization tool," *The Spang Robinson Report on Supercomputing and Parallel Processing*, vol. 4, pp. 2–6, Jan. 1990.

[64] D. Cheng and D. Pase, "An evaluation of automatic and interactive parallel programming tools," in *Proceedings of Supercomputing '91*, (Albuquerque, NM), Nov. 1991.

[65] V. Guarna, D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur, "Faust: An integrated environment for parallel programming," *IEEE Software*, vol. 6, pp. 20–27, July 1989.

[66] D. Cheng and K. Fletcher, "Private communication," July 1991.

[67] J. Stein, "Private communication," July 1991.

[68] A. Dinning and E. Schonberg, "An evaluation of monitoring algorithms for access anomaly detection," Ultracomputer Note 163, Courant Institute, New York University, July 1989.

[69] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pp. 1–10, Mar. 1990.

[70] J. M. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of Supercomputing '91*, (Albuquerque, NM), Nov. 1991.

[71] I. Nudler and L. Rudolph, "Tools for efficient development of efficient parallel programs," in *First Israeli Conference on Computer Systems Engineering*, 1986.

[72] E. Schonberg, "On-the-fly detection of access anomalies," in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 24(7), pp. 285–297, July 1989.

[73] G. L. Steele, Jr., "Making asynchronous parallelism safe for the world," in *Conference Record of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, Jan. 1990.

[74] S. L. Min and J. Choi, "An efficient cache-based access anomaly detection scheme," in *Proceedings of the 4th ACM International Conference on Architectural Support for Programming Languages and Operating Systems,* SIGPLAN Notices 26(4), pp. 235–244, Apr. 1991.

[75] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Tech. Rep. TR90-141, Dept. of Computer Science, Rice University, Dec. 1990.

[76] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng, "An overview of the Fortran D programming system," in *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, (Santa Clara, CA), Aug. 1991.

[77] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proceedings of Supercomputing '91*, (Albuquerque, NM), Nov. 1991.

[78] S. Hiranandani, K. Kennedy, and C. Tseng, "Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proceedings of the 1992 ACM International Conference on Supercomputing*, (Washington, DC), July 1992.

[79] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A static performance estimator to guide data partitioning decisions," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 213–223, July 1991.

[80] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty, "Scalable performance environments for parallel systems," in *Proceedings of Distributed Memory Computing Conference*, (Portland), pp. 562–569, Apr. 1991.