INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106
8517237

Torczon, Linda Marie

COMPILATION DEPENDENCES IN AN AMBITIOUS OPTIMIZING COMPILER

Rice University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1985
RICE UNIVERSITY

Compilation Dependences
in an
Ambitious Optimizing Compiler

by

Linda Marie Torczon

A Thesis Submitted
In Partial Fulfillment Of The
Requirements For The Degree

Doctor of Philosophy

Approved, Thesis Committee:

Kenneth W. Kennedy, Jr., Chairman
Professor of Computer Science

John E. Dennis, Jr.
Professor of Mathematical Sciences

Peter J. Varmán
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

May, 1985
Compilation Dependences
in an
Ambitious Optimizing Compiler

Linda Marie Torczon

ABSTRACT

When interprocedural analysis and optimization are employed in an optimizing compiler based in a software development environment, cooperation between the components of the environment is essential. Many components in such a software development environment are involved in the computation of either interprocedural information or the initial information from which the interprocedural information is derived. These components must be capable of both recording the information that they produce in the environment's database and interpreting information left in the database by other components.

To ensure that the components effectively cooperate, the impact of collecting interprocedural information and performing interprocedural optimizations must be carefully considered in the design of the environment. This dissertation presents an overview of the impact of interprocedural analysis and optimization on a software development environment designed to aid numerical programmers and an examination of the problems that arise in an
ambitious optimizing compiler which introduces compilation dependences between the procedures of a program.

The concept of a program compiler encapsulates the analysis required to collect interprocedural information, track compilation dependences, and make smart recompilation decisions. A program compiler is responsible for analyzing a program that has been altered and determining a course of action that will return the executable image of that program to a state that is consistent with its source. The design presented for a program compiler incorporates passes that perform the necessary interprocedural analysis. Interprocedural summary and aliasing information is collected using methods presented in the literature. Techniques for detecting interprocedural constants are presented. These methods range from expensive algorithms that detect most of the interprocedural constants to cheaper algorithms that detect fewer constants. Recompilation algorithms that detect the subtle dependences introduced between the procedures of a program when interprocedural information is used by the optimizer are described.
ACKNOWLEDGEMENTS

Earning a doctoral degree is an exacting, albeit rewarding, experience. Without the support and encouragement of the Rice community, my family, and my friends, completing the requirements for a doctoral degree would have been difficult, if not impossible.

In particular, I would like to thank my doctoral committee, Ken Kennedy, John Dennis, and Peter Varman, for their cooperation and technical advice. I would like to thank the National Science Foundation for sponsoring this research. Additionally, I would like to thank my family and friends for enthusiastically supporting my progress, and my parents for instilling in me a desire to learn. Finally, I would like to thank my husband, Keith Cooper, for the immense contribution he made to this research. His technical guidance, his proof reading skills, and the time he committed immeasurably improved the quality of this dissertation. His patience, enthusiasm, and understanding sustained me throughout my graduate career.
# Table of Contents

Chapter 1 - Impact of Interprocedural Analysis on a Programming Environment  
Chapter 2 - Previous Research  
Chapter 3 - The Program Compiler  
Chapter 4 - Interprocedural Constant Propagation  
Chapter 5 - Recompilation Algorithms  
Chapter 6 - Conclusions  

Bibliography
CHAPTER 1

Impact of Interprocedural Analysis on a Programming Environment

1.1. Introduction

One of the primary goals of the IR^n 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 IR^n programming environment [HoKe 84] changes the compilation process enough to
make computing such information palatable. Since all procedures 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
programmer must use the tools of the environment to change procedures or
programs, the compiler is assured that it will be notified of changes. Thus, it
can use information derived from previous analysis with the certain knowledge
that the information reflects the current state of the program and its
procedures.

One of the unique features of the \( \mathbb{R}^n \) 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 interprocedural information as an aid to global program
optimization. It also performs interprocedural optimizations such as linkage
tailoring and constant folding across procedure boundaries.

The decision to employ interprocedural techniques profoundly influences the
design of almost every aspect of a programming environment. This chapter
details the effect that performing interprocedural analysis and optimization has
had on the design of major components in the \( \mathbb{R}^n \) programming environment.
1.2. The \( IR^n \) Programming Environment

The \( IR^n \) 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 the 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 \( IR^n \) 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 features provided by the command processors running under the monitor include access to the \( IR^n \) database, the ability to edit source text and program compositions, interpretive debugging facilities, and an optimizing compiler.

The command processors use and record information in the \( IR^n \) 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 module editor, or intelligent FORTRAN editor, combines a knowledge of FORTRAN and access to the database to simplify the process of writing a
program. It helps the programmer enter syntactically correct versions of modules, the smallest editable units of source code, and obviates the need for a parser by directly constructing an abstract syntax tree representation of the program. All components of the environment use the abstract syntax tree as the standard module representation.

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 denotes a version of the program and ensures that a definition of each entry point used in the program version is included in this collection.

The IR\textsuperscript{n} programming environment includes an ambitious *optimizing compiler* to satisfy the numerical community's strict requirements for efficient execution of compiled code. In addition to the usual global optimization techniques, the IR\textsuperscript{n} compiler employs interprocedural analysis and uses interprocedural information as a basis for both global and interprocedural optimization. To accommodate interprocedural analysis and optimization, the task of compilation is divided between two distinct compilers, the *module compiler* and the *program compiler*.

The *module compiler* generates optimized code for a single module. 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 $\mathbb{IR}^n$ module compiler will use interprocedural information computed by the program compiler and attempt interprocedural optimizations suggested by the program compiler.

The *program compiler* is responsible for directing the construction of an executable image of a version of a program. 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 *interpretive debugger* enables the programmer to interactively step through parts of a given program, allowing him to interrupt and monitor execution. The interpreter is able to execute hybrid programs consisting of some compiled and some interpreted versions of modules. Thus, control passes quickly through stable, compiled sections of the program to a module under development, which is then interpreted. Without such hybrid execution, interpreters are impractical tools for large programs. One goal for the interpretive debugger is to support reversible execution.

Because the database, module editor, program editor, module compiler, program compiler, and interpretive debugger all deal with the internals of versions of programs and modules in the $\mathbb{IR}^n$ programming environment, the
impact of interprocedural analysis and optimization must be considered carefully in their design. Other command processors in the environment, like the text editor, calculator and mail system, do not handle source code or program compositions and, therefore, are not affected by interprocedural information. The remainder of this chapter discusses interprocedural analysis and its impact on the design of the database, module editor, program editor, module compiler, program compiler and interpretive debugger in the $\mathbb{R}^n$ programming environment.

1.3. 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 both the calling and 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 and calling 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 called procedure, \( q \). Notice that \( \text{GMOD}(q) \) consists of two components:

- the set \( \text{IMOD}(q) \) 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 \( \text{IMOD}(q) \) for each entry \( q \) in a version of a module are independent of any other procedures in any program in which that version of 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 module editor and stored as an attribute of the module version. From these sets, \( \text{GMOD}(q) \) can be computed for a specific version of a program by solving a data flow problem on its call multi-graph [Coop 83][CoKe 84]. Since the sets \( \text{GMOD}(q) \) are dependent on the call graph, they must be stored with the specific version of the program for which they are computed.

This observation illustrates an important aspect of the division of labor that 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 module editor can perform most of the information gathering needed to support flow insensitive interprocedural analysis. When interprocedural information is needed, an independent process can compute the side effects for the whole program. Because interprocedural
information does not depend on output from the module compiler, the
environment can compute interprocedural information for each entry in the
program before a single module version is compiled.

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].

1.4. Impact on Environment Components

1.4.1. The Database

Central to the entire IR^n 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 or
documentation of a version of a module, as well as information used to manage
the compilation of programs, like data flow annotations.

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-version is a defined set of module-versions which implement a specific program. The individual versions that implement a program must conform to the specifications for that program.

• 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, the attributes of a program-version include a composition or list of the module-versions incorporated in the program-version, a call graph, an executable image, and an entry table which maps entry points to the module-versions that provide them. A module-version has source, a list of entries called, and annotations provided by the module editor about side effects of the module implementation.

One of the most obvious implications of the 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 that describe the interprocedural MOD side effect discussed in section 1.3. 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 that generated them.

### 1.4.2. 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 module editor is the first tool to examine the contents of any module.

The module editor detects and records modifications to the textual source of a module-version that alter its semantic meaning. In addition to changes to the module itself, the module editor detects and records semantic changes in a module that are the result of modifications made to declarations, type definitions, and defined constants used by the module, even when that
information is stored externally to the module. The module 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 editing time.
Declarations, definitions, and constants that are defined within a module can
easily be handled by the module editor because the necessary information is
contained within the module. Definitions shared by multiple modules pose a
harder problem for the module editor. Linking shared definitions to their uses in
other modules is one way that the module editor can track external definitions,
declarations, and constants used by a module [Capl 85]. The module editor can
create and maintain links between definitions and modules. Once the links are
formed, the module editor possesses enough information to perform syntactic
checking, even when the checking involves external shared definitions, because
the module editor knows where to find the needed definitions. These same links
also provide the module editor with a means of determining which modules, if
any, have been semantically altered due to a change to a shared definition.
Modules that have been semantically altered are marked for recompilation by
the module editor. This is only possible because all of the modules exist in a
single database.

If the environment is to produce and use interprocedural information, the
module editor is the appropriate place to perform certain analytical tasks,
including constructing initial information for use in later analytical passes. The
module editor cannot by itself compute interprocedural side effects, because interprocedural analysis requires a call 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 that 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 module 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 module 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 module editor can also differentiate between call sites that impact the analysis and call sites that are irrelevant to it.
Summary Information: The computation of flow insensitive interprocedural summary information is described by Cooper and Kennedy [CoKe 84]. This 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(q), described in section 1.3, are needed for each entry q provided by the module. All of these can be computed efficiently in the module editor.

Constant Propagation Information: Algorithms which compute interprocedural constant propagation information are presented in chapter 4. A variety of techniques are presented. Each of these methods relies on initial information which can be computed by the module 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 that are passed as actual parameters or as complex as performing a global constant propagation on a procedure to detect local constants passed as actual parameters or to detect constant valued global variables.

Recompilation Information: Chapter 5 presents methods to determine which module-versions in a program-version must be recompiled in response to an editing change, if the program-version has been compiled using interprocedural information. The algorithms presented rely on the module editor to mark
module-versions whenever they are semantically changed. This allows easy
construction of a list of all module-versions that have been semantically altered
since the last compilation of the program-version. Depending on the technique
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 Reference set, the set of all formal parameters and global variables of a
procedure that are either loaded or stored in the procedure body, is one type of
initial information used in these algorithms.

Interprocedural information can be useful in a number of ways in the
module editor. For example, when the user enters a constant as an actual
parameter to an external procedure, the module editor could present a warning
if that particular parameter might be modified, by consulting the MOD set.

Following the philosophy of the DAVE system [FoOs 76], Zadeck has
proposed using global data flow information in the module 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 that 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 module 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 module editor; editing a module-version is an intraprocedural task. If the module editor uses interprocedural information, it must take pains to ensure that the user knows which program-version is being considered.

1.4.3. The Program Editor

To specify a program configuration, a program editor is provided in the IRn 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 that implement them. Provided this mapping includes a main procedure and an implementation for every needed entry point, it can be used to generate an executable image.

The program editor 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 program 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 program 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 program editor must determine which module-versions actually exist; therefore it knows when a program-version cannot 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 program editor treats a composition as a mapping from entry points to their implementations, the program editor can use
existing program-versions as libraries. The program 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. The user simply creates a new composition containing the modules that 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 those 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 program 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.
1.4.3.1. 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. Information about interprocedural side effects not only helps produce better optimized code, it also reduces the amount of analysis the module compiler is required to do. Experience with an advanced vectorizer for FORTRAN [AIKe 82] shows that the number of DEF-USE chains constructed by the compiler can be drastically reduced by interprocedural analysis.

The desire to perform linkage tailoring has a direct impact on the choice of intermediate representations used for the program. If the optimizing 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 optimizing compiler introduces constructs that have no reasonable representation in a high-level intermediate form. This necessitates 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.

1.4.3.2. The Program Compiler

The program compiler examines the entire set of procedures that constitutes a program-version and computes information that directs the
construction of an executable image of the program. It embodies the essence of interprocedural analysis and optimization in the IR^n 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 interprocedural flow insensitive summary information [CoKe 84], interprocedural flow insensitive aliasing information [Coop 85], and the interprocedural constant propagation information discussed in chapter 4 of this dissertation.

Based on the interprocedural information that it computes and information obtained from the module editor and program editor, the program compiler determines which module-versions must be recompiled, where linkage tailoring should be performed, and where other optimizations can be performed. Once the interprocedural phase is complete, the individual module-versions are recompiled using the module compiler as needed.

Chapter 3 details the design of a program compiler which handles the intricacies of interprocedural analysis and the subtle and complex compilation dependences introduced between the procedures of programs that are compiled in the presence of interprocedural information. Chapter 5 presents algorithms that limit the amount of recompilation necessary when an editing change is made to a procedure in a program. Limiting the amount of necessary recompilation is essential to preserving the economies of separate compilation, an important feature in an environment that supports programming in the large.
1.4.4. The Interpretive Debugger

A major goal of the interpretive debugger design is to support integrated execution of interpreted and compiled modules. When, for debugging purposes, the interpreter is asked to report which variables are changed by each statement in the interpreted code, calls to compiled subroutines must be treated as single statements for the purpose of reporting changes. However, this means that the interpreter must compare all of memory before and after the call to determine what has actually changed.

In the IR\textsuperscript{n} programming environment however, the interpreter can instead use interprocedural information to determine which variables \textit{might} change, 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 \textit{reversible execution}, since implementation of this feature requires dynamic checkpointing of variables that \textit{might} change as a result of a call to a compiled procedure.

1.5. Overview

The decision to collect and use interprocedural information in the IR\textsuperscript{n} programming environment has had a substantial impact on the design of many components in the environment. In particular, the presence of interprocedural information at compile time has spawned the need for a program compiler which analyzes the compilation dependences between the procedures of a program, compilation dependences that only arise when interprocedural information and
optimization are present at compile time.

The program compiler and the algorithms it uses to both create and detect compilation dependences are described in this dissertation. To provide the requisite background in the area of interprocedural analysis and optimization, chapter 2 briefly describes previous research in the area. The design of the program compiler is presented in chapter 3. Chapters 4 and 5 present algorithms that can be used by the program compiler to detect interprocedural constants and compilation dependences, respectively.
CHAPTER 2

Previous Research

2.1. Interprocedural Analysis

In a traditional compiler, the individual procedures in a program are compiled separately. When a call site is detected in a procedure, information about the invoked procedure is not available to the compiler. As a result, the compiler is forced to make worst case assumptions about the impact, or side effects, of executing the procedure call on the values of variables accessible to the called procedure. Furthermore, the compiler has no description of the different environments from which a procedure is called. As a result, the compiler must assume that some pairs of external variables accessible by the procedure being compiled may be aliases, that is to say, they may both refer to a single storage location. These worst case assumptions made by the compiler may prevent it from performing valuable optimizations.

To more precisely determine when it is safe and potentially profitable to apply an optimization, the compiler needs information that describes the impact of invoking individual procedures in a program. Interprocedural analysis is used to derive interprocedural information, information that describes the impact of procedure calls. The interprocedural information computed is specific to a given
call graph or call multi-graph, \( G = (N,E) \). Here, the set \( N \) contains the nodes of the call graph, with each node representing one procedure from the program. The set \( E \) contains the edges of the call graph, representing individual call sites in the program.

Interprocedural information is characterized as either \textit{flow sensitive} or \textit{flow insensitive}. Flow insensitive information includes events that occur on at least one path through a procedure. This implies that there may be paths through a procedure on which the event does not occur. Flow sensitive information only includes events that occur on every path through the procedure. Flow insensitive summary information can be efficiently computed in the presence of aliasing. The computation of flow insensitive summary information in the presence of aliasing is Co-NP-complete [Myer 81].

Precision of the computed information is an important consideration when selecting an algorithm that computes interprocedural information. An algorithm yields information that is precise up to symbolic execution if it produces the most precise results possible under certain assumptions. The assumptions are:

- it is possible to take any path through a procedure when that procedure is invoked.
- the call site from which the procedure is invoked does not determine the path that will be taken through the procedure.

There are a variety of types of interprocedural information that are valuable to compute. The two most commonly discussed in the literature are interprocedural summary information, which describes the cumulative side
effects of procedure invocations, and interprocedural aliasing information, which describes the aliasing patterns inherited from a procedure's call sites.

Chapter 1 discussed the impact of interprocedural analysis and optimization on the design of the IR\textsuperscript{n} programming environment. Interprocedural analysis has been discussed in the literature since the early 1970's. It is only recently that our understanding of interprocedural analysis has progressed to the point where an environment like IR\textsuperscript{n} is feasible. In particular, interprocedural analysis is important to this dissertation because the design of the program compiler, that component of the environment which both computes and uses interprocedural information, and the compilation dependences introduced by the use of interprocedural information in an optimizing compiler are the major topics of this dissertation.

The principal focus of this dissertation is to provide a practical framework to overcome some of the difficulties caused by the use of interprocedural information in the compilation process. Before proceeding further, a brief review of the literature of interprocedural analysis is in order.

2.1.1. Spillman

Spillman, a member of the PL/I optimizing compiler implementation team, was the first to write about computing interprocedural information [Spil 71]. Spillman describes a method for computing three types of summary information:
• the set of variables that may be modified by executing statement s
• the set of possible aliases for a variable x
• the set of locations where control could be transferred after execution of statement s

His technique involves making a single pass over the program, visiting the procedures in any convenient order, to collect information about the call sites and procedures in the program. Imprecise aliasing information is also computed in this pass.

The information from the first pass is stored in a matrix, the Expose Matrix. Subsequent passes manipulate the Expose Matrix to compute interprocedural side effects. Information about the procedures in the program is first accessed in reverse invocation order, an ordering that guarantees that a procedure is visited after the procedures that it invokes. The information is then accessed in invocation order in the final pass through the program. When recursion is detected, iteration over the procedures involved in the recursion is necessary to compute the interprocedural side effects.

Spillman's work is unusual because it deals with computing interprocedural side effects for a language that allows aliasing, procedure variables, label variables, pointers, and other features that make the detection of the call graph and the control of flow in the program difficult. It also accounts for exceptional condition handlers, in the form of PL/I On Condition blocks. Subsequent work in the field often ignores one or more of these features.
2.1.2. Allen and Schwartz

Allen [Alle 74] discusses a technique for extending standard global data flow problems across procedure boundaries. In particular, Allen presents methods for constructing interprocedural information similar to the global REACHES information used in computing global DEF-USE chains. The interprocedural DEF-USE chains that can be constructed from this information support a variety of interprocedural optimizations, including restricted forms of interprocedural constant propagation.

In her work, Allen avoids dealing with the circularity inherent in computing interprocedural information by processing the procedures in a program in reverse invocation order and ignoring the problems introduced by recursion. This allows her method to compute its information in a single pass. Allen and Schwartz extend Allen's work to handle recursion [AlSc 74]. Their approach makes initial worst case assumptions about the information for a small core of the procedures involved in the recursion. This yields a safe, albeit imprecise, solution to the problem at the end of the first pass. Iteration over the procedures involved in the recursion can be used to further refine the computed interprocedural information. However, due to the initial worst case assumption, the iteration can converge to a stable solution that is refined, but still imprecise [Lome 77].

---

1Kennedy describes REACHES information and the construction of DEF-USE chains [Kenn 81].
Unlike Spillman, Allen and Schwartz do not expressly describe how to handle language features, other than recursion, that make computing interprocedural information for a real language difficult. In particular, the problems associated with transfer of control on exit from a procedure and language features like procedure, pointer, and label variables are not discussed. They do not discuss aliasing. It is also not clear how such a scheme works with separate compilation.

2.1.3. Rosen

Rosen describes a method for computing interprocedural information which answers the following questions:

- \textsc{MOD}(q,x)\quad \text{Can procedure } q \text{ modify variable } x? \\
- \textsc{USE}(q,x)\quad \text{Can procedure } q \text{ use variable } x? \\
- \textsc{PRE}(q,x)\quad \text{Can procedure } q \text{ preserve variable } x? \\

Rosen formulates a set of simultaneous equations which are solved to produce \textsc{MOD}, \textsc{USE}, and \textsc{PRE}. An iterative technique is used to solve the equations if recursion is present in the program [Rose 79]. The \textsc{MOD}, \textsc{USE}, and \textsc{PRE} information derived for each call site is the sharpest possible information. This implies that it would be necessary to perform symbolic execution to derive information that is more precise than Rosen's information.

Because his technique considers individual execution paths through the program's call multi-graph separately, the information derived is sharper than
that found by Spillman, or later methods like those of Barth, Banning, Myers and Cooper. Spillman and Barth compute less precise summary interprocedural summary information in the presence of aliasing because the aliasing information that they derive is imprecise. Banning and Cooper propose techniques that compute flow insensitive summary information which is precise up to symbolic execution. Unlike Rosen, however, their methods are incapable of producing precise flow sensitive summary information. Myers proposed two separate techniques, one for flow insensitive problems and one for flow sensitive problems.

Because it must consider all the alias equivalence relations associated with the procedures in a program to compute path specific information, Rosen’s method can be expensive. A procedure can potentially have an exponential number of alias equivalence relations associated with it, leading to a potential combinatorial explosion in the cost of Rosen’s method. Rosen argues, however, that, in actual programs, name scoping adequately limits the number of aliasing equivalence relations in most cases.

Rosen handles the problems, like aliasing, introduced in storage mappings by abstracting them out of the equations and encapsulating them in a data flow index. The data flow index for each variable specifies its name, its location in the control flow graph of the program, and pertinent aliasing information. The details of constructing data flow indices in the presence of language features like dynamic allocation and procedure, label, and pointer variables are left to the
implementor.

2.1.4. Lomet

Recognizing the potentially exponential impact that precise aliasing information can have on the computation of both summary data flow analysis and global data flow analysis, Lomet suggests a set of inequalities that approximate the effect of aliasing on the data flow information in a call-by-reference environment [Lome 77]. To approximate the interprocedural side effects of a procedure call, Lomet first computes a called procedure's side effects in the absence of aliasing. By combining the results of this initial computation with the aliasing present at the call site, he produces an approximation to the desired interprocedural information.

Lomet's inequalities can be used in conjunction with Rosen's technique [Rose 76] to produce MOD, USE, and PRE information, while avoiding the potentially combinatorial nature of Rosen's computations. While the MOD information produced in this manner is precise up to symbolic execution, the USE and PRE information is less precise than that produced by Rosen. Lomet's inequalities also improve the precision of the information produced by Allen's technique. This is primarily due to the fact that Lomet takes aliasing into account. Recall that aliasing effects were not covered in Allen's original work.

Lomet only intends to present a method for coping with the difficulty of computing summary data flow information and global data flow information in a
language that supports call-by-reference parameter bindings. As a result, he does not actually compute the needed aliasing information, nor does he handle language features like procedure or label variables which lead to dynamic structuring of the control flow graph.

2.1.5. Barth

Barth presents a set of techniques for computing summary information based on algebraic relations about the program [Bart 77]. He gives formulas for computing MOD, USE, and PRE information, the sets of variables that can be modified, used, or preserved by the execution of a procedure call. He also provides concrete definitions for the correctness and precision of a summary computation.

Barth's method is similar in flavor to Spillman's, in that the relations can be derived in a single pass over the program and then manipulated independently of the program. Further, the relations can be represented in bit-matrix form, allowing relatively efficient implementation. The primary shortcoming of Barth's work lies in the imprecision of the information it computes. The relational approach does not lend itself to dealing with call site specific aspects of aliasing. For example, the information is computed over the call graph, rather than the call multi-graph.
2.1.6. Banning

Banning formulated the computation of flow insensitive summary information as a data flow analysis problem [Bann 78]. He presents a data flow framework for describing interprocedural summary information, making it possible to compute this information using any of the standard techniques of global data flow analysis. The method he uses requires only a single pass through the call graph to gather the initial data needed. The flow insensitive summary information that he computes using this method is precise up to symbolic execution. This method can be extended to handle the flow sensitive side effects of procedure calls, but the resulting information produced is less precise.

Banning factors aliasing out of the problem of computing flow insensitive summary information. He handles the question of aliasing separately and suggests using a method based on depth first search to compute the needed aliasing information. He explains how the model for computation that he presents can be extended to handle language features, like procedure and label variables, that can make the call graph and the control flow graph of a program difficult to detect.

2.1.7. Weihl

Weihl considers the problems associated with computing interprocedural data flow information for languages that allow call-by-reference formal
parameters, pointers, procedure variables, and label variables [Weih 80]. To a large extent, the problems imposed by pointers, procedure variables, and label variables have been overlooked. Spillman [Spil 71] and Ryder [Ryde 79] have considered these problems for specific languages, PL/I and FORTRAN respectively, and made assumptions that limit the applicability of the methods they suggest. Other researchers have either ignored or limited the impact of pointers, procedure variables, and label variables on their data flow algorithms by placing restrictions on their use.

When procedure variables are present in a language, the call graph for a program can be difficult to deduce. Pointers can complicate the aliasing patterns in the program. The use of label variables makes it difficult to determine the control flow graph of a routine.

To find the call graph and control flow graph for a program, Weihl first computes lists of potential values for procedure variables and pointers and the aliasing patterns in the program. This information is computed without any knowledge of the program's call graph or control flow. Once potential values for the procedure variables are known, the call graph is constructed. Interprocedural summary information can then be computed. Furthermore, values for the label variables can be estimated and a control flow graph can be constructed.

Weihl proves that determining possible values for procedure variables is P-space hard. Because of the potentially great expense involved in computing
precise information about procedure variables, the method he proposes is conservative and sometimes imprecise.

2.1.8. Myers

Myers presents a data flow framework for computing interprocedural flow insensitive summary information [Myer 80]. The results he achieves on the flow insensitive side are equivalent to those discussed by Banning in his dissertation [Bann 78], but were derived independently.

Myers also discusses flow sensitive problems. He shows that the avail, live, and must-summary data flow problems are either NP-complete or Co-NP-complete in the presence of aliasing [Myer 81]. He then presents a $O(\text{EDGES} \times \text{SET})$ algorithm for solving these problems in the presence of aliasing. Here, EDGES is the size of the program's flow graph, and SET reflects the number of aliasing patterns in the program. The potentially exponential behavior of the algorithm arises from the SET term. Myers argues that, in practice, SET will be small. Hence, he suggests that computing avail, live, and must-summary information in this manner is practical.

2.1.9. Sharir and Pnueli

Sharir and Pnueli propose approaches to computing interprocedural data flow information that combine the process of computing intraprocedural and interprocedural information [ShPn 81]. They suggest two methods that combine the computation of interprocedural and intraprocedural information, yielding
the sharpest possible interprocedural information for each call site. Both suggested techniques are capable of handling recursion.

The first method is a functional approach. Under this scheme, procedures are treated as collections of blocks. The individual blocks have input-output relations, similar to those used in global data flow analysis, defined on them. When a procedure call is detected, it is treated as a super operation which involves all of the blocks of the procedure. The second technique discussed is the call-strings approach. When it is used, the whole program is effectively treated as a single flow graph. The interprocedural flow of information through the program is made explicit by tagging the information being computed with a history of the call multi-graph edges across which the information has propagated.

Sharir and Pnueli restrict their work to languages that contain only call-by-value parameters. This eliminates aliasing. They ignore procedure variables. Finally, they assume an environment without separate compilation, evidenced by their restriction prohibiting external procedures.

Their work combines the approach taken by Allen and Schwartz, namely computing global-style information interprocedurally, with a computational method reminiscent of Rosen's work. Unfortunately, this results in a method that appears impractical for large scale programs, since it cannot deal with separate compilation. The strong economic advantages of separate compilation for large programs argue against the application of techniques like theirs.
2.1.10. Cooper

In his dissertation, Cooper discusses the computation of flow insensitive interprocedural summary information in a programming environment [Coop 83]. His work builds on that of Banning and Myers. By splitting the summary computation into two distinct subproblems, he lowered the time bound on the summary computation to $O(E\alpha(E,N))$, assuming bit-vector operations. The algorithm also lends itself to efficient techniques for updating interprocedural information after some of the procedures in a program are modified [CoKe 84]. The flow insensitive summary information computed using this technique is precise up to symbolic execution.

Additionally, Cooper formulated the aliasing problem as a data flow analysis problem, allowing the application of standard techniques from global data flow analysis to solve it. In fact, even the $O(E\alpha(E,N))$ technique developed for computing summary information can be applied to this problem.

2.2. Linkage Tailoring

For the sake of uniformity, compilers normally generate standard calling sequences for procedures. The idea of tailoring procedure linkages to increase the run-time efficiency of the resulting code has been suggested many times in the literature but has been implemented relatively rarely. One of the earliest mentions of this idea is found in Ershov's work [Ersh 66]. Because linkage tailoring requires knowledge of both procedures involved in a procedure call, it is
difficult to accommodate this optimization in a scheme supporting separate compilation. Further, because the code generated as a result of this optimization is specific to a given program, performing this optimization introduces an entirely new set of program management problems into the compilation process.

For an optimizing compiler imbedded in a programming environment, the drawbacks of performing linkage tailoring may be reduced in a manner that makes this optimization worth considering. Linkage tailoring involves changing the calling sequence used at individual call sites to suit the specific situation at that site. The primary motivation is to speed up execution by reducing the overhead of individual procedure calls.

Work in this area is relevant to this dissertation because linkage tailoring is one optimization that is interesting to consider in the context of the program compiler. In particular, the program compiler computes interprocedural constant propagation information to assist in making an assignment of linkage styles to the call sites in a program. Further, any compiler that seriously tries to perform linkage tailoring will benefit from employing techniques for interprocedural constant propagation and recompilation analysis similar to those considered in this dissertation.

The four major linkage styles to be considered are closed, open, semi-open, and semi-closed.
• The standard linkage a compiler generates is a closed linkage. The compiler generates standard call and return sequences at a call site and standard entry and exit sequences for the called procedure.

• Generating an open linkage involves inline substitution of the procedure's body for the procedure call. This optimization does not really generate a linkage, it eliminates one at the cost of duplicating the procedure's body.

• Generating a semi-open linkage involves creating a private copy of the procedure body which is compiled together with the calling procedure. This type of linkage can be profitable in a situation where one procedure calls another procedure from multiple call sites. The compiler can use information about the two procedures to generate more efficient code for both the calling procedure and its private copy of the called procedure.

• Semi-closed linkages impose a compilation ordering on the procedures constituting a program. In a semi-closed linkage, the compiler uses information about the compilation of the called procedure to generate calling sequences for it. This information can describe register use, parameter passing mechanisms, and other run-time behavior. If the compiler is to generate such linkages, it must always compile called procedures before the procedures that call them.

These linkage styles are discussed in some detail by Allen and Cocke [AlCo 72].

The problem of constructing an optimal assignment of linkage styles to the call sites of a program is at least NP-hard [Sche 77]. In practice, approaching
this problem with even an approximative technique is difficult, because it involves estimating the improvement that is likely to occur for each potential linkage assignment. An accurate estimate would require very specific knowledge about the code generator and optimizer. Fortunately, even simple heuristics can lead to encouraging results [Hech 77].

This section discusses the research in the area of linkage tailoring done by Scheifler, Hecht, the ECS group, Ball, and Cooper. In particular, the estimation methods employed by the various researchers are explored and compared.

2.2.1. Scheifler

Scheifler analyzes the effects of inline substitution as a means of improving the performance of programs written in data abstraction languages like CLU [Sche 77]. In particular, Scheifler is interested in the tension between improving the execution time of the program and the growth in object code size resulting from inline substitution. When procedure integration is performed, the size of the object code produced by the compiler can increase dramatically. To prevent this from happening, Scheifler uses a size constraint on the generated code as his major criterion in deciding where to perform inline substitution.

Using simple heuristics to select call sites for procedure integration, Scheifler performs inline substitution until either all of the procedures in the program are integrated or the maximum allowable code size is reached. Call site selection is based on the estimated code size of the called procedure and the
execution frequency of the call site. The first candidates for substitution are procedures that do not increase the size of the object code when they are integrated. These include short routines and procedures which are called only once. Call sites are then preferentially selected based on the ratio of their execution frequency to the code size of the called procedure. Since inline substitution of these selected call sites can increase the size of the code generated for the program, integration must cease when the overall size constraint for the program is reached. At this point, there may be procedures in the program which are called only once, due to the changes in the program that result from inline substitution. These procedures are integrated if inline substitution can be performed without increasing the size of the program’s object code.

Scheifler’s results indicate that most of the procedure calls can be eliminated in programs that have a low degree of recursion. Furthermore, the object code produced for these programs is comparable in size and slightly faster than object code produced by traditional methods.

2.2.2. Hecht

Scheifler’s results portray the advantages of inline substitution as an isolated optimization. After substituting the procedure bodies, Scheifler performs no optimizations that take advantage of the improved quality of the data flow information available to the compiler as a result of the inline substitution. In his “modest quad improver” for the SIMPL-T language, Hecht
takes advantage of this information by performing local optimizations on the
code after the bodies of the procedures have been substituted [Hech 77].

To select call sites for integration, Hecht uses very simple heuristics which
require neither size estimates of the called procedure nor estimates of the
execution frequency of the call site. A procedure is considered for substitution if
it is invoked only once and it is not the main procedure. To preserve the
advantages of separate compilation, a procedure under consideration is rejected
if it is an externally defined procedure or an entry point. Because Hecht's
intermediate language is unable to handle arbitrary transfer of control,
procedures that have imbedded return statements must also be rejected.

Although he used a simple overall design for his quad improver, by doing
analysis and optimization after integration Hecht was able to reduce the number
of quads for the whole program by 5% to 15%, in spite of growth from
procedure integration.

2.2.3. Experimental Compiling System

The Experimental Compiling System (ECS) produced at IBM's T. J.
Watson Research Center explored a new methodology for constructing compilers
[AlCa 80]. The ECS approach was based on a uniform method of expressing
both the source language and the machine language representation of a
program. In ECS, the meaning of a source language construct is specified by a
defining procedure. The defining procedures are in turn specified by defining
procedures, until we reach a level of degenerate defining procedures. Because the semantics are expressed by a plethora of layers of small procedures, ECS uses inline substitution of defining procedures to produce machine code from the source language. To improve the quality of the resulting code, the inline substitution is followed by analysis and optimization phases.

2.2.4. Ball

Both Hecht's quad improver for SIMPL-T and the ECS compiler generator take advantage of the additional information about a called procedure which becomes available when the procedure is integrated. Yet, neither take into consideration potential optimization gains when selecting call sites for integration. Ball bases his selection of call sites for procedure integration on potential for improvement by optimization [Ball 79]. In particular, Ball looks for situations where inline substitution of the called procedure makes optimizations like constant propagation and test elision possible.

To estimate the improvements due to procedure tailoring which can result when a procedure is substituted, Ball computes the relationships between the values of variables in expressions contained in a procedure and the values of formal parameters and global variables on entry to the procedure. This relationship is expressed by Ball's parameter dependency sets, sets that are computed using standard global data flow techniques. The parameter dependency set for an expression contains those formal parameters of the procedure that are either directly or indirectly involved in the computation of
the expression. If the value of the expression can be determined from the initial values of the formal parameters and local constants of the procedure, the parameter dependency set for the expression is labeled *strong*. If the initial values of the formal parameters contained in the parameter dependency set for an expression are involved in the computation of the expression, but insufficient to determine its value, the set is labeled *weak*.

Using an execution frequency profile of the procedure and a model of the standard object code generated by the compiler for individual statements, Ball next computes estimates of execution time and code space for each statement in the procedure. These estimates are based on standard code templates that the compiler will generate for the statement in the absence of optimizations. For expressions that have strong parameter dependency sets, the savings that are possible if the values of the formal parameters in the parameter dependency set are known at compile time are computed and stored with the parameter dependency set. Using this information, Ball computes an estimate of the space and time savings that can be garnered by inline substituting a procedure at a call site and tailoring the procedure body for that call site.

2.2.5. Cooper

In his dissertation [Coop 83], Cooper suggests a scheme for performing linkage tailoring in a programming environment. Taking into consideration the advantages of performing interprocedural optimizations in a programming environment, Cooper proposes an algorithm that divides the analysis needed to
select call sites for linkage tailoring between an intelligent editor and an interprocedural analyzer.

Cooper's algorithm bases call site selection on both a call site's potential for optimization and its effect on the size of the object code generated for the procedure. A constraint is placed on the size of the object code generated for the program to prevent the exponential growth in code size that can result from linkage tailoring. A procedure's potential for optimization is estimated using analysis similar to that suggested by Ball [Ball 79]. Scheifler's heuristics [Sche 77], altered to take the optimization potential estimates into account, are then used to select call sites.

Based on information derived about both procedures involved at a call site, the analyzer identifies call sites where generation of customized procedure linkages appears profitable. In some cases, the analyzer may elect to clone the called procedure, assigning a specially optimized version of the called procedure to specific call sites. Because selecting an optimal assignment of linkage styles to call sites is impractical, Cooper proposes an approximate technique. The compiler only accounts for the impact of a custom linkage on procedures within a small call-chain distance of the call site. This simplification, coupled with hard constraints on growth in the overall size of the program, allows the compiler to assign linkage styles efficiently.

Cooper's algorithm is unique and interesting for a variety of reasons. Unlike previously described linkage tailoring techniques, it is designed to work
in a programming environment. It takes into account both optimization potential and growth in the size of the code generated for the program when selecting call sites for linkage tailoring. It also considers assigning linkages other than open and closed linkages to the selected call sites. Finally, it approximates the impact of assigning a custom linkage, allowing linkage types to be assigned more efficiently.
CHAPTER 3

The Program Compiler

3.1. Introduction

Traditionally, a compiler is a translator that takes the source language version of a program as input and produces an executable image of the program. In order to economically support large programs, the compilation process has been subdivided to improve its efficiency, notably with the introduction of separate compilation and linkage editing. To support programming in the large, interprocedural analysis, and ambitious optimization in the IR\(^R\) programming environment, the task of compiling a program has been split between two separate compilers: a program compiler and a module compiler.

The module compiler translates the source statements of an individual module into assembly code for the target machine. It performs global optimization and can, under instruction from the program compiler, take advantage of interprocedural information as a basis for both global and interprocedural optimization.

The program compiler examines the entire set of procedures that constitute a program and computes information that directs the construction of an
executable image of the program. This ranges from a list of modules that 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 that constitute the program.

3.2. Previous Work

One of the earliest examples of a program compiler is Feldman's make system [Feld 79]. Make allows the programmer to specify the relationships between the components of his program and the actions that make must take in response to a modification of some module in the program. In make's view, one file depends upon another if a change to the first file forces an update to the second file. In a UNIX-based program development environment, make is used to maintain programs by specifying relationships between source files, object files, and executable images.

The criteria that make uses to determine if an update is necessary is the time that a file was last modified. An update to a file is forced whenever a file that depends on it has been altered since the file's last modification. When make

1Notice that make defines a dependence such that the source depends on its executable.
is executed on a set of dependences, it uses the time stamp criterion to detect dependences that are no longer satisfied. *Make* then performs the specified actions to satisfy each of these dependences. In a typical programming application, this involves using compilers and the loader to create an executable image. *Make* relies on a user-provided description of the dependences in a program. Given this description, in the form of a "Makefile", *make* automates the process of constructing an executable image from the source files.

Because *make* relies on a user-provided graph of dependency information and a superficial criterion like the time stamp on a file to determine when a dependence is not met, its decisions on what should be recompiled are, by nature, somewhat imprecise. *Make* has no knowledge of how a module has been changed. It cannot differentiate between a modification that corrected a spelling error in a comment and one that added an assignment statement. Thus, changing a comment in a module leads to recompilation even though the modification has no semantic impact on the program. Furthermore, editing changes in a file that contains definitions, declarations, or constants shared by several modules will lead to the recompilation of every module or procedure that uses any of the information contained in the file, even procedures that do not use any of the definitions, declarations, or constants that were altered. In large software systems, the cost of these unnecessary compilations can be significant.

In practice, the problem of shared definitions in textual include files introduces enough unnecessary recompilations to merit special attention. Tichy
and Baker have developed an efficient method for reducing the amount of recompilation needed to ensure consistency in a program after a change has been made in one of its components [TiBa 85]. In particular, their method eliminates most of the redundant recompilations that arise when editing changes are made to a *context*, the part of the program that contains shared definitions, declarations, and constants. Their scheme is an interesting counterpoint to *make*. Where *make* bases recompilation decisions on a user provided graph of dependency information and an external attribute of the files that contain the program, Tichy and Baker base their recompilation decisions on information derived by analyzing the component parts of the program.

Under the scheme proposed by Tichy and Baker, a module in a program is recompiled if it has been modified or if a context which specifies definitions, declarations, or constants used in the module has been altered *in a way that will affect the module*. To determine if a module that uses some of the information specified in a context must be recompiled due to a modification of the context, the *change* set of the context and the *reference* set of the module are compared. The *change* set is computed by comparing the old and new versions of the context and isolating any changes between the two that could invalidate previous compilations. Items that were added, changed, or deleted would be included in the change set. The *reference* set associated with the module as a result of its dependence on the context contains items that were defined in the context and referenced either in the module or in another context upon which
the module also depends. Recompilation of a module that depends on a changed context is necessary only if the intersection of the change set of the context and the reference set associated with the module is non-empty.

Tichy and Baker's work is a significant advance over make for two reasons. First, it reduces the amount of recompilation required in response to changes in large systems. Second, it uses automatically derived information as a basis for its decision making process.

These two program compilers are important, not only as prior research in this area, but because they have had a practical impact on the development of software systems. Feldman's make has proven to be an incredibly versatile tool and has become a staple element of the UNIX environment. The cumulative hours of programmer time saved by make in its lifetime is inestimable. The ideas embodied in Tichy and Baker's work are a major step forward, in that they improve the resolution of its test for satisfaction of a dependence. Together, these ideas form the basis for a credible program compiler for use with traditional compilers.

3.3. The IR^n Program Compiler

The problem with performing interprocedural analysis and using the resulting information to do ambitious optimization is that it leads to complex compilation dependences between the procedures of a program. When a compiler uses interprocedural information to make decisions, the correctness of
the resulting executable code is a function of the state of the entire program at the time the decision is made. This introduces intricate compilation dependences between the components of a program. For a system using interprocedural information and separate compilation to be of practical interest requires the use of a program compiler that is far more complex than either of those already discussed.

Since the dependences caused by using interprocedural information are compiler induced dependences, and not dependences introduced by the programmer, they cannot be detected using the techniques proposed by either Feldman or Tichy and Baker. Instead, the dependences between the procedures of a program that arise as a result of interprocedural analysis and optimization must be tracked during the compilation process. Decisions to recompile to ensure a consistent executable image of a program or to identify procedures where recompilation is likely to achieve a major improvement in the efficiency of the generated code must be based on knowledge of these dependences.

The task of the IRⁿ program compiler is to detect the interprocedural compilation dependences in a program and determine what action should be taken to return the executable image of the program to a consistent state after changes have been made to the program. The ability to handle interprocedural information and dependences make the IRⁿ program compiler unique. Doing this requires several distinct passes over the program's call graph:
Pass 1: The *semantic modification detection pass* determines which modules must be recompiled due to alterations to the composition of the program or direct editing changes made to modules or contexts included in the program.

Pass 2: The *interprocedural information update pass* updates the flow insensitive interprocedural summary and aliasing information associated with the call sites and procedures of the program to a state consistent with the program's source text.

Pass 3: The *interprocedural constant propagation pass* calculates interprocedural constants used in the program.

Pass 4: The *compilation dependence detection pass* detects modules that require recompilation because changes in interprocedural data flow or constant information have invalidated assumptions made in their most recent compilation.

Pass 5: The *linkage tailoring pass* examines the prospects for generating customized procedure linkages in the modules already being recompiled.

Information from one pass can sometimes be used to limit the amount of analysis required in later passes of the program compiler. If, for instance, the semantic modification detection pass detects no modules that have been changed and no changes in the program's composition, then further analysis is not necessary. The executable image of the program is consistent with its source
text. When changed modules or changes in the program's composition are detected, the interprocedural information update pass can restrict its analysis to altered modules, modules involved in the program composition modifications, and modules that have interprocedural information that is affected by the changes in the program. The compilation dependence detection pass only needs to consider modules that have not already been marked for recompilation by the semantic modification detection pass. It can further restrict its attention to modules with interprocedural information that was modified by either the interprocedural information update pass or the interprocedural constant propagation pass. Finally, when looking for prospects for generating customized procedure linkages, the linkage tailoring pass only examines modules that were marked for recompilation in either the semantic modification detection pass or the compilation dependence detection pass.

3.3.1. Semantic Modification Detection Pass

The semantic modification detection pass computes a pair of lists. The first list consists of those modules added to the composition since the last compilation and those modules where semantic changes have been made to the source, declarations, type definitions, or constants since the last compilation. Modules in this list must be recompiled. The second list contains modules that have been deleted since the last compilation. These modules are of no further interest to the program compiler when it is compiling this specific program. If both lists are empty, neither further analysis nor recompilation is necessary. If
either of the lists is non-empty, the remaining passes of the program compiler must be invoked.

In the $\mathbb{R}^n$ programming environment, the program editor and the module editor simplify the process of detecting and recording information about changes to programs and modules. When the programmer makes a change, these editors record relevant information in the database for later use by the program compiler.

To create or modify source text in the $\mathbb{R}^n$ environment, the programmer invokes the module editor. The editor helps the program compiler in several ways. Because it understands the programming language, the editor differentiates between changes that do not affect the meaning of the module, like a change made to a comment, and semantic changes to the module. When a semantic change is made, the editor marks the module for consideration by the program compiler. Thus, some of the decisions that guide the execution of the program compiler are made at editing time.

Similarly, the program editor records the modules added to or deleted from the program's composition in each session. By leaving an updated list of modules added to the program and an updated list of modules deleted from the program in the database, the program editor simplifies the task of detecting modules that have been added to or deleted from the program since the last compilation.
Using information left by the editors, the program compiler can easily determine which modules must be recompiled due to editing changes. In a single pass through the composition, the program compiler sets the list of changed modules to include all of the modules marked by the module editor as changed. The modules included in the list of modules added to the program since its last compilation, information computed by the program editor, are also added to the list of modules to be recompiled. Because this list excludes modules where only non-semantic changes have been made, the program compiler avoids some of the unnecessary module recompletions that a system like `make` must undertake.

In fact, because the editor has more available contextual information than the analyzer designed by Tichy and Baker, the editor can, in some cases, make a more precise determination of which modules in a program must be recompiled. An environment with a module editor and a database for storing accumulated information about programs and modules provides a supportive framework for solving difficult information based problems like the one Tichy and Baker solved. Program compilers that are not imbedded in a part of a programming environment are forced to do analysis similar to Tichy and Baker's analysis to determine the impact of changing a shared definition.

From the information recorded by the program editor, the program compiler can identify all modules that have been deleted from the program's composition since the last compilation. These modules comprise the second list.
At this point, the program compiler can remove from the database any program specific information about modules that have been deleted.

3.3.2. Interprocedural Information Update Pass

Once the IR$^n$ program compiler has constructed the list of changed modules, it must incrementally update the flow insensitive interprocedural summary and aliasing information associated with the call sites and procedures in the program. The list of modules marked for recompilation and the list of modules deleted from the program since its last compilation are needed for the interprocedural information update pass. Using these lists, information stored in the database by the editors, and the results of previous interprocedural analysis passes, the program compiler brings the interprocedural information into a state that is consistent with the source text of the program. The algorithms involved and their time complexities are described by Cooper and Kennedy [Coop 83][CoKe 84].

3.3.3. Interprocedural Constant Propagation Pass

After the interprocedural summary and aliasing information for the program is updated, the program compiler performs interprocedural constant propagation on the program. When a variable has a constant value on all paths entering a given procedure, we say that the variable is set to an interprocedural constant. The interprocedural constants detected during the interprocedural constant propagation pass provide information useful in many optimizations,
including loop unrolling based on constant valued parameters, and eliminating code controlled by conditionals based on constant valued variables. They can also serve as a basis for estimating improvements gained from customized procedure linkages, particularly inline substitution.

Techniques for performing interprocedural constant propagation are presented in chapter 4 of this dissertation. The algorithms presented range from expensive, precise techniques to cheap approximations. Some of the interprocedural constant propagation algorithms presented do not rely on interprocedural summary and aliasing information. When this is the case, the interprocedural constant propagation pass can be merged with the interprocedural information update pass.

3.3.4. Compilation Dependence Detection Pass

Once the interprocedural data flow and constant information is known, the compiler can determine which modules need to be recompiled due to changes in their interprocedural information. At compile time, the module compiler uses interprocedural information about one procedure to reason about optimizations in other procedures. As a result, either a change in the composition of the program or a semantic modification made to a procedure in the program can invalidate optimization decisions made in previous compilations of procedures not directly modified by the programmer. Chapter 5 describes methods to detect the compilation dependences introduced between the procedures of a program as a result of using interprocedural information in the compilation
process.

This pass of the program compiler need only examine modules which are not already on the list of modules that must be recompiled. Any module placed on the list of changed modules in the semantic modification detection pass can be safely ignored in the compilation dependence detection pass.

3.3.5. Linkage Tailoring Pass

After the recompilation dependences have been computed and a list of modules for recompilation determined, the program compiler should consider the issue of generating custom procedure linkages for call sites in the modules already being recompiled. While generating customized linkages can produce a significant run-time savings, it is difficult to assess when a customized linkage is sufficiently important to justify an additional recompilation. For this reason, the program compiler takes a conservative attitude toward linkage tailoring and only reconsiders linkage tailoring decisions involving modules being recompiled for other reasons.

The program compiler examines each call site in the procedure being recompiled, applying an estimation technique similar to the technique proposed by Cooper [Coop 83]. Based on the estimated improvement, the estimated execution frequency, and space constraints, the program compiler suggests a linkage type for the call site. The program compiler will suggest an open, semi-open, or closed linkage, or suggest that the procedure body be cloned so that
different linkages can be considered for each clone.

An incremental approach is of use here because it avoids recompilations due to linkage tailoring. Linkage tailoring decisions for the entire call graph of the program are considered only when all of the procedures in the program are marked for recompilation.

3.3.6. Recompilations

After performing these analytical passes, the program compiler invokes the module compiler on each module in the list it has accumulated. Using the results of these compilations and the previously compiled versions of any modules unaffected by editing changes, it then produces an executable image for the entire program.
CHAPTER 4

Interprocedural Constant Propagation

4.1. Introduction

Constant propagation involves detecting variables whose values are constant and determining the range over which the value of the variable remains constant. The constants detected by constant propagation are interesting because they make it possible to perform a variety of optimizations. These include loop unrolling, dead code elimination, evaluation of expressions at compile time, and simplification of array addressing computations.

A variety of techniques exist to detect situations in which a variable has a constant value over some range of a single procedure [WeZa 85]. In general these global constant propagation techniques do not find all of the constants that are available in the procedure, but they do find most of them. In practice, there are a number of methods that may prove to be of practical value in the program compiler, even though they do not detect every available constant. The published algorithms for global constant propagation vary in both the class of constants they detect and their time complexity.
The effectiveness of global constant propagation is demonstrated by the fact that it is included in global optimizing compilers. Extending constant propagation to handle interprocedural constants, constants that are passed to procedures through global variables or formal parameters, has the potential to increase the number of constants that are available for use by the global optimizer.

Several complications introduced by procedure calls prevent the straightforward application of the techniques used in global constant propagation to the interprocedural problem. Because the interprocedural problem involves dealing with information passed at call sites, it faces several complications that do not arise in the global problem. The presence of aliasing complicates the detection of constants. To ensure that a variable is constant, it is not only necessary to ensure that its value is not changed, but also that the values of its potential aliases are not changed. Myers has shown that interprocedural constant propagation is Co-NP-complete in the presence of aliasing [Myer 81].

Because algorithms to find all of the constants in a program are potentially exponential, unless P = NP, the program compiler looks for a subset of those constants which are useful in optimization. Specifically, it attempts to discover, for each procedure, those formal parameters and global variables which have constant values on entry to the procedure from each possible call site. Such values are called interprocedural constants.
If the program compiler determines that different sets of constants enter a procedure from different call sites, it may elect to use that information to produce different versions of the called procedure tailored for use at the individual call sites. When the program compiler is constrained to produce a single compiled version of each procedure, it must tell the module compiler to assume only that information which is true at every call site that invokes the procedure. If the analyzer uses only those constants which hold at every call site in its propagation, the resulting information is only a flow insensitive approximation to the underlying flow sensitive problem. Additionally, when the aliasing information that is available for the called procedure is flow insensitive information, any constant information derived using it has a flow insensitive component.

This chapter examines the problem of propagating constants across procedure call boundaries and presents a set of algorithms for interprocedural constant propagation. First, a precise but expensive method based on inline substitution is discussed. Second, an inexpensive approximation technique is described. This method computes a rough approximation to the information derived by the precise algorithm. Third, a family of intermediate cost, intermediate precision techniques is presented. Finally, an alternative approach which uses information derived in the interprocedural summary computation is described.
Throughout this chapter, a program is represented by a call multi-graph \( G = (N,E) \). \( N \), the nodes of the call multi-graph, represents the procedures. \( E \), the arcs of the call multi-graph, represents call sites in the program. To simplify the discussion of interprocedural constant propagation, global variables will be treated as implicit formal parameters of every procedure.

4.2. A Precise Method

The first interprocedural constant propagation technique arises naturally from the desire to avoid the complications introduced by procedure calls. This technique performs inline substitution on all non-recursive call sites in the program. At recursive call sites, we can unroll the recursion once and then generate normal procedure calls to handle further recursion.\(^1\) Once the procedure calls have been removed, a standard global constant propagation algorithm, like Wegman and Zadeck's algorithm [WeZa 85], is applied to the expanded program to do the actual analysis. The class of constants detected by this technique and the precise time complexity of the process is dependent on the global constant propagation algorithm used.

It is important to keep in mind that the global constant propagation is being performed on an enlarged program. Multiple calls to a single procedure will cause the overall size of the program to grow as the substitutions are performed. Since this growth is potentially exponential [Sche 77], the time

\(^1\)Alternatively, an explicit stack can be introduced and the recursions converted into explicit iteration. While this suggestion is less than elegant, it does eliminate all recursive call sites in the program, allowing complete inline substitution.
complexity of the precise method is potentially exponential.

Even though inline substitution can lead to a significant increase in the size of the program, there are good reasons to examine this method. Inline substitution removes aliasing that may appear in a procedure by making those aliasing relationships explicit. Because inline substitution expands individual paths through the call multi-graph, the aliasing information represented in the explicit variable relationships is much more precise than that computed using flow insensitive techniques. Inline substitution restores flow sensitivity to the constant propagation information by making an individual copy of the procedure for every call site.

Because inline substitution removes aliasing, restores flow sensitivity, and allows us to use effective global constant propagation techniques to detect interprocedural constants, more interprocedural constants can be detected with this method than with any technique that leaves procedures intact. The flow insensitive nature of other techniques causes them to not propagate globals and parameters which may be constant in some, but not all, invocations of a procedure. Similarly, when only flow insensitive aliasing information is available, constant valued global variables and formal parameters that have aliases in some, but not all, invocations of a procedure may be missed due to potential aliases to non-constant valued variables.

Using inline substitution, the constant propagation algorithm detects situations in which a procedure sets a call-by-reference formal parameter or a
global variable to a constant value before returning. This happens because
global and interprocedural analysis are combined, and because the parameter
renaming is made explicit. The other methods discussed in this chapter will not
detect such constants.

When inline substitution is applied, a single algorithm can be used for both
global and interprocedural constants. Since only one constant propagation pass
is necessary, compilers that use this technique can be made smaller and more
maintainable. Furthermore, the analysis uses existing global constant
propagation techniques to detect both global and interprocedural constants.
Such techniques are already well understood. They have been shown to be
efficient and effective for detecting constants [WeZa 85].

It is important to note that this inline substitution technique discovers
every constant detectable by the underlying global algorithm, unless some
recursive call sites remain in the program. Dealing with these call sites would
require the introduction of flow insensitive approximations.

4.3. An Inexpensive Approximation Technique

Techniques that rely on flow insensitive information can only approximate
the results possible when the precise technique of section 4.2 is used. Because
the space explosion caused by inline substitution increases the complexity of the
analysis, it is worth considering less expensive methods which do not use inline
substitution, even though they produce less accurate information. This section
presents an inexpensive interprocedural constant propagation technique and several methods for computing the initial information required for the propagation pass. This technique computes only a rough approximation to the information produced by the inline substitution technique.

One of the least expensive approaches open to the program compiler is to simply detect local constants that appear in the parameter list of a procedure call and propagate these constants to the called procedure. The propagation can be done in a single pass through the procedures of the program. For each formal parameter of a procedure \( n \in N \), the value of the formal parameter is determined taking by the intersection of the values for the formal parameter that are propagated from the call sites that invoke \( n \). Note, that unless all of the call sites that invoke \( n \) propagate the \textit{same} constant value, the formal parameter will not be constant valued. Since this technique propagates only local constants, it is necessary to propagate the value of a parameter at a call site only once. Furthermore, the constants can be propagated in any order. The propagation analysis requires time proportional to \( O(AP) \) where \( AP \) is the sum over all call sites of the number of actual parameters at the call site. If the number of actual parameters at a call site can be bounded by a small constant, the propagation analysis takes time proportional to \( O(E) \).

Prior to performing the propagation phase of this technique, the local constants available at each call site must be computed. The literal constants available at a call site can be detected by the module editor as the programmer
enters the text of the call site and recorded as module-version specific information in the database.

If it is desirable to propagate more than just literal constants, additional initial analysis can be used to generate local constants that are not literal constants. One method of finding these initial constants is to perform a global constant propagation on each procedure. The cost of this analysis depends on the global constant propagation technique used.

This initial analysis can be done either in the module editor or in the program compiler, depending on the desired precision of the information. If the analysis is done in the module editor, the information produced is module-version specific because it is computed in the absence of interprocedural information. If the analysis is done in the program compiler, the global constant propagation can take advantage of the interprocedural aliasing and summary information computed in the interprocedural information update pass of the program compiler instead of making worst case assumptions whenever it detects a call site in the procedure. In this case, the information produced by the global constant propagation is program-version specific information.

The constants detected by this technique can be used to clone a procedure and tailor individual copies of it to take advantage of propagated constants. In particular, this simple scheme would easily detect opportunities for unrolling loops in which the loop stride is a literal valued actual parameter. Dongarra has demonstrated the effectiveness of this trivial interprocedural optimization
This method is inexpensive, but the class of constants that it detects is small. Only global constants are propagated at each call site. Constants that can propagate into a procedure from a call site that invokes it are not considered in this analysis. Hence, constants can only traverse a single edge of the call multigraph before they are ignored. This technique may also fail to detect some constants because the global constant propagation relies on either worst case assumptions about the side effects of procedure calls or flow insensitive interprocedural summary and aliasing information when it encounters a call site. As a result, the constant information computed can be, at best, a flow insensitive approximation.

The inexpensive technique presented in this section is limited in that it only propagates constants along a single edge in the call multi-graph. A constant that is passed to one procedure will not be propagated through any call sites in that procedure. In software libraries like LINPACK [DBMS 79], this is an important consideration. These libraries often take parameters for array dimensions and loop strides and pass them on to lower level routines like the BLAS. In the absence of constant propagation information, the optimizer will be unable to take advantage of situations where loops can be unrolled and array addressing calculations simplified. Methods that are capable of propagating constants through procedures and along multiple call multi-graph edges are capable of detecting constants that would make these optimizations possible.
4.4. Intermediate Approaches

This section presents another family of techniques for computing flow insensitive approximations to the constant information that can be derived using the inline substitution method from section 4.2. These techniques represent an improvement over the precision of the inexpensive approximations because they are capable of detecting constants that propagate across multiple edges of the call multi-graph. Because they are based on flow insensitive aliasing and summary information and contain the implicit assumption that a single copy of each procedure is being generated, they detect fewer constants than the inline substitution method.

These intermediate cost techniques involve two distinct types of analysis: tracking constant values across the edges of the call multi-graph and computing the impact of constant valued formal parameters on the set of constants available at each call site in the procedure. The first problem is an interprocedural problem. The second problem is primarily an intraprocedural problem; it may be desirable to use interprocedural summary and aliasing information to improve the precision of the intraprocedural phase.

4.4.1. Interprocedural Analysis

To solve the interprocedural problem, assume that each edge of the call multi-graph is annotated with a set $\text{PassedConstants}(e)$ containing constants propagated by the call site and that each node is annotated with a set
Constants\( (m) \) containing those constants known to hold on entry to the procedure. The elements of the \( \text{PassedConstants}(e) \) and \( \text{Constants}(m) \) sets are tuples of the form \(<\text{actual parameter, value}>\) and \(<\text{formal parameter, value}>\), respectively. Further, assume the existence of a function \( \text{jump}(\text{Constants}(m),e) \) which creates \( \text{PassedConstants}(e) \) from \( \text{Constants}(m) \).

The \( \text{Constants}(n) \) set for procedure \( n \) can be calculated directly from the \( \text{PassedConstants}(e) \) sets of the call sites invoking \( n \). This can be described by a set of simple equations.

\[
\text{Constants} \ (n) = \bigcap_{e = (m,n)} f_e (\text{PassedConstants} \ (e))
\]

where \( f_e \) maps the actual parameters at the call site in \( m \) into the corresponding formal parameters of procedure \( n \). Substituting the jump function into the equation yields:

\[
\text{Constants} \ (n) = \bigcap_{e = (m,n)} f_e (\text{jump} (\text{Constants} \ (m),e))
\]

which describes the values of \( \text{Constants}(n) \) in terms of the \( \text{Constants}(m) \) sets of the procedures that call \( n \). Given a mechanism for computing \( \text{jump}(\text{Constants}(m),e) \) for any \( e \), the set of simultaneous equations describing the interprocedural constant propagation can be solved using standard techniques from global data flow analysis. An iterative worklist approach, similar to the one described by Hecht [Hech 77], is one method that can be used to efficiently solve these equations. If the cost of computing the jump function is proportional to some function of the attributes of a procedure \( n \), \( g(n) \), then the cost of the intermediate approaches is proportional to the sum over all
procedures in the program of $FP(n)\cdot g(n)$. Here, $FP(n)$ is the number of formal parameters in procedure $n$.

This equation describes the process of propagating constants along all edges leading into a procedure. It does not describe the internal details of the jump function. Three different jump functions are presented in this section.

The inexpensive approximation technique of the section 4.3 is a trivial example of this type of framework. For the inexpensive technique, the jump functions return the same value independent of any knowledge about constant valued formal parameters or global variables. Thus, the interprocedural solver traverses each edge only once.

4.4.2. Intraprocedural Analysis

The evaluation of jump functions should be as inexpensive as possible because, during the solution of the interprocedural constant propagation problem, it may be necessary to make more than one pass over each procedure. The solver will need to evaluate the jump function for an edge every time it traverses the edge.

4.4.2.1. Global Constant Propagation

The most obvious method for computing jump functions is to use global constant propagation as the jump function. The results of the global constant propagation trivially indicate which actual parameters are constant at a given call site. As stated previously, a variety of such algorithms are available. These
algorithms can be differentiated by their time complexity and by the class of constants that they detect.

Using a global constant propagation to implement jump functions has the advantage that it produces information that is as precise as possible, given the flow insensitive nature of the interprocedural information that it uses. The main disadvantage is that it requires a complete global constant propagation on the procedure every time its incoming interprocedural constant information changes. Thus, in an iterative implementation, a global constant propagation is required at every node visit. Because this has the potential to be expensive, it is desirable to investigate other techniques for implementing jump functions.

4.4.2.2. DEF-USE Chains

An inexpensive mechanism for propagating constant valued formal parameters through a procedure can be based on DEF-USE chains. It is likely that DEF-USE chains will already have been computed in the editor and will be available in routines that are to be compiled. The DEF-USE chains can be used to detect any constant formal parameters that are passed, without change, as actual parameters to other procedures. Since passing parameters in this manner often occurs in code that involves calls to library routines, this is an important case to catch.

Given DEF-USE chains, the algorithm to detect these constant parameters is simple. If a parameter at a call site has a single DEF-USE chain which connects
the use as an actual parameter with a definition as one of the formal parameters
of the procedure, then that actual parameter is a candidate for constant
propagation. The DEF-USE chains can be examined to produce a mapping from
the formals of a procedure to the actuals at each of its call sites. During the
interprocedural constant propagation phase, if a formal parameter is determined
to be constant, this map can be used to propagate its constant value to each call
site with parameters that are solely dependent on the constant formal parameter
for their value.

This method of implementing jump functions is much less expensive than
the algorithm using global constant propagation at each step. This is true for
several reasons. First, the method relies on DEF-USE chains, which may already
have been computed for use in the module editor or in other optimizations.
Second, the information used to propagate the formal parameters of a procedure
to the actual parameters at call sites within the procedure can be calculated in
one pass through the procedure and stored, making it unnecessary to analyze
the procedure every time a new constant is propagated to the procedure.
However, this method has the disadvantage that it does not detect all of the
constants found when using global constant propagation. This method pipelines
consstants that enter the procedure as parameters through to the call sites in the
procedure. It ignores interaction which the incoming constants might have with
the procedure body, interactions that might create more constant valued actual
parameters.
Performing an initial global constant propagation on each procedure to detect those constants in the procedure that can be propagated as actual parameters is one means of detecting some of the constants that are missed by the pipelining method. With this addition, the pipelining technique will detect both formal parameters and local constants that are passed as actual parameters. Additional constants that arise from interactions of interprocedural constants with the procedure body will still be missed.

4.4.2.3. Symbolic Constant Propagation

To more closely approximate the results of using constant propagation and still maintain some of the advantages of pipelining constants through the procedure along the DEF-USE chains, we need to be able to calculate more accurately the relationship between the formal parameters of the procedure and the actual parameters at its call sites. Symbolic constant propagation provides one method of computing this relationship.

To symbolically compute the relationship between the formal parameters of a procedure and the actual parameters at call sites within the procedure, assign symbolic constants to each of the formal parameters and global variables associated with the procedure. Perform global constant propagation on the procedure using one of the many published global constant propagation algorithms. The chosen algorithm will need to have its test for equality modified because, when symbolic constants are used, it is necessary to be able to test expressions containing symbolic constants, not just literal constants, for
equality. The equality test can be greatly simplified if a canonical order for the symbolic constants is defined and the constant expressions are kept in this canonical order as they evolve.

The symbolic constant propagation will divide the actual parameters into three categories: actual parameters that are always constant and have a known value, actual parameters that can be constant if certain formal parameters and global variables are constant, and actual parameters that are not constants. When computing the jump function for the procedure, only the constants in the first two categories are of interest because the non-constant actual parameters are irrelevant to interprocedural constant propagation.

For actual parameters that are always constant, symbolic constant propagation will compute the value of each of the actual parameters. The values can be used to initialize the interprocedural constant propagation. The result of the jump function for each pass over the procedure will include these actual parameters as constant parameters.

For actual parameters that can be constant if certain formal parameters and global variables are constant, the symbolic constant propagation will have computed expressions containing symbolic constants. Given constant values for all of the symbolic constants in the expression associated with an actual parameter, the constant value of the actual parameter can be determined.

If the interprocedural constant propagation determines that some of the formal parameters and global variables associated with the procedure are
constant, there may be actual parameters in the second category that are constants. The value of these constants can be computed from the expression that is computed for the actual parameter during the symbolic constant propagation.

For each pass over the procedure during the interprocedural propagation phase, determining which of the potentially constant actual parameters are constant is inexpensive. Once the symbolic constant propagation is completed, for each call site e, NecessaryConstants(p,e), a set delineating which formal parameters and global variables each potentially constant actual parameter, p, depends upon, can be computed from the expression describing the value of that actual parameter. From the interprocedural constant propagation we have a set that describes the constant valued formal parameters and global variables of the procedure, Constants(m). Given this information, we can determine that the potentially constant actual parameters that are constant are those actual parameters whose NecessaryConstants(p,e) set is a subset of the Constants(m) set of the procedure. These constant actual parameters can then be propagated.

When we use symbolic constant propagation instead of DEF-USE chains to determine which constants can be propagated through a procedure, we detect a larger class of constants. Symbolic constant propagation can detect constant valued actual parameters that are computed from a combination of the values of local constants and constant formal parameters. The pipelining method only propagates formal parameters that pass unchanged through a procedure to a use
as an actual parameter. Like the information used in the pipelining method, the
information from the symbolic constant propagation can be computed once for
the procedure and the results used during each interprocedural iteration through
the procedure.

Symbolic constant propagation produces many of the constants that global
constant propagation on every interprocedural iteration would produce.
Symbolic constant propagation will not detect constant valued variables that are
constant if and only if the constant valued formal parameters upon which they
depend have a particular value or range of values. Because symbolic constant
propagation uses symbols to represent the values of formal parameters, the
equality test it must apply when two paths meet finds a subset of the constant
valued expressions detected by global constant propagation.

4.5. An Alternative Technique

The previous sections of this chapter propose a variety of methods for
detecting interprocedural constants, methods whose precision varies with the
complexity of the scheme proposed. These techniques are of general interest
because they do not require specific features inherent in the \( \mathbb{R}^n \) programming
environment for successful implementation, nor are the details of the \( \mathbb{R}^n \)
program compiler and the information that it collects intrinsic to any of these
schemes. While general methods have the advantage that they can be used in a
variety of environments to solve the interprocedural constant propagation
problem, they have the disadvantage that they do not effectively use
information and features that are unique to the environment in which they are being implemented.

The alternative scheme described in this section is based on information that the IR^n program compiler computes, collects, and stores during the interprocedural information update pass of the program compiler. A description of the algorithms involved in the pass and the information that they compute is presented by Cooper and Kennedy [CoKe 84][Coop 83].

This algorithm uses a relation produced by the interprocedural summary computation, map*. This relation describes, for each formal parameter, all of the formals to which it can eventually be bound. Used another way, it describes all of the formal parameters which can eventually reach a given formal parameter.

For formal parameters, the algorithm proceeds as follows:

1. Initialize all formal parameters to ⊤, indicating no knowledge of their value.
2. For each procedure, do a global constant propagation.
   Examine each call site in the procedure. For each constant valued actual parameter, one that is not in MOD for the call site, propagate it to each other parameter it can reach, using map*.
   The value of the target parameter is unchanged if the propagated constant and the target parameter share the same value. If only one of them has the value ⊤, the target parameter is set to the other value. Otherwise, the target parameter is set to ⊥, indicating a non-constant value.
3. For each formal parameter which is marked as a constant, use map* to check its ancestors for the value ⊤. If one is set to ⊤, set the parameter's value to ⊥.
To handle global variables, the MOD computation for global variables can be slightly modified to produce a mapping similar to $map^*$ for global names. Given the existence of this mapping, the technique can propagate constant values for global variables in a manner analogous to that used for formal parameters.

This algorithm produces conservative information about the constant values of formal parameters and global variables in the program. Because the MOD test used to determine which constants are propagated is very imprecise, only constants that are universally used as constants are marked as constants by this algorithm.

It is possible that further repetitions of the second step would reveal more constant values. The $n^{th}$ iteration of step two would reveal symbolic expressions discovered in the $(n-1)^{st}$ pass of an iterative implementation of the technique from section 4.4 which used symbolic constant propagation as the jump function. Because of the crude nature of the MOD test, it seems unlikely that more than one repetitions would be worthwhile.

It is difficult to compare the class of constants detected using this technique to the class of constants detected by other techniques. While the $map^*$-based technique described in this section is likely to catch many of the constants detected by the inexpensive technique of section 4.3, it will almost certainly miss some of those constants due to the lack of precision in the MOD test. On the other hand, the alternative technique does detect some constants that are
propagated over multiple procedure boundaries. The inexpensive technique detects constants that have traversed at most one procedure boundary. Like the technique described in section 4.3, this algorithm requires a constant number of passes over the program’s call multi-graph.

In practice, this method will detect a class of constants that may prove to be important to global optimization. Variables containing dimensioning information about arrays, information about loop strides, and other “structural” information that is defined once and used in multiple locations without modification will be detected.

4.6. Conclusions

This chapter discussed several techniques for performing interprocedural constant propagation, ranging from expensive techniques which rely on inline substitution to detect a large class of interprocedural constants to a simple, inexpensive technique which detects and propagates only those constants that appear as literal constants at call sites. An experimental study of the expense versus precision trade-off inherent in these techniques is needed to determine which of these interprocedural constant propagation algorithms should be used in the interprocedural constant propagation pass of the program compiler.
CHAPTER 5

Recompilation Algorithms

5.1. Introduction

One of the primary functions of the program compiler is identifying procedures that must be recompiled to ensure that the executable image of a program is consistent with the current state of its source text. The program compiler begins this process by constructing lists of procedures that must be recompiled because of direct modification of the program or the source text of modules included in the program. It then ensures that interprocedural summary, aliasing, and constant propagation information is current, in its second and third passes. The fourth pass of the program compiler uses information computed in the previous two passes to discover modules that need recompilation because of changes in the interprocedural environment in which they have been compiled.

This chapter focuses on the analysis required to discover this latter set of procedures. The analysis need only consider modules not included on the list of procedures developed in pass one; if a module is already being recompiled, there is no need to analyze it using the techniques presented in this chapter.
SUBROUTINE EXAMPLE()

COMMON A
...

DO 10 I = 1, 100
...

A = 2.0
...

CALL S1()
...

10 CONTINUE
...

RETURN
END

Figure 5.1

To understand the recompilation problem, consider the program fragment shown in Figure 5.1. Assume that the only modification to variable A in the procedure is in the assignment statement shown. Assume also that the call to S1 is the sole procedure call inside the loop. Under these conditions, an optimizing compiler might choose to move the assignment to A outside of the loop. Unfortunately, this changes the location of the assignment statement relative to the repetitive calls to S1.
If the value of A is not modified in S1, the optimizing compiler can move the assignment statement outside of the loop. Performing this optimization, however, makes the correctness of the resulting code dependent on the fact that A is not modified in S1. If S1 is later altered so that it contains the statement $A = A \times 5.0$, the optimized code produced for EXAMPLE is no longer correct. Not only is the value of A potentially modified when S1 is invoked, the value of A on entry to S1 will be referenced prior to the modification of A. In general, if S1 or any procedures invoked as a result of executing S1 are edited in a manner which adds A to the MOD set associated with the call to S1, recompilation of EXAMPLE may be necessary. If A is referenced in S1 or any procedures directly or indirectly called from S1 before it is modified, EXAMPLE must be recompiled.

In spite of the importance of this problem, there is little or no published research which deals directly with the recompilation problem. Burke and Kennedy discussed the problem, resulting in a brief note [Burk 83]. The approach taken there is similar to that described in section 5.4.1; they also looked at common subexpression elimination.

The remainder of this chapter presents techniques for analyzing the impact of the compilation dependences introduced by the use of interprocedural information in an optimizing compiler.
5.2. A Simple Recompilation Algorithm

When working with an optimizing compiler that uses interprocedural information, the simplest way to ensure that an executable image of a program is correct, consistent, and fully optimized is to recompile all of the components of the program whenever any part of the program is changed. Previous compilers that used interprocedural information appear to have taken this approach [Hech 77][AlCa 80]. While safe, this method is expensive, discarding all of the economies of separate compilation. If it is necessary to recompile an entire program as a result of making an editing change that has only local side effects, using interprocedural information as a basis for optimization becomes less attractive.

The amount of recompilation required in response to a local change can be drastically reduced by examining precisely what interprocedural information the module compiler relied upon in compiling each module. The IRn programming environment uses the techniques developed by Cooper and Kennedy [CoKe 84] to annotate each statement in a procedure with MOD and USE information and computes aliasing information using methods similar to those presented by Cooper [Coop 85]. The algorithms discussed in this chapter deal with a program's call multi-graph, G = (N,E). A node n ∈ N in the call multi-graph represents a procedure in the program. An edge e = (m,n) ∈ E of the call multi-graph represents a call site in the program.
MOD and USE information are associated with the statements in a procedure. The information required for recompilation analysis is call site specific information. Since a single statement can contain multiple call sites, we need a set which is call site specific. Thus, for each call site, we require a set $E_{mod}(e)$ and a set $E_{use}(e)$ containing the call site's contribution to the statement's MOD and USE sets. For a procedure $m$, we need a set $E_{mod}(e)$ and a set $E_{use}(e)$ for each call site in procedure $m$. The $E_{mod}(e)$ set contains the names of variables that may be modified as a result of calling procedure $n$ from procedure $m$ at a specific call site. $E_{use}(e)$ includes variables that may be used, or referenced, in procedure $n$ when it is invoked at call site $e$. Both $E_{mod}(e)$ and $E_{use}(e)$ account for aliasing effects in procedure $m$. If a variable $v$ is in $E_{mod}(e)$ or $E_{use}(e)$, any variable in procedure $m$ to which $v$ is aliased is also in the appropriate interprocedural summary information set. $E_{mod}(e)$ and $E_{use}(e)$ are flow insensitive sets, so they contain variables that might be modified as a result of the call, as opposed to variables that must be modified.

Consider a program compiled with optimizations based on this interprocedural information. Modifying one of its procedures in a manner that impacts the procedure's interprocedural summary sets can necessitate the recompilation of the procedures that invoke it, either directly or indirectly. This happens when the compilation of the calling procedure depends on an interprocedural fact that the editing change invalidates. Similarly, modifying a procedure in a way that changes the aliasing environment it passes to a called
procedure can necessitate the recompilation of the called procedure and other procedures it invokes.

It is important to realize that not all changes in the summary information require recompilation of other procedures. Recall that $E_{mod}(e)$ and $E_{use}(e)$ are flow insensitive summary sets associated with a call site $e$. They contain the variables that can be modified and used as the result of executing a procedure call. Because these sets describe events that may not take place, the optimizer does not perform optimizations based on the presence of a variable in a MOD or USE. Since the optimizer already accounts for the flow insensitive nature of the information, recompilation is not necessary when a modification or use is removed from an interprocedural summary information set.

Similarly, a modification of the aliasing patterns passed on at a procedure call may not necessitate recompilation. Like $E_{mod}(e)$ and $E_{use}(e)$, the aliasing information associated with a procedure $m$, $A_{alias}(m)$, is flow insensitive. As a result, optimizations are not performed based on the presence of an alias pair in the aliasing information. Recompilation is, therefore, not necessary when an alias pair is removed from the aliasing information.

If an element is removed from one of the interprocedural information sets associated with procedure $m$, it is not necessary to recompile $m$ for the sake of correctness. Removing an element from an $E_{mod}(e)$, $E_{use}(e)$, or $A_{alias}(m)$ set associated with $m$ simply means that object code generated for $m$ using the previous information is conservative but correct. It may be desirable to
recompile procedure $m$ to capitalize on new optimizations opened up by the changed information, but such recompilation is not required to maintain a correct and consistent compilation of $m$.

Whenever an element is added to an interprocedural set associated with $m$, it may be necessary to recompile $m$ to insure the correctness of its compilation. Optimizations based on the invalidated data flow information may be incorrect under the new information. Using this fact, we can construct a simple test to determine if it is necessary to recompile a procedure based on a change elsewhere in the program. Recompilation of a procedure $m \in \mathcal{N}$ may be necessary if any alias pairs are added to its $Alias$ set or any variables are added to the $Emod(e)$ or $Euse(e)$ sets associated with a call site $e$ in the procedure. More formally, recompilation of a procedure $m$ is not necessary if

$$Alias(m) \subseteq Alias_0(m)$$

and for all edges $e = (m,n) \in \mathcal{E}$

$$[Emod(e) \subseteq Emod_0(e)] \land [Euse(e) \subseteq Euse_0(e)]$$

where $Alias_0(m)$, $Emod_0(e)$, and $Euse_0(e)$ denote the interprocedural information sets used in the previous compilation. The cost of performing this test is proportional to the number of call sites, so it requires $\mathcal{O}(E)$ set operations.

An improvement in the code generated for the procedure may be possible if information is deleted from one of the interprocedural aliasing or summary sets. There may be optimizations that were inhibited during the last compilation by the presence of the deleted information. Those procedures that have modified
interprocedural information but need not be recompiled for safety should be considered for this type of recompilation.

The simple algorithm decreases the amount of recompilation required to ensure consistency in response to an editing change to some procedure in the program. This test does not eliminate all unnecessary recompilation; newly added interprocedural information may actually be irrelevant to the correctness of the previous compilation of a procedure. However, this algorithm does reduce the amount of recompilation at a minimal cost, the cost of storing the interprocedural information used in the last compilation of the procedure and the cost of computing the answer to the test above.

5.3. Restricted Simple Recompilation Algorithm

The simple recompilation algorithm of section 5.2 reduces the number of unnecessary recompiations without relying on analysis of the source of the procedure under consideration. As a result, recompiations are initiated when variables or alias pairs have been added to the $E_{mod}(e)$, $E_{use}(e)$, or $Alias(m)$ information of a procedure $m$, regardless of whether or not the addition of the variable to the interprocedural information can actually change the correctness of the compiled version of $m$. In particular, if a variable not referenced in a procedure $m$ is added to one of the interprocedural summary sets associated with $m$, procedure $m$ will be recompiled even though a change in the interprocedural information associated with the unused variable could not possibly affect the correctness of its previous compilation.
Such recompilations are clearly unnecessary, yet they do arise in practice. Global variables, like those in FORTRAN common blocks and C source files, are often accessible by many procedures that do not use them. If recompilations were restricted so that they occur only when the elements added to an interprocedural information set associated with a procedure \( m \) are referenced in \( m \), further savings would be possible. The recompilation tests presented in the remainder of this section are inexpensive. Detecting the need for recompilation in a program as a result of changes in interprocedural summary and aliasing information can be accomplished in \( O(E) \) set operations using these tests.

5.3.1. Aliasing

Consider interprocedural aliasing information. When a new alias pair is added to the \( \text{Alias} \) set associated with a compiled procedure, a new dependence is created between the two variables. Since the dependence was not there at compile time, the module compiler may have made optimizations that placed one or both of the variables in registers. The compiled version of a procedure can be invalidated by addition of an alias pair to its aliasing information if and only if the members of the alias pair are being accessed in a manner that causes their loads and stores overlap. Were it possible to efficiently compute, for every possible alias pair in a procedure, information about potentially dangerous overlaps between the use of one member of the pair and the storage of the other member of the pair in a register, we could determine precisely when recompilation due to the addition of a new alias pair was necessary. However,
computing this information is difficult and expensive because it depends intimately on the design of both the optimizer and the code generator.

Less precise information about the global variables and formal parameters used in a procedure can be computed by the module editor at editing time. In a single pass through the source text, it is possible to determine which variables are referenced, either loaded or stored, in a procedure. Since recompilation due to a new alias pair is necessary only in cases where both members of the pair are referenced in an overlapping manner, recompilation is only needed if one member of the pair is referenced in the procedure and the other member of the pair is referenced in the procedure or a procedure that it invokes, directly or indirectly.

Let $\text{Referenced}(m)$ represent variables visible outside procedure $m$ which are loaded or stored in $m$. $\text{Referenced}(m)$ can trivially be computed in the module editor using initial information sets that the module editor computes for the interprocedural information update pass of the program compiler. These sets, $\text{IMOD}(m)$ and $\text{IUSE}(m)$, contain variables visible outside $m$ that are modified or used in $m$, respectively. Modifications and uses that arise when procedures are invoked in $m$ are not included in these sets. $\text{Referenced}(m)$ is the union of $\text{IMOD}(m)$ and $\text{IUSE}(m)$.

Let $\text{Referenced}^+(m)$ represent variables visible outside procedure $m$ which are loaded or stored in $m$ or procedures that are invoked by $m$. $\text{Referenced}^+(m)$ can trivially be computed from information derived in the interprocedural
information update pass. This pass computes GMOD(m) and GUSE(m), sets containing those variables visible outside procedure m which are modified or used, respectively, in m or a procedure that it directly or indirectly invokes [CoKe 84]. Referenced⁺(m) is the union of GMOD(m) and GUSE(m).

Using the Referenced(m) and Referenced⁺(m) sets described above, the recompilation test can be stated as follows. Recompilation of a procedure m ∈ N as the result of a change in its aliasing information is necessary only if there exists an alias pair <u,v> such that

\[<u,v> \in [\text{Alias}(m) - \text{Alias}_0(m)]\]

\[\land\]

\[u \in \text{Referenced}(m) \land v \in \text{Referenced}⁺(m)\]

When this test is applied, at most \(O(N)\) set operations are needed to determine which procedures in a program should be recompiled due to changes in aliasing information.

Using information that it has acquired about the procedure from the module editor, the restricted simple recompilation algorithm reduces the number of recompilations caused by the addition of new alias pairs. By employing a similar strategy, the number of procedures that should be considered for recompilation in an effort to increase the run-time efficiency of the compiled code can be reduced. When it is not necessary to recompile a procedure to insure correctness, it is only desirable to recompile a procedure if the recompilation could lead to an improvement in the code generated for the
procedure. In the case of aliasing, recompilation as the result of the deletion of an alias pair from \textit{Alias} could only lead to an improvement in the code generated for the procedure if the alias that existed in the previous compilation prevented optimizations from occurring. Since an existing alias can only prevent optimization of procedure \( m \) if one of the variables in the alias pair is referenced in the source code of \( m \) and the other variable is referenced in \( m \) or a procedure that \( m \) directly or indirectly invokes, it is only necessary to consider recompilation to achieve a code improvement if there exists an alias pair \(<u,v>\) such that

\[
<u,v> \in [\text{Alias}_0(m) - \text{Alias}(m)] \\
\land \\
[u \in \text{Referenced}(m) \land v \in \text{Referenced}^+(m)]
\]

5.3.2. Summary Information

When the simple recompilation algorithm is used, a procedure is recompiled if a variable is added to one of the interprocedural summary information sets associated with a call site in the procedure. As in the case of aliasing, this criterion forces recompilation of the procedure even when neither the added variable nor any of its aliases are used by the procedure.

The recompilation test for summary information can be made more precise by insuring that recompilation of a procedure as the result of the addition of a variable to an interprocedural summary information set only occurs when the
variable or one of its aliases is used in the procedure. Since $E_{mod}(e)$ and $E_{use}(e)$ already account for aliasing information in the calling procedure, $m$, the addition of a referenced variable to the interprocedural summary information sets can easily be detected. Under this scheme, recompilation of a procedure $m \in N$ is necessary only if there exists a variable $v$ and an edge $e = (m,n) \in E$ such that

$$v \in [E_{mod}(e) - E_{mod}(e)] \cup [E_{use}(e) - E_{use}(e)]$$

\[\land\]

$$v \in \text{Referenced}(m)$$

As with aliasing, we can also restrict the times when recompilation for the sake of improving the code generated should be considered. The deletion of a variable from an $E_{mod}(e)$ or $E_{use}(e)$ set associated with a procedure $m$ can lead to a situation where the code previously compiled for $m$ can be improved. This can only happen if that variable or one of its aliases was referenced in procedure $m$ when it was last compiled. The program compiler should only consider recompilation to improve optimization in a procedure $m \in N$ if there exists a variable $v$ and an edge $e = (m,n) \in E$ such that

$$v \in [E_{mod}(e) - E_{mod}(e)] \cup [E_{use}(e) - E_{use}(e)]$$

\[\land\]

$$v \in \text{Referenced}(m)$$

If the module compiler restricts itself to preserving the relative ordering of loads, stores, and procedure calls, this test can be made more precise. The
Referenced sets in the interprocedural summary information tests can be replaced with a set containing only those variables loaded in the procedure. Stores to variables that have not been loaded can only lead to incorrect code where the position of the store relative to a procedure call has been changed based on an interprocedural summary set.

By restricting the tests used by the simple algorithm to consider only those global variables and formal parameters that are either loaded or stored in a procedure, many of the spurious recompilations that are irrelevant to the correctness of the program are eliminated. This additional accuracy is possible because the restricted test takes the source of the procedure into account when making the decision to recompile. The additional accuracy is available at minimal extra cost: the cost of determining which global variables and formal parameters are referenced in the procedure, the cost of storing Referenced information, and the cost of the additional set membership operations required in each test.

5.4. More Precise Recompilation Algorithms

Were precise information describing the set of interprocedural information upon which the module compiler relied available, it would be possible to further reduce the amount of recompilation necessary in response to an editing change. In particular, the simple algorithms of sections 5.2 and 5.3 force recompilation of a procedure m when information is added to the Emod(e), Euse(e), or Alias(m) sets associated with the procedure, regardless of whether or not the change
impacts the correctness of the generated code. In many cases, the module compiler may not have found a specific interprocedural fact helpful in optimizing a procedure. There are likely to be facts that the module compiler ignored, but the recompilation algorithms will deem significant. With more precise knowledge of how the interprocedural information is used in the compilation, some of these recompiations could be eliminated.

Studying the types of optimization performed in an optimizing compiler and the *global* data flow information required for these optimizations provides insights into the construction of more precise algorithms. We will consider two classes of *global* optimizations: in the first, removing an element from a *global* data flow set mandates recompilation, while in the second, adding an element to a global data flow set mandates recompilation. Global common subexpression elimination is an example of the former, while eliminating register stores after the last use of a variable is an example of the latter. Each class of optimization requires a different type of recompilation information.

To understand how recompilation information can be collected for these two types of optimizations, we will examine the *global* data flow information necessary to perform global common subexpression elimination and register store elimination in a procedure.\(^1\) This requires a small amount of notation. Let the procedure be represented by its data flow graph, \(G = (N, E, n_0)\). Each node \(n \in N\) represents one of the procedure's basic blocks, a sequence of statements

\(^1\)Kennedy's formulation of the data flow equations is used in this discussion [Kenn 81].
with no control flow branches. The edges $e = (m,n) \in E$ represent control flow between two basic blocks, represented by $m$ and $n$. The control flow enters the procedure through its entry node $n_0$.

### 5.4.1. Global Common Subexpression Elimination

When two or more instances of the same expression are separated by code that does not redefine any of the variables present in the expression, the result of the first evaluation of the expression can be saved and used to replace the later re-computations of that expression. This optimization is known as global common subexpression elimination. In order to perform this optimization, it is necessary to know which expressions are available at various points in the procedure. To represent this information, we compute a set $\text{AVAIL}(b)$ for each basic block $b$. $\text{AVAIL}(b)$ contains expressions which have been computed and not redefined by assignment to one of the variables in the expression, along every path entering $b$. These expressions are available on entry to $b$. $\text{AVAIL}$ information can be calculated from local information, using standard data flow techniques. This calculation is rapid, in the sense of Kam and Ullman [KamU 78]. Solution of the following system of equations yields the desired result:

$$\text{AVAIL}(b) = \bigcap_{a \in P(b)} (\text{DEF}(a) \cup (\text{AVAIL}(a) \cap \text{NKILL}(a)))$$

where $P(b)$, the predecessor set of $b$, is defined as

$$P(b) = \{ a \in N \mid (a,b) \in E \}$$
and $\text{DEF}(a)$ and $\text{NKILL}(a)$ are sets of local information for basic block $a$. $\text{DEF}(a)$ contains those expressions computed in $a$ and not subsequently redefined in $a$. $\text{NKILL}(a)$ describes the set of expressions that are not redefined in $a$.

In the absence of interprocedural information, the AVAIL analysis must assume that a procedure $n$ kills every variable it can access when invoked at a call site $e$. This means that the NKILL set for a block $b$ containing the call site must exclude every actual parameter and every global variable that can be modified in $n$. In the presence of interprocedural information, only variables in $E_{\text{mod}}(e)$ must be excluded from $\text{NKILL}(b)$. $E_{\text{use}}(e)$ plays no role in the AVAIL computation.

If a variable $v \in E_{\text{mod}}(e)$, $v$ is not in $\text{NKILL}(b)$ for the block $b$ containing the call site $e$ because any expression containing $v$ may be redefined when the called procedure is invoked. Thus, any expression $x \in \text{AVAIL}(b)$ for some block $b$ depends on the fact that its constituent variables are not in $E_{\text{mod}}(e)$ for any call site between the expression's most recent evaluation and $b$, on each path leading to $b$. If the module compiler uses the fact that $x \in \text{AVAIL}(b)$ to eliminate a reevaluation of $x$, the correctness of that decision relies on the $E_{\text{mod}}(e)$ sets for the appropriate call sites. To insure correct code, recompilation becomes necessary when any of the variables in $x$ are added to one of these $E_{\text{mod}}(e)$ sets. Since $E_{\text{use}}(e)$ information is not pertinent to the AVAIL information collected for global common subexpression elimination, only $E_{\text{mod}}(e)$ and $\text{Alias}(m)$ information need be considered when determining whether to recompile a
procedure m.

If it were possible to determine the set of call sites, $Calls$, between the last
computation of an expression and the point where a re-evaluation is replaced,
along all paths leading to the statement containing the replaced expression, it
would be possible to construct a more precise set of information upon which to
base our recompilation decision. Assume $Calls(x, b)$, for an available expression $x$
and a basic block $b$, is easy to construct. Whenever an available expression is
used in an optimization at some location in procedure $m$, the module compiler
can construct, for each of the procedures invoked by a site in $Calls(x, b)$, a set
$MustNotMod(e)$ containing variables that the optimizer assumes are not in the
$Emod(e)$ set for the call site. Since interprocedural information can include only
the globals of procedure $m$, the actual parameters at the call site, and aliases of
these variables, $MustNotMod(e)$ need only contain such variables.

Given $MustNotMod(e)$ for all call sites in $m$, recompilation of $m \in N$ is
necessary only if there exists an alias pair $<u, v>$ such that

$$<u, v> \in [Alias(m) - Alias_0(m)]$$

\[ u \in Referenced(m) \land v \in Referenced^+(m) \]

or there exists a variable $v$ and an edge $e = (m, n) \in E$ such that

$$v \in [Emod(e) \cap MustNotMod(e)]$$

This recompilation test is inexpensive, requiring at most $O(E)$ set operations to
determine which procedures in the program must be recompiled.
Let the elements of $AVAIL(b)$, $DEF(b)$, and $NKILL(b)$ be replaced with tuples of the form $<\text{name, calls}>$ where name is the literal name associated with the available expression, $x$, under consideration and calls is the set $\text{Calls}(x, b)$. The components of a tuple $z \in AVAIL(b)$ are referenced as $z$.name and $z$.calls. Furthermore, let the calls sets of each $x \in DEF(b)$ contain all call sites occurring in $b$ after $x$ is evaluated. Let the calls sets of each $x \in NKILL(b)$ contain all call sites in $b$. By carefully redefining the operators and local sets in the $AVAIL$ computation, the necessary information about call sites can be accumulated as a part of the $AVAIL$ computation. The intersection and union operators used in the $AVAIL$ computation are defined as follows:

$$X \cap Y = Z = \{ z \mid \begin{cases} 
  \text{t.e. } x \in X, y \in Y \\
  \text{with } x.\text{name} = y.\text{name} = z.\text{name} \\
  \text{and } z.\text{calls} = x.\text{calls} \cup y.\text{calls} 
\end{cases} \}$$

$$X \cup Y = Z = \{ z \mid \begin{cases} 
  \text{if t.e. } x \in X, y \in Y \text{ with } x.\text{name} = y.\text{name} \\
  \text{then } z.\text{name} = x.\text{name} \\
  \text{and } z.\text{calls} = x.\text{calls} \cup y.\text{calls} \\
  \text{or} \\
  \text{if t.e. } x \in X \text{ and t.e. no } y \in Y \text{ with } x.\text{name} = y.\text{name} \\
  \text{then } z = x \\
  \text{or} \\
  \text{if t.e. } y \in Y \text{ and t.e. no } x \in X \text{ with } y.\text{name} = x.\text{name} \\
  \text{then } z = y 
\end{cases} \}$$

Using these definitions, for each $x \in AVAIL(b)$, $x$.calls contains the set $\text{Calls}(x, b)$ needed for producing the $\text{MustNotMod}(e)$ information described above. In spite of the change in the operators and local sets in the $AVAIL$ computation, calculation of the new $AVAIL$ and $\text{Calls}(x, b)$ information is still rapid, in the sense of Kam and Ullman [KaUl 76]. The expense involved in this recompilation test
may lie in producing a $\text{MustNotMod}(e)$ set for each call site. $\text{MustNotMod}(e)$ is computed by the module compiler during its optimization phase. As a result, the cost of computing this information is dependent on the design of the optimizing compiler and the type of optimizations that it performs.

5.4.2. Eliminating Unneeded Register Stores

The module compiler need not store a value back into memory from a register if it is certain the value cannot be used later in the program. This eliminates an unneeded register store operation. LIVE information specifies when a value will be used again. Using LIVE information, the module compiler can perform the register store elimination optimization.

A variable is live at a point in a procedure if there exists a path from that point to a use of the variable and if that path contains no assignments to the variable. $\text{LIVE}(b)$ contains variables that are live on entry to basic block $b$. Using standard global data flow techniques, $\text{LIVE}$ information can be calculated using the following equation:

$$\text{LIVE}(b) = \text{IN}(b) \cup \bigcup_{a \in \mathcal{S}(b)} (\text{THRU}(b) \cap \text{LIVE}(a))$$

In this equation, $\mathcal{S}(b)$, the successor set of $b$, is defined as

$$\mathcal{S}(b) = \{ a \in \mathcal{N} \mid (b,a) \in E \}.$$  

$\text{IN}(b)$ is the set of variables that are used in $b$ before being redefined. Variables in $\text{IN}(b)$ are live on entry to $b$. $\text{THRU}(b)$ is the set of variables that are not redefined in $b$. 
When LiVE information is computed in the absence of interprocedural information, the module compiler must assume that any variables visible to a called procedure $n$ at a call site may be used in procedure $n$ before being redefined. This assumption can increase the live ranges of variables and inhibit application of the register store elimination optimization. Using interprocedural $Euse(e)$ sets reduces the set of variables assumed LiVE because of a call. $Emod(e)$ information is not pertinent to the computation of LiVE information because both LiVE and $Emod(e)$ information are flow insensitive. Regardless of whether or not the variable is in $Emod(e)$, there may be a path through the called procedure along which the variable is not redefined. It is also possible that the variable in $Emod(e)$ is used before it is redefined on a path through the procedure. To perform the elimination of unneeded register store backs, it is necessary to know which variables are not live at a point. Hence we must assume that a variable is still in LiVE($b$), even if it is in the $Emod(e)$ set of a call site that appears in $b$.

Since $Emod(e)$ information is not used in the LiVE computation, only $Euse(e)$ and $Alias(m)$ information need to be considered when making the recompilation decision. If, for each call site in the procedure, we know which uses of variables in the called procedure could invalidate optimizations made in the calling procedure, recompilation tests more precise than those described in sections 5.2 and 5.3 are feasible. Given a $MustNotUse(e)$ set for each call site in $m$, recompilation of $m$ is necessary only if there exists an aliasing pair $<u,v>$.
such that

\[
\langle u, v \rangle \in [\text{Alias}(m) - \text{Alias}_0(m)] \\
\& \\
[u \in \text{Referenced}(m) \& v \in \text{Referenced}^+(m)]
\]

or there exists a variable \( v \) and an edge \( e = (m, n) \in E \) such that

\[v \in [\text{Euse}(e) \cap \text{MustNotUse}(e)]\]

This computation is inexpensive, requiring at most \( O(E) \) set operations to determine which procedures in the program must be recompiled.

In order to annotate each call site in the procedure with a \( \text{MustNotUse}(e) \) set, we must first compute, for each basic block \( b \) in the procedure, the set of call sites that could invalidate an optimization made in \( b \). Optimizations based on flow insensitive live information are invalidated when the life of a variable is extended. This occurs when a use of the variable or one of its aliases is added after the current last use of the variable. Hence, the only call sites that can potentially invalidate an optimization are those that occur after the point where the optimization occurred. For each basic block \( b \), this set of call sites can be collected by simultaneously solving

\[\text{Calls}(b) = \bigcup_{a \in S(b)} (\text{NewCalls}(a) \cup \text{Calls}(a))\]

with the live equation. \( \text{NewCalls}(a) \) is the set of call sites in basic block \( a \). \( \text{Calls}(b) \) is the set of call sites that occur after basic block \( b \). This calculation is rapid in the sense of Kam and Ullman [KaUl 76].
An optimization made in block $b$ for variable $v$ can be invalidated by call sites in $b$ and $Calls(b)$. To insure the correctness of this optimization, we add $v$ to the $MustNotUse(e)$ set associated with a call site for each site in $Calls(b)$ if $v$ or any potential alias of $v$ is an actual parameter of that call site or a global variable of the calling procedure. It must also be added to call sites in $NewCalls(b)$ that occur between the optimization and the end of the block. This provides us with the $MustNotUse(e)$ set that we used in our recompilation test. As with the $MustNotMod(e)$ sets, the cost of computing the $MustNotUse(e)$ sets is dependent on specific details about the optimizer.

5.4.3. Generalization

By examining two specific optimizations, we discovered two techniques which lead to recompilation tests more precise than those given in sections 5.2 and 5.3. The two problems differed in the nature of the global data flow information on which the optimization is based. In global common subexpression elimination, the underlying data flow problem is AVAIL analysis, which is flow sensitive, while the LIVE problem which is the basis for eliminating unnecessary register stores is a flow insensitive problem.

For flow sensitive global problems, the type of call site tracking required for AVAIL analysis is necessary. In these problems, recompilations are caused by the deletion of an element from the global data flow set. Thus, the recompilation algorithm must know which call sites occurred on the path over which the data flow information propagated, from the data flow event that created the specific
fact to the site of the optimization based on the fact.

For flow insensitive problems, recompilations are mandated by the growth of global data flow sets. Thus, the LIVE analysis needs to know the set of call sites that can potentially add to the global information. This requires the simpler $Calls(x,b)$ computation of section 5.4.2.

5.5. Multiple Procedures in a Compilation Unit

In the treatment of the recompilation problem in this chapter, we have usually referred to recompiling individual procedures as a result of the dependences inherent in a program that has been compiled using interprocedural aliasing and summary information. In the $\mathbb{IR}^n$ programming environment, it is possible to have multiple procedures treated as an indivisible compilation unit known as a module. When multiple procedures are contained in a module, it is not necessary to determine the dependences between the procedures in the module because, by definition, these procedures will always be compiled together. Variables that are global to the procedures in the module but not visible outside of the module need not be added to $MustNotMod(e)$ and $MustNotUse(e)$ sets unless they are used as actual parameters, because they cannot be changed. The recompilation test only requires call site specific information that describes the dependences between the procedures in a module and the procedures called inside the module but not contained in the module. Such information can be trivially constructed from the recompilation information computed in this chapter.
5.6. Interprocedural Optimizations

When interprocedural optimizations are applied to the procedures of a program, dependences are introduced between the procedures of the program. Changes made to one procedure in the program may invalidate interprocedural optimizations made to another procedure in the program. Unless the dependences introduced by the interprocedural optimizations are known, the entire program must be recompiled when semantic changes are made to any of the modules.

In order to preserve the advantages of separate compilation, it is necessary to determine what information the optimizer relied upon when it decided to perform the interprocedural optimization. The types of information used by different optimizations vary depending on the optimization being performed. For example, the optimizer depends on the information provided by interprocedural constant propagation when it performs optimizations, like loop unrolling and simplification of array addressing, which are based on the presence of interprocedural constants. Using tests similar to the simple tests of section 5.3, recombinations forced by changes in the interprocedural constant information can be detected.

As discussed in chapter 4, \textit{Constants}(m) contains those constants known to hold on entry to procedure \( m \). Elements of \textit{Constants}(m) are tuples of the form \(<\text{v}, \text{value}\>\). The variable under consideration as a constant is \( \text{v} \). Its value is stored in the value field of the tuple. Since the optimizer may have performed
optimizations in procedure m based on the presence of interprocedural constants in \( Constants(m) \) and the use of those constants in procedure m, deleting a constant from \( Constants(m) \) could make it necessary to recompile m. Let \( Constants_0(m) \) represent the interprocedural constants that were known to hold on entry to procedure m at the time that it was compiled. Then, recompilation of procedure m, as a result of a change in its interprocedural constant information, is necessary only if there exists a tuple \(<v,\text{value}>\) such that

\[
<v,\text{value}> \in [Constants_0(m) - Constants(m)]
\]

\[
\land
\]

\[
v \in Referenced(m)
\]

This recompilation test is inexpensive. At most \( O(N) \) set operations are required to perform this test on a program.

Recompilation may be desirable if a new constant is added to \( Constants(m) \) because further optimizations might be possible, given this additional knowledge. Recompilation to improve the run-time speed of the generated code for procedure m, as a result of a change in its interprocedural constant information, should be considered only if there exists a tuple \(<v,\text{value}>\) such that

\[
<v,\text{value}> \in [Constants(m) - Constants_0(m)]
\]

\[
\land
\]

\[
v \in Referenced(m)
\]
The tests in this section indicate when recompilation is necessary for correctness or desirable for possible code improvements as a result of a change in the interprocedural constant information. Similar recompilation tests, which detect when the dependences introduced by other interprocedural optimizations have been invalidated, can be developed in an analogous fashion.

5.7. Conclusions

Compiling a program in the presence of interprocedural information introduces dependences between its procedures which complicate the question of what to recompile when a change is made in the program. In the absence of information about these dependences, all procedures in the program must be recompiled when a change is made to insure the correctness of the code. This chapter described several techniques for reducing the amount of spurious recompilation that is required when all procedures are recompiled. Sections 5.2 and 5.3 discussed simple tests for reducing the amount of recompilation required when interprocedural information is used in the compilation process. Section 5.4 described more expensive tests that further reduce the number of unneeded recompileations by collecting information about the optimizations that use the interprocedural information during compilation. An example of the type of recompilation decisions that are required when interprocedural optimizations are involved is given in section 5.6. Further experimental work is needed to determine whether the simple tests or precise, more expensive tests are preferable in an actual compiler.
CHAPTER 6

Conclusions

If the impact of collecting interprocedural information and performing optimizations is carefully considered when an environment is designed, an optimizing compiler can benefit from being located in the environment. The database and tools of an environment can effectively be used to collect and record information which the compiler needs. In particular, the database can serve as a repository for the source text and information associated with procedures and programs in the environment, making it easy for the compiler to access information as needed. Since an ambitious optimizing compiler is planned for the IR\textsuperscript{n} programming environment, the impact of performing interprocedural analysis and optimization in the environment has been considered in its design. An overview of the impact of interprocedural analysis and optimization on the IR\textsuperscript{n} programming environment was presented in this dissertation. Emphasis was focused on the manner in which interprocedural information impacts the ambitious optimizing compiler included in the environment.

To support interprocedural analysis and ambitious optimization in the IR\textsuperscript{n} programming environment, the task of compiling a program was split between
two separate compilers, a program compiler and a module compiler. This structure separated the interprocedural concerns, embodied by the program compiler, from the intraprocedural concerns, which were handled by the module compiler.

The program compiler updates the interprocedural information computed for each of the call sites in the program, determines which procedures in a program should be recompiled, selects call sites where linkage tailoring can be effectively applied, and invokes the module compiler to generate optimized code for each of the procedures marked for recompilation. Techniques for computing interprocedural constant propagation information and for detecting when a procedure should be recompiled due to changes in its interprocedural information were examined in detail.

A variety of algorithms that find interprocedural constants were presented in chapter 4. The algorithms range from a simple, inexpensive technique which detects and propagates only those constants that appear as literal constants at call sites to expensive techniques which rely on inline substitution to detect a large class of interprocedural constants. A class of intermediate approaches was proposed in section 4.4. These methods all use the same interprocedural propagation technique, but differ in the mechanism used to model intraprocedural propagation. This propagation is modeled by jump functions, that compute the values of constant-valued actual parameters as a function of constant-valued formals. An alternative technique which detects only constants
that retain the same constant value throughout the program, a class of constants that may include dimensioning information, loop strides, and other structural information about the program, was detailed in section 4.5.

The use of interprocedural information in compilation gives rise to subtle and complex compilation dependences between the procedures of a program. Algorithms that, based on an understanding of the interprocedural dependences involved, limit the amount of recompilation necessary after procedures in the program have been changed were discussed in chapter 5. The methods described included those which rely only on interprocedural information to make recompilation decisions as well as those which base recompilation decisions on a combination of interprocedural information and information derived about the procedures when they were last compiled.

Further work is needed to determine which of the proposed interprocedural constant propagation methods and recompilation algorithms should be employed in the IR* programming environment. Since an expense versus precision tradeoff is involved in both the interprocedural constant propagation methods and the recompilation algorithms, an implementation of the algorithms discussed and pragmatic experience running programs through the optimizing compiler could be used to rank the various algorithms. Finally, good heuristics for assigning linkage types to call sites and better techniques for estimating the improvement that will result if a special linkage is created at a call site are also areas in which future research is warranted.
**BIBLIOGRAPHY**


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


[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.


