The Impact of Interprocedural Analysis and Optimization on the Design of a Software Development Environment

Keith D. Cooper
Ken Kennedy
Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251

1. Introduction

One of the primary goals of the programming environment project is to mount a concerted attack on the problems of performing interprocedural analysis and optimization in a compiler. Few commercial optimizing compilers employ interprocedural techniques because the cost of gathering the requisite information in a traditional compiler is too great. 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. Gathering this information would require examining the source of every procedure in the program - an expensive process, particularly unfortunate since the primary goal of separate compilation is to reduce the amount of recompilation required in response to changes in an individual procedure.

The existence of a software development environment like the programming environment [HoKe 84] changes the compilation process enough to make computing such information palatable. Since all modules are developed and all programs are 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. Whenever the compiler needs information about possible side effects of a particular procedure, it can simply extract this information from the environment's central database. Because the only mechanism for changing modules or programs is through the tools provided by the environment, the compiler is assured that it will be notified of any changes. Thus, it can use information derived from previous analysis with certain knowledge that the information reflects the current state of the program and its procedures.

One of the unique features of the programming environment is that it automatically collects and uses interprocedural data flow information. Recent results [Coop 83] allow the environment to efficiently collect such information. The compiler uses this interprocedural information as an aid to global program optimization. It also performs interprocedural optimizations such as linkage tailoring and constant folding across procedure boundaries.

This paper examines the effect of performing interprocedural analysis and optimization on each of the major components of the environment. It should become clear from the discussion that the decision to employ interprocedural techniques has profoundly influenced almost every aspect of the design of the environment.

2. The Programming Environment

To appear in *Proceedings of ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments,* **June 1985.**

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

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 [Teit 77], C [HaNo 82], Mesa [MiMS 79], PL/I [TeRe 80], and Smalltalk 80 [GoRo 83], little effort has been expended to provide similar support for Fortran, the language used by most numerical programmers. The programming environment is intended to fill this void.

The environment consists of a collection of command processors, which run cooperatively under a *monitor*. 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.

The principal components of the current environment are the module editor, the program editor, and an execution manager.[1] An interactive debugger and an optimizing compiler are under development. The optimizing compiler is structured as a pair of separate compilers, a program compiler and a module compiler [Torc 85]. We will discuss each of these components in turn.

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. A module is the smallest editable unit of source text in the environment, consisting of one or more entry points and their implementations. The 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 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. Module 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 *module compiler*. Because much of the work performed in the front end of a traditional compiler is performed in the environment's module and program editors, the module compiler consists primarily of an optimizer and a code generator. In addition to the standard techniques from global optimization, the module compiler will use interprocedural information and attempt interprocedural optimizations.

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

2.4. Program Compiler

The *program compiler* is responsible for directing the construction of an executable image of a program-version. It automates the process of reconstructing an executable in the tradition of *make* [Feld 79] and its successors [TiBa 85], calling on the module compiler as needed. In addition to this function, it performs the interprocedural analysis needed by the module compiler, determines which modules must be recompiled in response to editing changes, and provides the module compiler with directives about interprocedural optimizations like constant folding and linkage tailoring.

The program compiler embodies the essence of interprocedural analysis and optimization in the programming environment. It uses the information that other command processors, like the module editor and the program editor, have computed and stored in the database to produce flow insensitive interprocedural summary information [CoKe 84], flow insensitive interprocedural aliasing information [Coop 85], and interprocedural constant propagation information [Torc 85].

2.5. 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 which 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 would like the interpretive debugger to support reversible execution.

3. Interprocedural Analysis

As an introduction to the subject of interprocedural data flow analysis, consider the problem of computing the interprocedural MOD side effect for a statement s. 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. Let GMOD(q) be the set of such variables associated with the procedure, q. 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 multi-graph [Coop 83] [CoKe 84]. Since the sets GMOD(q) depend on the specific call multi-graph, they must be stored as 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. The traditional compiler is split into multiple parts implemented in different tools of the environment. In addition to parsing and type checking, the editor can perform most of the information gathering needed to support flow insensitive interprocedural analysis. An independent process can then compute the interprocedural side effects for the whole program. This makes interprocedural data flow information available to the code generator when it needs it. Because interprocedural information *does not depend on output from the compiler*, the environment can compute interprocedural information for each entry in the program before a single module

is compiled.

As a part of our preliminary research, we have discovered extremely fast algorithms for computing flow insensitive interprocedural side effects. In particular, we can compute these for call graphs with recursion in time **O**(E α(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 [CoKe 84]. This function grows extremely slowly. The algorithm can be adapted to efficiently perform incremental updates to the interprocedural information when a module is edited. In most cases, these updates can be performed in time linear or nearly linear in the size of the affected region of the call graph [CoKe 84].

Few commercial systems have employed interprocedural information, primarily because of the costs and the conflict it presents with separate compilation. Two notable exceptions are IBM's PL/I Optimizing Compiler and Xerox's Interlisp system. The former system computes MOD-like information for the entire set of procedures compiled together [Spil 71]. The latter system provides an interactive query system which computes interprocedural information as an aid to understanding and debugging programs [Masi 80].

4. Impact on Environment Components

Having described each of the individual components in a cursory manner, we will now examine the impact of the decision to collect interprocedural information and use interprocedural optimizations on each of them.

4.1. The Database

Central to the entire programming environment is its database. The database is a repository for all of the component parts of programs managed by the environment. It is the common structure for storing information and knowledge. This includes objects used by programmers, like the source text of a procedure or its documentation, as well as information used to manage the compilation of programs, like data flow annotations.

4.1.1. Structure of the Database

There are five major entity types in the database: *projects*, *programs*, *program-versions*, *modules*, and *module-versions*.

- A *project* is an administrative, managerial, or technical grouping, intended to incorporate a set of related programs.
- A *program* is a set of specifications for a computation. A program may have different versions which implement the specified computation.
- A *program-version* implements a specific program. An implementation consists of the set of module-versions of which the implementation is composed. All versions of the same program obey the same specifications and hence serve the same underlying purpose.
- A *module* is a set of specifications for a group of entry points.
- A *module-version* implements the entry points of a specific module. The various versions of a module must all meet the same entry point specifications, although they can differ considerably in internal implementation detail.

Most of the information manipulated by the end users of the environment is stored as attributes of program-versions and module-versions. For example, a program-version has, among others, a *composition* (list of module-versions incorporated), a *call graph*, an *executable image*, and an *entry table* (mapping from entry points to the module-versions that provide them). A module-version has *source*, a list of *entries called*, and *annotations* (information provided by the editor about side effects of the module implementation).

4.1.2. Call Graph Dependence

One of the most obvious implications of the environment's emphasis on interprocedural optimization is that *object code* for a module-version must be an attribute of the *program-version in which the module-version is compiled*. When the system confines its attention to intraprocedural analysis and optimization, the object code for a module-version is independent of any other module-version and can be stored as an attribute of the module-version to prevent duplication. But when compilations depend on interprocedural information, the code generated is specific to the program-version in which the compilation is performed. As a result, a single module-version incorporated into six program-versions could have six different object modules, each stored as an attribute of the

appropriate program-version.

In general, there are two types of information that the environment must deal with: *module-version specific* and *program-version specific*. The division must be carefully drawn in the design of the environment. For example, consider the annotations to the call graph which describe the interprocedural MOD side effect, discussed in section 3. While both the IMOD and GMOD sets are entry point specific, the IMOD sets can be stored as attributes of the module-versions since they are independent of the rest of the call graph. However, the GMOD sets are program-version specific and must therefore be stored as attributes of the program-versions which generated them.

4.1.3. Version Control

The decision to perform interprocedural analysis and optimization has a subtle impact on the attitude which the environment takes towards version control. Because the interprocedural information must be updated for every program containing module *m* whenever *m* is changed, it is not attractive to have many versions of a program containing a single module. As it happens, this also implies that differentiating between ''minor versions'' [Mull 83] of a module is unattractive. Systems that track minor versions create new minor versions of a module for every editing session [Roch 75].

In the environment, if we wish to test a new version of a module, we make a completely new version of the program containing it, so that the testing does not invalidate an existing working version. However, doing this increases the number of programs in which each of the unchanged modules is incorporated. It complicates the task of building and maintaining call graphs and has the potential to drastically increase the number of interesting call graphs containing a single module. Thus we encourage the programmer to create a new version only when a logically complete set of modifications has been made, irrespective of the number of editing sessions involved. This approach decreases the number of minor versions which must be tracked, at the expense of losing historical data about the sequence of operations leading to the current version.

4.2. The Module Editor

The module editor is the primary mechanism for modifying the source code associated with module-versions under the environment's control. In this role, the editor is the first tool to examine the contents of any module.

4.2.1. Responsibilities

The module editor detects and records modifications to the source of a module-version which alter its semantic meaning. In addition to changes to the module itself, the editor detects and records semantic changes to a module which result from modifications made to declarations, type definitions, and defined constants it uses, even when that information is stored externally to the module. This type of change is one of the motivating factors behind Tichy and Baker's work [TiBa 85]. The editor's ability to detect these semantic changes is a natural consequence of its need to understand the declarations, type definitions, and defined constants used in a module in order to provide syntactic checking of the module's source at edit time. Declarations, definitions, and constants that are defined within a module can easily be handled by the editor because the necessary information is local to the module. Definitions shared by multiple modules pose a harder problem for the editor; techniques for dealing with such definitions are discussed in Caplinger's dissertation [Capl 85].

If the environment is to produce and use interprocedural information, the editor is the appropriate place to perform certain analytical tasks, including constructing initial information for use in later analytical passes. The editor *cannot* by itself compute interprocedural side effects, because interprocedural analysis requires a call multi-graph of the program-version being analyzed, the existence of a source code implementation of each procedure in the call graph, and initial information about each procedure. For a program under development, there may be procedures which do not yet exist or are incomplete. Other users may be working on module-versions incorporated into a given program-version, forcing the interprocedural analyzer to wait until such time as all procedures are available. On the other hand, the editor must cooperate if interprocedural analysis is to be possible. In particular, it must contribute the initial information needed to compute basic interprocedural data flow information. Specifically, the module editor must compute four types of information:

Aliasing Information: Cooper presents an algorithm for annotating a program with aliasing

information [Coop 85]. The algorithm divides the work to be done into *introduction analysis* and *propagation analysis*. The introduction analysis is completely independent of specific program-versions and should be performed in the editor. This work includes computing the set of aliases introduced at each call site and a mapping from actual parameters of the call site to the formal parameters of the *calling* procedure. During this analysis the editor can also differentiate between call sites which impact the analysis and those which are irrelevant to it.

Summary Information: The computation of flow insensitive interprocedural summary information is described by Cooper and Kennedy [CoKe 84]. Their algorithm requires as input a substantial amount of information about each procedure and each call site. This includes name scoping information, descriptions of formal and actual parameters, and mappings of parameter bindings at the call sites. Additionally, sets like IMOD from section 3 are needed for each entry point provided by the module. All of these can be efficiently computed in the editor.

Constant Propagation Information: A number of algorithms which compute interprocedural constant propagation information are presented by Torczon [Torc 85]. Each of the methods relies on initial information which can be computed by the editor whenever a module-version is edited. Depending on the specific interprocedural constant propagation algorithm used, computing the initial information may involve an operation as simple as scanning the call site to detect literal constants used as actual parameters or as complex as performing a global constant propagation on a procedure to detect both local constants passed as actual parameters and constant valued global variables.

Recompilation Information: The recompilation algorithms used in the program compiler rely on the editor to mark module-versions which have been semantically changed [Torc 85]. This information is used to construct a list of all module-versions that have been semantically altered since the last compilation of the program-version. Depending on the specific algorithm used to make recompilation decisions, additional information may be needed to determine which unmarked module-versions must be recompiled due to changes in the interprocedural information assumed at the time that they were compiled. The REFERENCED set, which contains all formal parameters and global variables which are either loaded or stored in the procedure body, is one type of initial information used in these algorithms.

4.2.2. Applications

Interprocedural information can be useful in a number of ways in the editor. For example, when the user enters a constant as an actual parameter passed to an external procedure, the editor could warn the user that the constant might be modified, if that particular parameter is a member of the MOD set.

Simply providing a facility to display interprocedural information can be useful. The ability to display the interprocedural summary information that results from using the same module-version in two different program-versions might prove to be a helpful debugging tool. An examination of the interprocedural MOD information for each instance of a call site contained in several program-versions might help the user understand why the called procedure behaves differently in one specific program-version.

Following the philosophy of the DAVE system [OsFo 75], Zadeck has proposed using global data flow information in the editor to point out *data flow anomalies* to the programmer. In his dissertation [Zade 83], he suggests that this information should capitalize on the presence of interprocedural knowledge to improve the precision of the global data flow information. In considering this application, we must be careful to remember the sensitivity of interprocedural information to specific call graphs.

Interprocedural summary information describes possible data flow events along the set of paths through a call graph. Because the side effects of a single procedure call are a function of the entire body of the called procedure, including procedure calls imbedded in it, the resulting information depends on specific details of the called procedure and any other procedure which can be invoked indirectly by the call. Because the binding of procedure entry point names to implementations in module-versions is wholly controlled by the composition of the program-version, the results of an interprocedural summary information computation are wholly a property of a specific composition and its call multi-graph.

If interprocedural information is used in the editor to augment or refine the results of global

analysis, the resulting global information will be correct *only* for the specific program-version for which the interprocedural analysis was performed. When a module-version is included in multiple program-versions, conflicting diagnostic information may be reported when different program-versions are considered. This is a fundamental problem with using interprocedural information in the editor; editing a module-version is an intraprocedural task. If the editor uses interprocedural information, it must take pains to ensure that the user knows which program-version is being considered.

4.3. The Program Editor

To specify a program's configuration, a program composition editor is provided in the environment. It is the primary vehicle for programming in the large in the environment. The program editor allows a user to specify a collection of entry points and the module-versions which implement them. The program editor provides facilities for checking the consistency of a program-version. For example, it ensures that the actual parameters of a procedure call match the formal parameters of the called procedure in number and type. If the composition is *complete*, that is, it includes a main procedure and an implementation for every needed entry point, then the composition can be used to generate an executable image.

The composition processor creates program-versions. In this role, it has several responsibilities for interprocedural analysis. As it builds the composition, it constructs a call multi-graph for use in the interprocedural analyzer. It also generates annotations describing the graph's structure. When a composition is modified, the editor must update the call multi-graph to reflect the new composition and construct a list of all additions and deletions since the last compilation. This list is used by the program compiler in making its recompilation decisions.

Because the composition editor has intimate knowledge of each program-version's structure, it marks each composition as either eligible for compilation or not. In constructing the composition, the editor must determine which module-versions actually exist; therefore it knows when a program-version can not be compiled because of unimplemented entry points. This simple marking process keeps the optimizing compiler from spending large amounts of time analyzing incomplete programs.

The concerns of interprocedural analysis also impact the design of a command set for manipulating compositions. A case in point is the library search mechanism. Because the editor treats a composition as a mapping from entry points to their implementations, the editor can use existing program-versions as libraries. The editor provides a mechanism for specifying individual program-versions as libraries and searching them, in a given order, to resolve unmapped entry points.

The presence of a library search mechanism makes it tempting to use the following paradigm for creating new program-versions. (In fact, early versions of the program editor relied on this notion.) The user simply creates a new composition containing the modules which differ from the old program-version, along with the main procedure. Next, the old program-version is specified as the sole search library. Finally, the search mechanism is invoked to resolve unmapped entry points. It finds implementations of all these entry points in the old program-version's composition.

Using library search to implement a copy operation in this manner increases the amount of work required to compute interprocedural information about the new program-version. In a real copy operation, the editor would understand that the call multi-graph is preserved, allowing it to copy the interprocedural information associated with the old composition. Under the library search paradigm, each module is copied individually, with the result that most interprocedural information is lost and a complete re-analysis of the program-version is required. In a real copy implementation, the small collection of new modules can be treated as incremental updates to an existing program-version, with potentially large savings in work.

The program composition editor also provides a mechanism for defining a new module from a collection of modules. In this process, a collection of modules are marked to be treated as a single module. This presents special problems for interprocedural analysis. First, the fundamental quantities used by the program compiler in the interprocedural computations must be determined for the new module group. In the case of the MOD problem of section 3, this means computing IMOD(q) for every entry q provided externally by the group. Computing this set requires solving a small interprocedural data flow analysis problem

on the call graph of the module group. Second, there must be a mechanism for updating the interprocedural information on edges of the call graph internal to the module group, given a change externally. This requires extending the incremental updating algorithms of Cooper [Coop 83] to a hierarchical form. We are currently working on this problem.

4.4. The Module Compiler

The module compiler produces object code for an individual module. It capitalizes on the interprocedural information and optimization directives created by the program compiler to improve the efficiency of the code generated for individual modules.

4.4.1. Uses for Interprocedural Information

In the module compiler, interprocedural summary information is used to improve the precision of the computed intraprocedural information. For example, in the absence of MOD information about a call site, the module compiler must assume that the call results in modifications to every variable which is accessible to the called procedure and to every actual parameter used at the call site. With interprocedural MOD information, the number of variables believed to be modified can be greatly reduced.

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

Finally, information about interprocedural side effects not only helps produce better optimized code, it can also reduce the amount of analysis required in the module compiler. Our experience with an advanced vectorizer [AlKe 82] shows that the number of use-definition chains constructed by the compiler can be drastically reduced through the use of interprocedural analysis.

Because the module compiler uses interprocedural information as a basis for optimization decisions, the correctness of the compile-time decision making process is a function of the state of the entire program-version at the time of the *compilation*. Thus, changes to one module-version can invalidate the correctness of the optimized code previously generated for other module-versions. This introduces the recompilation problem addressed in section 4.5.

4.4.2. Optimizations

The desire to perform linkage tailoring has a direct impact on the choice of intermediate representations used for the program. If the compiler considers only strictly open and strictly closed linkages, then it may be possible to perform linkage tailoring on a high-level representation like an abstract syntax tree. In generating either semi-open or semi-closed linkages, however, the compiler will almost certainly introduce constructs which have no reasonable representation in a high-level intermediate form. This will likely necessitate use of a relatively low-level representation to accommodate optimizations like moving loop invariant procedure prologue code out of a semi-open call inside a loop.

Additionally, it is possible to perform a number of optimizations in the compiler that are difficult in a more traditional batch compiler. These optimizations are available to the compiler for little or no additional cost simply because the database provides a mechanism for coordination and cooperation between separate compilations. One such optimization is interprocedural constant folding, particularly as applied to actual parameters. Ball suggests that recognizing places where constants are passed is important to understanding the improvement to be gained by inline substitution [Ball 79]. This opens up the opportunities for optimizations like automatically unrolling loops when the loop stride is passed as an actual parameter. In a study of the BLAS routines, Dongarra demonstrated that this is a consistently profitable optimization [Dong 80]. Because these constants are available to the compiler as a natural consequence of the database's existence, this optimization is easy to perform.

4.5. The Program Compiler

The program compiler examines the entire set of procedures which constitute a program-version and computes information which directs the construction of an executable image of the program. This information ranges from a list of modules which need recompilation to directives about

which procedures are good targets for customized procedure linkages.

In the simplest sense, a program compiler analyzes a program and determines what action, if any, must be taken to make the executable image of the program consistent with the source after editing changes have been made to individual procedures in a program. The program compiler directs the compilation of the individual modules which constitute the program. Thus, Feldman's *make* utility [Feld 79] is an ancestor of the program compiler.

The program compiler addresses a more complex problem than previous program compilers. Performing interprocedural analysis and using the resulting information to do ambitious optimization introduces complex compilation dependences between the procedures of a program-version. Because the correctness of the executable code produced by the module compiler is a function of the state of the entire program *at compile time*, changes to one module-version in a program may invalidate previous compilations of other module-versions in the program. For a system using interprocedural information and separate compilation to be of practical interest requires the use of a program compiler which is far more complex than either *make* or the recompilation system discussed in [TiBa 85]. The ability to handle interprocedural information and dependences make the
program compiler unique.

Since the dependences caused by using interprocedural information are induced by the compiler rather than the programmer, they are best dealt with by tracking them during the compilation process. The task of the program compiler is to detect the interprocedural compilation dependences in a program and determine what action must be taken to return the executable image of the program to a consistent state after changes have been made to the program. Doing this requires several distinct passes over the program's call graph.

The first pass determines which modules must be recompiled because of direct editing changes to the program's composition or to the module itself. The second pass updates the interprocedural data flow information to a state consistent with the program's source text. The third pass performs interprocedural constant propagation. The fourth pass detects modules which need recompilation because changes in interprocedural data flow or constant information have invalidated assumptions made in their most recent compilation. Finally, the last pass examines the prospects for generating customized procedure linkages in the modules already being recompiled.

Each pass acts in some way to limit the number of modules which must be considered by later passes. Once the program compiler determines that a module must be recompiled, the later passes can ignore any analysis intended to determine recompilation information for that module. For example, if a module must be recompiled because the programmer added three statements to it, it is unnecessary to consider whether a change in the interprocedural summary information for that procedure also requires its recompilation. The summary information must be computed in pass two, but pass four can ignore the question of recompiling it. Similarly, after passes two and three, a module whose interprocedural and constant information has not changed need not be considered by the recompilation analysis algorithms in pass four.

Since compilations consume substantial resources, we would like to limit recompilation to as great an extent as possible. In her dissertation [Torc 85], Torczon presents a number of ways that the module compiler can generate information that helps reduce the number of required compilations. However, even without the assistance of the module compiler, there are a number of simple tests the program compiler can apply in its fourth pass.

In considering recompilation, the program compiler must first remember the underlying nature of the data flow information being used by the module compiler. For example, the MOD information is flow insensitive, so it describes variables which may be modified by executing a procedure call. Any optimization made on the basis of MOD information must already account for the fact that the variable *might not* be modified by the call. Thus, any change which removes a variable from MOD(s) cannot invalidate optimizations based on MOD(s). Only changes which add information to MOD(s) can invalidate optimizations based on MOD(s).

Torczon describes a series of recompilation tests based on interprocedural summary and constant information. These range from simple tests like the one just described to more complex tests which account for specific optimizations

performed in the module compiler.

4.6. The Interpretive Debugger

A major goal of the debugger design is to support hybrid execution of interpreted and compiled modules. Interprocedural information can simplify the interface between compiled code and the interpretive debugger. Similarly, the design of the debugger may preclude the application of some interprocedural optimizations.

Consider a session in the debugging interpreter. The programmer might ask the interpreter to report, after each statement executes, which variables have had their values changed. For statements which do not include procedure calls, this is a simple task. If, however, the statement involves a call to a compiled procedure, the interpreter must compare the values of each variable in memory against its value before the call in order to compute the debugging information. For any non-trivial program, this is a prohibitively expensive proposition.

If, however, the interpreter has access to interprocedural summary information, it can use the summary information to determine which variables *might* change as a result of the call, and the search for changed variables can be restricted to a much smaller set. This same technique can be used to improve the support for *reversible execution*, since implementation of this feature requires dynamic checkpointing of variables that *might* change as a result of a call to a compiled procedure.

The requirement for hybrid interpreted and compiled execution seems likely to rule out the application of certain optimizations. For example, the preceding discussion assumes that the compiler understands the mapping from variable names to storage locations for compiled code. This simple assumption may prohibit the compiler from applying sophisticated storage optimizations like those proposed by Fabri [Fabr 79].

Finally, hybrid execution of interpreted and optimized code has implications for the design of the interpreter. It is easy to imagine a programmer changing the value of a variable inside the interpreter during a debugging session. In a separate compilation environment without interprocedural information, this is a relatively safe action. Once the compiler has used interprocedural information as a basis for optimization decisions, though, the possibility arises that changing the value of a variable inside one procedure has implications for the correctness of the code already compiled for other procedures in the currently executing program-version. Thus, the interpreter may need to understand the manner in which the compiler uses interprocedural information in order to allow safe and correct responses to user requests during execution. Comer is investigating these and similar issues in the design of a debugger for the programming environment [Come 85].

5. Summary and Conclusions

Although a sophisticated software development environment is the only practical setting for performing optimizations based on interprocedural analysis, the discussion in this paper should convince you that the required analysis has a substantial impact on the design of every component of the environment. We have presented the design of the environment as an illustration of the issues that arise.

A preliminary implementation of the programming environment already exists. 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.

References

[AlKe 82] Allen, J.R. and Kennedy, K., ‘‘PFC: a Program to Convert Fortran to Parallel Form’’, Report MASC TR 82-6, Dept. of Mathematical Sciences, Rice University, Houston, TX, March 1982.

[Ball 79] Ball, J.E., ‘‘Predicting the Effects of Optimization on a Procedure Body’’, *Proceedings of the Sigplan Symposium on Compiler Construction*, SIGPLAN Notices, 14(8), 1979.

[Capl 85] Caplinger, M.A., ‘‘A Simple Intermediate Language for Programming

Environments'', Ph.D. Dissertation, Department of Computer Science, Rice University, Houston TX, May 1985.

[Come 85] Comer, R.S., ''Data Breakpoints'', Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX, expected Fall 1985.

[Coop 83] Cooper, K.D., ''Interprocedural Data Flow Analysis in a Programming Environment'', Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston TX, May 1983.

[Coop 85] Cooper, K.D., ''Analyzing Aliases of Reference Formal Parameters'', *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985.

[CoKe 84] Cooper, K.D. and K.W. Kennedy, ''Efficient Computation of Flow Insensitive Interprocedural Summary Information'', *Proceedings of the Sigplan Symposium on Compiler Construction*, SIGPLAN Notices, 19(6), 1984.

[Dong 80] Dongarra, J., ''LINPACK Working Note #3: FORTRAN BLAS Timing'', Technical Report ANL-80-24, Argonne National Laboratory, February 1980.

[Fabr 79] Fabri, J., ''Automatic Storage Optimization'', *Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN Notices, 14(8), 1979.

[Feld 79] Feldman, S., ''Make - A Computer Program for Maintaining Computer Programs'', *Software Practice and Experience*, Vol. 9, 1979

[GoRo 83] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and its Implementation*, Reading MA: Addison-Wesley, 1983.

[HaNo 82] Habermann, A.N. and D.S. Notkin, ''The Gandalf Software Development Environment'', Research Report, Computer Science Department, Carnegie-Mellon University, Pittsburg PA, January 1982.

[HoKe 84] Hood, R. and K. Kennedy, ''A Programming Environment for Fortran'', *Proceedings of the Eighteenth Annual Hawaii International Conference on Systems Sciences*, 1985.

[Masi 80] Masinter, L., ''Global Program Analysis in an Interactive Environment'', Xerox Palo Alto Research Center Technical Report SSL-80-1, January 1980.

[MiMS 79] Mitchell, J., W. Maybury and R. Sweet, ''Mesa Language Manual'', Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto CA, April 1979.

[Mull 83] Muller, H., ''A Conceptual Framework for Configuration Management Systems'', Rice University, Houston TX, September 1983.

[OsFo 75] Osterweil, L.J. and L.D. Fosdick, ''Dave - A Validation, Error Detection and Documentation System for FORTRAN Programs'', Technical Report CU-CS-071-75, Computer Science Department, University of Colorado, Boulder CO, June 1975.

[Roch 75] Rochkind, M.J., ''The Source Code Control System'', *IEEE Transactions on Software Engineering*, SE-1, Vol. 4, December 1975.

[Spil 71] Spillman, T., ''Exposing Side-Effects in a PL/I Optimizing Compiler'', *IFIPS Proceedings 1971*, North-Holland Publishing Co., 1971.

[Teit 77] Teitelman, W., ''A Display-Oriented Programmer's Assistant'', *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

[TeRe 80] Teitelbaum, R.T. and T. Reps, ''The Cornell Program Synthesizer: A Syntax Directed Programming Environment'', *Communications of the ACM*, 24(9), 1981.

[TiBa 85] Tichy W. F. and M. C. Baker, ''Smart Recompilation'', *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, 1985.

[Torc 85] Torczon, L.M., ‘‘Compilation Dependences in an Ambitious Optimizing Compiler’’, Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX, May 1985.

[Zade 83] Zadeck, F.K., ‘‘Incremental Data Flow Analysis in a Structured Program Editor’’, Ph.D. Dissertation, Department of Mathematical Sciences, Rice University, Houston TX, October 1983.