A Practical Environment for Scientific Programming

Alan Carle
Keith D. Cooper
Robert T. Hood
Ken Kennedy
Linda Torczon
Scott K. Warren

Department of Computer Science*
Rice University
Houston, Texas 77251-1892

1. Introduction

Since the appearance of the first electronic computers over forty years ago, software for science and engineering has been an important consumer of computer resources. Almost every industry uses scientific software in the development of their products. Energy companies use scientific programs to simulate the behavior of oil and gas reservoirs and to analyze seismic data for evidence of new reservoirs. The automobile and aircraft industries use very large programs to analyze and experiment with new vehicle designs. The National Center for Atmospheric Research is working with a computerized model of world weather in search of better weather prediction techniques. Computer models are also indispensable tools for theoretical biochemists, chemists, and physicists.

The total world investment in scientific programs is enormous, and the development and maintenance costs continue to grow. Support for scientific software development presents most of the problems found in systems programming support—the codes are large, the interfaces are complex, and debugging is difficult. In addition, there are three unique aspects to scientific programming that must be taken into consideration when designing support environments.

- First, most scientific codes are written in FORTRAN and, although the language can be criticized on a variety of grounds, its standardization together with the early availability of good compilers has led to the wide acceptance of the language by the numerical software community.
- Second, there is vast repository of accumulated software and the scientific community actively exchanges software, particularly in the form of libraries of FORTRAN subroutines such as LINPACK and EISPACK.
- Third, since scientific calculations are extremely computation-intensive, efficiency of the code generated by the compiler is critical to the FORTRAN programmer. The first FORTRAN compiler project implemented a very sophisticated optimization phase. Today, any manufacturer desiring to sell machines for scientific problem-solving must devote significant resources to the development of a optimizing compiler for FORTRAN.

In spite of all the recent research activity on programming support environments, not much has been done for the support of scientific programming[1]. As a part of a research project on the use of workstations for scientific problem-solving, we embarked in 1982 on a project to design and implement a software development environment, called $\mathbb{R}^n$, for numerical programmers. In addition to providing the usual environment tools—intelligent editor, debugger, data base—we designed $\mathbb{R}^n$ to address the three unique aspects of scientific programming: FORTRAN, software libraries, and efficiency of execution.

* This work has been supported by both IBM Corporation and the National Science Foundation.

[1] An exception is TOOLPACK, a widely-distributed library of programming support tools for FORTRAN.

Of these, efficiency has had the most important implications for the environment. In exploring ways that IR^n might help an optimizing compiler produce efficient code, it is reasonable to ask if there is any room for improvement over the very good code generated by today's commercial compilers. The largest remaining impediment to high efficiency in code generated from FORTRAN is the presence of calls to independently-compiled procedures. When the compiler encounters such a call, it must assume that every variable accessible to that procedure will be both used and changed. In other words, every variable in common and every parameter must be assumed to be corrupted by the call. This assumption severely inhibits optimization around a procedure call. For example, *redundant subexpression elimination* is an optimization in which computation of an expression is replaced by a load of the value of a previously-computed identical subexpression. When attempting to eliminate redundant subexpressions, the compiler must assume that any expression involving either a global variable or a formal parameter must be recomputed after a call site, severly limiting the effectiveness of this important optimization.

These problems have been acknowledged by compiler-writers for a long time, yet there has emerged no general solution suitable for adoption into commercial compilation systems, in spite of the extensive literature on algorithms for interprocedural optimization and analysis (see Richardson and Ganapathi's recent bibliography [8]). The problem is that interprocedural optimization conflicts with FORTRAN's notion of *independent compilation*—the ability to compile a subroutine in the absence of any information about the program that calls it. In a typical FORTRAN compilation system, the user prepares and compiles each of the subroutines separately. The program is then constructed by joining the pieces together during the linkage edit phase. If we are to take advantage of knowledge about other procedures in the program during compilation, this approach is no longer feasible.

A naive solution is to insist that the entire program be submitted to the compiler at one time, enabling it to garner all the information it needs to perform interprocedural analysis and optimization from its reading of the code. This approach suffers from the drawback that FORTRAN programs often grow to hundreds of thousands of lines and compiling such a program at once would take an intolerable length of time. One of the main reasons for independent compilation is to make it possible to rapidly reconstruct an executable after a change to a small part of the program. Our experience with PFC, a compilation system for parallel computers which uses the naive approach has taught us that the overhead of simply reading the entire program makes the system too slow to be usable.

A software development environment like IR^n offers an opportunity for a more elegant solution. If the programmer is willing to define which procedures constitute the program before compilations begin, then the system components can collaborate to deliver the information needed to perform interprocedural analysis and optimization while preserving the efficiency of independent compilation. For example, the source editors can provide enough information about the contents of every procedure in a given program to permit the compilation system to develop enough information about interprocedural data flow to improve the quality of optimization in a procedure with subroutine calls. In addition, the system can use information from the editors to perform sophisticated interprocedural optimizations.

This paper describes IR^n, paying particular attention to its role in the compilation process. Section 2 contains a description of the important programming tools in IR^n and a discussion of the mechanics of program preparation. Section 3 discusses the design and implementation of the environment's two principal editors. Section 4 describes the *execution monitor*, which runs a program and provides debugging support. The compilation system is treated in detail in Section 5, with particular emphasis on compiling the whole program, rather than a single subroutine. Finally, Section 6 outlines the evolution of the environment to support parallel programming.

2. IR^n Overview

IR^n is intended to be used by expert programmers in developing, testing, debugging and maintaining large FORTRAN programs. The current version of the environment includes a graphical window manager, a database, tools for creating whole programs from FORTRAN source, tools for compiling these programs, and tools for debugging them.

2.1. The Monitor and Database

The entire environment is structured as a collection of command processors running inside a single UNIX† process. A window-based *monitor* provides mechanisms to control sub-processes, facilitates communication between command processors, and manages the display, mouse, and keyboard. It also provides a set of abstractions for the objects used to construct the user interface.

The command processors use a central *database* as a long term repository for the objects used in preparing and translating programs. This includes objects created by the programmer, such as source text and data, and objects created by the compiling system, such as subroutine interface information. The two principal types of objects stored in the database are *modules* and *compositions*. A module is a set of one or more entry points and the code to implement them. Thus, a module may include several subroutines. The decision about what to include in a module is up to the user, and is made on the basis of taste about how much should go into a single editable or compilable unit. A *composition* is a hierarchical specification of a program's structure in terms of modules and other compositions.

2.2. Program Preparation Tools

One of the explicit design goals of $\mathbb{R}^n$ was to provide a mechanism for supporting interprocedural analysis and optimization of compiled code in a practical compiling system. To achieve this, we have divided the compilation system into two compilers: a *program compiler*, which deals solely with optimizations and analysis that cross procedure boundaries, and a *module compiler*, which compiles and optimizes the procedures in a single module, using interprocedural information developed by the program compiler.

Similarly, to support construction of whole FORTRAN programs, $\mathbb{R}^n$ provides a pair of editors: a *composition editor*, which is used to define the contents and structure of programs and composite modules, and a *module editor*, which is used to enter and modify FORTRAN source.

The steps required to prepare for execution in $\mathbb{R}^n$ are depicted in Figure 1. We briefly discuss each of these tools.

Module Editor

The *module editor* is intended to be the primary vehicle for editing FORTRAN modules in the environment. It provides a convenient blend of structure and text editing, allowing the user to shift freely between these two paradigms. It is an intelligent editor that not only assists the programmer in producing syntactically correct programs, but also participates in gathering information for use by other tools. The editor directly constructs an *abstract syntax tree* representation of the module; this internal form for procedures is used throughout the environment. In addition, the module editor collects module-specific information required to support the interprocedural analysis and optimization performed in the compilation system.

Composition Editor

The user creates and modifies compositions with the *composition editor* which checks for consistency (the agreement of entry point specifications with corresponding calls) and completeness (the inclusion of all required entry points). In essence, the composition editor is a syntax-directed editor for a module interconnection language, albeit one with a screen-oriented concrete syntax. The composition editor allows the user to conveniently complete a program with missing entry points by searching through a sequence of libraries. If the user wishes to take advantage of the support for interprocedural optimization, the composition of a program must be defined before the compilation process begins.

Program Compiler

The principal task of the program compiler is to automatically construct an executable image for a program that is, at once, fully optimized and fully consistent with the source code [5]. Thus, it performs the bulk of the work of interprocedural analysis. This entails deriving a graph of the procedure call

† UNIX is a trademark of AT&T Bell Laboratories.

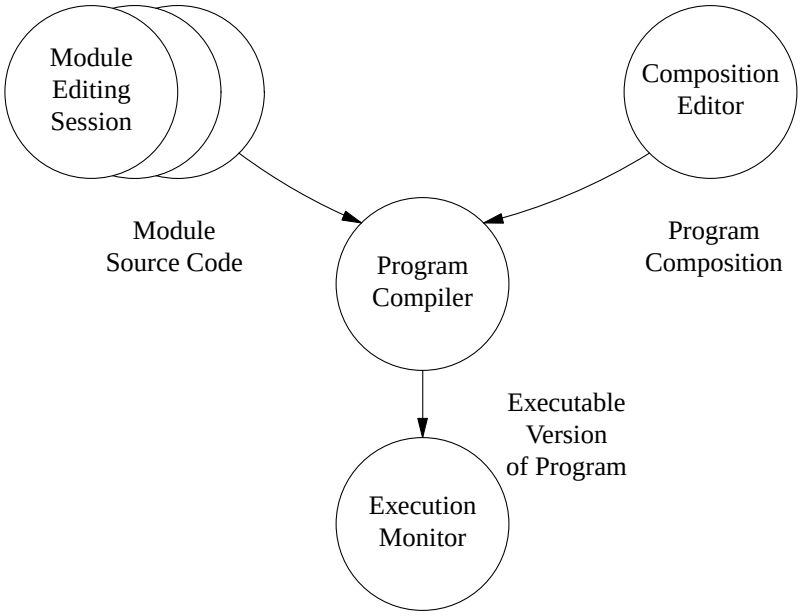

Figure 1: Program preparation in IR^n

structure of the program, as well as computing summary, aliasing, and constant sets. It also develops an optimization plan for the whole program. The plan details what type of procedure linkage should be used for each call site. The actual translation of source code to object code for the target machine is performed by the module compiler.

Module Compiler

The module compiler resembles the optimizer and code generator of a conventional compiler. It deals with a single module or a small collection of modules that are being optimized together because of directives provided by the program compiler. It translates the source code for the module or modules into optimized code for the target machine, capitalizing on the presence of interprocedural information about the program. The module compiler has been carefully designed to allow experimentation with different strategies for optimization and for recompilation analysis.

Execution Monitor

The execution monitor allows the programmer to run a program constructed by the environment. It provides an editor-like setting for program execution and debugging. For example, operations such as breakpoint setting or expression evaluation are accomplished with a small number of mouse selections. In addition, it allows a range of instrumentation levels, including execution of hybrid programs in which some modules are run from compiled, optimized code and other modules are run interpretively. Instrumentation levels are dynamic; the programmer can change them whenever execution is paused at a breakpoint.

While the interprocedural information collected by the environment is primarily of interest to the optimizer, it finds application in other areas. In particular, the two editors and the execution monitor capitalize on the availability of this information. The composition editor uses information about the number, type, and dimension of parameters to check for consistent use of interfaces. The module editor uses the same

information to generate editing templates for calls to known procedures. Further, if the programmer enters a literal constant as an argument to a subroutine invocation, and the parameter in that position may be modified, the editor generates a warning. The execution monitor uses interprocedural information about which variables may be modified by a procedure call to improve the efficiency of interpreted code. This is discussed further in Section 4.

2.3. The Compilation Process

In designing the environment, one of our explicit goals was to distribute the program analysis required for optimization over the entire program preparation process. By carefully making the information flow in the environment match the steps a user takes to develop a program, we have been able to collect the requisite information efficiently and unobtrusively. As shown in Figure 2, the module editor collects local data-flow information and the composition editor collects program structure information. The program compiler uses this information to compute interprocedural data-flow information which is then supplied to the module compiler. In effect, the editors unobtrusively record information needed to support optimization while the user is developing a program, reducing both the actual and the perceived expense of the analysis. To see this, consider a specific interprocedural problem.

Suppose we wish to determine, for each call site *s* in a procedure, the set MOD(*s*) that contains all the variables that might be modified upon return from the call. Any variable that is passed as a parameter to the called procedure or any variable that is global to the called procedure is a candidate for MOD(*s*). Conventional optimizing compilers have no access to information about called procedures, and hence must assume that MOD(*s*) consists of all variables that are either actual parameters at the call site or global variables of

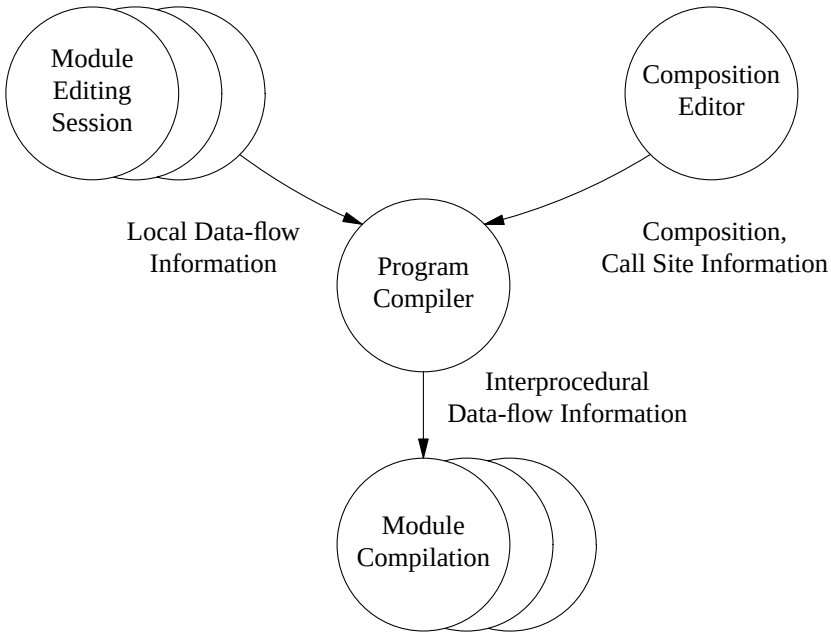

Figure 2: Flow of interprocedural information

the called procedure—the only assumption that is guaranteed to be safe in these circumstances.

To compute a more precise MOD(s), we need to look at the variables that might be changed, directly or indirectly, by the called procedure, q. Let GMOD(q) be the set of variables that may be changed by a call to q. If we can determine GMOD(q) for each procedure q in the program, then MOD(s) for a call to q can be derived by determining which variables in the calling procedure are bound to members of GMOD(q) by the call.

A strategy for computing the GMOD sets can be derived by noticing that GMOD(q) consists of two components:

- the set IMOD(q) of variables that might be directly modified by statements in q other than procedure calls, and
- the set of variables that might be modified as a side effect of a secondary procedure call from within q.

The sets IMOD(q) for each entry q in a module are independent of any other procedures in any program in which the module is incorporated. Hence, these sets can be *computed by the module editor* and stored with the module that generated them. From these sets, GMOD(q) can be computed for a program by solving a data-flow analysis problem on the program's call graph. Since the sets GMOD(q) depend on the specific call graph, they must be stored with the program for which they are computed.

By distributing the analysis over both editing and compiling, we are able to decrease the time it requires. Because the editors derive initial information like the IMOD sets and the call graph components, the program compiler need not reread each of the procedures in the program for each compilation. The required local information, recorded by the editors as procedures are developed and modified, is saved in the database. Instead, the program compiler need only read the IMOD sets, recompute the interprocedural information, and recompile the changed subroutine along with any subroutines for which interprocedural information has changed[2]. This avoids the necessity to read the source of every module in the program at each compilation, a real savings.

These observations illustrate an important aspect of the division of labor in IR^n. In addition to parsing and type checking, the editors can gather most of the information needed to support interprocedural analysis. An independent process can then compute the interprocedural side effects for the whole program. Because the analysis of these side effects *does not depend on output from the compiler*, the environment can compute interprocedural information for each entry point in the program before any module is compiled.

3. IR^n Editors

In this section we will describe in more detail the two editors that are used to prepare programs in IR^n. The purpose of this discussion is to illustrate the philosophy behind the design, rather than to provide a level of detail more suitable for a user's manual. Hence, our primary focus will be on the functions performed by these editors and upon the principal design decisions.

3.1. The Module Editor

As discussed in the previous section, the module editor is not only used for entering source text, it also plays a major role in gathering information to support the compilation process. In the IR^n database, all modules are represented as abstract syntax trees. The editor is responsible for producing such trees as a by-product of the editing process. In addition, the editor must produce information about the usage of variables in the program as an input to interprocedural analysis. Such information can only be gained by examining the program in parsed form.

Since we needed to have the program in parsed form, it seemed logical to design the module editor as a *structure editor*, one in which the user enters whole syntactic constructs into a tree. In a typical structure

[2] The subject of limiting the number of recompilations is discussed in Section 5.

editor, the program is entered through a series of steps in which the user expands a placeholder template at each step. For example, the user might wish to expand a <statement> into a <conditional statement>. After executing the command to do this, the statement placeholder is replaced by an if-then-else template which contains placeholders for the conditional expression, the then-statement and the else-statement. Hence, the program is developed from the top down using a system based on the language's grammar.

Structure editors have many advantages. They provide assistance for the programmer in correctly entering unfamiliar statements and moving entire syntactic structures around the program. As an additional advantage from the implementor's standpoint, they obviate the need for a source language parser[3].

Unfortunately, users find structure editors awkward for a variety of reasons. The main problem is that such editors deal with trees that look like text. Hence, the rules concerning legal and illegal operations are far from self-evident. For example, there are parts of the displayed text that cannot be edited, namely the fixed tokens in a template. The user who wishes to convert an if statement to a while statement soon discovers that changing the if token will not work. In fact, this change requires a non-intuitive sequence of transformations that are far more complex than the operation deserves. Another problem with structure editors is that the cursor motion commands are difficult to deal with because they represent motion in trees. Our original editor provided a partial solution by relying on point-and-click semantics using a mouse, but this was not sufficient for the sophisticated user who wanted to move the cursor without leaving the keyboard.

For these reasons, we have completely reimplemented the IR^n module editor to accommodate a style that mixes the text and structure paradigms. This solution retains most of the advantages of structure editing while providing a much more understandable interface to the user. The new editor has all the capabilities of the old editor but now can be used as a pure text editor as well. Furthermore, there is a natural and helpful connection between text and structure. The user who wishes to get a template for a particular construct need only type an unambiguous prefix and hit the EXPAND key.

The key to the implementation is to use a line-oriented approach to parsing. Text is parsed on a line-by-line basis, with errors reported immediately. This is a good match to FORTRAN, with its line-oriented statement structure. The whole program is parsed in two levels—with one parse for a single line and a second to determine global program structure. After any text editing operation is performed, the FORTRAN parser is invoked on the changed text to reconstruct the abstract syntax tree for the module.

Selection in the module editor also employs the mixed paradigms. Textual selections are made simply by dragging the mouse over a range of characters. A MORE operation expands the selection to the smallest enclosing structural subtree. Selections can be deleted, copied or pasted by making a selection from an *edit* menu or by directly entering the appropriate key sequence.

Manual *view control* is an important feature of the new module editor. The user specifies the view through the use of explicit CONCEAL *and* *REVEAL* commands, which can be applied to text ranges or whole substructures. If applied to substructures, concealment is persistent; that is, the concealed substructure remains concealed when the user scrolls to another part of the program and then returns. The editor employs filtering mechanisms to provide automatic view control.

A final feature is of interest because of its use of information about the program that contains the module being edited. If the user specifies a *context program*, in which the module is included, and then asks the editor to expand a call statement, he will be prompted with a menu of entry points provided in the context program. Once he selects an entry point and expands again, a call template will be displayed with placeholders for each parameter. These names of these placeholders indicate the type and meaning of the parameters they represent. This feature makes it easy for the programmer to get the parameter bindings right without searching through the code of the program for the entry point code.

---

[3] IR^n includes a parser to allow importing of programs developed under other systems.

3.2. The Composition Editor

To create a *program* in $\mathbb{R}^n$, the software developer constructs a hierarchical description known as a *composition*. A composition is simply a list of modules and other compositions—hence, a composition can be used to specify not only a program's contents but also its structure. For example, if the program designer wishes to organize the code into three phases, he builds the program composition by including three subcompositions, each representing one of the phases.

Compositions are the basis for the configuration management and module interconnection facilities in $\mathbb{R}^n$. They contain the information needed to perform efficient cross-procedural consistency checks. At each level of the composition, import and export lists provide a scoping and renaming facility for entry point names. In essence, compositions provide the numerical programmer with a MODULA-like abstraction mechanism without extending the base FORTRAN syntax. Compositions can represent libraries and composite modules as well as programs.

Compositions are constructed and modified with the *composition editor*, an interactive editor that uses the same mixture of structure and text editing found in the module editor. The composition editor checks the composition for consistency and completeness after each modification. *Consistency* requires that all call site specifications match the entry point specifications to which they are bound and that no two subunits included by a composition export identically named routines. *Completeness* requires that a composition implement all entry points that it exports, except those that it explicitly imports from some external source.

Little distinction is made between composite modules, libraries and programs. A composite module encapsulates a particular indivisible abstraction, much like a MODULA module does. A library is a logical collection of indivisible subunits supporting problem solving in some domain. A program is simply a complete and consistent composition having a designated ''main'' entry point. The composition editor can be used to build complete programs and libraries of subroutines, and to compose collections of subroutines and lower-level abstractions into new abstractions. Just as easily, a complete program may be used as a library, or as a single cohesive abstraction for use within another program.

These ideas are best illustrated through an example. Suppose we are implementing a program that requires an efficient sorting routine, say *quicksort*, and we wish to use the linear-time median finding algorithm to select the pivot element. The median finder needs another sorting routine for small sets, for example *bubblesort* to deal with sets of fifty or fewer elements.

Figure 3 shows an example composition that describes one possible implementation of our sorting program. The innermost composition, named *median*, contains two modules needed for the linear median algorithm, *median* and *bubblesort*. This composition is included, in turn, in a composition named *quicksort*, which is one of the subunits of the outermost composition, named *QuickTest*.

The example illustrates the facilities for managing name scopes. Because the *provides* set of the composition *median* doesn't contain *sort*, the entry inside module *bubblesort* can't be seen outside that composition. Thus, it doesn't conflict with the entry *sort* provided by the module *quicksort*. The composition *quicksort*, in turn, makes the entry *sort* available externally under the name *qsort* by renaming it in the *provides* set. The example also illustrates the power of this system for structural description. The median finder is encapsulated into one composition while the main sorting procedure is encapsulated into another. Throughout, comments are used to elucidate the program map.

Support of both *bottom-up* and *top-down* program development paradigms has been an important goal in the design of the composition editor. Bottom-up development constructs programs from existing code. Using the composition editor, the programmer can build a portion of a program and have the environment efficiently complete it by including existing routines with matching subroutine interfaces from some base library or previously-existing program. All automatic additions are tentative, and can be easily ''undone.'' This scheme is ideal for constructing a new program that is very similar to an existing program, or for constructing a program that primarily invokes common library routines.

Top-down development allows the system developer to describe the entire structure of his program phase by phase, before any FORTRAN code has been written; global storage can be defined and subroutine names and parameter definitions can be chosen. These subroutine interfaces and global storage definitions

```
comp QuickTest
  provides main;
  {
    mod driver
      provides main;
      requires qsort;
    comp quicksort
      provides qsort=sort;
      {
        /* Large-data quicksort--uses linear median finder */
        mod quicksort
          provides sort;
          requires median;
        comp median
          provides median;
          {
            /* Linear median - uses bubblesort to find
               median for sets with fewer than 50 elements */
            mod median
              provides median;
              requires sort;
            mod bubblesort
              provides sort;
          }
      }
  }
```

Figure 3: A composition

will be made available to the programmer automatically. In this scenario, interface inconsistencies are either avoided or immediately discovered. Errors will be tolerated within a composition, however, in order to support graceful changes to the top-down design of a system. It is, of course, possible to mix bottom-up and top-down development schemes.

4. The Execution Monitor

4.1. Design Motivation

One of the major concerns in designing a system for debugging large, programs is the fact that increased power in the debugger slows the execution of the program being debugged. Although interpreters can provide many desirable features for instrumenting a running program, they are typically too slow to be seriously considered for debugging computationally intensive programs. On the other hand, a fully optimized program is exceedingly difficult to debug, primarily because there may be little correspondence between the original source and the program being executed.

We need a paradigm for program execution that combines the best of both execution methods. If the debugger permits the substitution of interpreted for optimized code during execution, it will be possible to provide the increased debugging instrumentation precisely when it is needed without being penalized for it elsewhere. For example, if a large program has recently had one of its component modules changed, it

should be possible for execution to proceed quickly (*via* compiled code) to the module under development and at that point bring the full instrumentation power of an interpreter to bear. Such a facility would make it possible to implement several important features:

Incremental Modification

Having interpreted code in the form of an abstract syntax tree will make it easy to support modifications of the program during execution. This is particularly important when running large programs, because of the time required to recompile, relink, and return to the same point in the execution.

Exact Real Arithmetic

One of the most promising reasons for the inclusion of interpreted code in large scientific programs is that it will be possible to perform floating point arithmetic with *no* roundoff or representation errors. By representing a real number by instructions for producing arbitrarily precise approximations, Boehm, *et al.*, [3] have provided a way to perform *exact* arithmetic operations on real numbers. Because the numbers are not represented by finite precision floating point representations, problems such as roundoff errors do not occur. Although a severe efficiency penalty rules out the possibility of using the exact representation of real numbers for performing all calculations in a large program, the techniques are perfectly adequate for evaluating the accuracy of answers obtained by conventional floating point arithmetic.

Reversible Execution

Interpreted code will also make it feasible to support reverse-stepping the program counter. Such a facility would permit the user to back up execution from a fault to determine what went wrong. For example, if the program stops because of a ''zero divide'', the user could reverse-step the program counter, looking for the place where the denominator became zero.

4.2. Implementation

To handle the execution of compiled code, the execution monitor is implemented in the style of other UNIX debuggers such as *dbx* and *sdb*—the debugger resides in a process that has the program being debugged as a child process. To access or change values in the program state of the child, the debugger uses a system call that enables a process (such as a debugger) to read and write the target's address space. To determine where various identifiers are in the executable, it uses symbol table data placed in the executable by the compiler. Having separate address spaces for parent and child has the obvious advantage that the existence of the debugger will not interfere with memory utilization in the program being debugged. The organization of the execution monitor is depicted in Figure 4.

To handle interpretation during program execution, the debugger process includes the code that implements the interpreter. When interpretation is requested, the debugger reads an abstract syntax tree (AST) representing the code to be interpreted into the address space of the parent process. To accomplish this, it uses symbol table information in the executable to map from the address of an instruction to the name of the object module that contains that address. The module name is then mapped to the appropriate AST in the database.

To support the existence of interpreted and compiled code in the executable we must worry about two things: the control flow interfaces and the data interfaces between the two types of code. For example, when a call is made from compiled code to a subprogram that is to be interpreted, we must ensure that execution in the child process stops and that the interpreter is started at the correct location in the forest of AST's. In addition, references to data that cross the interpreted-compiled boundary must be handled correctly. For example, there may be references in compiled code to data that is declared in a module being interpreted.

Fortunately, a simple invariant makes it easy to avoid the data interface problems: *the entire state of the program being debugged is kept in the child process*. Thus, correct values for variables and a correct procedure call stack are always available to the compiled code in the subprocess. The other side of the coin is that with this invariant, interpreted code must use a system call to access variable values. We can relax this invariant and allow caching of some parts of the program state in the parent to improve efficiency. This will not cause any problems as long as the state of the child process is restored before any compiled code is

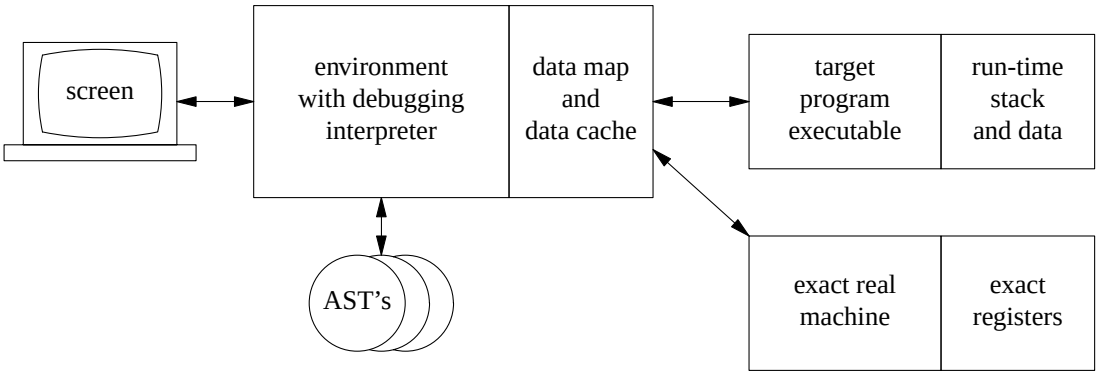

Figure 4: Organization of the execution monitor

executed. Furthermore, data-flow information provides a description of precisely those values in the cache that need to be written through to the debugged process on entry to compiled code. The data-flow information also describes those values in the cache that must be refreshed after execution of the compiled code.

To handle the control flow interfaces between interpreted and compiled code, we must first solve the problem of *detecting* the change and then take action to ensure that the invariant is maintained. Detecting the interfaces is trivial in interpreted code. In compiled code, we need only set breakpoints at the points of interface. If we require that these be at statement boundaries, this poses no special problems. The hardest part of maintaining the invariant involves updating the procedure call stack in the child when a call or return is made. In fact, because of the invariant, we must also be concerned with calls from interpreted code to interpreted code. The tables below summarize what needs to be done on calls to and returns from a subprogram. In addition, Figure 5 shows an example of the changes in control flow on a call from compiled code to interpreted code.

call	to compiled	to interpreted
from compiled	*nothing*	A • break at first executable statement and start interpreting
from interpreted	B • push parameters & return address on stack in child • resume execution in compiled prologue code	• do B then A

return	from compiled	from interpreted
to compiled	*nothing*	C • resume execution at compiled epilogue code
to interpreted	D • trap on RETURN machine instruction • pop parameters • resume interpreter	• do C then D

The interpreter is able to implement operations on variables of type exact precision by communicating with the exact arithmetic server, running locally or on another machine on the network. The server program stores the exact representations of the numbers. It also accepts requests to initialize the numbers, combine them with standard arithmetic operations, and return printable representations of their values.

In order to maintain the invariant that requires the values of variables to be stored in the child process, after every assignment to an exact variable, the interpreter requests a printable representation of the variable from the server, and then converts that to a double precision value, which it stores in the child[4]. Unfortunately, because exact values are computed in a demand-driven fashion, this ''display'' operation which produces a printable representation is the costliest of all exact operations. As described above, the existence of interprocedural data-flow information will allow us to limit display operations to only those that are necessary because of actual references to the values in output statements or compiled code.

In order to single-step the program counter in reverse we need to keep track of all the states that the program enters. One way to achieve this is to make sure that each change to the program state can be undone. To reverse program execution we need to be able to reverse each different kind of statement:

- If successive values assigned to a variable are kept in a push-down store, then an assignment statement can be reversed by popping from the stack.

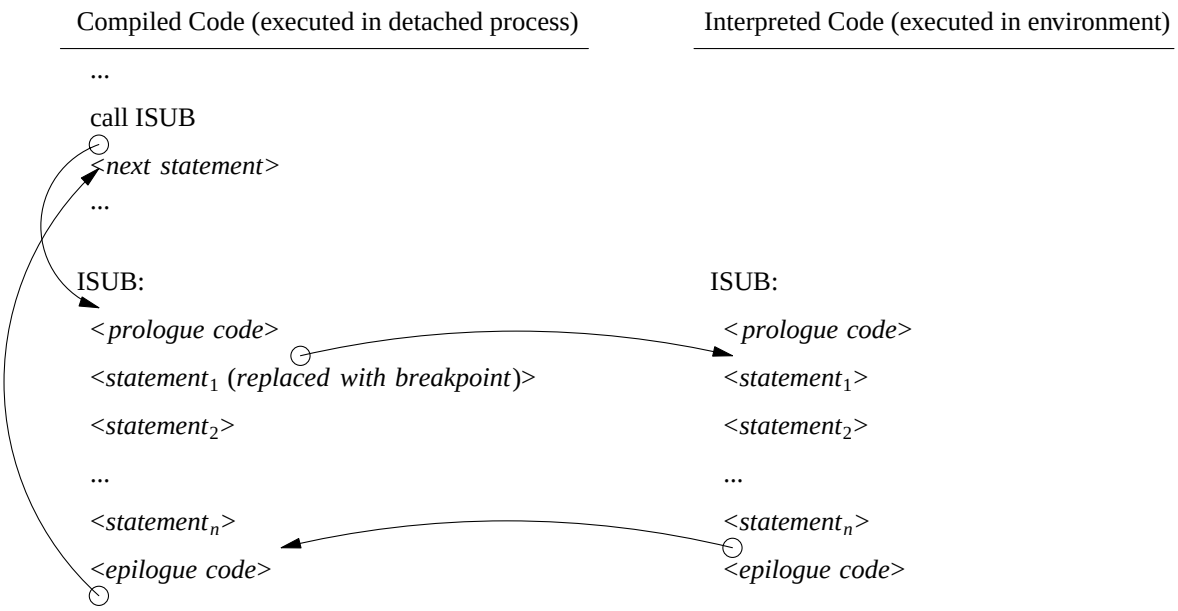

Figure 5: Flow of control during call from compiled code to interpreted code

[4] The double precision **values are used only by compiled code. The interpreter continues to use the values stored on the exact machine.**

- To reverse a branch, we need to keep track of where execution came from at all branch targets (both explicit and implicit). Again, this needs to be done in a last-in, first-out fashion.
- Subroutine or function calls to compiled code are similar to assignment statements, except that instead of a single scalar value being saved, we will need to save all variables that are potentially modified by the compiled code. In the absence of interprocedural data-flow information, this amounts to saving all parameters, static variables, and globally accessible variables (*e.g.* those in common in a Fortran program). With the data-flow information in IR^n, however, we have a more precise description of what will need to be checkpointed.
- I/O statements are perhaps the most difficult to handle efficiently. Stream I/O can be handled fairly simply by keeping counts of how much input or output was performed by a statement. Non-stream output could be handled as an assignment statement on a very large array, although this has obvious efficiency problems on large files.

5. Whole Program Analysis and Optimization

Numerical programmers have long demanded that compilers produce efficient code. This is particularly true in the FORTRAN realm, where programmers have come to expect that their existing ''dusty decks'' will be translated, automatically, to take maximal advantage of new machines and new architectures. This attitude presents a major challenge to designers of optimizing compilers. Techniques in intraprocedural optimization have reached a fairly mature state. To further improve the efficiency of compiled code will require an optimizer that considers the whole program, planning optimizations based on program-wide information. This is a natural evolutionary step in the development of optimizers, which have progressed from statement-by-statment code improvement, through techniques that handled sequences of statements, to global optimizations that take an entire subroutine into account.

In IR^n we are building an interprocedural optimizer that analyzes the entire program and plans optimization from a whole-program perspective. Conradi has estimated that aggressively pursuing such an approach can result in a speed improvement of twenty percent or more [4].

In Section 2, we saw how the tools in IR^n cooperate to provide the information necessary to perform interprocedural analysis and optimization. However, this introduces a new problem. Once the compiler uses interprocedural information as a basis for compile-time decisions, it introduces subtle and complex compilation dependences between the procedures in the program. To see this, consider two procedures *a* and *b* that are components of a single program. In compiling *a*, the compiler applies optimizations that it knows are safe, using interprocedural facts gathered from analyzing the whole program. If the user makes an editing change to *b* that adds some fact to one of its interprocedural data-flow sets, and the previous compilation of *a* relied on the absence of the fact to justify the safety of some optimizing transformation, then the change makes that transformation unsafe, retroactively[5]. Thus, if the compiler uses interprocedural facts as a basis for compile time decisions, it must have the ability to later retract the decision in response to editing changes that invalidate the correctness of the transformed program.

In a traditional compiler, the only way to achieve this is by compiling the whole program at once—an expensive proposition that removes the benefits of separate compilation. In the IR^n compilation system, we track the compilation dependences introduced by the use of interprocedural information to limit the amount of analysis and compilation required to return the program to a state consistent with its source code. This ''damage control'' strategy allows the compilation system to retain some of the benefits of separate compilation while aggressively using interprocedural information to improve the efficiency of compiled code. This approach should make the system practical for large programs.

To allow it to efficiently perform whole program optimization, the IR^n compiler has been divided into two parts: a *program compiler* that handles the interprocedural aspects of the compilation process and a

[5] Of course, the analogous case where a fact is deleted and the compiler relied on its presence in a previous compilation requires similar treatment.

module compiler that performs intraprocedural optimization. The program compiler is responsible for building a consistent executable for a program, computing interprocedural information for use by the module compiler, and determining which procedures should be recompiled. The program compiler also determines where interprocedural optimizations should be performed. The module compiler produces optimized object code for a single module based on information provided by the program compiler about the context program and its own analysis of the module source code. An artifact of this compilation strategy is that the code generated by the module compiler is a function of the program in which the module is incorporated as well as its own source. Hence, the object version cannot be safely re-used in other programs.

5.1. Program Compiler

The program compiler is assigned the task of directing the compilation of a program and computing information about the program which other tools in the environment can use. Since the programs being compiled may be large, efficiency is a major concern. This is particularly important when the program compiler is recompiling an existing program that has been slightly modified. To recompile such a program, it employs efficient algorithms for computing the interprocedural information. (If possible, an incremental update is applied.) It next performs analysis aimed at limiting the number of procedures that must be recompiled. Finally, it selects interprocedural optimizations for the module compiler to perform on the procedures slated for recompilation. These three phases of program analysis and optimization are described below.

Interprocedural Information

Before a program can be compiled, interprocedural information for the program must either be derived or updated. The program compiler is the tool that actually computes this information, so its first task when compiling a program is to perform interprocedural analysis on the program. It uses initial information provided by the module editor, the call graph components provided by the composition editor, and any interprocedural information for the program computed by previous compilations. It stores the results of its analysis in the database. Interprocedural summary sets, like the MOD sets described in Section 2.3, are one type of information that it produces. Interprocedural constants are another type of information computed.

An interprocedural constant is a variable that has a known constant value on all invocations of a procedure in a given program. The process of discovering interprocedural constants and their values is known as *interprocedural constant propagation*. These constants may provide valuable information about loop strides and bounds that can improve the compiler's ability to optimize the whole program. This is often the case in large programs that call library routines. Unfortunately, the precise solution to the constant propagation problem, whether interprocedural or intraprocedural, is not computable. In fact, even the standard approximations solved intraprocedurally are co-NP complete in the presence of aliasing between call-by-reference formal parameters and global variables [7]. Fortunately, it is possible to find approximate solutions to this problem, solutions that contain interprocedural constants that are valuable for optimization.

In $\mathbb{R}^n$ we have developed a method for computing interprocedural constants [5] in which the creation and transmission of constants within a procedure is modeled by a function that is constructed by the source editor. The interprocedural analysis phase invokes these functions at program compilation time to model the flow of constants between procedures.

Recompilation

To make the $\mathbb{R}^n$ compilation system practical for large programs, we have tried to retain some of the benefits of separate compilation. Recall that using interprocedural information as a basis for optimization introduces subtle compilation dependences between the procedures in a program. In $\mathbb{R}^n$, the compiler analyzes these dependences to predict how source code changes can affect the correctness of previously compiled code. This information, in turn, allows the compiler to determine which procedures must be recompiled to insure consistency in the program after an editing change [5].

After the program compiler updates the interprocedural information for a program, it constructs a list of the procedures requiring recompilation. It initializes the list to contain all those procedures that must be recompiled because of direct editing changes. This list is actually produced by the two editors. If the

initial list does not contain all of the procedures in the program, the program compiler then examines each procedure where interprocedural sets have changed to see if it must be recompiled because of changes in its interprocedural sets.

All of our techniques for recompilation analysis apply the same test to determine when a procedure must be recompiled. The test compares the current interprocedural information against *annotation sets*. These sets contain those interprocedural facts that can be true without invalidating the procedure's previous compilation. The various methods that we have proposed differ in the precision with which they assign values to the annotation sets.

Interprocedural Optimization

The program compiler examines the whole program to develop a plan for applying interprocedural optimizations. Some parts of the plan are obvious: for example, any constants discovered during the interprocedural analysis will be folded into the individual procedures where they hold. Other parts of the plan require explicit and complex decision processes, like the decision over where to perform inline substitution.

The approach we are pursuing is implemented in the program compiler. The program compiler constructs the call graph, gathers information about each procedure, and computes interprocedural data-flow information. Next, it competitively evaluates the possible transformations, applying those with the largest estimated improvement. Finally, it invokes the module compiler on individual modules, passing it directives to guide the optimization process. By evaluating on a competitive basis the opportunities for applying these transformations, the program compiler directs the effort of the module compiler into those areas that are most likely to pay off, from the perspective of the whole program.

5.2. Module Compiler

While the program compiler effectively directs the whole program optimization by providing interprocedural information and interprocedural optimization directives, the module compiler actually implements these optimizations. It does this both indirectly, by using interprocedural information to improve the precision of its intraprocedural information, and directly, by applying the interprocedural transformations as directed.

The module compiler uses information about side effects of procedure calls to improve the precision of the intraprocedural data-flow information around the call sites. The increased precision of the data-flow information leads to an improvement in the module compiler's ability to optimize procedures. For instance, consider performing common subexpression elimination. To determine which expressions are eligible for common subexpression elimination at a given point in a procedure, it is necessary to compute which expressions are *available*. An available expression is an expression that was computed and not subsequently changed along any path leading to the point of interest in the program.

If, when computing the available expressions, the compiler encounters a procedure call, it is forced to make conservative assumptions about the side effects of invoking that procedure. In the absence of interprocedural side effect information, the compiler must assume that any variables that were visible to the called procedure may have been changed by the procedure call. When any of these variables appear in an expression that is considered available before the procedure call, the expression must be marked as unavailable because its value may have changed as a result of the procedure invocation. If interprocedural information is present, the compiler need only invalidate those expressions that rely on a variable contained in the MOD set for the call site. As a result, expressions that contain only variables that cannot be modified are still available after the call. This leads to a larger set of available expressions and, potentially, allows additional common subexpressions to be eliminated. This improvement speeds up the resulting executable.

The module compiler uses information about aliases between formal parameters and global variables to improve intraprocedural optimization. For example, in the absence of interprocedural aliasing information, the compiler cannot determine whether or not pairs of global variables or formal parameters occupy disjoint storage locations. To avoid incorrect code, the compiler cannot store such variables in registers. If the compiler knows that the aliases do not exist, it can store these variables in registers, with a resulting improvement in the speed of the object code. Knowledge of aliasing patterns comes into play in nearly

every optimization decision.

Similarly, the module compiler uses information about interprocedural constants as input to its own intraprocedural constant propagation analysis. This allows the intraprocedural analyzer to recognize constants that are are inherited from the calling environment. In practice, important information like array dimensions and loop strides are likely to be detected by the interprocedural constant propagation; this information can play an important role in purely intraprocedural optimizations, like loop unrolling.

Information about interprocedural side effects not only helps produce better code, it can also reduce the amount of analysis required in the module compiler. Our experience with PFC shows that the number of use-definition chains constructed by the compiler can be drastically reduced through the use of interprocedural analysis. This occurs because the compiler is able to use the interprocedural side effects information at a call site to determine which variables visible to the called procedure have been used or redefined as a result of invoking the procedure. The compiler then is not forced to make conservative assumptions about the side effects of the procedure call and create unnecessary use-definition chains.

6. Support for Parallel Programming

In the next phase of research we plan to evolve $\mathbb{R}^n$ to support parallel programming. This is a natural extension of our work on automatic detection of parallelism and software for supercomputers [1]. We begin with a summary of work to date on the supercomputer software project. Then we present a conceptual picture of the parallel programming environment we envision, along with a discussion of implementation strategy.

6.1. PFC and PTOOL

Since 1978, a group at Rice has been conducting an active program of research in software for vector and parallel supercomputers. This research has led to two important experimental systems: PFC and PTOOL.

PFC (Parallel FORTRAN Converter) is a system that automatically vectorizes FORTRAN programs by performing a sophisticated analysis of *dependences*. A dependence exists between two statements if one statement can store into a location that is later accessed by the second statement. Although most optimizing compilers analyze dependences in a program, they use a particularly naive treatment of arrays. Vectorization systems employ a much more powerful analysis that is fairly effective in dealing with subscripted references in loops [6, 2].

To illustrate the analysis of dependences, consider the code fragment below (expressed in FORTRAN 8x notation):

```
DO I = 1, N
     X(I)   = Y(I) + C
     Y(I+1) = X(I) + A(I)
REPEAT
```

The second statement depends on the first by virtue of the value stored in X(I) and then loaded on the same iteration. This dependence is called a *loop-independent dependence* because it forces a particular ordering among the statements in the body of the loop. The first statement depends upon the second one, because of the value stored into Y(I+1) *on one iteration and loaded from Y(I)* on the next one, when I *has been incremented. This dependence is said to be carried* by the loop because it forces an ordering among the different iterations of that loop. To put it another way, the existence of a carried dependence prevents the separate iterations from being run completely in parallel unless some synchronization is inserted to insure that the proper values of Y *are used on each iteration. Such a synchronization effectively serializes the loop.*

Research on PFC has concentrated on finding vectorization algorithms efficient enough for use in a compiler. An indication of its success is that PFC served as the prototype for the IBM VS FORTRAN Version 2 vectorizing compiler for the 3090 Vector Feature [9], which achieves excellent results while remaining reason-

ably efficient[6].

Since completion of the original PFC in 1982, research has continued in three areas. First, algorithms to perform interprocedural analysis on whole programs have been added. The approach used is based upon experimentation with an implementation of the algorithms developed for $\mathbb{R}^n$. The aim of this work is to determine the impact of whole program knowledge on vectorization and parallelization. Currently, PFC analyzes side effects of procedure calls, aliasing patterns, and constants propagated across procedure boundaries.

Second, we have been developing parallel programming tools that use PFC's analysis phase. Our first effort, called PTOOL, is an interactive adviser designed to assist in the prevention of errors arising from unintentional data sharing or unforeseen load-store orders for shared data in parallel programs. PTOOL is really a sophisticated browser for an interstatement-dependence database created by PFC. It permits the user to select a loop in a sequential FORTRAN program and ask whether or not the iterations may be run in parallel. If the answer is ''no'', PTOOL will display all the carried dependences that would prevent parallelization.

While PTOOL is extremely helpful for identifying loops that can be parallelized, it provides no assistance in the generation of parallel code. A third project is investigating automatic generation of code for multiprocessors. This has led to another derivative of PFC, called PFC+, that not only recognizes loops whose iterations can be run in parallel but also performs sophisticated transformations, such as loop interchange, loop alignment, and code replication to enhance the parallelism available [2].

PFC now contains approximately 95,000 lines of PL/I code and runs on an IBM 4341. PTOOL is already in use at Los Alamos National Laboratory and is scheduled to be installed at Livermore and Argonne. Both PTOOL and PFC+ will be installed at the Cornell Theory Center. These systems represent a significant research resource because they are relatively easy to modify. This makes it possible to add and evaluate new transformation methods rapidly. In addition, PTOOL has value as an educational tool—it is now used to teach compiler students about dependences in programs.

6.2. A Parallel Programming Environment

Building on $\mathbb{R}^n$ and our experience gained with PFC and PTOOL, we are enhancing $\mathbb{R}^n$ with extensive support for parallel programming. This involves both continued work on the fundamental tools of the environment and the addition of entirely new features.

The programming environment we envision will provide support for an extended version of FORTRAN that we call Parallel FORTRAN. This language will be similar to the emerging class of macro libraries that provide parallel programming support in FORTRAN. Programming in the environment would take place in two phases. In the first phase, the programmer would construct a program that is correct in a version of the language that has a single, non-parallel execution schedule. This is similar to writing the program in sequential FORTRAN. In the second phase, the programmer would evolve the program into an efficient parallel program by eliminating dependences that prohibit a highly parallel schedule.

It is in this second phase that the parallel programming tools would be used. The environment will support a transformational style of programming in which the system provides automatic assistance. In this style, the user will ask the system where the performance bottlenecks are and can ask for suggestions or make a change. If the request is for suggestions, the system will reach into its library of transformations to find the ones that will improve the parallelism without changing the meaning from the fixed-schedule version of the program. The user may then select a transformation to be performed. If, on the other hand, the user makes a manual change, the system will attempt to prove that it preserves the meaning of the fixed-schedule program. If it does not succeed, it then records the change as one to be tested later.

---

[6] The IBM compiler, based on an early version of the PFC system, is quite effective at vectorization and requires only a modest increase in compile time over the scalar VS FORTRAN compiler with optimization.

To support this style of programming, many changes must be made to the existing software. First, we must have ways of incrementally analyzing dependence and making transformations in the source editor. Second, we must have ways of analyzing a program in enough depth to verify that a change preserves meaning. Finally, we must have ways to use information about questionable changes in the testing and debugging process. In addition, a good parallel programming environment must include support for the *design* of parallel programs. The following subsections discuss the ways we propose to address these requirements.

Dependence Analysis in a Structure Editor

In the automatic detection of parallelism, the key analytical information used by a compiler is a *dependence graph* [6, 2] for the procedure. Our experience with the PTOOL system has convinced us of the value of presenting dependence information to explain why particular loops will not run in parallel. Unfortunately, because PTOOL relies on a batch analysis of the program, users have found the delay between typing a proposed change into the source and getting feedback about its impact to be frustrating. To build a more responsive parallel programming advisor, we plan to maintain a current dependence graph in the structure editor for FORTRAN, incrementally rebuilding dependences whenever the source is modified. In addition, we will provide automated support for transformations that increase the parallelism available in a FORTRAN program. The resulting editor should make experimentation with parallel programming reasonably easy.

Improvements to Interprocedural Analysis

The analysis of interprocedural side effects is an important component of the system because accurate information about interprocedural data flow is essential if we are to verify the manual transformation steps discussed earlier. Our experience with interprocedural data-flow analysis in both IR^n and PFC has convinced us that improvements are needed in both the theory and the implementation. Two particular problems present themselves: the need for more precise treatment of arrays and the need for incremental updating techniques. We intend to further our work on the theoretical front while at the same time pursuing implementations in the new parallel programming environment.

Our experience with using interprocedural summary information in a working system for detecting parallelism has shown that the granularity of conventional summary information is too coarse to allow effective detection of parallelism in loops that contain call sites. The problem is that the current analysis treats whole arrays as single units. Thus, it is able to determine whether an array is modified somewhere, but not whether it is modified in only a single column or row. This limitation is disastrous for parallelization because the most effective way to parallelize a loop is through data decomposition, in which each parallel iteration works on a different subsection of a given array.

Fortunately, a generalization of the approach currently used to solve interprocedural data-flow analysis problems can be used to develop more precise information about side effects. The standard algorithms for interprocedural analysis can be extended in a natural way to deal with a lattice of *regular sections* of an array. Each regular section represents a common pattern of access to a subarray such as a single element, a whole row or a whole column. If the regular section lattice is carefully designed, the analysis will produce information precise enough to do a good job of parallel decomposition without becoming unacceptably inefficient. Nevertheless, we expect that regular section analysis will require substantially more computing resources than the current techniques. This will increase the desirability of incremental methods for updating interprocedural information in response to a program change. In addressing this problem, it is important to evaluate the tradeoff between incremental and parallel evaluation methods, since it seems likely that batch algorithms will be more amenable to parallel execution than incremental techniques.

Parallel Debugging

Currently, the IR^n execution monitor supports debugging a sequential program on the local machine. We plan to enhance it to support debugging programs executing on remote machines and to support the management of parallel background processes, using an approach similar to those employed by execution monitor was implemented in a manner that should facilitate extending it to deal with processes on remote machines. All that is required is to produce distributed version of a UNIX system call that permits a foreground process to inspect and change locations in the address space of a background process. The

extension to the management of several processes is also straightforward, but requires more work. This is the sort of approach being used in the emerging class of parallel UNIX debuggers, such as those offered by Sequent and Alliant for use with their machines.

However, what may well be the most difficult problem in debugging code for shared memory multiprocessors still remains: it is very hard to recreate with a debugger the sequence of events that led up to an error of unintentional data sharing. To address this problem, we plan to use the information from the environment discovered during analysis and compilation to provide clues to the location of errors at run time.

For example, if an incorrect value is detected in a parallel program at a point where the sequentially scheduled version produced the correct value, the debugging system would be invoked. It would trace back to locations inside parallel regions that might have contributed to the computation of the erroneous value. Then it would use *adversary scheduling*, a technique that employs dependence analysis to pick schedules likely to lead to errors, to help locate the problem. In other words, the debugger would step the processors in a sequence that static analysis suggests is likely to produce a different answer from the standard sequential schedule.

Whole Program Planning

In a system for automatically decomposing a large program for parallel execution, optimizing transformations should be planned from a global perspective. For example, to make effective use of inline substitution, the compiling system must plan the optimizing transformations, including parallelization, for the whole program. As outlined in Section 5, planning optimizations on a whole program basis is the responsibility of the program compiler.

The planning phase of the program compiler is the natural place to identify larger granularity parallelism. The program compiler will select the loops to be run in parallel, considering loop nests that span procedure call boundaries. It will use this information as input to its decision process that selects procedure calls for inline substitution. The program compiler will include this information in the directives passed to the module compiler. Thus, the module compiler will know which routines contain parallel loops and which ones will be run from inside parallel loops.

7. Summary and Conclusions

The IR^n project has broken new ground in the use of components of a software development environment to support the compilation process, while developing a number of new ways to build those component tools. The research has contributed new algorithms for interprocedural analysis and methods for managing the recompilation problem. It has also explored an approach to language-oriented editing that combines the advantages of text and structure editing and a debugging system that permits the mixed execution of compiled and interpreted routines.

The current implementation, scheduled for distribution to a number of sites this fall, consists of approximately 105,000 lines of C and runs on both the IBM RT PC and SUN workstations. Over the next several years the implementation will evolve as scientific programming evolves, with the principal change being the incorporation of advanced features for the support of parallel programming.

8. Acknowledgments

Many people have contributed to the IR^n project at Rice. They have provided us with a marvelous vehicle for conducting our research. The editors and referees provided valuable insights on the form and content of this paper. To all these people go our heartfelt thanks.

9. References

[1] J.R. Allen, D. Callahan and K. Kennedy, ‘‘Automatic Decomposition of Scientific Programs for Parallel Execution,’’ *Conf. Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1987, pp. 63−76.

[2] J.R. Allen and K. Kennedy, ‘‘Automatic Transformation of FORTRAN Programs to Vector Form,’’ *ACM TOPLAS*, Vol. 9, No. 4, Oct. 1987.

[3] H-J. Boehm, R.S. Cartwright, M.J. O’Donnell, and M.S. Riggle, ‘‘Exact Real Arithmetic: A Case Study in Higher Order Programming,’’ *Proc. of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pp. 162−173.

[4] R. Conradi, ‘‘Interprocedural Optimization of Object Code,’’ TR 25/83, Division of Computer Science, University of Trondheim, Trondheim-NTH, Norway, 1983.

[5] K.D. Cooper, K. Kennedy, and L. Torczon, ‘‘The Impact of Interprocedural Analysis and Optimization in the IR^n Programming Environment,’’ *ACM TOPLAS*, Vol. 8, No. 4, October, 1986, pp. 491−523.

[6] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, ‘‘Compiler Transformation of Dependence Graphs,’’ *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, pp. 207−218.

[7] E.W. Myers, ‘‘A Precise Inter-procedural Data Flow Algorithm,’’ *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1981, pp. 219−230.

[8] S. Richardson and M. Ganapathi, ‘‘Interprocedural Analysis: A Bibliography,’’ SIGPLAN Notices, 22(6), Jun. 1987, pp. 12−17.

[9] R.G. Scarborough and H.G. Kolsky, ‘‘A Vectorizing FORTRAN Compiler,’’ *IBM J. Research and Development* Vol. 30, No. 2, Mar. 1986, pp. 163−171.

[10] R. Triolet, F. Irigoin, and P. Feautrier, ‘‘Direct Parallelization of Call Statements,’’ *Proc. SIGPLAN ‘86 Symposium on Compiler Construction, SIGPLAN Notices* Vol. 21, No. 7, Jul. 1986, pp. 176−185.