Optimization of Compiled Code in the Programming Environment

Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251

Abstract

Despite substantial progress in the theory of interprocedural data flow analysis, few practical compiling systems have applied such techniques to produce more efficient object programs. For a compiler to perform interprocedural analysis and optimization, it needs access to information about the entire program during the compilation of each procedure in the program. Imbedding the compiler in a programming environment simplifies the task of efficiently providing this support. The compiling system in the programming environment has been designed to optimize whole programs. This paper discusses the design of that compiler. It sketches the algorithms used to perform interprocedural analysis and discusses issues involved in constructing a practical interprocedural optimizer.

1. Introduction

The programming environment is an integrated collection of tools to assist programmers in building numerical software in Fortran. Although, sophisticated systems exist to support programmers who write in high level languages that are popular among computer scientists, like Lisp [21], C [13], Mesa [17], PL/I [22], and Smalltalk 80 [12], little effort has been expended to provide similar support for Fortran, the language used by most numerical programmers. The programming environment [14] is intended to fill this void.

Since Fortran users have come to expect compilers that generate excellent code, proper concern for efficiency is important to the programming environment project. Indeed, one of its primary goals is to mount a concerted attack on the problems of performing interprocedural analysis and optimization in a compiling system. Few commercial optimizing compilers employ interprocedural techniques because the cost of gathering the requisite information is too great in a traditional batch compiler.[1] Computing the side effects of a procedure call requires detailed knowledge of the internals of both the called procedure and any procedures invoked either directly or indirectly from it. Thus, the compiler potentially needs information about the internals of every procedure to determine the side effects of procedure calls, even separately compiled procedures. Without the structure of an external database, gathering this information would require examining the source of every procedure in the program - an expensive process. This is particularly unfortunate since the primary goal of separate compilation is to reduce the amount of recompilation required in response to incremental changes to the program.

A software development environment like changes the compilation process enough to make computing such information palatable. Since all modules are developed and all programs are

This research has been supported by IBM Corporation, and by the NSF through grants MCS 81-04006, MCS 81-21844, and MCS 83-03638.

[1] Two notable exceptions are IBM's PL/I Optimizing Compiler and Xerox's Interlisp-D system. The former computes side effects for a set of procedures compiled as a unit, but makes no attempt to pass such information between independent compilations [19]. The latter provides an interactive query facility that computes interprocedural information as an aid to understanding and debugging programs [16].

defined using tools of the environment, these tools can cooperate to record the information necessary to do a good job of interprocedural analysis and optimization. For example, the source editor can record information about the uses and definitions of variables that the environment needs to compute the side effects of each procedure in a program. If this information is saved in a central database, it will be available for use by the optimizing compiler on demand. Furthermore, since the only mechanism for changing modules or programs is through the tools provided by the environment, the environment can assure that the information the compiler finds in the database always reflects the current state of the program and its procedures.

This paper presents an overview of the optimization process as it is designed into the environment. From the discussion it should become clear that the concern for interprocedural analysis and optimization has profoundly influenced the design of nearly every component of the environment.

2. The Programming Environment

The programming environment is designed to run on a high-performance workstation with bit-mapped screen and mouse. It is currently being developed on Sun Microsystems workstations, but the implementation has been crafted to make it easy to move to other machines with similar capabilities. The environment is organized as a collection of command processors running cooperatively under a single *monitor* program. The monitor controls interactions between these command processors and provides primitives for handling the mouse, bit-mapped screen, keyboard, menus, and windows.

The command processors record and use information in the *database*. The database is used as a repository for information about programs and procedures in the environment. In particular, it provides a convenient mechanism for communication between tools. Two types of objects are found within the database: modules and programs. A *module* is an editable unit containing one or more Fortran subprograms. A *program* is a collection of modules that can be executed. A single module may be incorporated into several programs. The externally known names defined within a module are *entries*. Entries usually represent the names of subroutines that can be called.

The environment provides tools for dealing with both modules and programs. The principal components of the current environment are the module editor, the program editor, and an execution manager.[2] An interactive debugger and an optimizing compiler are under development.

2.1. Module Editor

The *module editor*, or *intelligent Fortran editor*, combines a knowledge of Fortran together with access to the data base in order to simplify the programming process. The module editor helps the programmer enter syntactically correct programs by providing commands that generate templates for the major language constructs. For example, to insert a DO-loop, the programmer need only invoke the DO-loop command and the cursor is replaced by a DO-loop template with place markers in the positions where further text should be entered. The editor obviates the need for a parser by directly constructing an abstract representation of the program. All components of the environment use this abstract syntax tree as the standard program representation.

The editor makes use of information stored in the project data base to help the programmer construct subprograms that are consistent with the program being developed. For example, when a programmer wishes to insert a call to an external subroutine, the editor queries the data base to retrieve the parameter specifications for the called routine and uses these to prompt the user for parameters. From the user's viewpoint, the editor automatically inserts a template for each actual parameter that explicitly identifies its name and type.

2.2. Program Editor

The *program editor*, or *composition editor*, assists the user in defining a consistent and complete version of a program. It helps the user specify the collection of module-versions that make up a version of the program and ensures that a definition of each entry point used in the program-version is included in this collection. When the desired specification differs only slightly from an existing one, the composition editor simplifies the

[2]The execution manager uses standard tools to compile and execute a program. We do not discuss it in this paper.

specification process by providing the ability to edit an existing program-version. Additionally, it has a library search mechanism to allow an automatic search to complete the composition.

2.3. Interpretive Debugger

The *interpretive debugger* will enable the programmer to step through parts of a given program, allowing him to monitor and interrupt execution. By cooperating with the compiler, the interpreter will be able to handle hybrid execution of a program consisting of both compiled and interpreted modules. Thus, during debugging, the programmer can execute stable module-versions in a compiled form while interpreting modules under development. This makes interpretive debugging a practical tool for large programs, by allowing control to pass quickly through those parts of the program that are uninteresting to the debugging process and bringing the full power of the interpreter to bear on those parts of the program where it is needed. We also plan to have the debugger support reversible execution.

2.4. Optimizing Compiler

In addition to tools designed to help the numerical programmer develop, test, and maintain programs, any serious programming environment for Fortran must pay careful attention to the efficiency of compiled code if it is to satisfy the numerical community's strict requirements for efficient execution. Therefore, the programming environment will include an ambitious optimizing compiler. Because much of the work of a traditional compiler's front end is performed in the environment's module and program editors, its ''compiler'' consists primarily of an optimizer and a code generator. In addition to the standard techniques from global optimization, the compiler will use interprocedural information and attempt interprocedural optimizations. In order to effectively collect and use interprocedural information, we have divided the optimizing compiler into a pair of compilers, the module compiler and the program compiler. The motivation for this subdivision and the responsibilities of the program compiler are the subject of much of the rest of this paper.

3. Optimizing Whole Programs

Compiler optimization researchers have long believed that interprocedural effects are the last remaining major source of inefficiency in languages, like Fortran, with mature optimizing compilers. Why then do so few compilers attempt interprocedural analysis or optimization? The reason is cost. To effectively deal with interprocedural effects, the compiler needs access to information from every procedure in the program. In other words, to compile a single module, the system would need to read and analyze all the modules contained in the program. Since the number of modules can grow quite large, the cost in compilation time would far outweigh the benefits in improved optimization. This is particularly vexing in Fortran, because the goal of separate compilation is to reduce the amount of work that must be done to prepare a large program for execution after an editing change.

In the context of a sophisticated programming environment like , a more palatable solution is possible. If the programmer exclusively uses the tools of the environment to prepare his program for execution, we can have those tools *spy* on him record in the data base the information needed to support interprocedural data flow analysis. The environment can then perform interprocedural data flow analysis over the call graph[3] of the program and save the information in the database, where the module compiler can reference it, on demand, to get a precise specification of the side effects of a called procedure. This permits the module compiler to build a more accurate representation of data flow *within* a module.

An environment also makes it possible to perform *interprocedural optimizations*, like *in-line substitution* and *linkage tailoring*, that improve code across procedure boundaries. Such optimizations are possible because the environment has access to the source text for all the component modules of a program. Furthermore, the notion of program composition supported in the environment naturally provides a degree of control over the consistency of the executable that is difficult to achieve in traditional compilation schemes.

While using interprocedural analysis and optimization can result in more efficient object code, it introduces a new difficulty: the *recompilation problem*. Whenever the compiler uses

[3]Actually, the call graph is a multi-graph, with an edge for each call site. We always refer to it as a graph for the sake of consistency.

interprocedural information as the basis for compile-time decisions in one procedure, the correctness of the resulting object code becomes a function of the state of the entire program. To see this, consider the compilation of a procedure p that invokes another procedure q. At the call site, the module compiler refers to the environment's database to determine which variables can be changed as a side effect of executing q. The information about q incorporates information about the side effects of procedures invoked, directly or indirectly, from within q; hence it may encapsulate knowledge about all of the procedures in the program. After p has been compiled, a change to some other procedure in the program may change the interprocedural information on which the compiler relied to generate correct code for p, requiring a recompilation of p.

The environment addresses the recompilation problem by performing a *recompilation analysis* [4] [24] to minimize the number of recompilations actually performed in response to an editing change. Section 7 will provide an overview of this approach.

4. Design of the Optimizer

4.1. Interprocedural Analysis

As an introduction to the subject of interprocedural data flow analysis, consider the problem of computing the interprocedural MOD side effect. For any statement s, MOD(s) is the set of all variables that might be changed as a result of executing s. Usually, MOD(s) is easy to determine. However, if s contains a procedure call, the problem is more complex. 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 assume that MOD(s) consists of all variables that are either actual parameters at the call site or global variables of the called procedure. This is the safest assumption possible in the absence of information about the called procedure.

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 an invocation of q. If we can determine GMOD(q) for each procedure q in the program, then MOD(s) for an invocation of q can be determined by seeing which variables in the calling procedure are bound to members of GMOD(q) by the call.

Notice that GMOD(q) consists of two components:

- the set IMOD(q) of variables that might be modified by statements in q other than procedure calls;
- the set of variables that might be modified as a side effect of a 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. They are, however, a function of the specific implementation, or version, of the module. Hence, the sets can be computed by the editor and stored with the version of the module which generated them. From these sets, GMOD(q) can be computed for a specific version of a program by solving a data flow problem on the call graph [5] [7]. Since the sets GMOD(q) depend on the specific call graph, they must be stored with the version of the program for which they are computed.

This observation illustrates an important aspect of the division of labor which occurs in the programming environment. In addition to parsing and type checking, the editor can perform most of the information gathering (i.e., finding all the variables used and defined in the body of the procedure) 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 in the program before any module is compiled.

4.2. Program Compiler

Within the programming environment, the responsibility for optimization of compiled code is distributed among several components of the environment, rather than remaining the sole responsibility of the optimizing module compiler as it is in more traditional systems. The module editor is responsible for producing local information about the behavior of individual procedures — for example, the procedures called and the variables used or modified. The program editor is responsible for defining the composition of the program — which modules are actually included in the program — and producing the call graph which describes how

they are related. Once interprocedural information has been computed and stored in the database the optimizing compiler can use it to produce better object code.

Absent from this discussion is the component that is responsible for truly interprocedural compilation tasks, such as propagating interprocedural information around the call graph or making decisions about interprocedural optimizations. In the environment, these tasks are the responsibility of a new tool - the *program compiler*. Thus the tasks traditionally assigned to an optimizing compiler are divided between the optimizing *module compiler*, which produces object code from the source for modules, and the program compiler.

The *program compiler* is responsible for directing the construction of an executable image of a program. Its task is threefold: to determine which procedures must be recompiled, to ensure that those compilations have the correct interprocedural information available to them, and to suggest interprocedural optimizations to the module compiler. Thus, it is a combination of an interprocedural data flow analyzer, a dependence-based program constructor like *make* [11] and its successors [23], and an optimizing advisor for the module compiler. In the role of data flow analyzer, it uses information saved by the module and program editors to compute flow insensitive interprocedural summary information [7], flow insensitive interprocedural aliasing information [6], and interprocedural constant propagation information [24]. As a program constructor, it uses a program's call graph and composition to derive dependence relationships and it uses information from the module editor, the program editor, and previous compilations to precisely determine which modules need recompilation. In the role of optimization advisor, it uses interprocedural information to make decisions about where it is profitable to perform interprocedural optimizations like constant folding and linkage tailoring.

The program compiler is the central tool in the optimization strategy for the environment. The remainder of this paper is devoted to discussing the functions it must perform and algorithms for efficiently accomplishing those functions.

4.3. The Optimization Process

The information flow in the compilation system is depicted in Figure 1. The program

Figure 1. Compilation System.

editor produces a prescription for composing the program which is used as the input for compiling the program. But the program compiler also performs interprocedural data flow analysis. To do this it requires information about the usage of variables in the various modules of which the program is composed. This information is collected and recorded in the data base by the module compiler as it edits each of the individual routines. Hence, the program compiler is invoked after the program editor and the module editor. However, it is invoked *before* any module compilations. Results of interprocedural data flow analysis are therefore available for use in the the optimization of individual modules.

As we shall see, there is also systematic feedback of information from the program compiler to the module and program editors and from the module compiler to the program compiler. These effects are primarily related to incremental

program compilation, discussed in Section 7. However, one application of the interprocedural data flow information computed in the program compiler is to perform interactive detection of data flow anomalies in the module editor. If the user revises a module by attempting to pass a constant in a parameter position that might be changed at run time the editor informs the user of the potential modification.

5. Interprocedural Optimization

The optimizing module compiler in the programming environment will capitalize on the presence of interprocedural information and module source code in the database to perform optimizations that span procedure boundaries. The goal of all interprocedural optimization is to tailor the code generated for a procedure to execute more efficiently by taking into account, at compile time, facts about the run-time environment in which the procedure will be executed. The optimizer uses stored knowledge about the calling procedure and the values of its variables at the call site, along with information about the side effects of any call sites in the procedure being compiled, to generate less general, more efficient code.

5.1. In-line Substitution

The most straightforward interprocedural optimization is in-line substitution. While this type of optimization has been discussed in the literature since the mid-1960's [10], few practical compiling systems have actually implemented it. To perform in-line substitution, the compiler simply treats the procedure body as if it were a macro definition and the call site as an instantiation of the macro. Actual parameters are used in place of formal parameters in the expanded body, and local storage for the called procedure is merged with the local storage for the calling procedure.[4] This transformation completely eliminates the overhead associated with making the procedure call. When followed by a decent global optimization pass on the expanded procedure, in-line substitution results in optimizing both procedures together, producing the best possible code for them within the given compiler.

[4]Of course, the compiler must carefully avoid name conflicts and ensure that the parameter binding semantics are preserved.

For those procedures which are called from only one place and those whose body, when compiled, is smaller than the linkage code, in-line substitution is always desirable. In the general case, however, the optimization benefits must be weighed against the increased object code size resulting from multiple copies of a single procedure.

5.2. Linkage Tailoring

Strictly speaking, in-line substitution is a type of linkage tailoring. However, there are a number of less radical forms. For example, the compiler might use *semi-closed* linkages in which some, but not all, of the code from the called procedure is moved into the calling procedure at the call site. The main advantage of doing this is to simplify the linkage code in the called procedure by tailoring it to the environment at each call site.

The program compiler will attempt to discover places where *cloning* a procedure can lead to improved optimization. When the interprocedural environment in which a procedure is to be executed differs radically between two sets of call sites, the program compiler can clone the procedure, tailoring the interface in different ways to the different call sites. To illustrate the usefulness of this technique, consider a procedure from a library that employs an option parameter to specify which of two different ways the procedure is to be used. Frequently, the option is specified by a constant parameter at the point of call. For one of the two option values, the procedure reduces to a small fragment of code suitable for in-line substitution; for the other, the reduced code is much longer. The program compiler might clone this procedure into two copies, one called from all sites in which there is a constant parameter specifying the short code fragment and the other called from the remaining sites.

Because, on the surface, it appears that cloning may cause significant space problems, it is important to understand its application. Rather than generating clones to attack specific opportunities, the compiler should generate clones to isolate uniquely bad call sites. In the case of an outstanding opportunity for improvement, in-line substitution is likely to result in greater improvement than cloning; in the case of a few disastrously bad call sites, cloning will allow the optimizer to generate general code for them while optimizing for the remaining sites. This optimization, then,

should be considered a technique for isolating problems rather than a direct means of improving optimization.

5.3. Other Techniques

Finally, a number of interesting optimization opportunities arise because the compiler operates inside a programming environment, where it has easy access to the text for each procedure and records of previous compilations. An environment can perform transformations which are difficult to implement in a traditional separate compilation system. One example is cross jumping, in which a simple pattern matching process is used to unify identical instances of procedure epilogue code. Extraneous copies of this code are replaced by branches into the remaining instance, with a concomitant saving in object code size.

6. Interprocedural Algorithms

6.1. Flow Insensitive Side Effects

The problem of computing GMOD(q) for each procedure q in the program, as described in Section 4.1, is a *flow insensitive* data flow problem. Intuitively, a data flow problem is flow insensitive if the information being computed does not depend on the flow of control in the individual procedure bodies. For example, if we assume that any path can be executed, the problem of computing MOD is flow insensitive because a variable *may be modified* if there is a statement somewhere in the procedure body that can modify it as a side effect. This can be determined by a simple scan of the procedure source, along with an appropriate analysis of the indirect effects of calls to other procedures. By contrast, the computation of the set of variables that *must be modified* as a side effect of a call is flow sensitive, because a variable is in the set only if it is modified on *every* path through the procedure. Determining this will require an analysis of control flow in the procedure.

We restrict our attention to the computation of flow insensitive side effects, since Myers has proved that the precise computation of flow sensitive side effects is intractable [18]. In addition, the flow insensitive problems have the nice property that the computation of preliminary information can be achieved without resorting to complex data flow analysis in the editor. For example, the set IMOD(q) of variables that may be modified by a procedure q is simply the set of variables mentioned on the right hand side of an assignment or in a read statement.

Although Banning [2] suggests solving flow insensitive problems by formulating them as data flow problems on the call graph, the usual fast data flow algorithms cannot be applied for technical reasons. However, as a part of our preliminary research, we have discovered a decomposition of such problems into two subproblems, each of which can be solved by fast elimination techniques. In particular, using an algorithm due to Tarjan [20] we can compute solutions to both problems for reducible call graphs with recursion in time $\mathbf{O}(E\alpha(E,N))$ where E is the number of edges in the call graph, N is the number of vertices, and α is a function related to an inverse of Ackermann's function [7]. This is the best known time bound for solution of such problems.

6.2. Aliasing

The program compiler also computes information about aliasing relationships involving formal parameters and global variables. Whenever a procedure can access a single storage location using more than one name, those names are said to be *aliases*. It is essential that the module compiler understand which names are potential aliases; in the absence of such information it must treat *every* global variable and *every* formal parameter as if they were potential aliases. This implies, for example, that their values cannot be retained in registers across statement boundaries.

To understand this, consider the following sequence of assignment statements:

```
a = 10
b = 12
c = a * b
```

In the absence of aliasing, the compiler would probably retain the values of a and b in registers, since they are referenced almost immediately after being computed. If, however, this procedure can be invoked in a manner such that a and b refer to a single storage location, this simple optimization to eliminate apparently redundant stores and loads would result in *incorrectly* assigning the value 120 to c.[5]

[5]Strictly speaking, the Fortran standard makes aliases illegal in the sense that the compiler need not pay attention to the problem: if the programmer uses aliases, he may encounter unexpected results. However, many compilers handle them correctly, creating an expectation of such handling in programmers

To provide the module compiler with a more precise understanding of which variables are potentially aliased, the program compiler solves an interprocedural data flow analysis problem. For each procedure p, it computes a set ALIAS(p) containing all the aliases that can hold on entrance to p. Each potential alias is represented by a pair <x,y>. The presence of the pair <x,y> in ALIAS(p) implies that some sequence of call sites in the program results in an invocation of p with both x and y as names for a single storage location.

The aliasing problem can be solved using any of several algorithms from global data flow analysis [6]. In particular, the $\mathbf{O}(E\alpha(E,N))$ technique developed for computing flow insensitive summary information can be used, albeit with a slightly larger constant than that found in the summary computation.

6.3. Constant Propagation

Even though the general constant propagation problem is undecidable [15] and the usual approximate constant propagation problem is flow sensitive, hence intractable in an interprocedural setting [18], it is important to propagate some constants interprocedurally to compensate for the widespread use of constant parameters in programs that incorporate routines from common libraries like LINPACK [9]. For example, in LINPACK's BLAS library every routine has a parameter to hold the stride of elements in a vector passed to the routine. In practice this parameter is almost always equal to the constant 1, a fact that would be enormously useful in optimizing routines from LINPACK, as Dongarra has shown [8].

In her dissertation, Torczon presents a set of algorithms that compute useful approximations to the complete set of interprocedural constants [24]. These algorithms differ in the sets of constants they detect, in their expected running times, and in their suitability for application in an incremental framework. The optimizer will make use of one of these techniques, based on the notion of *jump functions*. A jump function approximates the manner in which constants are propagated through procedures, using techniques which are much less expensive than performing global constant propagation. Most of the work involved in computing jump functions can be pushed back into the module editor, greatly decreasing the cost of repetitive evaluations of the function during the actual propagation phase.

By using jump functions and by having the editor determine which parameters are constant at the point of call to a procedure from within a module being edited, we can compute a useful approximation to the set of interprocedural constants [24]. Our method identifies a set of variables which are provably constant on entry to each procedure in the program. While this set is a subset of the provably constant variables in the program, it does include many of the constants which are interesting to optimization. In particular, it detects the loop strides passed around inside LINPACK, our example above. Given this information, the module compiler can automatically generate code that is identical to the hand optimized code used in Dongarra's study.

7. Incremental Program Compilation

Although the module compiler is a monolithic program, recompiling from scratch each time it is invoked, the program compiler must be able to cope with incremental change in a reasonably efficient manner. Otherwise, editing a single module in a program could lead to recompilation of every module in the program, an expensive process.

Hence, the program compiler must have ways of limiting the amount of work that is required in response to changes in the behavior of individual modules and changes to the composition of the program. We subdivide these issues into two categories — incremental updating of interprocedural information and limiting module recompilation after change.

7.1. Incremental Update

Once interprocedural information has been computed for a program, it must be updated in response to each change of a module incorporated in the program. To handle this problem, the editor must record any changes in the interprocedural sets, such as IMOD, each time a module is changed. When the program compiler is invoked, it must find all the modules for which the data flow information has changed and propagate the effects of those changes around the call graph until the values stabilize. Since program call graphs can be quite large and since the effect of a small change

---

who use them. Therefore, the environment will generate correct code in the presence of aliases.

```
begin
  worklist ← {p};
  while worklist ≠ ∅ begin
    remove arbitrary element q from worklist;
    let oldGMOD = saved value of GMOD(q);
    compute newGMOD, the new value of GMOD(q);
    if newGMOD ≠ oldGMOD then
      add each procedure that calls q to worklist;
  end
end
```

Figure 2. Change Propagation Algorithm.

may be fairly local, it seems unreasonable to recompute the interprocedural information from scratch after each change. Some form of directed incremental change propagation is much more appropriate.

For example, consider recomputing the GMOD sets after a change to a single procedure p. The straightforward strategy would use a worklist of procedures for which GMOD must be recomputed. The algorithm is given in Figure 2. When the worklist is finally empty, the change has propagated as far as it can through the graph. Unfortunately, this can also take a significant amount of time, especially when there are cycles in the call graph.[6]

Additionally, we must recognize that having an incremental algorithm is not enough; we must structure the development tools to provide the information needed to drive these incremental techniques without requiring extensive re-analysis of the program. We have divided the responsibility for producing this information between the editors. Recall that the module editor is responsible for ensuring the syntactic and semantic soundness of modules; this makes it the obvious place to detect when a module has been changed in a manner which necessitates recompilation. Changes to the call graph can also require some degree of recompilation. The program editor maintains the program's call graph; it also constructs a separate list of additions to and deletions from the call graph to assist the incremental data flow algorithms. This list is reset on every program compilation.

Using this information, the program compiler can incrementally update the interprocedural summary, aliasing, and constant information which is kept for each program. When the program compiler is invoked, it collects these records left behind by the two editors and uses them to understand what significant changes have been made since the last program compilation for this program. This allows it to effectively use incremental updates to bring the information into a state consistent with the new state of the program's source text. In each of our algorithms, we have taken some care to ensure that an effective incremental technique can be used to update the information in response to editing changes in the program [5] [7]. For example, the algorithm discussed in section 6.1 can be adapted in a natural way to efficiently update information on side effects — for most cases in time linear or nearly linear in the size of the affected region of the call graph [7].

7.2. Limiting Recompilation

To make interprocedural analysis and optimization practical for large programs, the compiler must do a credible job of solving the recompilation problem introduced in Section 3. Without a good solution to this problem, the compiler can forfeit nearly all the benefits of separate compilation — effectively requiring a complete recompilation of the program in response to minor editing changes. One of the major goals of the program compiler is to eliminate unneeded recompilations. As a part of the preliminary design work for the program compiler, Torczon has developed a family

[6] The programming environment supports a recursive Fortran.

of techniques for limiting recompilations [24] [4].

Each of these techniques is characterized by a test based upon some method for reasoning about how interprocedural information is actually used by the module compiler. For example, consider the flow insensitive MOD sets described in Section 4.1. If a variable x ∈ MOD(s) for some call site s, then x *can* be modified as a consequence of executing statement s. There may, however, be execution paths through the called procedure along which x is not modified. Thus, the optimizer can only rely on MOD information as an assertion of what *may* happen, not what *must* happen. Because of this fundamental fact about the nature of flow insensitive summary information, no recompilation can be *required* by a change that deletes an element from a MOD set. Since a variable in MOD may or may not be modified, the code generated by the compiler must account for either possibility. The compiler can only rely on the absence of a variable from MOD for an optimization.

For a call site s, this observation leads to one of the simple recompilation tests. A recompilation is not necessary if

$$\text{MOD}(s) \cap \text{REFERENCED}(p) \subseteq \text{OLDMOD}(s)$$

where MOD(s) is the new set, OLDMOD(s) is the old set, and REFERENCED(p) contains the names of all variables either used or defined in the calling procedure. Note that this test requires no knowledge about what the module compiler has done in the past; it only uses previous versions of the interprocedural sets.

If the compiler records which interprocedural facts it actually uses, the recompilation analysis can be made more precise by comparing the changed information with the information upon which the module compiler actually relied. This requires, however, that the module compiler keep careful records about its compile-time decisions. In practice, it will require experience to determine whether the increased precision in the recompilation algorithm compensates for the added cost of collecting this information.

7.3. Program Compilation Algorithm

We are now ready to provide an overview of the strategy for program compilation in the environment. To accomplish its responsibilities, the program compiler employs the following algorithm, organized into several distinct passes over the program.

(1) Using records left behind by the program editor and module editor, construct an initial list of modules requiring recompilation. This list contains precisely those modules where editing has made a semantic change since last compilation, plus any modules added to the call graph since the last compilation.

(2) Incrementally update the interprocedural summary and aliasing information stored in the database. The updating techniques have been described in previous papers [5] [7].

(3) Compute interprocedural constant propagation information. Because this computation relies on information produced in the second pass, it cannot be not done concurrently with the other interprocedural analysis. The algorithm for this analysis was sketched in section 6.3.

(4) Examine information about each procedure *not* already slated for recompilation and apply recompilation tests like those described in section 7.2 to determine which procedures must be recompiled because of a change to interprocedural information used in the last compilation.

(5) Examine the complete recompilation list in search of opportunities for interprocedural optimization. In particular, identify call sites where custom linkages appear profitable, and direct the module compiler to produce the appropriate linkages.

8. Expected Improvement

While the potential benefits of interprocedural optimization have long been discussed in the literature of compiler construction [1] [10], there is only a small body of published work dealing with actual implementations which use these techniques. This is probably due to several independent factors, including the difficulty of supporting these techniques in a traditional compiler and the lack of an adequate solution to the recompilation problem. Finally, in the few compilers where such techniques have been tried, the published results almost always describe total improvements for the optimizer without attempting to assess the role of interprocedural analysis or optimization.

In his proposal for an interprocedural optimizer in the PQCC project, Conradi makes a

detailed assessment of the potential improvement to be had from interprocedural analysis and optimization [3]. He suggests that a well-done interprocedural optimizer could result in a 10-20% speed-up in execution of the resulting object code. This includes the direct impact of interprocedural optimizations like in-line substitution and the indirect effects caused by improved precision in global flow analysis. He arrives at this number by bringing together studies of the empirical properties of programs, run-time profiling of code generated by a number of compilers, and close examination of the numbers actually published for compilers which implement interesting interprocedural techniques.

9. Summary and Conclusions

Interprocedural optimization has been largely neglected in commercial compilers because it would be too costly. However, the advent of sophisticated software development environments may make it possible to perform interprocedural analysis and optimization in a natural way. We have presented the optimization strategy used in the programming environment as an illustration of the issues that must be considered. To our knowledge, is the only environment project with the support of interprocedural analysis and optimization as a central goal. The discussion in this paper should convince you that adopting such a goal has a substantial impact on the design of every component of the environment.

A preliminary implementation of the programming environment already exists. Written in C for the Sun Microsystems line of interactive workstations, it includes stable versions of the monitor, the module editor, the program editor, and the execution manager. A single-user database has been in use for over a year; the multi-user version of the database is under construction. Implementations of the interactive debugger, the module compiler, and the program compiler are underway. Finally, a number of ancillary command processors, like a calculator, a terminal emulator, a HELP processor, and a documentation editor are also included in the current system.

Acknowledgements

The programming environment has been implemented by a group of faculty, research associates, and students. In addition to the authors, the following people have written or designed parts of the system: Scott Alexander, Don Baker, Hans Boehm, Mike Caplinger, Ben Chase, David Chase, Scott Comer, Mike Donegan, Soroor Ebnesajjad, Mike Fagan, Robert Hood, Bill LeFebvre, Kathryn McKinley, Paul Milazzo, Doug Monk, Hausi Muller, Brandon Rigney, Kim Taylor, Alan Weingarten. Some of the algorithms for interprocedural analysis were first tested in the Rice Vectorizer (PFC) by Randy Allen and David Callahan. Our thinking was also influenced by the many people who have reviewed the project. Prominent among these have been Rick Adrion, Fran Allen, Horace Flatt, Bill Gear, Steve Kiely, Leon Osterweil, Fred Ris and Bill Tuel. When the project was in its infancy, Nico Habermann and the Gandalf group provided us with a great deal of assistance. The resulting system has been strongly influenced by Gandalf. Finally, our colleagues Corky Cartwright, John Dennis, and Scott Warren have had a substantial impact on the philosophy underlying the project. To all of these people go our heartfelt thanks.

References

[1] F.E. Allen, and J. Cocke. A catalogue of optimizing transformations. *Design and Optimization of Compilers*, R. Rustin, ed. Prentice Hall, New Jersey, 1972.

[2] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Proceedings of the Sixth POPL*. 1979.

[3] R. Conradi. Inter-procedural optimization of object code. Division of Computer Science Technical Report 25/83, University of Trondheim, Trondheim-NTH, Norway, 1983.

[4] K.D. Cooper, K. Kennedy and L. Torczon. Recompilation algorithms for an optimizing compiler based in the programming environment. Computer Science Technical Report 84-7, Rice University, Houston, TX. November 1984.

[5] K.D. Cooper. Interprocedural data flow analysis in a programming environment. Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston TX . May 1983.

[6] K.D. Cooper. Analyzing aliases of reference formal parameters. *Proceedings of the Twelfth POPL*. 1985.

[7] K.D. Cooper, and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. *Proceedings of the Sigplan Symposium on Compiler Construction*. SIGPLAN Notices, 19(6). 1984.

[8] J.J. Dongarra. LINPACK working note #3: FORTRAN BLAS timing. Technical Report ANL-80-24, Argonne National Laboratory. February 1980.

[9] J.J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia. 1979.

[10] A. Ershov. ALPHA - an automatic programming system of high efficiency. *JACM*, 13(1). Jan. 1966.

[11] S. Feldman. Make - a computer program for maintaining computer programs. *Software Practice and Experience*, 9. 1979.

[12] A. Goldberg, and D. Robson. *Smalltalk-80: the language and its implementation* . Reading MA: Addison-Wesley. 1983.

[13] A.N. Habermann, and D.S. Notkin. The Gandalf software development environment. Research Report, Computer Science Department, Carnegie-Mellon University, Pittsburg PA. January 1982.

[14] R. Hood, and K. Kennedy. A programming environment for Fortran. *Proceedings of the Eighteenth Annual Hawaii International Conference on Systems Sciences*. 1985.

[15] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7. 1977.

[16] L. Masinter. Global program analysis in an interactive environment. Xerox Palo Alto Research Center Technical Report SSL-80-1. January 1980.

[17] J. Mitchell, W. Maybury and R. Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto CA. April 1979.

[18] E.W. Myers. A precise inter-procedural data flow algorithm. *Proceedings of the Eighth POPL*. 1981.

[19] T. Spillman. Exposing side-effects in a PL/I optimizing compiler. *IFIPS Proceedings 1971*, North-Holland Publishing Co. 1971.

[20] R. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3). 1981.

[21] W. Teitelman. A display-oriented programmer's assistant. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*. 1977.

[22] R.T. Teitelbaum, and T. Reps. The Cornell Program Synthesizer: a syntax directed programming environment. *CACM*, 24(9). 1981.

[23] W.F. Tichy and M. C. Baker. Smart recompilation. *Proceedings of the Twelfth POPL*. 1985.

[24] L.M. Torczon. Compilation dependences in an ambitious optimizing compiler. Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX. May 1985.

Abbreviations:

CACM	Communications of the ACM
JACM	Journal of the ACM
POPL	Joint SIGACT - SIGPLAN Symposium on the Principles of Programming Languages