RICE UNIVERSITY

Managing Interprocedural Optimization

by

Mary Wolcott Hall

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Chairman
Professor of Computer Science

Keith D. Cooper
Associate Professor of Computer Science

Linda Torczon
Research Associate

Robert Bixby
Professor of Mathematical Sciences

Houston, Texas

April, 1991

Managing Interprocedural Optimization

Mary Wolcott Hall

Abstract

This dissertation addresses a number of important issues related to *interprocedural optimization*. Interprocedural optimization is an integral component in a compilation system for high-performance computing. The importance of interprocedural optimization stems from two sources: it increases the context available to the optimizing compiler, and it enables programmers to use procedure calls without the concern of hurting execution time.

While important, interprocedural optimization can introduce some significant compile-time costs. When interprocedural information is used to optimize a procedure, the procedure is then dependent on those interprocedural facts. Thus, even if the procedure is not edited, it may require recompilation due to changes in the interprocedural facts. In addition to these effects on recompilation, interprocedural information can also be expensive to compute. Furthermore, interprocedural optimizations can increase program size which can in turn increase compile time. To make interprocedural optimization feasible in a compilation system, it must be possible to manage the compile-time costs.

This dissertation explores some open questions in interprocedural optimization. An efficient algorithm is developed for constructing the call multigraph, the underlying program representation for interprocedural optimization. A procedure cloning algorithm is also described. This algorithm avoids the significant code growth and increased compilation time possible with cloning, while focusing optimization on the most profitable opportunities. We present results of an in-depth study of inline substitution. These results led to a new approach for interprocedural optimization, focusing on enabling only high-payoff optimizations. All of these ideas are brought together in a compilation system supporting interprocedural optimization, which significantly reduces the costs of interprocedural optimization. The compilation system is further adapted to support optimizations for enhancing parallelism.

Acknowledgments

I would like to thank my committee, particularly Ken Kennedy, Keith Cooper and Linda Torczon. All three have provided insight, guidance and nurturing from the beginning of my graduate career.

Throughout the course of this research, countless people have given feedback on my work. I would particularly like to thank Preston Briggs, Steve Carr, Paul Havlak, Kathryn McKinley and Tom Murtagh for their significant contributions to this dissertation. The ParaScope research group has offered support and insight, as well as the underlying implementation upon which mine is based. Paul Havlak, Seema Hiranandani, Ivy Jorgensen, Chuck Koelbel, Kathryn McKinley, Rene Rodriguez, Jerry Roth, Leah Stratmann and Chau-Wen Tseng helped proofread my thesis and provided helpful comments.

The experiment described in Chapter 3 would have been impossible without support from many individuals and organizations. The list of supporters is quite long, but a few people offered significant assistance. In particular, Argonne National Labs provided access to their machines, Steve Wallach provided access to a Convex, Randy Allen helped me understand the Titan compiler, and Vicky Dean and the system support staff at Rice assisted in numerous ways.

I would like to thank the National Science Foundation, IBM Corporation and Control Data Corporation for supporting this research.

Finally, I would like to thank my husband Mark, my parents and the rest of my family for everything they did to make this goal achievable.

Contents

Illustrations

Chapter 1

Programming Environment Support for Interprocedural Optimization

Scientific programs are typically characterized by numerous floating point computations, which result in long execution times. Accordingly, scientific programmers are often very concerned about execution time performance. Their programs are usually written in FORTRAN, a fairly natural language for expressing mathematical formulas, but limited enough that it allows a compiler to generate good quality object code.

The $\mathbf{R}^n$/ParaScope programming environment was designed to support the special needs of scientific FORTRAN programmers [CCH+87] [CCH+88].[1] A primary goal of ParaScope is to provide a vehicle for investigating code optimization. From the beginning, a major component of the compilation system has been support for optimization across procedure boundaries.

There are two reasons why interprocedural optimization is needed to support compilation of scientific programs. First, the optimizing compiler can be much more effective with interprocedural information [MS91]. Otherwise, in the presence of procedure calls, the compiler must assume that a procedure will both use and change every variable accessible to it. These worst-case assumptions restrict optimization of procedures that contain calls to other procedures. This restriction is particularly limiting for high-level optimizations that span multiple statements in the program source. If the statements involved cross a procedure boundary, the compiler cannot perform the optimization. This is unacceptable since high-level optimizations can produce significant improvements in program performance.

The second reason for interprocedural optimization is to enable programmers to use a good programming style. Programmers requiring efficiency may modify their programming style to compensate for limitations in the compiler. In particular, programmers concerned about the overhead of procedure calls often write monolithic

[1] $\mathbf{R}^n$ was originally designed to support programming for scalar architectures. In recent years, $\mathbf{R}^n$ has been extended to include support for user-assisted and automatic parallelization, and has been renamed to ParaScope. For the rest of the dissertation, we refer only to ParaScope.

procedures. This leads to programs that are difficult to build, debug and maintain. If provided with a compiler that can effectively optimize across procedure boundaries, the programmers may be willing to make extensive use of procedures, resulting in a better programming style.

Interprocedural optimization has been gaining attention recently in the research community. Still, very few commercial compilers perform optimizations across procedure boundaries. This is because interprocedural optimization introduces compile-time costs that are considered too great to make them worthwhile. First, interprocedural analysis interferes with separate compilation. Separate compilation restricts recompilation to only those modules that have been edited since the last compile. Once optimizations cross procedure boundaries, a procedure is dependent upon the interprocedural facts used to optimize it. This makes it possible for a procedure to require recompilation even if it has not been edited.

There may be additional costs associated with certain interprocedural transformations. In this dissertation, we cite as a further cost growth in program size. Increases in program size can substantially increase compile time, as well as storage requirements for the program. With program growth, there is also the danger that increased memory requirements will cause execution time performance to suffer.

ParaScope is designed to overcome these difficulties of interprocedural analysis and optimization. Through a shared central database and a group of cooperating tools, a large portion of the information needed to perform the optimizations is known prior to compilation. Preliminary information for interprocedural analysis is accumulated during editing, eliminating the need to examine source code during analysis. We further manage the costs by limiting interprocedural optimization to situations that are likely to produce a noticeable run-time improvement. Recompilation requirements are reduced with analysis to determine compilation dependences. This research addresses the following important issues related to interprocedural optimization:

- What interprocedural optimization techniques are worth consideration, and when is one technique more effective than another?
- How can the compiler predict whether application of a transformation is likely to improve execution-time performance?
- How can the need for recompilation be minimized when interprocedural transformations have been applied?
- How can all of these ideas be effectively incorporated into a practical programming environment?

This chapter introduces the transformation techniques to be discussed in the dissertation and provides the ParaScope framework for interprocedural optimization. The first section briefly describes the transformation techniques. Section 1.2 describes the support required from the programming environment to permit efficient interprocedural optimization. Section 1.3 presents the *program compiler*, the tool in the environment that provides program-level management of optimization. In section 1.4, we explain the differences in the interprocedural optimization strategy for parallelizing compilers. Section 1.5 presents work related to the ParaScope framework for interprocedural optimization. The final section gives an overview of the remaining chapters of the dissertation.

1.1 Interprocedural Transformations

One of the important goals of this research is to understand the relative advantages of interprocedural transformation techniques. In this dissertation, we describe a system where all interprocedural optimization is realized with some combination of three techniques: (1) *inline substitution*, (2) *procedure cloning*, and (3) global optimization enhanced by *interprocedural data-flow information*.

With inline substitution, a call site is replaced by the body of the called procedure. Formal parameters in the procedure body are replaced by actual parameters at the call. Because the code appears in place of the call site, inline substitution can provide the best possible context to the optimizer. (However, this is only true if all calls in the program are inlined.) An additional benefit of inlining a call is that the overhead associated with procedure calls is eliminated. Previous research has shown that inline substitution on a limited class of procedures can improve the object code by eliminating call overhead, without dramatically increasing the program size [Sch77]. Others have suggested that inlining can have a more significant impact on program performance if followed by optimization [Hec77]. We undertook a study to understand the efficacy of inlining, and the significance of its associated costs. In this dissertation we describe our experience with inlining and present a strategy for restricted inlining.

Procedure cloning involves making a *clone*, or copy, of a procedure. Each copy can then be optimized to more closely match the interprocedural environment for a particular group of calls to the procedure. Cloning is useful when calls to a procedure can be partitioned into groups, with each group having distinctly different interprocedural information. This permits the refining of incoming interprocedural information. The utility of cloning had not been fully explored when this research began. This dissertation presents a general strategy for cloning as well as a *goal-directed* strategy that applies cloning where it anticipates benefits. The effectiveness of this goal-directed strategy is demonstrated the program matrix300 from the SPEC benchmark suite.

Global optimization of a procedure based on interprocedural information is only limited in the feasibility of the analysis required. Experimental evidence has demonstrated that interprocedural analysis designed for parallelizing compilers can significantly reduce dependences assumed in the presence of procedure calls [TIF86] [LY88b] [LY88a] [HK91], resulting in significant execution-time performance improvements [MS91]. For scalar compilation, research on the effectiveness of interprocedural information has produced mixed results, ranging from moderate [Con83] to marginal improvement [RG89b]. This dissertation does not address the issue of making interprocedural analysis effective. The focus of this research is to determine when interprocedural transformations such as inline substitution and cloning can be more effective than global optimization based on interprocedural information.

Figure 1.1 presents a graphical description of the differences between the three techniques for interprocedural optimization. In the figure, both procedures A and B call procedure C, and C has three calls to procedures not shown. With interprocedural information, effects coming in from above include effects from A and its ancestors as well as effects from B and its ancestors. As a result, the amount of optimization possible in C is limited if A and B provide very different information to C. Cloning separates the incoming information from A and B by making two copies of C, with A and B calling separate copies. Then C can be more effectively optimized by tailoring the code to reflect its calling environment. Inline substitution also results in copies of C tailored to its callers, but it also moves the body of C into its callers. This allows movement of code across the call boundary, the most important use of inlining.

Interprocedural optimization is needed to increase the context of information available to the optimizer. However, the cost of the increased context may be very significant. The costs of inlining can be significant: it introduces compilation dependences among all component procedures in an inlined module; it can greatly increase program size [Sch77]; and, increased program size can lead to substantial increases in compile time [RG89a]. The costs of cloning may also be significant: it introduces compilation dependences among clones of a procedure, and it also increases program size. In the worst case, the program growth can be as great as with inlining. However, cloning permits sharing of specially optimized versions of a procedure among multiple

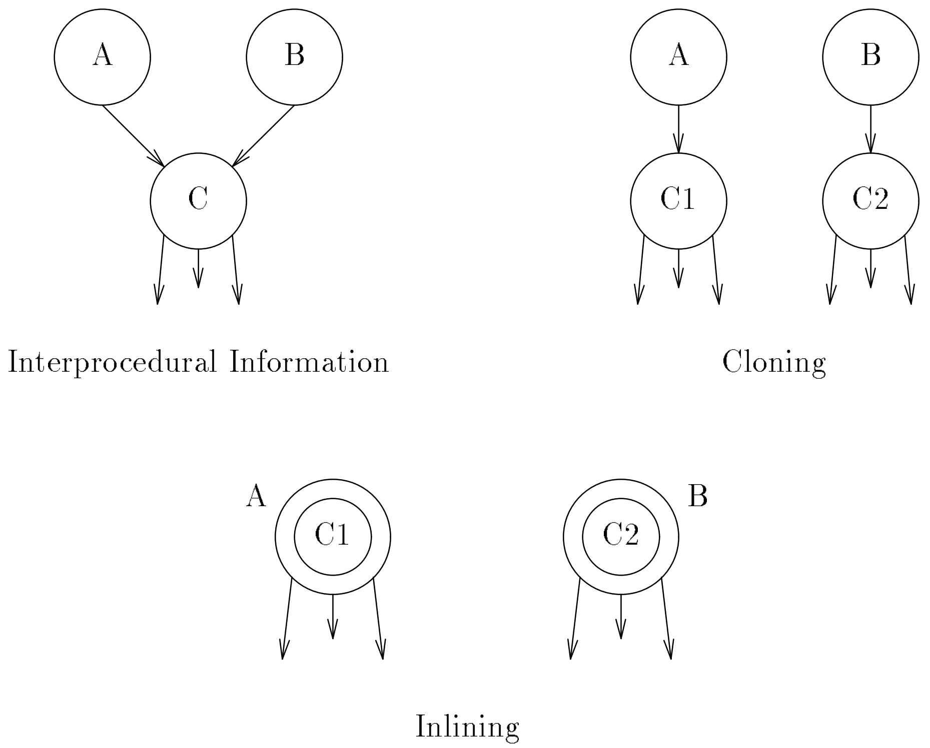


Figure 1.1 Graphical comparison of the effectiveness of interprocedural optimization techniques.

callers. The main cost associated with using interprocedural information in global optimization is that it introduces compilation dependences. The relationship between increased costs and increased context associated with interprocedural optimization techniques is depicted in Figure 1.2.

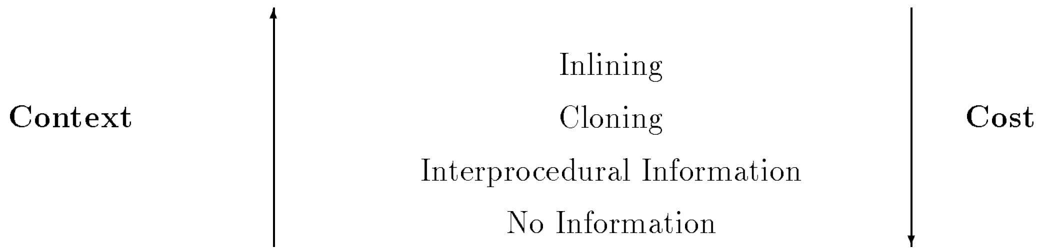


Figure 1.2 Tradeoffs between increased context and cost.

1.2 The ParaScope Programming Environment

ParaScope is designed to support all the requirements of a scientific programmer. Editing, program building, and debugging tools, as well as some auxiliary tools such as an infinite precision calculator and a text editor for documentation, all share a common database. To make interprocedural analysis efficient, the tasks usually left for the compiler are distributed throughout a number of tools in the environment. Information gathered in the editing and compiling tools is coordinated by the database. The flow of information among these tools, depicted in Figure 1.3, is described in this section.

Module editor. The module editor supports a combination of structure and text editing. The source representation of a FORTRAN module is an Abstract Syntax Tree (AST). Structure editing is performed directly on the AST. Even during text editing, each time a new line is completed it is parsed into the AST. Such a representation obviates the need for parsing in the compiler, and facilitates the gathering of local information needed for interprocedural analysis.

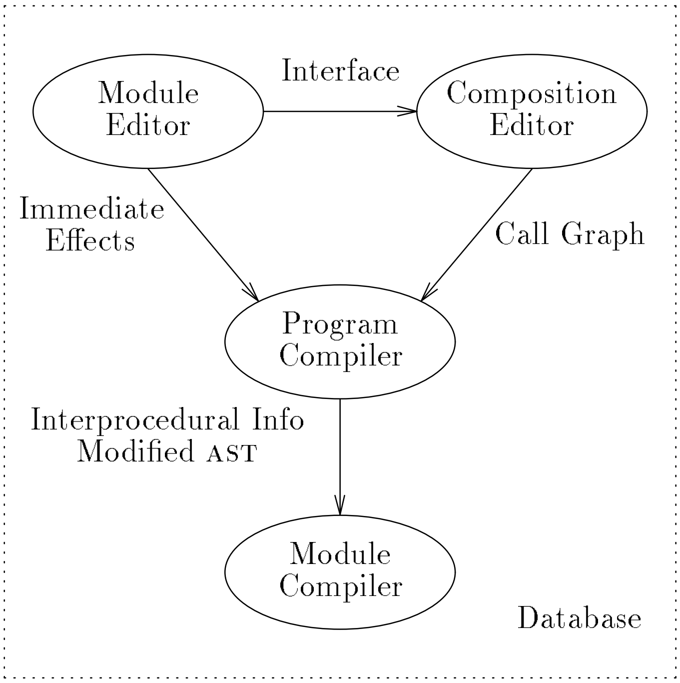


Figure 1.3 Flow of information in ParaScope.

Upon completion of an editing session, the module editor writes two sets of information to a file separate from the AST representation. The first of these is the module's interface. This consists of a list of the procedures declared in the module, with information about the number and type of the procedures' formal parameters. There is also a list of the call sites in each procedure, with information about the number and type of the actual parameters at the calls. The interface information is used by the composition editor to build the program representation and the call multigraph.

The second set of information describes the potential interprocedural effects of the procedure. For example, for the MOD problem, the editor provides global variables and parameters directly modified within each procedure. This is exactly the information that a compiler must collect from source code prior to calculating interprocedural information.

Composition editor. The composition editor is a structure editor for defining programs, or *compositions*, from module interfaces and other compositions. When adding a module to a composition, the composition editor reads in the module's interface rather than the module itself.

In addition to providing extensive facilities for building programs from existing components, it also tracks errors in interfaces such as inconsistencies in the number or type of parameters, and maintains a list of missing entry points. From a composition that is complete and without any serious errors, the program compiler can derive the call multigraph.

Program compiler. The program compiler provides all of the program-level support necessary for compilation and interprocedural optimization. It supplies the interprocedural information for a procedure to the module compiler. Since some interprocedural transformations are applied to source code, it also provides the module compiler with transformed source. The next section describes the program compiler in depth.

Module compiler. The module compiler performs the remaining duties typically considered to be the responsibilities of the compiler. It takes the transformed AST from the program compiler and translates to an intermediate representation known as ILOC (Intermediate Language for an Optimizing Compiler). Then, it optimizes the ILOC and generates object code. An important job of the AST-ILOC translator is to preserve on translation the interprocedural information provided by the program compiler. Then, the interprocedural information can be used in optimizations and register allocation.

1.3 The Program Compiler

The tool in ParaScope that provides program-level support for compilation is the *program compiler*. Many aspects of interprocedural optimization require knowledge about the whole program. As examples, gathering interprocedural information, performing source-level interprocedural transformations and minimizing recompilation requirements all need program-level support. The program compiler is designed to support all these tasks in an efficient way.

The program compiler can be broken down into five phases as shown in Figure 1.4. The first phase is building the call multigraph, the representation of the program used to calculate interprocedural information. The second phase calculates interprocedural information over the call multigraph. In the third phase, the program compiler plans which interprocedural transformations it will perform. Because of the inherent costs of interprocedural optimization, it is important that transformations only be performed if they are likely to be profitable. In the fourth phase, the program compiler performs all source-level interprocedural transformations. Additional optimizations may be performed by the module compiler, but even these are directed by hints from the program compiler. In the final phase, the program compiler invokes the module compiler for each module it has determined requires recompilation. It must locate not only modules that have been edited, but also those that have been invalidated by changes to their interprocedural environment. The rest of this section provides a detailed description of the phases of the program compiler.

1.3.1 Building the Call Multigraph

The *call multigraph* is a static structure describing the possible run-time interactions between the procedures in a program. The nodes of the call multigraph represent

call graph → information gathering → planning → performing transformations → compilation

Figure 1.4 Phases of the program compiler.

the procedures in the program. An edge ($p \rightarrow q$) exists if procedure p can invoke procedure q. Such an edge will be added for each call site in p invoking q; thus, the structure is a multigraph. Edges are annotated with information about the passing of actual parameters at the call site they represent. Since the call multigraph summarizes the relationships between procedures in a program, it serves as the framework for interprocedural data-flow analysis.

In ParaScope, we rely on properties of FORTRAN to simplify the problem of call multigraph construction. Specifically, procedure-valued parameters are allowed, but no other procedure-valued variables. Although the FORTRAN 77 standard does not allow recursion, it is often supported by FORTRAN compilers, and it is allowed in the FORTRAN 90 standard. Accordingly, our algorithm also supports recursion.

1.3.2 Gathering Interprocedural Information

The second phase of the program compiler gathers interprocedural information over the call multigraph. There are two distinct types of information collected in this phase: information used to enhance global optimization and information used to locate good targets for interprocedural transformations.

For each call site, we calculate MOD and REF information. These are the sets of variables possibly modified and referenced as a result of the call, respectively. For each procedure, we calculate ALIAS and CONSTANT information. The former represents pairs of variables that might refer to the same location in memory within the procedure. The latter indicates variables known to have the same constant value for all invocations of the procedure. The MOD, REF, ALIAS and CONSTANT information is used to enhance global optimization.

Other interprocedural information is used to locate good targets for interprocedural transformations. As an example, we use execution frequency estimates to order the application of interprocedural transformations so that the most important procedures are transformed first. Other special-purpose interprocedural information is also used to target good call sites for inlining and cloning.

1.3.3 Planning Optimizations

Once the interprocedural information has been gathered, the program compiler can examine the information, planning the interprocedural optimizations it will perform.

Some of the information calculated in the previous phase is used solely to locate good opportunities for optimization. Based on this information, we select procedures for optimization. We may also have a number of constraints that need to be satisfied. For example, if we are concerned about code growth after inline substitution, we may impose a limit on the increase in code size that will be tolerated. We may have other constraints designed to limit compilation dependences. Such constraints are needed to reduce the costs associated with interprocedural optimization.

Some transformation choices may make the resulting interprocedural information at a procedure more precise. For example, as a result of inlining, after replacing formal parameters with constant-valued actual parameters, it may be possible to evaluate some branches in the procedure body. This enables eliminating unreachable code, which may in turn reduce the list of modified and used variables in the procedure. When interprocedural information is used to make decisions about applying transformations, the decision process is improved by incrementally updating interprocedural information after each transformation.

1.3.4 Performing Interprocedural Transformations

After the planning phase, the program compiler performs the interprocedural transformations on the source code. This is the stage when inline substitution and procedure cloning are performed. Optimizing intermediate code based on interprocedural information is done later by the module compiler. The transformations done by the program compiler, and some done by the module compiler, are those that the planning phase has deemed profitable.

1.3.5 Compiling Modules and Building an Executable

The final phase of the program compiler is to direct the module compiler to compile the modules with the benefit of the interprocedural data-flow information. In this phase, the program compiler is responsible for determining which modules require recompilation, and which modules make up the executable. The program compiler builds an executable of the program for use in the execution monitor.

When interprocedural information is used to perform optimizations, every time the program is compiled the validity of these optimizations must be checked. This is true even if a procedure has not been edited since the previous compilation. Modules optimized based on interprocedural facts will need to be recompiled if any of those interprocedural facts change.

The program compiler must also determine recompilation requirements due to interprocedural transformations. If a cloned procedure has been changed, all cloned versions of that procedure must be recompiled. Editing of any procedure making up an inlined version forces that all the inlining be repeated and that the inlined version be recompiled. When a change makes a transformed procedure invalid, the decision to perform the transformation should be reevaluated. The program compiler must also track what source modules are used to build the program executable. This information is needed to understand what versions make up an executable, both for building the executable and for determining recompilation requirements in some subsequent compilation.

1.4 Parallelizing Program Compiler

The phases of a parallelizing program compiler are given in Figure 1.5. The difference between this diagram and the one in Figure 1.4 is the inclusion of dependence analysis before the planning phase. Dependence analysis is a key component in a parallelizing compiler, used to understand the pattern of memory accesses in a program [Kuc78]. Parallelization preserves the meaning of a program as long as the order of accesses to an individual memory location is retained. In dependence analysis, pairwise comparisons of memory accesses are performed to determine if they can reference the same location.

In a compiler performing dependence analysis, the analysis is typically in the back-end of the compiler immediately preceding code generation. We have separated the analysis from code generation to be able to use the results from dependence analysis within a single procedure to make decisions about optimizations across procedure boundaries.

The approach for parallelization differs from the previous program compiler framework because we have added an additional pass over the program source. This is because the information needed for parallelizing transformations must be precise. While

call graph	→	information gathering	→	dependence analysis	→	planning	→	performing transformations	→	compilation

Figure 1.5 Phases of the parallelizing program compiler.

we could include the dependence analysis at the end of the editing session as we do with other interprocedural information, the time required to perform precise analysis may be longer than the programmer is willing to accept. Also, the precision of the dependence analysis benefits from having solutions to the interprocedural problems available. During recompilation analysis, dependence testing is avoided in most cases when the procedure does not require recompilation.

1.5 Related Work

1.5.1 Early Work

The earliest known work on interprocedural optimization is due to Ershov [Yer66]. He suggested procedure calls (and loops) as good targets on which to concentrate optimization. The ALPHA translator optimized parameter passing using information about accesses to the parameters within the called procedure.

The Allen-Cocke optimization catalog defined four different ways of implementing procedure call linkages [AC72]. These are *open*, *closed*, *semi-open* and *semi-closed* linkages. Open linkage is another name for inline substitution. Closed linkage is the usual linkage style for separate compilation. With semi-open linkage, a procedure definition is compiled with its caller. The called procedure and the caller are optimized together, and the procedure calls are converted to branches to the procedure body. This type of linkage gives some of the benefits of open linkage without any increase in code size. Semi-closed linkage requires the called procedure to be compiled before the calling procedure. In this way, the compiler can optimize the passing of parameters at the call site, as Ershov did in the ALPHA system. Semi-closed linkage can be realized in ParaScope by using interprocedural information during global optimization. However, the ParaScope compilation system avoids ordering dependences and also uses information the caller propagates to the callee.

Much of the early work on interprocedural optimization focused on interprocedural analysis. This began with the description of an implementation in 1971 [Spi71]. Other work attempted to accomodate unusual language features [Wei80], calculate more precise information [Ros79] [Mye81], or produce information more efficiently [Ban79]. The design in ParaScope draws heavily from Banning, who recognized that interprocedural side-effect analysis can be separated from alias analysis with the results of side-effect analysis updated to include the effects of aliasing.

Another early work on interprocedural optimization, the Experimental Compiling System at IBM, describes a compilation system centered around inline substitution [Har77b] [ACF+80]. Instructions in the intermediate language represent function calls, not only to source procedures but also to primitive operations. After translation to the intermediate language, inline substitution and global optimizations are repeatedly performed until the program is represented completely in terms of primitives. The goals of the system were to improve programming style by making procedure calls inexpensive; to simplify optimization since it can occur at any time after translation to intermediate code; and to provide a mechanism for supporting multiple source and target languages.

1.5.2 Previous Work in ParaScope

The ParaScope strategy for interprocedural optimization has been evolving for a number of years. Supporting interprocedural analysis efficiently was a goal from the beginning [HK83]. The first step was the development of efficient algorithms for interprocedural side-effect and alias analyses, separated into a local phase and a propagation phase [Coo83] [CK84] [Coo85]. This two-phase analysis required cooperation of other programming environment tools and the concept of a program compiler [Tor85] [CKT85] [CKT86a]. Recompilation analysis was incorporated to minimize the need for recompilation as a result of changes to interprocedural information [CKT86b] [BCKT90]. Also, new algorithms were developed for interprocedural constant propagation [CCKT86], nearly linear side-effect and aliasing analysis [CK88b] [CK89] and array side-effect analysis [CK88a] [HK91].

This earlier work provided a good framework for the research in this dissertation. One aspect that was lacking in the previous design was adequate support for interprocedural transformations such as inlining and cloning. Based on the previous program compiler framework, this dissertation has developed a strategy for interprocedural optimization that includes interprocedural transformations.

1.6 Overview of Dissertation

In this first chapter, we have laid out the requirements for a programming environment that efficiently employs interprocedural optimization. Because a great deal of the work involved in interprocedural optimization takes place in the program compiler, we have focused this chapter on the program compiler's design.

In Chapter 2, we present algorithms for building the call multigraph. The call multigraph is important because it forms the framework for calculating all interprocedural information. Chapter 3 describes a study of inline substitution. Because inline substitution provides the best possible interprocedural information, and therefore the best possible optimization opportunities across procedure calls excluding secondary effects, a careful look at inline substitution provided insight into the usefulness of alternative interprocedural optimizations. Chapter 4 presents a *goal-directed* strategy for inlining and cloning designed to enable specific high-payoff optimizations. Chapter 5 thoroughly explores procedure cloning, presenting a general algorithm that exploits the potential of cloning while avoiding extensive compilation costs. Chapter 6 completes the treatment of scalar interprocedural optimization, describing the interprocedural information calculated in ParaScope and providing guidelines for inlining and cloning. Most importantly, Chapter 6 describes a general system to combine inlining, cloning and optimization with interprocedural information, while managing the recompilation requirements for each of these. Chapter 7 considers interprocedural optimization for parallelization, describing how interprocedural parallelizing transformations can be supported efficiently. Chapter 8 concludes the dissertation, describing its contributions and implications for the future.

Chapter 2

Call Multigraph Construction

Any technique performing analysis or optimization across procedure boundaries requires some underlying representation of the program structure. Most often the structure used is the *call multigraph*. The call multigraph is a static structure describing the dynamic invocation relationships between procedures in a program. A node in the call multigraph represents a procedure, and an edge $(p \rightarrow q)$ exists if procedure p can invoke procedure q.

An algorithm for call multigraph construction is somewhat dependent upon the features of the language the compiler supports. In this chapter, we make the following assumptions about the language:

- Recursion is allowed.
- Procedure-valued parameters are allowed.
- Assignments to procedure variables are *not* allowed.

Of the points above, the only restriction is the third point. The algorithm does not handle assignments to procedure variables; instead, we require that procedure variables only receive their values from the parameter passing mechanism.

This chapter presents an algorithm for call multigraph construction designed for use in ParaScope. As a result, the language features described above exactly match features of FORTRAN and its extensions. However, this algorithm can be used in compiling a variety of other programming languages. With minor extensions to the algorithm, we can relax the requirement that procedure variables only receive their values from parameter passing. It appears that such extensions would enable call multigraph construction for languages such as Scheme and ML [Shi88].

When procedure-valued variables do not exist in a language, constructing the call multigraph only requires a single pass over the procedures and call sites in the program, adding edges $(p \rightarrow q)$ whenever a call to q appears in procedure p. Building a call multigraph when procedure-valued formal parameters exist is difficult for two reasons. First of all, although propagation of values for the procedure formals is fairly straightforward, once new bindings for invoked procedure formals are located, edges are added to the graph to reflect the new calls that the invocation can represent. Thus, values are propagated on a changing graph. Secondly, as edges are added, we propagate information along the new edges, possibly returning to nodes that have already been visited. This revisiting of nodes can be a source of inefficiency.

In this chapter, we present an efficient algorithm for call multigraph construction. The algorithm is more efficient than previous techniques because it delays propagation of new information until it can be used. It is based on the binding multigraph β, a structure used in the formulation of effectively linear interprocedural analysis algorithms by Cooper and Kennedy [CK88b] [CK89]. β is a specialized call multigraph, containing a node for each formal parameter in the program, and an edge between nodes f_p^i and f_q^j if the i^{th} formal parameter of p is passed as the j^{th} actual parameter of q at some call site in p invoking q. Thus, β explicitly represents the bindings of formal parameters in the program. In this chapter, we are only concerned with nodes in β representing procedure-valued formal parameters and the edges between them.

The algorithm presented here produces the same result as a method due to Burke, but has a better asymptotic time bound. Both approaches are more efficient, though less precise, than our previous algorithm [CCHK90]. The next section provides background information and defines what the algorithm computes. Section 3 presents the algorithm, and Section 4 gives an example that exercises all of the steps of the algorithm. In Section 5, we prove correctness of the algorithm, and Section 6 proves its time complexity. Section 7 discusses previous work, including a comparison with two existing algorithms [Bur87] [CCHK90]. Section 8 summarizes the chapter.

2.1 Background and Definitions

Since processing *statically bound calls*, (i.e., calls invoking procedure constants) is straightforward, the challenge is to add edges representing *dynamically bound calls* through procedure variables. This requires that we determine the set of possible values for a procedure variable. Since we assume procedure variables only receive their values through parameter passing, determining procedures invoked at dynamically bound calls is equivalent to locating all the possible values bound to the invoked procedure variable via parameter passing.

The algorithm simulates parameter passing of procedure constants and procedure variables during execution. For a given procedure formal f, the algorithm calculates the set $Boundto(f)$, the procedure constants that can be bound to f. Thus, for some call site invoking formal f, $Boundto(f)$ is the set of procedures that may be invoked at the call. The final *Boundto* values are described by the following simultaneous equations:

$$\begin{aligned} Boundto(f_p) = & \bigcup_{c\ invokes\ p} StaticBindings(f_p, c)\ \cup \\ & \bigcup_{c\ invokes\ f_q\ \wedge\ p\ \in\ Boundto(f_q)} DynamicBindings(f_q, f_p, c) \end{aligned}$$

For some formal f_p of procedure p, $Boundto(f_p)$ receives its values from parameters passed either at statically bound calls to p or at dynamically bound calls where p is one of the possible bindings for the invoked parameter. The equations for *StaticBindings* and *DynamicBindings* are as follows:

$$\begin{aligned} & StaticBindings(f_p, c) = \\ & \qquad ConstantPassed(f_p, c) \cup Boundto(FormalPassed(f_p, c)) \end{aligned}$$

$$\begin{aligned} & DynamicBindings(f_q, f_p, c) = \\ & \qquad ConstantPassed(f_p, c)\ \cup\ DynamicFormalBindings(f_q, f_p, c) \end{aligned}$$

Based on the above equations, a procedure formal f_p receives its bindings either directly from the procedure constants passed to f_p, or indirectly from bindings of the procedure formals passed to f_p. This is true for both statically and dynamically bound calls.

$ConstantPassed(f_p, c)$ and $FormalPassed(f_p, c)$ contain the procedure constants and procedure formals of the caller passed as actuals to f_p at c, respectively. We derive $ConstantPassed(f_p, c)$ and $FormalPassed(f_p, c)$ by examining the call c, locating the actual parameter passed to f_p. Either a procedure constant or a procedure formal is passed at the call, so the sum of the number of elements in $ConstantPassed(f_p, c)$ and $FormalPassed(f_p, c)$ is exactly 1 (assuming c invokes p).

The set $DynamicFormalBindings(f_q, f_p, c)$ gives the bindings for f_p at some dynamically bound call c invoking f_q where a formal procedure parameter is passed to f_p. The equations for *DynamicFormalBindings* are as follows:

$$DynamicFormalBindings(f_q, f_p, c) =$$

$$\begin{cases} \{p\} & \text{if } FormalPassed(f_p, c) = f_q \\ Boundto(FormalPassed(f_p, c)) & \text{otherwise} \end{cases}$$

The *DynamicFormalBindings* equations are only needed for the special case when the procedure parameter invoked at a call also appears as one of the actual parameters at the call. Otherwise, the definitions for *StaticBindings* and *DynamicBindings* are identical. To understand the difference, consider a procedure parameter f_q with multiple values in its *Boundto* set, one of which is p. If f_q is invoked at call c and also appears as the actual parameter passed to f_p, the formulation of *StaticBindings* would propagate all of the values in $Boundto(f_q)$ to f_p. However, we know that p can only be invoked at this call when f_q has binding p, so we are introducing imprecision by propagating bindings other than p to f_p.[2]

The simultaneous equations in this section could be adapted in a straightforward way into an iterative algorithm for call multigraph construction. However, the resulting algorithm could potentially require a pass over all the procedure formals each time a new binding was located for an invoked procedure formal (i.e., a new edge in the call multigraph). The algorithm presented in this chapter is efficient because it propagates new bindings individually as they are located.

2.2 Algorithm

The algorithm for call multigraph construction is given in Figures 2.1 and 2.2. The procedure *InitializeNode* in Figure 2.1 is called on a newly reachable procedure to initialize the *Boundto* sets for its procedure parameters, and to locate elements for *Worklist*. It recursively calls itself to initialize procedures that become reachable through static edges from the current node.

Procedures are initialized as they become reachable rather than initializing all of the procedures in the program at the beginning. This is to ensure that only reachable nodes are placed in the call multigraph so that procedure parameter bindings result only from reachable calls. This is important because by adding bindings from unreachable nodes, we are potentially adding extra edges to the final graph.

The main algorithm *Build*, shown in Figure 2.2 relies on a list of pairs *Worklist*. An element $\langle a, f_p \rangle$ is in *Worklist* if a can be bound to f_p through parameter passing along some chain of calls and a is possibly not yet in $Boundto(f)$. Initially, *Worklist* contains those pairs representing bindings from statically bound calls; that is, a would be the actual parameter passed to f_p at some call to p. The job of *Build* is to propagate bindings for procedure formals at statically bound calls, and handle both procedure constants and procedure formals at dynamically bound calls. Dynamically bound

[2]Burke's algorithm does not handle the special case that *DynamicFormalBindings* is designed to capture but could with a slight modification.

```
/* Add a reachable node and all of its static edges */
procedure InitializeNode(p)
    add p to Nodes
    foreach formal procedure parameter f of p
        Boundto(f) ← ∅
    foreach call site (p → q) in p
        if q is a procedure constant then
            if q ∉ Nodes then call InitializeNode(q)
            add edge (p, q) to call multigraph
            foreach procedure constant a and formal g of q such that a is passed to g
                add ⟨a, g⟩ to Worklist
        endif
    endfor
end /* InitializeNode */
```

Figure 2.1 Initializing information at a newly reachable node.

---

calls are more difficult since we must wait for bindings of the invoked formal to know what procedure variable is being passed a value.

When a pair $\langle a, f_p \rangle$ is removed from *Worklist*, we must examine all call sites in f_p's procedure, looking for uses of f_p. There are three distinct situations that may exist at each call site:

1. **The call site is statically bound.**
   Then, f_p may appear as an actual at the call; the binding a is propagated to the corresponding parameter of the called procedure (step 1a of the algorithm).
2. **The call site is dynamically bound, but invokes some formal other than f_p.**
   Again, f_p may appear as an actual at the call; the binding a is propagated to the corresponding parameter of all procedures currently bound to the invoked formal (step 2a).
3. **The call site invokes f_p.**
   First, we can propagate procedure constants at the call to the corresponding parameters of a (step 3a). Also, we can propagate known bindings for the other procedure variables at the call to the corresponding parameters of a (step 3c). In the special case that f_p appears as an actual at the call, we propagate a to the corresponding parameter of a (step 3b).

Steps 2 and 3 of the algorithm complement each other. If we receive the binding for a formal passed at a dynamically bound call before we know all the bindings of the

```
      /* Build call multigraph starting at root procedure */
      program Build
      Worklist ← ∅
      Nodes ← ∅
      call InitializeNode(main)

      while Worklist ≠ ∅
          select and delete an element ⟨a, f⟩ from Worklist
          let p denote the procedure to which f is a formal parameter

          if a ∉ Boundto(f) then
             add a to Boundto(f)
             foreach call site (p → q) in p
(1)             if q is a procedure constant then
                   /* Add to chains in β ending at f through this call to q */
                   foreach formal g of q such that f is the actual passed to g at the call
(1a)                  add ⟨a, g⟩ to Worklist

(2)             else if q ≠ f then /* q is some procedure parameter other than f */
                   /* Add to chains in β ending at f through all procedures bound to q */
                   foreach b ∈ Boundto(q)
                      foreach formal h of b such that f is the actual passed to h at the call
(2a)                     add ⟨a, h⟩ to Worklist
                endif

(3)             else /* f is invoked at the call */
                   if a ∉ Nodes then call InitializeNode(a)
                   add edge (p, a) to call multigraph
                   /* Locate beginnings of chains in β */
                   foreach procedure constant b and formal g of a such that b is passed to g
(3a)                  add ⟨b, g⟩ to Worklist
                   /* Add to chains in β through this call to a */
                   foreach formal h of p and formal g of a such that h is passed to g at the call
                      if h = f then /* special case from DynamicFormalBindings */
(3b)                     add ⟨a, g⟩ to Worklist
                      else
                         foreach procedure constant c ∈ Boundto(h)
(3c)                        add ⟨c, g⟩ to Worklist
                endif
             endfor
          endif
      endwhile
      end /* Build */
```

Figure 2.2 Main algorithm for computing call multigraph.

invoked formal, when we eventually receive the binding for the invoked formal, step 3c propagates the binding of the parameter at the call. If we receive the binding for an invoked formal before bindings for the formals passed at the call, when we eventually receive bindings for the formals passed at the call, step 2a propagates their bindings.

2.3 An Example

This section works through the steps of the algorithm to make them clearer. Consider the following program:

program $main$
    call $a(b, c)$
end

procedure $a(f_1, f_2)$
    call $b(a, f_2)$
    call $f_1(f_1, f_2)$
end

procedure $b(f_3, f_4)$
    call $f_3(f_3, d)$
    call f_4
end

For this example, the initialization step would locate the passing of procedure constants in $main$ at the call to a, and further, in a at the call to b. Thus, the initial elements of *Worklist* are $\{\langle b, f_1\rangle, \langle c, f_2\rangle, \langle a, f_3\rangle\}$, and the initial edges are $(main, a)$ and (a, b).

Now the elements on *Worklist* are processed. In this example, as elements are added to *Worklist*, we annotate the element with the step in the algorithm that caused it to be added to *Worklist*. The steps are as shown in Figure 2.3.

The remaining elements on *Worklist* are $\{\langle b, f_3\rangle, \langle d, f_4\rangle, \langle a, f_1\rangle, \langle c, f_2\rangle, \langle d, f_2\rangle\}$, all of which have already been processed by the algorithm. So, the algorithm terminates. The final values for *Boundto* and the final call multigraph are shown in Figure 2.3.

2.4 Proof of Correctness

Lemma 2.1 The algorithm *Build* terminates after no more than EP iterations of the *while* loop.

Proof. Each iteration of the *while* loop selects and removes a pair from *Worklist* and processes this pair. Each time an element is added to *Worklist* represents the propagation of a new binding of a formal procedure parameter along a distinct edge. (Note that we refer to an edge rather than a call here. For a dynamically bound

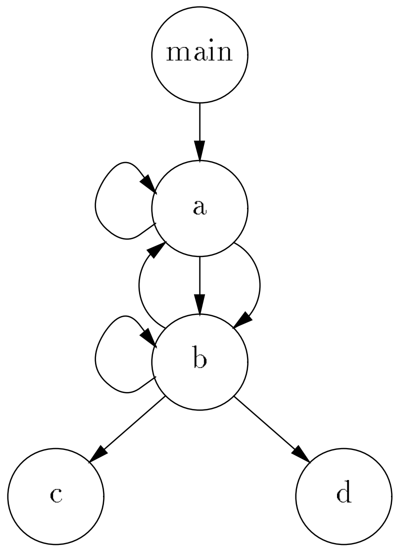
Process $\langle b, f_1\rangle$: $Boundto(f_1) \leftarrow \{b\}$ add edge (a, b) to call multigraph add $\langle b, f_3\rangle$ to $Worklist$ (3b) *Process* $\langle c, f_2\rangle$: $Boundto(f_2) \leftarrow \{c\}$ add $\langle c, f4\rangle$ to $Worklist$ (really added twice) (1a)(2a) *Process* $\langle a, f_3\rangle$: $Boundto(f_3) \leftarrow \{a\}$ add edge (b, a) to call multigraph add $\langle d, f_2\rangle$ to $Worklist$ (3a) add $\langle a, f_1\rangle$ to $Worklist$ (3b) *Process* $\langle b, f_3\rangle$: $Boundto(f_3) \leftarrow \{a, b\}$ add edge (b, b) to call multigraph add $\langle d, f_4\rangle$ to $Worklist$ (3a) add $\langle b, f_3\rangle$ to $Worklist$ (3b)	*Process* $\langle c, f_4\rangle$: $Boundto(f_4) \leftarrow \{c\}$ add edge (b, c) to call multigraph *Process* $\langle d, f_2\rangle$: $Boundto(f_2) \leftarrow \{c, d\}$ add $\langle d, f_4\rangle$ to $Worklist$ (added twice) (1a)(2a) *Process* $\langle a, f_1\rangle$: $Boundto(f_1) \leftarrow \{b, a\}$ add edge (a, a) to call multigraph add $\langle a, f_1\rangle$ to $Worklist$ (3b) add $\langle c, f_2\rangle$ to $Worklist$ (3c) add $\langle d, f_2\rangle$ to $Worklist$ (3c) *Process* $\langle d, f_4\rangle$: $Boundto(f_4) \leftarrow \{c, d\}$ add edge (b, d) to call multigraph

Figure 2.3 Steps of algorithm for example program.

call, we actually can introduce separate sets of bindings for each possible value of the invoked formal. However, each unique value of the invoked formal is represented by a distinct edge in the final call multigraph).

For each edge, the maximum number of pairs added to *Worklist* is equal to the number of unique bindings of the actual procedure parameters appearing at this call. Let P be the total number of unique procedure constants passed as parameters in the program. If we assume the number of parameters of a procedure is bounded by some small constant, then the number of elements added to *Worklist* as a result of a single call site is $O(P)$. Thus, the number of pairs appearing on *Worklist*, and also the number of iterations of the *while* loop, is bounded by $O(EP)$.

Lemma 2.2 The algorithm *Build* builds the portion of the static call multigraph reachable from the root node through direct calls before entering the *while* loop.



$Boundto(f_1) = \{b, a\}$
$Boundto(f_2) = \{c, d\}$
$Boundto(f_3) = \{a, b\}$
$Boundto(f_4) = \{c, d\}$

Figure 2.4 Resulting call multigraph and *Boundto* sets from example.

When visiting a node, the procedure *InitializeNode* adds all outgoing static edges to the call multigraph, recursively initializes newly reachable procedures, and adds pairs to *Worklist* from bindings of procedure constants to procedure parameters at static call sites. Since it recursively visits nodes reachable through static edges, and since we are starting with the root node, the partially constructed graph upon entry to the *while* loop contains the static call multigraph reachable from the root node through direct calls.

Lemma 2.3 The algorithm adds a pair $\langle a, f_{n_m} \rangle$ to *Worklist* if and only if through some chain of calls $n_0 \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_m} n_m$, a is bound to f_{n_m} at c_m.

Proof.

$\Rightarrow$: By induction on the *while* loop iteration count.

Basis. As established in Lemma 2, on entry to the *while* loop *Worklist* contains the set of pairs $\langle a, f_p \rangle$ such that procedure constant a is passed directly to f_p at some statically bound call site invoking p. All such bindings correspond to the *ConstantPassed* term in the *StaticBindings* portion of the *Boundto* equation from Section 2.

Induction. Assume the lemma is true after the first n iterations of the loop. The pair $\langle a, f_p \rangle$ selected on iteration $n+1$ must then be an element in the final set $Boundto(f_p)$. If the binding has already been propagated, we ignore this pair and continue to the first iteration that contributes a new binding.

We visit each call site in p such that f_p is either invoked or is one of the actual parameters. In the call sites where f_p appears as an actual parameter, we must treat separately the statically bound call sites from those that invoke procedure parameters.

For statically bound calls, we add pairs $\langle a, f_q \rangle$ to *Worklist* for any formals of the called procedure q where f_p appears as the corresponding actual parameter at the call c. Here, $FormalPassed(f_q, c) = f_p$, and $a \in Boundto(f_p)$. So, this corresponds to the second term in the *StaticBindings* portion of the equation for $Boundto(f_q)$. For call sites invoking procedure parameters ${f_p}' \neq f_p$, we add pairs $\langle a, f_q \rangle$ if f_p is an actual at the call, $q \in Boundto({f_p}')$, and f_q is the corresponding formal of q. This matches the second case in *DynamicFormalBindings*, which makes up part of the *DynamicBindings* portion of the equation $Boundto(f_q)$.

At call sites where f_p is invoked, we have located a new binding for the invoked parameter, and so we add an edge to the call multigraph. For any procedure constants passed as actual parameters at the call, we add the binding for parameters of a to *Worklist*, corresponding to the *ConstantPassed* term in the *DynamicBindings* portion of the *Boundto* equation for the formals of a. If f_p appears as an actual parameter, we add $\langle a, f_a \rangle$ to *Worklist*, the special case from *DynamicFormalBindings*. Also, for any other formal procedure parameter ${f_p}'$ passed as an actual at the call, we add to *Worklist* pairs $\langle b, f_a \rangle$, where $b \in Boundto({f_p}')$ and f_a is the corresponding formal at the call. These bindings are included in the general case of *DynamicFormalBindings*.

Note that since a may become a reachable procedure as a result of this call, we invoke $InitializeNode$, so we may also add bindings from any static calls through a. That these statically bound calls are correctly processed follows from Lemma 2.

$\Leftarrow$: By induction on the length of the call chain.

Basis. The only length-0 call chain is the root node. It has no parameters, so no bindings.

Proof. Assume that for every call chain $n_0 \xrightarrow{c_1} \ldots \xrightarrow{c_m} n_m$ of length m, pairs $\langle a, f_{n_m} \rangle$ are inserted into Worklist representing bindings for the formal parameters of n_m. Let $n_0 \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_m} n_m \xrightarrow{c_{m+1}} n_{m+1}$ be some call chain in the program. By the induction hypothesis, all bindings for the procedure formals of n_m that result from this call chain have been inserted into *Worklist*.

Consider the effect of propagating the pairs in *Worklist* representing bindings at c_m at a particular call c_{m+1} in n_m. Suppose c_{m+1} is a statically bound call. Then values for procedure parameters of n_{m+1} come from the procedure constants and procedure parameters passed as actuals at the call. The procedure constants are added in $InitializeNode$. The bindings for the procedure formals of n_{m+1} are added whenever the bindings for the procedure formals of n_m are processed.

If c_{m+1} is a dynamically bound call invoking some formal f_{n_m}, then one of the bindings for f_{n_m} must be n_{m+1}. By the induction hypothesis, the pair $\langle n_{m+1}, f_{n_m}\rangle$ must have been added to *Worklist* when actuals passed at c_m were propagated to n_m. Bindings for all other formals of n_m through this call chain have also already been added to *Worklist*. The bindings of the formals of n_{m+1} arise from the actuals at c_{m+1}. For actuals that are procedure constants, the bindings are added during processing of the pair $\langle n_{m+1}, f_{n_m}\rangle$. Bindings through procedure formals are added during processing of the bindings for formals of n_m passed as actuals at c_{m+1}.

Once we process the pairs for all of the procedure parameters of n_m, we will have propagated these values to n_{m+1} along the call site c_{m+1}, and this is guaranteed to happen since we have proven termination of the algorithm. □

Theorem 1 *Build* correctly builds the *Boundto* sets as defined in Section 2, and as a result, correctly computes the call multigraph for any input program.

Proof. By Lemma 1, the algorithm terminates, and by Lemma 2, *InitializeNode* correctly builds the static portion of the call multigraph. Lemma 3 establishes that each chain of formal procedure parameters is correctly located and that such chains correspond to chains of calls in the call multigraph. Since the chains in β are located and bindings for nodes in these chains are correctly computed, bindings for invoked procedure parameters must all be correct. With correct bindings at the invoked procedure parameters, edges representing dynamic calls can all be added to the call multigraph. Since we have shown that both statically and dynamically bound calls are correctly added, we have the desired result. □

2.5 Time Complexity

The procedure *InitializeNode*, which initializes *Boundto* sets and *Worklist*, examines a procedure and its call sites once for each formal procedure parameter. It is invoked once for each node in the final call multigraph. Let c_p be the maximum number of procedure parameters of any procedure. Then the initialization step requires $O(c_p(N+E))$, where N and E are the number of nodes and edges in the final call multigraph, respectively. In the following discussion, we assume c_p is bounded by a small constant and eliminate it from consideration, consistent with assumptions made in other work [CK88b] [CK89].

The procedure *Build* processes an element of *Worklist* on each iteration of the loop. Lemma 1 asserted that the maximum number of pairs appearing on *Worklist* is bounded by $O(EP)$. Consider the number of unique pairs appearing on *Worklist*. As part of the processing of a pair $\langle a, f \rangle$, the procedure constant a is added to the *Boundto* set of the procedure parameter f. If this pair is ever selected from *Worklist* again, it will not be processed since a already appears in $Boundto(f)$. Thus, the number of iterations of the *while* loop that proceed past the initial test is no greater than the number of possible pairs in *Worklist*. These are $\langle$ procedure constant, procedure parameter $\rangle$ pairs, so the maximum number is NP. Since the maximum number of unique *Worklist* elements for a given procedure is bounded by P, in the *for* loop we visit a distinct call site $O(P)$ times. As a result, examining call sites requires $O(EP)$ time.

Consider the inner loops that propagate newly determined bindings of procedure parameters. Each operation inside these loops either adds an element to *Worklist*, or adds an edge to the final call multigraph. Since we have already established that the total number of elements added to *Worklist* is $O(EP)$, the step where an element is added to *Worklist* can only be executed a total of (EP) times. The step where edges are added is only executed once for each edge in the final call multigraph that did not occur in the static call multigraph. Thus, the total number of times this step occurs is $O(E)$ times.

The entire algorithm is $O(N + EP)$. Although $P \leq N$, in our experience with FORTRAN this number is almost always ≤ 1, so we expect the algorithm to have linear behavior in practice.

2.6 Related Work

2.6.1 Comparison with Burke's Algorithm

Figure 2.5 presents a summary of the algorithm proposed by Burke [Bur87]. The set $ActualBound(f)$ represents all procedure constants that can be bound to a procedure parameter f that is invoked. For procedure parameters that are not invoked, it contains only those values directly bound to the procedure parameter (and not those bound through a chain of calls). The set *Bound*(f) contains all of the procedure parameters in the program that may be bound to procedure formal f through some chain of call sites. The *Bound* relation can be thought of as representing the transitive closure of the edges in β.

The algorithm initializes the *ActualBound* sets from direct bindings appearing at call sites, and adds edges to the call multigraph if bindings for any invoked procedure parameters are set. Then the *Bound* set is built using interval analysis, based on the edges in the current call multigraph. Subsequently, new bindings for calls through procedure parameters are located, and the appropriate edges are added to the call multigraph.

The problem with building the call multigraph in this way is that adding edges may add new values to the *Bound* sets. This is the case if call sites invoking procedure formals also pass procedure formals as actuals at the call site. As a result, if any changes are made to the call multigraph, the cycle of interval analysis and updating the call multigraph must be repeated.

```
build static call multigraph
initialize ActualBound(f) to empty for all procedure formals f in the program
foreach procedure constant passed as an actual at a direct call
   add procedure constant to ActualBound(f) for the corresponding formal f

repeat until no changes in ActualBound(f) for any procedure formal f
   foreach procedure formal f that is invoked
      foreach new q ∈ ActualBound(f)
         add edge (p, q) to call multigraph
         if callsite has procedure constants passed as actuals then
            update ActualBound sets for procedure formals of q

   calculate Bound(f) for all procedure formals f using interval analysis
      on the current call multigraph

   foreach procedure formal f that is invoked
      foreach procedure formal g ∈ Bound(f)
         ActualBound(f) ← ActualBound(f) ∪ ActualBound(g)
end /* repeat */
```

Figure 2.5 Summary of Burke's algorithm.

Time complexity. Building the static call multigraph and initializing *ActualBound* sets requires examination of each procedure and call site in the program, and their parameters. Again assuming that the maximum number of procedure parameters for any procedure in the program is bounded by a constant, this step requires $O(N+E)$ time. Updates to the call multigraph based on changes to *ActualBound* can be made as such changes are located. Thus, the time complexity for this step is $O(E)$, since these changes result in edge additions to the multigraph.

To calculate the *Bound* sets, the current call multigraph is broken down into intervals. The time required for interval analysis is $O(dE)$ bit vector steps where d is the loop nesting depth of the call multigraph for a reducible call multigraph, or the loop connectedness for an irreducible call multigraph. Here the length of the bit vectors is the maximum size of *ActualBound* sets, which is equal to the number of procedure parameters in the program. Since we have assumed the number of procedure parameters for a single procedure is bounded by a small constant, the resulting length of the bit vectors is bounded by $O(N)$.

Recognizing changes to *ActualBound* requires visiting each procedure parameter that is invoked, and each parameter in its *Bound* set. Since only *ActualBound* sets for invoked procedure formals are updated, the outer loop executes $O(E_p)$ time, where E_p is the number of call sites invoking procedure formals ($E_p < E$). Since the size of the *Bound* set is bounded by the number of procedure formals in the program, the number of times the inner loop executes is bounded by $O(N)$. Assuming bit vector operations for the union of *ActualBound* sets, this requires $O(NE_p)$ bit vector steps with the length of the bit vector bounded by P. (Recall that P is the number of unique procedure constants passed as parameters in the program.)

So one cycle of interval analysis and updating the *ActualBound* sets requires $O(NE_p + dE)$ steps. As mentioned before, this cycle is repeated when procedure parameters are passed as actuals at call sites invoking procedure parameters. Thus, the number of times the cycle is repeated is one more than the length of the longest chain of procedure parameters passed as actuals through indirect calls. This is no greater than $O(PE_p)$, where $PE_p < E$, but is likely to be much less. Thus, the final algorithm requires at most $O(PE_p(NE_p + dE))$ time.

Comparing the two algorithms, the asymptotic complexity of our algorithm is $O(N+EP)$ unit steps while Burke's is $O(PNE_p^2 + dPE_pE)$ bit vector steps. Clearly, our algorithm has a better asymptotic time complexity. However, we expect the algorithms to perform comparably on typical FORTRAN programs. Let us consider the complexity of Burke's method for most FORTRAN programs as we did with our algorithm. Typically, chains of indirect calls do not exist, so the cycle of interval analysis and call multigraph updates will only be executed once. Also, the value for d is small, since call multigraphs are unlikely to have deeply nested loops. Thus, building the call multigraph for a typical FORTRAN program using Burke's method requires $O(NE_p + E)$ bit vector steps. At least on FORTRAN programs, we expect the two methods to perform comparably.

2.6.2 Comparison to Precise Algorithm

In [CCHK90], we present a precise algorithm for call multigraph construction, based on work by Ryder [Ryd79]. Ryder's algorithm propagates simultaneous bindings of values to procedure parameters. The algorithm computes a precise call multigraph and requires only a single pass over the program, but it cannot be used for languages permitting recursion.

To be able to use Ryder's algorithm in ParaScope, we extended it to an iterative algorithm, thus allowing recursion. While precise, the algorithm has a worst-case time bound of the maximum number of simultaneous bindings of values to procedure parameters over all calls. If c_p is the maximum number of procedure formals for any procedure, the time required by the algorithm is bounded by $O(N+EP^{c_p})$. Even with the assumption that c_p is a constant (used in the previous analyses), the algorithm is still polynomial in the size of the graph.

Although in [CCHK90] we pointed out that a loss of precision could result from using Burke's algorithm, the conditions for this loss of precision rarely occur in FORTRAN. The potential for loss of precision using the algorithm in this chapter is demonstrated with the following example:

program main
  call $p(a, b)$
  call $p(c, d)$
end

procedure $p(e, f)$	**procedure** $a(g)$	**procedure** $c(h)$
call $e(f)$	**call** g	**call** h
end	**end**	**end**

The call multigraph for this example is shown in Figure 2.6. The solid lines show edges added to the graph by either method, and the dashed lines represent additional edges added by the new method.

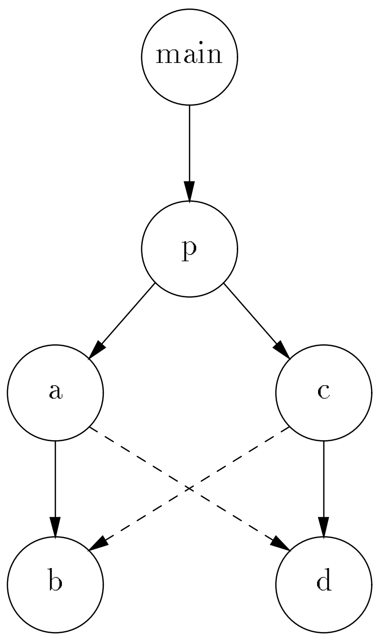


Figure 2.6 Call multigraph generated by precise method, with additional edges added by new method shown in dashed lines.

The extra edges are added because the new algorithm tracks the bindings for the formals *individually*, while the precise algorithm maintains *simultaneous* bindings contributed from each call. Thus, the new algorithm produces a less precise graph whenever there exist two distinct calls to a procedure propagating bindings where at least two of the bindings for the formals differ. This is not possible when at most one procedure formal exists for any procedure. Thus, we do not expect that it will happen often in FORTRAN.

2.6.3 Early Work

Early algorithms for call multigraph construction in the presence of procedure formals were suggested by Spillman [Spi71] and Walter [Wal76]. Spillman builds the call multigraph as part of an overall interprocedural analysis approach for PL/I. Using the "expose matrix" he captures information about values for procedure parameters as well as variable aliasing, values for label variables, modification side-effect information, available expressions and exception handling. The algorithm formulation requires two passes over the program, but in some cases, such as with invoked procedure parameters, after propagation we may find new values that require additional passes over the program. The worst case scenario would require repeating passes over the program for each edge added to the graph; that is, for each unique binding of an invoked procedure formal. Thus, the worst-case time bound is $O(PE_p)$ passes over the program. The resulting call multigraph is imprecise.

Walter builds the call multigraph to locate recursion in the program. His method supports Algol 60, which adds the requirement of handling call-by-name parameter passing semantics.[3] The call multigraph is described using boolean relations, requiring repeating transitive closure and and composition of the relations until the graph stabilizes. As a result, the algorithm is not very efficient.

2.6.4 Extensions for Other Languages

The next step with this algorithm is to extend it to handle other language features such as assignments to procedure-valued variables. Shivers' work on control flow analysis in Scheme suggests that such extensions are possible, although the result might not be very precise [Shi88]. He converts Scheme programs to *Continuation Passing Style* so that all transfers of control can be represented as function calls. Then the problem of representing the control flow in the program is exactly the same as constructing the call multigraph. The method he describes could use the algorithm that has been presented in this chapter with some minor extensions.

When a function is *stashed* into a data structure (similar to assignment), he makes worst-case assumptions about the effects of its callers. This is because its callers can include functions not currently part of the program. Similarly, when unknown functions are *fetched* from a data structure (a use of a function variable), he assumes that any of the stashed functions could be invoked. In dealing with other languages, these worst-case assumptions could be improved. For example, taking advantage of scoping rules in the language or types of procedure variables could refine the worst-case assumptions.

2.7 Chapter Summary

This paper has presented a very efficient algorithm for calculating the call multigraph in the presence of procedure-valued parameters. The new algorithm is less precise than the modified Ryder algorithm. However, we believe this loss of precision will not occur in FORTRAN where procedure parameters are infrequently used.

[3]Actual call-by-name parameters are treated as function definitions, and their uses as function calls.

The algorithm will perform efficiently on any input program. However, since procedure parameters appear so infrequently in FORTRAN, it will probably not perform much better than any other method in the typical case. The new algorithm will only prove itself to be much more efficient than other methods in situations where procedure parameters are more prevalent – either special classes of FORTRAN programs, or programs in other languages.

In using this algorithm over the precise one, there is a tradeoff between efficiency and precision. Unfortunately, in languages where procedure parameters are frequently used, loss of precision is more likely using the new method. Thus, selecting an algorithm for building the call multigraph requires consideration of whether efficiency is more important than the precision that may be lost by the more efficient technique.

The next step in call multigraph construction is dealing with other language features, including assignments to procedure-valued variables. A precise algorithm would require tracing data-flow in the source. However, a straightforward adaptation of this algorithm could produce conservative results. The value of the adapted algorithm would depend on the prevalence of assignments and invocations of procedure-valued variables.

Chapter 3

Inline Substitution

As we stated in Chapter 1, interprocedural optimization is needed because it increases the context available to the optimizing compiler. Of all interprocedural optimizations, inline substitution provides the most context to the compiler since it makes code for the called procedures directly available to the optimizer.[4] By understanding the effect of inline substitution on optimization, we can also anticipate the kinds of improvements possible from other interprocedural techniques.

This dissertation research began with a study of the interaction between inline substitution and aggressive code optimization. We sought to divide the improvements due to inlining into two categories: those that can be approximated with interprocedural information, and those that necessitate inlining. For the latter, we also hoped to define some simple heuristics to predict cases when inlining is profitable.

The previous work on inlining, discussed at the end of the chapter, did not adequately explain the impact of inlining on optimizing compilers. Moreover, the results seemed to depend on the programming language, the amount of optimization performed in the compiler and architectural features of the machine.

To explore these issues, we conducted the experiment depicted in Figure 3.1, an investigation into the efficacy of inlining in FORTRAN. (FORTRAN is a particularly interesting language because it has a long tradition of high quality optimizing compilers.) We built a user-controlled, source-to-source inlining facility. This tool allows us to examine a FORTRAN source program, apply an inlining strategy by manually marking call sites, and automatically produce a transformed FORTRAN source that reflects the inlining. Because both the original source and the transformed source are valid FORTRAN programs, we can then compile and execute them on a variety of target machines. To serve as a basis for our study, we transformed a set of eight programs. We then compiled and ran them on five different machines, taking mea-

[4] This is not exactly true since inlining is usually performed on a limited number of call sites, while optimization based on interprocedural information can be performed around all call sites.

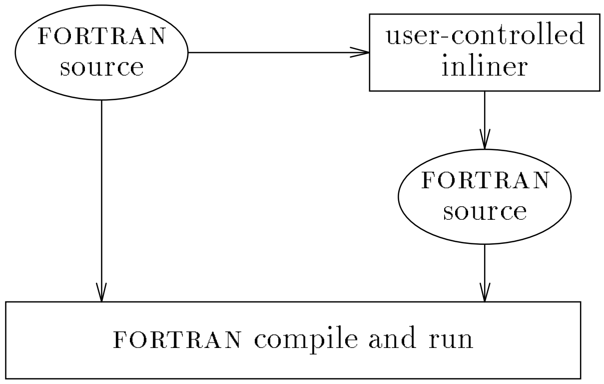


Figure 3.1 Structure of the inlining experiment.

surements throughout the process. A discussion of the inlining study can also be found in [CHT90b].

The remainder of this chapter is divided into five main sections. The next section discusses the experimental methodology in more detail. Section 3.2 summarizes our measurements and discusses what these findings show about changes in source code size, object code size, compilation time, and execution time. Section 3.3 draws together conclusions from the preceding sections and summarizes them. One of the conclusions from the study is that register allocation is significantly affected by inlining. A discussion of the effects of inlining on register allocation is in Section 3.4. In Section 3.5, we discuss important issues that arose in the implementation of the inlining tool itself. It provides an overview of the implementation, followed by some detailed discussion of cases where the tool cannot inline a call site. The chapter closes with a discussion of previous work in inlining and a summary.

3.1 Experimental Methodology

The experiment took place in two phases. In the first phase, we created a set of transformed programs. In the second phase, we compiled both the original source and the transformed source on each machine, and took measurements of the compile time and run time of each.

3.1.1 Phase 1: Transforming Programs

To create the transformed source, we built a two-part facility for user-directed inline substitution. First, we built an interface to the program compiler that allows the user to specify, on a call site by call site basis, those calls that are to be inlined. The level of granularity is important; selecting individual call sites allows maximal freedom in designing inlining strategies to evaluate. The tool allows the user to navigate around the call multigraph, simplifying the task of finding all such call sites.

The second step of the process actually constructs the transformed source. The inliner reads the program description and the annotations that specify the inlining pattern. With this information, it constructs a new abstract syntax tree (AST) for each procedure in the transformed program. To do this, it reads in the AST for the base procedure and then merges into that tree the ASTs for the inlined procedures. It then prettyprints the tree to produce a text file containing the transformed FORTRAN source.

To avoid redundant inlining, the call sites are processed in reverse topological order. Thus, if we want to inline a call from some procedure p to another procedure q, we do this before we inline p into any of its callers. This way, the inlining of q into p occurs only once, rather than once for each of p's callers. A topological order exists for non-recursive programs, which was the case for all of the programs used in the study. When recursion exists, the call sites are visited in essentially topological order, with the exception of calls to procedures involved in recursive cycles.

The tools inside the environment imposed a limit on our ability to inline. The AST used to represent FORTRAN source is relatively large, around 1000 bytes of tree per line of source text. This imposed a practical limit of 2500 lines on the size of any single procedure. For larger trees, the performance of the tools on our workstations degraded quickly due to the large virtual working sets. Given this restriction, we inlined any call site that met one of the following criteria:

1. It invoked a procedure of fewer than 25 source code lines.
2. It was the sole call to a procedure of less than 100 source code lines.
3. It was contained in a loop and invoked a procedure of less than 175 source code lines.

Due to limitations imposed by features of FORTRAN or details of the tool's implementation, we could not inline some call sites. The five cases that arose are discussed in detail in section 3.5.

Initially, we examined twelve numerically intensive FORTRAN programs. As the study progressed, we dropped four of the programs due to specific nonstandard properties of the individual programs that made inlining illegal. The remaining programs are of moderate size, ranging from 297 to 5979 non-comment lines of code. The eight programs included in the study are:

vortex	a particle dynamics code simulating the dynamics of a 1-dimensional vortex sheet via discrete vortices;
shal64	a simple atmospheric dynamics model based on the "shallow-water" equations;
efie304	solves electromagnetic scattering problems involving arbitrarily shaped conducting surfaces;
wanal1	boundary control of the wave equation by Conjugate Gradient Method;
wave	a 2-dimensional relativistic electromagnetic particle simulation used to study plasma phenomena;
euler	a 1-dimensional spectral code modeling shock waves propagating in a tube, bursting diaphragm flows and colliding shock wave flows;
cedeta	an implementation of the Celis-Dennis-Tapia method for equality constrained global minimization; and
linpackd	the classical "Dongarra" benchmark of LINPACK routines.

Figure 3.2 gives some basic information about each of the programs included in the study, in both the original and transformed state. The programs are ordered by percentage growth in text size. We use this ordering throughout the paper, in both tables and graphs, except where explicitly stated otherwise. The percentage of calls inlined give a direct measure of our heuristics' effectiveness. (The number of call sites represented includes calls to libraries, such as LINPACK, but not calls to intrinsics, such as *abs*.) A simple count of the call sites inlined shows that, on average, 75 percent of the call sites were eliminated from the programs. All of the programs except vortex grew in the process. The average procedure length grew in every case.

The column labeled "% Inlined — Dyn" shows the number of procedure call executions eliminated. This number is expressed as a percentage of the calls executed by the original program on the same data. On average, our heuristics were able to eliminate 89 percent of the executed procedure calls. Thus, the transformations eliminated the vast majority of the time spent in procedure call overhead at run time. (In five of the programs, we eliminated over 99 percent of the dynamic calls.)

Name	Total Calls	% Inlined Stat	% Inlined Dyn	Original Source Total Lines	Original Source Total Procs	Original Source Avg Proc Length	Transformed Source Total Lines	Transformed Source Total Procs	Transformed Source Avg Proc Length
vortex	19	100	100	534	19	28	527	1	527
shal64	25	96	100	297	8	37	321	2	161
efie304	40	83	100	1248	18	69	1456	8	182
wanal1	43	84	100	1252	11	114	1751	8	219
wave	223	52	75	5979	92	65	8820	53	166
euler	31	65	57	1098	13	84	1646	4	412
cedeta	247	79	82	4269	48	89	9296	20	465
linpackd	34	44	100	417	10	42	988	4	247
Average	83	75	89	1887	27	66	3101	13	297

Figure 3.2 Characteristics of original programs.

3.1.2 Phase 2: Measurement

To evaluate the effectiveness of the inlining strategy, we compiled each program, both the original and transformed version, on each of our target machines. We recorded compile time and run time for each program at each level of optimization provided by the compiler. To obtain reliable timing information, we repeated each operation multiple times.

This part of the experiment was performed on each of five machines. Three of them are scalar machines: an IBM 3081d, a MIPS 120/5, and a Sequent Symmetry (used as an 80386 uniprocessor). The other two machines are vector multiprocessors: a Stardent Titan and a Convex C240. These two machines provided us with some insight into the expected results on a large class of modern supercomputers or minisupercomputers. Figure 3.3 shows configuration information about each of the target machines. While this collection of machines is small, we feel that it is reasonably

Machine	CPU	Number CPUs	Memory Size (mb)	Compiler & Version
IBM 3081d	3081	1	32	VS FORTRAN 2
MIPS	R2000	1	16	MIPS FORTRAN 2.0
Sequent S81	80386	1	80	SVS FORTRAN 3.2
Convex C240	Convex	4	512	Convex FORTRAN V5
Stardent Titan-p2	R2000	4	64	Stardent FORTRAN 2.1.1

Figure 3.3 Target machines and compilers.

	Execution Times — Base Numbers (hours:minutes:seconds)				
	3081	M120	S81	C240	Titan
vortex	17:20.9	15:10.4	49:05.8	8:21.5	38:41.2
shal64	1:37:00.5	42:20.4	2:57:36.6	11:03.1	32:27.8
efie304	16.8	14.6	1:02.9	7.9	29.1
wanal1	6:57:45.0	4:51:03.9	15:22:09.6	54:36.1	2:53:53.8
wave	1:23:18.3	1:11:55.7	2:52:40.2	25:12.8	1:11:55.5
euler	1:09.7	47.1	3:09.6	20.8	32.9
cedeta	34.1	30.8	1:17.1	14.5	41.0
linpackd	34.7	25.1	1:59.4	14.6	31.0

	Compilation Times — Full Optimization (minutes:seconds)				
	3081	M120	S81	C240	Titan
vortex	2.6	10.3	7.2	13.4	39.9
shal64	1.7	7.9	5.1	7.8	21.5
efie304	7.6	52.6	13.3	26.2	1:32.0
wanal1	19.8	8:42.4	20.2	1:37.0	3:49.6
wave	35.0	2:54.7	1:05.2	2:14.6	9:25.4
euler	5.0	29.6	12.9	23.4	1:15.0
cedeta	18.1	1:39.1	2:01.1	1:12.4	8:55.2
linpackd	2.6	11.3	5.8	12.6	31.6

Figure 3.4 Baseline data for comparisons.

representative of the market today. Each machine is a well established platform that, within its market niche, is regarded as having relatively stable and solid software.

Figure 3.4 provides baseline data for later comparison. The first table shows execution times for each program on each machine. The measurements were made using the original source code, with no code optimization. Throughout the paper, execution times are given as percentages of these base times. The second table shows compile times for each program on each machine. In each case, the measured compilation is at the highest level of optimization available on the machine (vector multiprocessor mode on the Convex and Stardent machines).

3.2 Experimental Results

One of our primary goals was to gain an understanding of the efficacy of inline substitution in reasonable quality commercial FORTRAN compilers. Three key issues arise: (1) growth in object code size, (2) growth in compile time, and (3) improved run-time efficiency. In this section, we examine the data collected in our experiments and draw some conclusions about each of these issues.

In general, the conclusions that can be drawn from our experiment present a mixed picture. To simplify the presentation, we will first discuss overall trends and conclusions, and then consider any differences between the scalar machines and the vector multiprocessors. In the experiment, we gathered data on each compiler at each of its various levels of optimization. Unless otherwise stated, we will cite numbers for each compiler at its highest level of optimization.

3.2.1 Program Characteristics

The three issues described above arise because inlining changes fundamentally the characteristics of the user's program. As shown in Figure 3.2, the transformed program is almost always larger than the original program, both in terms of total source lines and the size of an average procedure.

Because the procedures in the transformed source are constructed mechanically, their name spaces are likely to be larger than those of equivalent procedures written by humans — a human would re-use temporary names where the inliner will merge the two name spaces and rename to avoid conflicts. Thus, if we inline procedure q into p at two distinct call sites, the resulting code will contain two complete copies of q's local name space, rather than the single copy (or less) that a version of the same code written by a human would use.

The transformed source, with its larger procedures, should provide more contextual information to the compiler and its optimizer. Expanding the callee's body at the call site exposes a wealth of information to the compiler. It exposes constant-valued formal parameters and makes explicit the aliases that arise from parameter binding. Definitions and uses are now visible to standard single-procedure analysis techniques. Loops that previously contained calls are now susceptible to analysis with standard techniques, like strength reduction, that are not applied to a loop that contains a call. (On the SVS compiler for the Sequent, this effect was particularly noticeable. The improvement in execution time due to enabling strength reduction was always larger in the inlined programs than in the original versions.) In some cases, this information could be derived by an aggressive interprocedural analysis of the original source code. In many cases, however, the program is subject to more precise analysis after inlining because of the additional information that is exposed through inlining.

During the study, we encountered a number of coding practices that made in-line substitution impossible. Most of these also violate the FORTRAN standard. Procedures that rely on a default assumption of static allocation for all local variables was the most common problem encountered. In general, these variables appeared in DATA statements. Explicitly passed array dimensions that varied across the invocations of call sites was a second common problem. In most cases, this arose in connection with an array used to provide temporary storage. (The lack of dynamic allocation has given rise to this style of programming.) We encountered several call sites where actual parameters and their corresponding formals had different types. Finally, two of the programs contain jumps into loops from outside the loop. The programs containing these problems were either corrected or removed from consideration in the study.

3.2.2 Object Code Size

Most discussions of inlining include a warning to suggest that growth in object code size is a possible concern. Two specific problems may arise: increased loop size overflowing instruction caches and increased working set sizes swamping demand paged virtual memory systems. Certainly, one can construct pathological examples that will exhibit geometric growth in source code size. The arguments presented in the introduction to justify inlining as a code improvement technique also suggest that optimization should moderate object code expansion. Optimization improvements come from two sources: eliminating code and generating less general, more specialized code. The former directly shrinks the object code. The latter is expected to eliminate control flow, expose code as dead code, and increase the effectiveness of techniques like common subexpression elimination and constant folding. These, in turn, should lead to some reduction in object code size.

Figure 3.5 plots the data on source text growth and object code growth for our programs on the five compilers. While the results varied both with program and with compiler, none of the compilers exhibited linear growth in object code size. Four of the five compilers exhibited average growth of less than 6 percent, while the fifth (the

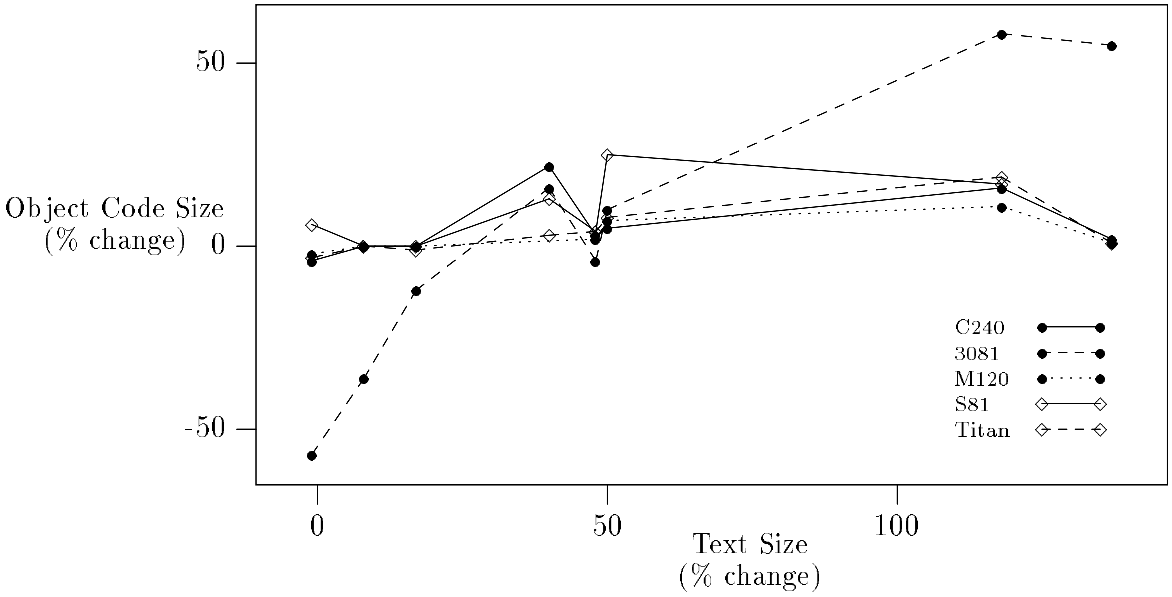


Figure 3.5 Object size vs. source text size.

SVS FORTRAN compiler) averaged less than 10 percent. The MIPS compiler showed minimal variation; its growth numbers all fall within 11 percent of the original code. The largest variation came with the VS FORTRAN compiler; it showed a range from -57 percent on `vortex` to 58 percent on `cedeta`. Almost all of the individual data points lie in the range between -5 and 25 percent growth.

This suggests that optimization did indeed mitigate growth in object code. To examine this issue in more detail, we compared object code size with no optimization against that produced with full optimization. We looked at changes to both the original and inlined versions of the program. Figure 3.6 shows the change in object code size due to optimization, expressed as a percentage of the pre-optimization object code size. For each compiler, the changes to the original program are reported in the left column, and the changes to the inlined program are reported on the right. (For the Convex and Stardent machines, we used compilations with full optimization targeted for a scalar uniprocessor configuration — no attempt to use vector or parallel hardware.)

The ability of the individual compilers to reduce object code size through optimization differs greatly. The Convex and Stardent compilers show average decreases in the range of 1 to 2 percent. The MIPS and VS FORTRAN compilers exhibit more substantial average decreases between 5 and 28 percent. The SVS compiler exhibits

	Text	Change in Object Code Size %									
	Size %	3081		M120		S81		C240		Titan	
vortex	-1	-17	-28	-3	-3	-6	6	0	-2	0	0
shal64	8	-25	-35	-1	-2	0	0	-2	-2	0	*
efie304	17	-19	-23	-6	-7	0	0	-2	-2	-3	-3
wanal1	40	-36	-36	-10	*	0	-6	-2	0	-7	-7
wave	48	-21	-25	-8	-8	-1	-1	0	0	0	0
euler	50	-22	-27	-4	-13	0	-3	0	-2	0	-3
cedeta	118	-21	-26	-13	-24	11	9	-1	-1	-2	-2
linpackd	137	-17	-25	-1	-9	-5	*	0	0	0	0
Average	52	-22	-28	-5	-9	-1	1	-1	-1	-2	-2

Figure 3.6 Effect of optimization on object code growth.

code growth as a result of optimization in three cases. To isolate the problem, we compared object code sizes generated by different settings of the optimization flag. On the inlined version of vortex, the problem first appears when global register allocation is enabled. On both versions of cedeta, the increased size appears when, according to the compiler manual, only constant folding is enabled.

The general trend in the data in Figure 3.6 suggests that decreases in object code size as a result of optimization are greater for the inlined version of the program than for the original version. With the exception of the SVS compiler, all compilers showed a greater percentage decrease in object code size in the inlined version for at least one program. The differences are very small for the Convex and Titan, but are significant for the 3081 and the MIPS. Comparing the 37 pairs of entries in the table, there are 17 pairs where the reduction for the inlined program is greater than for the original program. There are only three where the reduction is greater for the original program, and two of these occurred with the SVS compiler. (Three pairs are excluded from the count because the numbers for the inlined versions were unavailable.)

Taken as a whole, these numbers suggest that the object code growth resulting from inlining is manageable. Given the large percentage of dynamic calls eliminated by our heuristic, it appears that, in most situations, the compiler can eliminate the majority of calls, and, hence, most procedure call overhead, without a substantial increase in object code size.

We have demonstrated that inlining does not disastrously increase object code size. This suggests that code growth due to inlining will not have an adverse effect on paging and caching. However, the actual impact of changes in object code size on

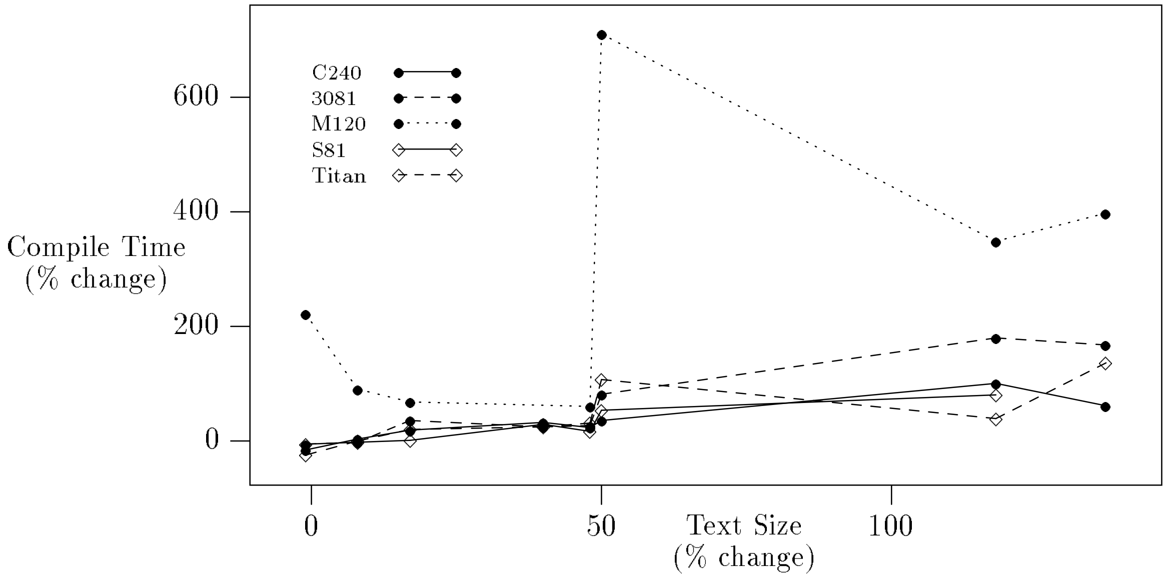


Figure 3.7 Compile time vs. source text size.

execution time performance is much harder to assess and is outside the scope of this dissertation.

3.2.3 Compile Time

A second argument used against inline substitution is the potential for dramatic increases in compile time. In practice, compilers use some algorithms that have non-linear asymptotic complexities. This raises the concern that applying inlining can lead to unacceptable increases in compile time.

Figure 3.7 plots increase in compile time against increase in program size. All compilations are shown with full optimization enabled. From the plot, it appears that euler has some fundamentally different characteristics from a compiler's viewpoint. Each compiler showed a large jump in growth of compile time between wave and euler. The times for cedeta and linpackd appear to better fit the trend established without euler.

Most of the compilers had fairly stable compilation times. Compile time on the Convex never grew faster than source text size. The Stardent and SVS compilers each had one program where growth in compile time exceeded growth in source text. The VS FORTRAN compiler had mixed results; its compile time growth exceeded source text growth half of the time. (On the other hand, it alone showed substantial decreases in object code size.) Finally, the MIPS compiler appears to be extremely sensitive to average procedure size. It alone showed consistent superlinear growth in compile time.[5]

The performance of both the Convex and Stardent compilers is particularly satisfying, since both perform sophisticated data dependence analysis to support their vector and parallel hardware. Despite the costs associated with this analysis, neither compiler saw extraordinary growth in compile time when dependence analysis was enabled. While the numbers in Figure 3.4 suggested that the Stardent compiler is somewhat slower than the other compilers in the study, the speed problem shows up when compiling the original source without optimization for a uniprocessor machine. This strongly suggests that it is not related to either optimization or dependence analysis.

3.2.4 Execution Time

Given that compilers can mitigate the potential for explosion in object code size and compile time, one key question remains. Can compilers capitalize on the opportunities presented by inlining to improve the actual execution time of programs? Figure 3.8 shows the overall effectiveness of inlining by plotting the change in execution times for each combination of program and machine. Figure 3.9 plots the change in execution time as a function of change in compile time. In both figures, the times shown for the Convex and Stardent machines are for multiprocessor vector execution.

Figure 3.10 gives the raw data from which these plots are derived. The numbers in the figure represent the execution time for the program versions at each optimization as a percentage of the execution time of the unoptimized original program. The baseline data is reported in Figure 3.4. In Figure 3.10, the averages for a compiler were computed by completely discarding any program that the compiler failed to successfully translate. (Such programs are denoted by asterisks in the tables.)

Scalar Results

In Figure 3.8, it is clear that there is no real trend, either by compiler or by program. The results are mixed, with many instances each of improvement and of degradation.

[5]The MIPS compiler also showed a surprising growth in working set size for compilation. To obtain timings that were not dominated by paging overhead required a machine with 48 megabytes of memory. Running the compiler on a machine with 16 megabytes of memory resulted in radical increases in compile time. For example, wanal1 *before inlining* required over 95 hours of wall time to compile. On the 48 megabyte system, that compilation required around nine minutes.

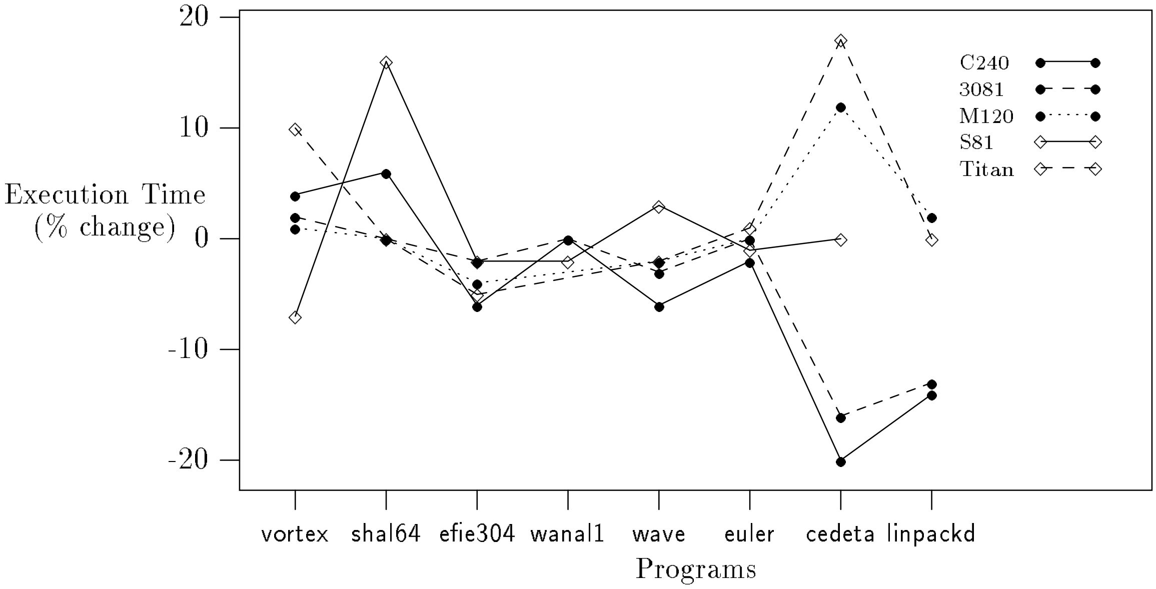


Figure 3.8 Change in execution time from inlining.

- Across the board, the IBM compiler had the most stable performance. It showed the best improvement of the three scalar machines, 16 percent on `cedeta`. There were two cases where inlining caused it to produce slower code; in each case the degradation amounted to less than 1 percent of the execution time. In general, the IBM compiler was able to improve the code after inlining.
- The MIPS compiler exhibited relatively stable performance. It showed little improvement or degradation as a result of inlining, with one exception being a 12 percent degradation on `cedeta`.[6]
- The SVS compiler had mixed results — four wins and two losses. It produced the single largest degradation among the scalar machines, 15 percent on `shal64`.
- The Convex compiler, targeting a uniprocessor vector configuration, profited from the transformation. Six of the programs showed improvement; the two

[6] At its highest level of optimization, the MIPS compiler performs interprocedural register allocation. Because the effects of interprocedural register allocation could mask the effects of inlining, the numbers reported for the MIPS are without interprocedural register allocation. In most cases, the performance change after interprocedural register allocation was negligible, with four exceptions: the inlined version of `cedeta` exhibited a 15 percent improvement due to interprocedural register allocation; both versions of `linpackd` were slowed down by about 5 percent after interprocedural register allocation; and the inlined version of `efie304` was slowed down by 27 percent.

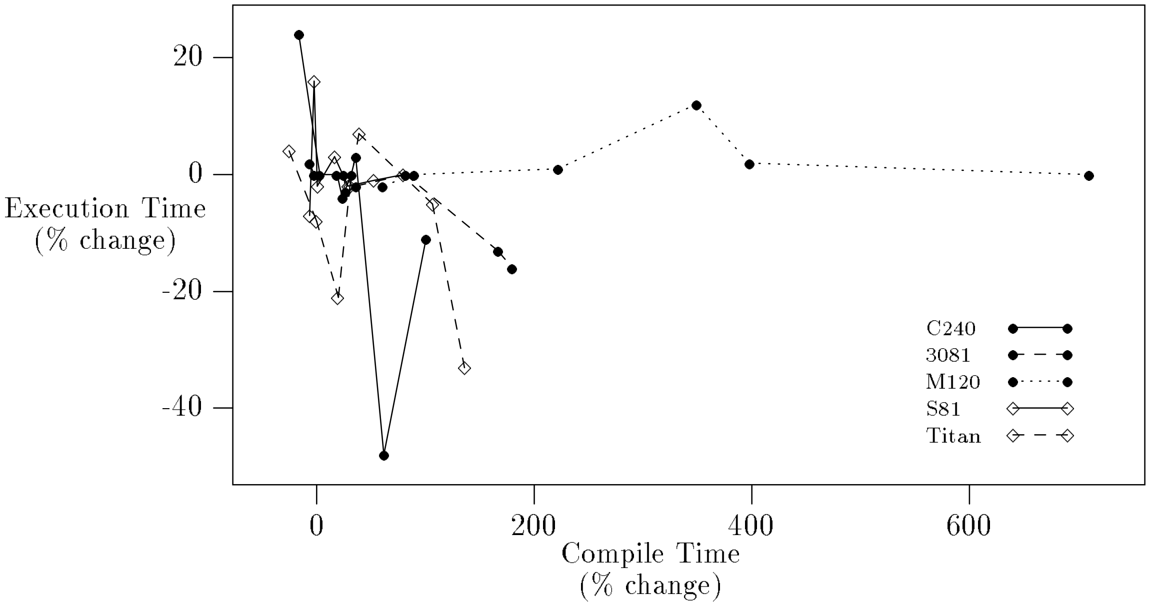


Figure 3.9 Change in execution time vs. compile time.

---

degradations were both small. It showed the largest overall improvement, 20 percent on cedeta.

- The Stardent compiler, targeting a uniprocessor vector configuration, showed little improvement from inlining. Its best improvement was 5 percent on efie304. Its worst degradation was 18 percent on cedeta.

There does not appear to be a clear trend in the data, either by compiler or by program. Overall, wins outnumber losses. But, the vast majority of the wins fall in the range between the infinitesimal improvement and 5 percent.

Figure 3.9 suggests that increased compile time also fails to predict improved execution time. Three of the compilers get marginally better results at the high end of the compile time scale; the MIPS compiler often fails to get back to the original program's execution time.

These results provide insight into another issue: the relative importance of procedure call overhead. The transformation process eliminated most procedure calls; in five of the programs, we eliminated over 99 percent of the executed calls. This did not lead to consistently faster execution.

Two conclusions are possible. Either call overhead is a negligible part of program execution, or program properties introduced by inlining resulted in decreased code

	IBM 3081d			MIPS M120/5			Sequent S81		
	uopt + inl	opt	opt + inl	uopt + inl	opt	opt + inl	uopt + inl	opt	opt + inl
vortex	100	65	66	102	68	69	99	82	76
shal64	100	19	19	100	33	33	100	38	44
efie304	97	46	45	88	50	48	100	92	90
wanal1	100	16	16	100	34	*	100	46	45
wave	101	37	36	100	41	40	100	67	69
euler	100	34	34	103	30	30	100	79	78
cedeta	96	61	51	115	43	48	108	85	85
linpackd	193	39	34	171	44	49	137	67	*
Average	110.9	39.6	37.6	111.3	44.1	45.3	101.0	69.9	69.6

	Convex C240					Stardent Titan				
	uopt + inl	vect	vect + inl	para	para + inl	uopt + inl	vect	vect + inl	para	para + inl
vortex	93	26	27	21	26	63	60	66	28	29
shal64	100	18	19	6	6	*	32	32	12	11
efie304	94	64	60	40	40	87	79	75	96	76
wanal1	117	19	19	7	7	101	*	13	*	*
wave	101	32	30	26	25	98	63	62	60	59
euler	99	51	50	69	71	93	84	85	200	190
cedeta	92	89	71	160	143	101	101	119	168	179
linpackd	109	21	18	27	14	92	20	20	21	14
Average	100.6	40.0	36.8	44.5	41.5	89.0	67.8	71.2	97.5	91.2

Figure 3.10 Changes in execution time.

quality that masked the savings in call overhead. The former point may be the result of the FORTRAN programmers' reluctance to use procedure calls. If procedure calls make up only a small amount of the work in the program, eliminating all of them does not have a significant effect on performance. However, it appears from Figure 3.10 that interactions with the compiler are also hiding the call overhead savings. The unoptimized execution of the transformed source on the three scalar machines shows little improvement and sometimes degradation from eliminating calls; the Convex and Stardent compilers do a somewhat better job. As an example of an adverse effect of inlining, we know from experience with ParaScope's compiler that inlining often increases register pressure.

Parallel Results

Two of the machines are vector multiprocessors, the Stardent Titan and the Convex C240. They both have stable restructuring compilers — compilers that attempt to automatically discover opportunities for vector and parallel execution. Both compilers use techniques based on data-dependence analysis. Thus, they provide us with the opportunity to examine the issue of inlining as an aid in automatic parallelization.

In general, the compilers were able to generate better code for the transformed source text. For both compilers the difference was just under 7 percent. Looking at specific programs, each compiler had two cases where the transformed code ran slower than the original. On the Convex, vortex ran 24 percent slower; on the Stardent cedeta slowed down by 7 percent. The remaining cases range from minimal differences to major improvements — 48 percent for linpackd on the Convex and 33 percent for linpackd on the Stardent.

This suggests that seven of the programs could be improved by inlining. (Both compilers failed to improve vortex.) However, the execution time improvements were small compared to what might be expected. We wondered if the loops with inlined call sites still had dependences that were preventing parallelization, or the compilers were parallelizing unprofitable loops.

To make this determination, we examined each loop containing procedure calls in our set of programs. Wherever a call appeared in a loop nest, each loop in the nest was counted. For each loop, we looked at the diagnostic information provided by both compilers. The results appear in Figure 3.11.

Line 1 of the table represents the static count of loops with calls. The numbers in line 2 represent those loops from line 1 where inlining has eliminated the calls.

	vortex	shal64	efie304	wanal1	wave	euler	cedeta	linpackd
1. DO loops with calls	1	0	15	20	20	6	23	10
2. no calls after inlining	1	0	14	20	15	0	22	8
3. Convex improved	0	0	4	6	1	0	0	1
4. Titan improved	0	0	1	4	1	0	1	1

Figure 3.11 Static count of loops with call sites.

Because the inlining was guided by heuristics, there are a few cases where the calls remain. Lines 3 and 4 indicate the loops from line 2 that each compiler was able to at least partially parallelize.

The outcome of this exercise was the observation that in 5 of the programs – efie304, wanal1, wave, cedeta and linpackd – a large number of loops have been opened up to analysis, but few have any additional parallelism. This prompted a more detailed study of the loops that the compilers were unable to parallelize. Inlining had exposed parallelism in some of the loops, but the compilers failed to locate it. We observed certain properties of the loops unlike code that would be generated by a human. These properties suggested some optimizations that should be performed between inlining and dependence analysis to aid the compiler in locating additional parallelism. The optimizations and an experiment demonstrating their effectiveness is described in Chapter 7.

The table in Figure 3.11 shows that the compilers did not find parallelism in a majority of the loops containing inlined calls. However, it also responds to the second question — were the compilers parallelizing unprofitable loops?

Both compilers had problems deciding when parallel execution was profitable. Compare the execution times of the original code in uniprocessor vector mode against the results for multiprocessor vector mode. On the Convex, three of the programs ran slower as parallel programs than as uniprocessor programs. On the Stardent, four programs exhibited this behavior. This was particularly bad for both euler and cedeta. This problem exists before inlining, but can be compounded as a result of inlining by exposing parallel loops that are not profitable to parallelize, as in the case of cedeta on the Stardent. It appears that both compilers assume that parallel execution is profitable unless it can discover a tight bound on loop iterations that conclusively proves otherwise.

3.3 Implications

Overall, the five compilers were unable to consistently capitalize on the opportunities provided by inlining. Recall, from Figure 3.2, that our inlining strategy eliminated nearly all procedure calls — on average, 89 percent of the procedure calls executed in the original source were eliminated in the transformed source. The expected run-time savings do not appear during execution, as shown in Figure 3.10. We did not find a consistent and appreciable improvement in run-time speed. Several factors appear to account for this phenomenon.

First, inlining often increases register pressure. Both vortex and euler contain call sites that pass global variables from several COMMON blocks as actual parameters. In the original code, the actual parameters in the callee have one-register names. After inlining on some machines, they have two-register names, a base address plus an offset.[7] On the MIPS machine, the transformed version of euler executed 2 percent more loads and stores — that is, 1,900,000 more loads and stores — than the original version. Those cycles mask other improvements. On those compilers where the application of global register allocation is controlled by a user-supplied switch, the results of global allocation were mixed. To better understand why inlining affects register pressure, we compared assembler code before and after inlining. The results are presented in the next section.

Second, the inlined code may have fundamentally different properties. For example, in linpackd on the MIPS, inlining resulted in a 6 percent degradation in run-time speed. Closer investigation showed that the number of floating point interlocks per floating point operation rose from 0.62 to 1.1 after inlining. Since the program executes almost 20,000,000 floating point operations, that factor is significant.

Closer investigation revealed that the call from *dgefa* to *daxpy* passes two regions inside a single array in two distinct parameter positions. Unfortunately, complicated range analysis would be required to show that the regions do not overlap. Thus, when the call is inlined, the body of the key loop in *daxpy* is a single statement that both reads two locations and writes one location inside the array. Without complex analysis, the compiler must assume that the references can overlap. This introduces the data interlocks that we observed. A discussion of the analysis required to detect the changes brought about by inlining is found in [CHT90a].

Finally, design decisions that are justifiable for code written by a programmer may have unforeseen consequences when applied to code generated by the inliner. For example, many compilers place a hard upper limit on the number of variables subject to data-flow analysis; typically, they summarize all remaining variables with

---

[7]This accounts for the two instances of degradation on the IBM 3081. It may also account for the degradation that occurs when vectorization and inlining are combined on the Stardent. Although no additional loops were vectorized on these programs due to inlining, the execution time increased on both programs.

a single bit position.[8] With inlined code, the growth in name space as a function of procedure length is probably much greater than in human-generated code. This may exacerbate the adverse effects of summarization in data-flow analysis.

On the two parallel machines, we observed several cases where the decision to run a loop nest in parallel resulted in a disastrous slowdown. This happened with both euler and cedeta. It appears that both compilers assume that the parallel loop is profitable in those cases where the number of iterations is unknown. A simple strategy would allow the compilers to avoid these major slowdowns: in those cases where the number of iterations is unknown, generate both the uniprocessor and multiprocessor versions of the loop and insert a simple run-time test. Our results suggest that the growth in object code may be small. If doing so eliminated these slowdowns while allowing the compilers to aggressively generate parallel code, it would almost certainly make up for the minor object code growth.

3.4 Effects of Inlining on Register Allocation

To understand the impact of inlining on register allocation, we carefully compared the assembler code for the original and inlined versions of the programs generated by the MIPS compiler. The MIPS compiler was particularly amenable to this exercise for two reasons: (1) it annotates its assembler code with the source code, and (2) statement-level profiling information is available through the tool pixstats.

We observed the following effects of inlining on register allocation:

- The live ranges of values in the caller were sometimes extended.
- Renaming from formals to actuals changed the number of registers needed for address calculation.
- By inlining all calls within a procedure, registers were available for longer spans of the procedure.

Each one of these issues is described in this section.

[8]This strategy makes all bit-vector operations have a known, uniform length. It allows the compiler writer to hard code the bit-vector operations, and eliminate the loop based on bit-vector length on each operation.

3.4.1 Sharing Values across Procedure Boundaries

If a value is used by both the caller and the called procedure, there are two possible effects on register usage as a result of inlining. If the caller uses the value both before and after the call, then the value will already be in a register across the call. Thus, inlining will open up a reuse of a value already in a register, so one fewer register is required for the callee body.

On the other hand, suppose the caller uses the value only on one side of the call site, either before or afterward. Then the value stays in a register from its use in the caller to its use in the callee (or vice versa), rather than being loaded into a register immediately before each use. As a result of inlining such a call site, the caller will require one additional register in the region between the call site and the caller's use of the value.

An example from linpackd demonstrated how extending a live range could result in register pressure. Suppose an expression $n - i$ is calculated in a procedure, where n and i are formal parameters of the procedure. In the caller of this procedure, the expression $x - y$ is calculated, where x and y are the same values passed to n and i at the call site. As a result, after inlining the call site, the expression is calculated twice in the loop body. This causes the expression to stay in a register for both uses. Because the expression is loop-varying in the caller, it had previously been in a register only for the single use. Note that if the expression had been loop-invariant in the caller, it would have remained in a register across the call anyway, so this would not have increased register pressure.

3.4.2 Effects of Renaming from Formal to Actual

Inlining can have a tremendous impact on the number of registers used to perform address calculations. Because of the call-by-reference parameter passing semantics of FORTRAN, the compiler assumes that all actuals at call sites can be modified by the called procedure. As a result, most compilers implement parameter passing by passing the address of the actuals rather than their value. At some time in the body of a called procedure, each parameter has its address in a register. (On the MIPS, if it is one of the first four parameters, it is placed in a register for the call.)

In the case of an expression-valued parameter, placing the address in a register is a bit unnatural. The expression is calculated, and the value of the expression is placed on the stack. Then, this stack address is either passed in a register or on the stack. By inlining, the need for placing the expression's value on the stack and passing the stack address is eliminated. When parameters are constant integers, two registers are freed up since integer variables are used directly in instructions rather than being accessed from registers.

Both positive and negative effects on register pressure occur as a result of renaming arrays from formals to local or global variables in the caller. Addresses for an array access are calculated in the position in the code in which its most quickly varying subscript expression is being updated. Two array accesses from the same global or from local storage whose addresses are calculated in the same loop share a pointer to the beginning of the global or local storage, respectively. Two accesses handled at the same place in the code may also share other portions of the addressing calculation.[9] This sharing would not occur if the two accesses were to independent formals.

A special case occurs for 1-dimensional arrays. One register must be used to hold the beginning of the local variable storage, but offsets from there can often be calculated at compile time. Figure 3.12 shows an example from vortex where the change from parameter accesses to local variable accesses greatly decreases the number of registers used in the caller.

The placement of array address calculations has another effect as a result of inlining. Since addresses for array accesses are calculated in the loop in which the subscript expression is most quickly varying, the renaming from formal to actual can also affect the registers used to calculate subscript expressions. When formal parameters are used in subscript expressions, the address calculation may be moved to a different location in the caller after inlining. This would increase the portion of code in which the address for the array access is live.

3.4.3 Inlining All Calls in a Procedure

Many compilers divide the register set into *caller-saved* and *callee-saved* registers. The caller-saved registers must be saved and restored around procedure calls. For this reason, caller-saved registers are usually used for expressions with short live

[9] For example, accesses with identical subscript expressions from the rightmost dimension up to some previous dimension and equal constant dimension sizes will share the calculation of the identical dimensions. Also, accesses with identical subscripts from the leftmost position to some later position may share the identical portions if dimension sizes are the same for all dimensions, and if differing subscript expressions are constant (so that they can be calculated statically).

```
/* Before inlining, accesses in loop are to parameters */
procedure p(a, b, c, n)
     dimension a(n), b(n), c(n)

     /* Assume r1=addr(a)+i, r2=addr(b)+i, r3=addr(c)+i */
     do i = 1, n
         a(i) = ··· /* Access is to r1 */
         b(i) = ··· /* Access is to r2 */
         c(i) = ··· /* Access is to r3 */
     enddo
end

procedure q
     dimension l1(10), l2(10), l3(10)
     call p(l1, l2, l3, 10)
end

/* After inlining, accesses are to local variables */
procedure q
     dimension l1(10), l2(10), l3(10)

     /* Assume r1=beginning of local area + i */
     do i = 1, n
         l1(i) = ··· /* Access is to r1 */
         l2(i) = ··· /* Access is to 10(r1) */
         l3(i) = ··· /* Access is to 20(r1) */
     enddo
end
```

Figure 3.12 Accessing parameters vs. accessing local variables.

ranges, so that it is unlikely they remain live across the procedure call. The callee-saved registers are saved in a procedure before it uses them and are restored before exit from the procedure. Callee-saved registers are used for values that span a long range within a procedure [Cho88].

Upon inlining the last call in a procedure, the caller is able to freely use the caller-saved registers without ever having to save and restore them. Thus, inlining the final call in a procedure allows all registers to be used freely throughout the entire procedure body. Even though the same number of registers open up after inlining a call regardless of whether or not it is the last remaining call in the procedure, the span for which the register allocator can use the registers has changed.

This effect of inlining on register allocation was observed on the LINPACK routine *dgefa*. This procedure contains a loop nest of two loops. In the outer loop, there are calls to *idamax* and *dscal*. In the inner loop is a call to *daxpy*. Comparing the number of loads and stores when all calls are inlined versus inlining just *idamax* and *dscal*, 12 fewer accesses to memory occur in the fully inlined version. Inlining eliminates *all* unnecessary memory accesses.

3.5 Implementation Issues

Although our principal goal in performing this study was to increase our understanding of the interactions between inline substitution and global optimization, a subtask of the experiment was to construct the tools required to produce the transformed sources. In this section, we describe some of the lessons learned from the implementation.

3.5.1 Implementation Overview

As discussed earlier, we built the inliner as part of the program compiler. Inline substitution is implemented as a source-to-source transformation. It is performed on the AST representation of the program. The first step in inlining is a check to ensure that inlining is possible; this test checks for the five separate conditions that might make the transformation illegal in our system, described in Section 3.5.2.

After the compiler has proven that a particular substitution is legal, the actual transformation takes place in two phases. The first phase iterates over all of the symbols appearing in the called procedure, assigning unique names to its local variables and labels, and building a symbol table of the new names hashed on their original names. The second phase walks the AST, updates the names of all variables and labels, and moves the actual statements from the body of the callee to their new locations in the caller.

During the first phase, all global variables retain their original names. Common block definitions are added to the caller as needed. In the absence of name conflicts, local variables and labels retain their original names. If a conflict arises, local variable names are textually altered, and new labels are generated by incrementing the label number until a unique label is generated.

Formal parameters are renamed to their corresponding actual parameters from the call site. Two cases of interest arise: expression-valued parameters and array parameters. If the actual is an expression rather than a variable, the inliner generates a temporary and inserts an assignment before the procedure body to evaluate the expression and save its value. Constant-valued expressions are a special case. Unless the formal parameter appears on the left-hand-side of an assignment, no temporary is generated. (Although the FORTRAN standard forbids assignments to constant valued formals, it happens often enough in real programs to warrant handling it.)

The most complicated mapping of formals to actuals occurs with array valued actuals. The mapping relies on the fact that Fortran uses column-major storage. For an n-dimensional formal *f*, the inliner currently requires that the actual must match the formal in each of its first n-1 dimensions. The actual may contain more dimensions than the formal. Alternatively, the actual parameter may specify a location other than the first element of the array. This results in passing a subsection of the actual array.

The second phase of the inliner renames all references to variables and labels based on the translation table built in the first phase. Next it moves the statements that comprise the callee's body into an appropriate position in the caller. The executable statements are moved together.

- For a subroutine call, these statements simply replace the CALL statement.
- For a function call, they are moved immediately above the statement containing the call site. A temporary is created to hold the function's return value.

If the statement containing the call site has a label, the label is moved to the beginning of the inserted code. Declarations for the variables from the callee are inserted with those of the caller. COMMENT, IMPLICIT, and ENTRY statements are removed.[10] A number of minor issues arise in translating CALL and RETURN statements.

- RETURN statements are converted to GOTO statements that refer to a label immediately following the inlined procedure body.
- If the callee is a FUNCTION, then an assignment to the temporary designated for the return value is generated at each RETURN statement.
- If the procedure uses FORTRAN's alternate return mechanism to change the return address, then the RETURN statement is translated to a GOTO that targets the appropriate label-valued parameter.

[10]Comments are removed to limit the growth of the inlined program. The implicit statements are removed because they may conflict with implicit typing in the caller. Instead, type declarations are added when they do not already exist for every variable in the called procedure including those temporaries generated by the inliner. Finally, entry statements are removed so as to not have multiply defined subroutines.

- If the call site specifies an entry point rather than a procedure, the inliner must inline the entire procedure and add a jump to the location of the ENTRY statement. (That part of the callee's body preceding the ENTRY statement must remain intact in case the subsequent code jumps back to it.)

The inliner handles each of these appropriately.

Figure 3.13 shows an example which may clarify some of the issues. The placement of declarations and data statements ensures that meaning is preserved. Rather than attempt to resolve implicit typing conflicts between procedures, the inliner creates declarations for all implicitly typed variables from the callee. The declarations and data statements originating from b are sandwiched between a's declarations and data statements so that they remain in their original order. Any new declarations for b precede b's declarations to ensure that dependences between declarations are satisfied. An example of this might be a *parameter* statement that is used to dimension an array.

3.5.2 Failure to Inline

Five situations can prevent the inliner from constructing a valid FORTRAN program. Some of these are fundamental problems that any inlining tool will encounter; others are idiosyncratic to our implementation.

1. Inlining replaces the formal parameters of the callee with the actuals from the call site. For this to be meaningful, the formal and its corresponding actual must agree in type. If the formal and actual have different types, the transformation

```
      subroutine a                 subroutine a
      a's declarations             a's declarations
      a's data statements          new declarations for b
         ...                       b's declarations
10    if (b(x)) then               b's data statements
         ...                       a's data statements
                                      ...
                             10    begin body of b
                                   ret = return value for b
                                   goto 1
                             1     if ret then
                                      ...
```

Figure 3.13 An example of inlining a call site.

cannot be performed. A program that contains such a call site does not conform to the FORTRAN standard.

Nonetheless, few FORTRAN compilers enforce this restriction. We found such call sites in our study; others have reported similar results [CS85]. Some, quite obviously, had been carefully crafted to achieve specific behavior. For example, in euler, one call site passes an array of reals to a formal that is a complex array, relying on the standard's requirement that complex numbers must be implemented as pairs of reals.

2. Another class of problems arises when an actual parameter and its corresponding formal parameter are declared with different dimension information. These problems manifest themselves in two different ways: the number of elements in each dimension and the number of dimensions.

   The inliner requires that the sizes of the first $n-1$ dimensions specified in the callee be identical to the corresponding dimensions in the caller. While more complex mappings can conform to the storage, such remapping can introduce a level of complexity into all the subscript expressions.

   If the actual parameter has more dimensions than its corresponding formal, the inliner translates references to the formal into references to the actual in a straightforward manner. (This case is the classical FORTRAN dimension reduction at a call site.) If the actual parameter has fewer dimensions than the formal parameter, the references can still be translated. However, the resulting references can be substantially more complex than their original forms.

   In both cases that remap storage, changes to the size of an inner dimension or an increase in the number of dimensions, we disallow inlining. This decision stems from our caution about generating extra multiplies in subscript expressions as a result of inlining, and thus, potentially hurting program performance and analysis for optimization.

3. To allow for safe application of anchor pointing, the transformer does not inline any call site that appears in the second term or subsequent terms of a boolean expression. This optimization, also called a *short circuit* optimization, cuts short the evaluation of an expression as soon as its value has been fully determined. In the expression `(a .and. b)`, the compiler can avoid evaluating b if evaluating a yields *false*.

Strictly speaking, any program whose behavior changes under this transformation does not conform to the FORTRAN standard. Nevertheless, we felt that the inliner should preserve the original program's behavior in this case. To transform the source in a way that preserves its original behavior under inlining and anchor pointing requires the introduction of additional control flow operations. To simplify this situation, we elected to disallow inlining of any call except the leftmost in a boolean expression.

4. A call site that invokes a procedure-valued parameter cannot be inlined unless the compiler can, through interprocedural analysis, determine that the procedure variable has a single value across all invocations of the caller. If the variable has multiple values, the transformer cannot replace the call site with the body of any single procedure.

   Even with this restriction, the transformer handles the most common use of procedure-valued parameters in scientific FORTRAN programs. Programmers often pass into a procedure the name of another procedure that implements the mathematical function being manipulated. This simplifies applying the overall algorithm to different functions, but retains the property that, within a single compilation, the procedure-valued parameter has a single consistent value. In our experience, this is by far the most common use for procedure-valued parameters.

5. FORTRAN provides a mechanism to declare a variable static, the SAVE statement. The value of such a variable is preserved across invocations of the procedure in which it is declared. To preserve the correct behavior of these variables requires the introduction of a generated COMMON block in every instance of the procedure body. To date, we have not implemented this transformation.

As shown in Figure 3.2, even with these restrictions, we were able to eliminate most of the dynamically executed procedure calls.

3.6 Related Work

This section presents descriptions of numerous other implementations of inline substitution. Some of these include experimental results. The presentation is divided between inlining to reduce call overhead, and inlining to enhance optimization. Two distinct themes can be derived from this discussion. First, improvements due to inlining depend upon the programming language, the level of optimization in the compiler, and to some extent, the architecture of the target machine. Second, little experimental data is available to understand which optimizations are enabled by inlining.

3.6.1 Inlining to Reduce Call Overhead

For a long time, the primary goal of inline substitution was to reduce call overhead, the saving and restoring of state around procedure calls. The first study undertaken to understand the impact of inlining on call overhead was Scheifler's [Sch77]. He implemented inlining in a compiler for Clu, selecting call sites for inlining based on execution profiles, and restricting overall program growth to twice its original size. He observes that in data abstraction languages such as Clu, programs consist of many very short procedures. Moderate execution time improvements of 5 to 28 percent were demonstrated on 4 programs.

Hwu and Chang used a similar inlining strategy on C programs, considering not only the increased code size as the cost of inline substitution of a call site, but also the size of the control stack during execution [HC89].

Davidson and Holler also implemented source-to-source inlining for C programs, giving results for 4 different target compilers [DH88]. They report an average execution time improvement of 12 percent on 13 programs. However, the improvements range from a 60 percent improvement to a 9 percent degradation. They attribute the cases of increased execution time to be caused primarily by register pressure. The only variables allocated to registers by the observed C compilers are those explicitly designated by the programmer using register declarations. After inlining, the number of register declarations in a procedure may increase, in some cases exceeding the available number of registers.

In Holler's dissertation, she further investigates the effects of inlining on execution-time performance [Hol91]. By avoiding inlining when the number of register declarations will become too large, she still observes adverse effects of register allocation. Many compilers divide the burden of saving registers around procedure calls between the caller and the callee. Inlining moves the location of these saves and restores. It can potentially move them into a place in the program that is more frequently executed than their previous location, resulting in execution time degradation. Holler also provides extensive evidence to suggest that inlining does not usually adversely affect paging and caching.

3.6.2 Inlining to Improve Optimization

Much of the literature suggests that the value of inline substitution should be enhanced by subsequent global optimization. Hecht included inline substitution in an optimizing compiler for a structured programming language [Hec77]. A very restricted inlining strategy eliminated about 20 percent of the calls, with about a 2 percent decrease in the program intermediate representation. The general-purpose optimizer in the Experimental Compiling System at IBM used inline substitution as a key component in the implementation of an optimizing compiler [Har77b] [ACF+80].

For these two compilers, the kinds of improvements anticipated by inlining are not given. However, other work suggests that improvements will arise from (1) propagating constant-valued parameters through the body of the called procedure, (2) enabling code motion across the former call site, and (3) exposing more information to the register allocator [Bal79] [RG89a] [WZ85]. Unfortunately, very little experimental evidence has been published to prove these assumptions.

Prior to this study, only two others have analyzed through experimentation the impact of inlining on optimization. Richardson and Ganapathi's study on 5 Pascal programs demonstrated an average of 20 percent improvement [RG89a]. However, the compile times grew on average by a factor of five. They provide evidence to suggest that the performance improvement was primarily due to eliminating call overhead. Huson investigated inlining in Parafrase, an automatic parallelization system for FORTRAN [Hus82]. His results were mixed, with only a third of the procedures demonstrating any improvements. However, a single example yielded a speedup of 4 due to inlining. Huson's study was performed on the LINPACK library of subroutines. Recall that the linpackd program was the only one from our study that showed substantial improvement when inlining and parallelization were combined.

3.7 Chapter Summary

To close this chapter, we review the findings from the study that will be used in the rest of this dissertation. First, secondary effects of inlining mask the improvements due to elimination of call overhead and improved optimization. We have suggested ways the compiler can avoid secondary effects in some cases. Secondly, the impact of code growth on execution time should not be a tremendous concern. However, the adverse impact of procedure size on compile time suggests that limiting procedure growth is indicated.

Based on the results of this study, one could conclude that inlining is not a worthwhile optimization for FORTRAN compilers. At least for the scalar compilers, the benefits were not significant enough to make the costs tolerable. The potential for performance degradation is also discouraging. However, if it were possible to restict inlining to high-payoff optimizations, the significant improvements from the optimizations would make up for any degradations associated with inlining. An example of such optimizations is parallelization. The improvements in execution time of the parallelized versions of linpackd due to inlining are an indication that inlining can significantly improve parallelization when applied to appropriate call sites. In the next chapter, we give a strategy for inlining that is designed to enable memory optimizations. This approach is similar to what would be used to enable parallelization.

Chapter 4

Goal-Directed Interprocedural Optimization

During the study in Chapter 3, we observed that execution time improvements due to inlining were small, with occasional performance degradations. The failure of inlining to improve the code suggests that either (1) secondary effects mask the true improvements, or (2) the negative effects of procedure calls on code optimization are not substantial.

We were interested in determining which of these was the case. If secondary effects were masking improvements, then other interprocedural techniques might still be effective. Improved constant propagation has been suggested as the most important effect of inlining [Bal79] [WZ89]. To test this, we performed an experiment to isolate the effects of interprocedural constant propagation. We also studied improvements when constants information is further refined by cloning. The experiment, described in the next section, proved that interprocedural constants are reasonably important — yielding greater improvements than occurred with inlining. However, further improvements due to cloning were not very significant.

The failure of inlining and cloning to yield improvements prompted the following question: when would the improvements due to inlining and cloning be significant enough to outweigh the costs and any degradations caused by secondary effects? We discovered the answer following a second experiment, described in Section 4.2. The second experiment was in response to a challenge to improve performance of a benchmark program using any known optimization techniques. The optimizations had to be possible on an automatic system. Inlining and cloning turned out to be critical to improving the program; they were needed to enable high-payoff optimizations. These high-payoff optimizations required surrounding context to be applicable, and the interprocedural transformations were used to provide the needed context.

We call the use of interprocedural transformations to enable certain high-payoff transformations *goal-directed interprocedural optimization*. This chapter presents a goal-directed approach to inlining and cloning. Since the two experiments were critical in forming this strategy, the chapter begins with a description of the experiments in the next two sections. Section 4.3 presents an overview of the strategy. Analysis for the cloning portion is described in Section 4.4, and the analysis for the inlining portion is described in Section 4.5. Related work is described in Section 4.6, and Section 4.7 concludes the chapter with a summary.

4.1 Constant Propagation and Cloning Experiment

We performed the experiment depicted in Figure 4.1 to assess the benefits of interprocedural constant propagation and cloning based on interprocedural constants. By hand, we applied interprocedural constant propagation to the 8 programs from the inlining study. The constants were located using the "Pass Through" method improved by side-effect information, as described in [CCKT86].

After applying constant propagation, we looked for procedures invoked from multiple call sites in the program. We examined values of global variables and actual parameters at each call site. Whenever a unique set of constants appeared at a call site, we formed a clone of the called procedure and propagated the newly exposed constants to the cloned version. The result was a maximal cloning of the program based on interprocedural constants information.

Cloning was possible in only 5 of the 8 programs. The other 3 programs were eliminated from consideration. For the remaining 5 programs, we examined the effects of constant propagation and cloning on both code size and execution time.

As a simple measure of the changes in code size, we counted number of non-comment lines in the text size for the original program, the program after constant propagation, and the cloned version. The changes in program size are given in Figure 4.2. As a result of constant propagation, the program size is significantly

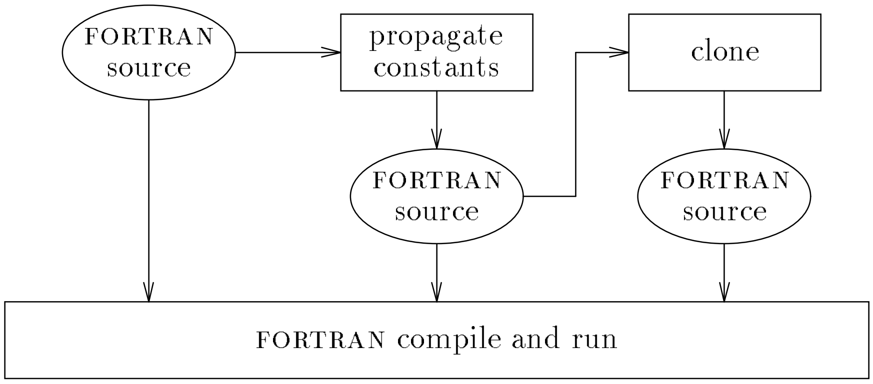


Figure 4.1 Structure of cloning and constants experiment.

reduced; on average it is 23 percent smaller than the original program. After cloning, in only two cases the programs are larger than their original size, and on average the programs are 14 percent smaller. Even the difference in size between the constant propagated version and the cloned version, represented by the column labelled δ, is small. On average, this difference is 11 percent. These results suggest that even with the maximal amount of cloning, growth in program size is manageable.

We also measured execution times for the original program, the constant propagated program and the cloned program. The measurements were gathered on the MIPS 120/5 compiler, version 2.1. Effects on execution time for the program version with constant propagation and the version with constant propagation plus cloning are shown in Figure 4.3. The numbers shown are percentages of the execution time of the original program. All execution times result from full global optimization[11].

The results in Figure 4.3 show that fairly significant improvements are possible with constant propagation – nearly 9 percent in two cases, and overall an average of 4.8 percent improvement. However, the results for cloning do not demonstrate a significant improvement over constant propagation. Only two cases demonstrated noticeable improvement due to cloning – cedeta and linpackd. It turns out that these two programs gained certain "important" constants from cloning. When we explain

	constants	cloning + constants	δ
cedeta	-4	5	9
euler	-52	-51	11
linpackd	-24	-5	3
wanal1	-4	6	8
wave	-29	-23	24
avg	-23	-14	11

Figure 4.2 Percentage change in text size after constant propagation, and after cloning.

[11]The MIPS compiler performs interprocedural register allocation at its highest level of optimization [Cho88]. The result of increased constants information appeared to interact poorly with interprocedural register allocation. We speculate that simplification of inner loop bodies may have permitted loop unrolling in a few cases. Since loop unrolling requires additional registers for addresses and floating point values, leaf procedures were using extra registers, causing register pressure in procedures higher in the call multigraph. The results reported in the table use the second highest optimization level, which does not perform interprocedural register allocation.

	constants	cloning + constants
cedeta	8.8	10.4
euler	8.9	8.7
linpackd	0.3	1.6
wanal1	0.0	0.0
wave	5.9	6.0
avg	4.8	5.3

Figure 4.3 Execution-time improvements due to constant propagation and cloning.

what constitutes an important constant later in the chapter, we will return to this experiment.

4.2 Optimizing matrix300

Colleagues at Rice have been demonstrating impressive results with optimizations designed to improve memory hierarchy performance [CCK90]. In particular, *strip mining* is used to improve cache blocking, *scalar replacement* is used to place array values in registers, and *unroll and jam* is used to adjust register pressure and loop balance. In response to this work, a researcher at SUN Microsystems became interested in the effectiveness of these techniques on the SPARC microprocessor. He challenged us to apply these techniques to the SPEC benchmark matrix300.

Using a combination of interprocedural transformations and memory-management transformations, we were able to achieve substantial improvements on both the SPARC and the MIPS[12]. The success of the experiment suggests a promising approach to interprocedural optimization: interprocedural transformations are useful in enabling important high-payoff intraprocedural optimizations. This section describes the experiment with matrix300.

4.2.1 Structure of the Program

The call multigraph of matrix300 is shown in Figure 4.4. The bulk of the computation is carried out in *dgemm* and its components.

- *dgemm* computes a matrix-matrix product, using *dgemv.*

[12]This experiment was done with Preston Briggs.

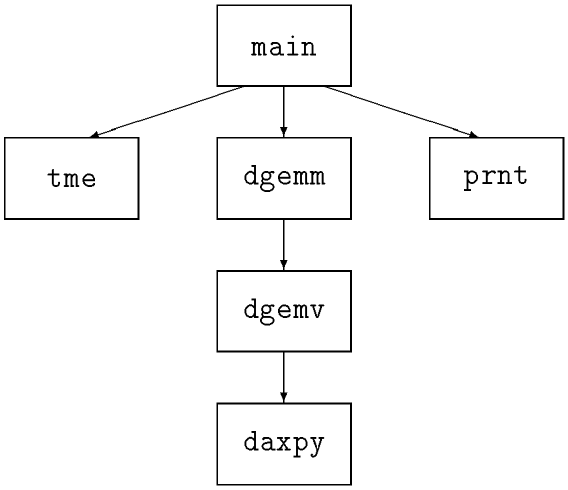


Figure 4.4 Original call multigraph for matrix300.

- *dgemv* computes a matrix-vector product, using *daxpy*.
- *daxpy* computes $\vec{y} \leftarrow \vec{y} + \alpha\vec{x}$.

All of the floating-point computations are actually performed in *daxpy*. Its form is very simple:[13]

```
subroutine daxpy(m, x, A, ia, Y, iy)
    dimension A(ia, m), Y(iy, m)
    do i = 1, m
        Y(1, i) = Y(1, i) + x * A(1, i)
    enddo
end /* daxpy */
```

The unusual form of the subscripts is simply a notational convenience, allowing the original authors a handy way of expressing access to non-unit stride vectors.

4.2.2 Improving Memory Performance

Examining the code for *daxpy*, we notice that each iteration of the loop contains three memory accesses and two floating point computations. Since floating point operations take less time than memory accesses, most of the execution time for the loop will be spent waiting on the memory accesses. Also, because each iteration crosses an entire column of the arrays X and Y, the loop will probably generate a large number of cache misses.

[13]Note that some variables have been renamed for clarity. Further, the BLAS routines were renamed to emphasize that all floating-point computations are double-precision.

We would like to apply optimizations to reduce the number of memory accesses in the loop. The key to avoiding memory accesses is recognizing *reuse* [CCK90]. When values are used multiple times, they can be kept in registers (or cache). By exposing reuse, accesses to memory are converted to register or cache accesses, which are much faster. Techniques for exposing reuse include *scalar replacement* and *unroll and jam*.

Unfortunately, *daxpy* offers no opportunities for scalar replacement; that is, there is no reuse of any of the array locations and therefore no profit in allocating array elements to registers. Further, the absence of reuse suggests that cache will be ineffective. Unroll and jam is often suitable for exposing opportunities for scalar replacement; however, to perform unroll and jam requires nested loops, and *daxpy* contains only a single loop. Examining *dgemv* below, we notice that the call to *daxpy* is contained inside a loop. This suggests that inlining *daxpy* would create a doubly-nested loop, suitable for transforming with unroll and jam.

```
subroutine dgemv(m, n, A, ia, X, ix, Y, iy, job)
    dimension A(ia, n), X(ix, n), Y(iy, n)
    if ((iabs(job) - 1) / 2) = 0 then
        ii = ia
        ij = 1
    else
        ii = 1
        ij = ia
    endif
    do j = 1, m
        y(1,j) = zero
    enddo
    do j = 1, n
        k = 1 + (j - 1) * ij
        call daxpy(m, X(1, j), A(k, 1), ii, Y(1, 1), iy)
    enddo
end /* dgemv */
```

Inlining *daxpy* involves an ugly array reshape of A. Difficulty arises because the variable ii is passed to formal ia in *daxpy*, representing the leading dimension size in A's declaration. If the compiler were to automatically perform the translation and substitute for k, the resulting reference to `A(1,i)` in *daxpy* would be `A(1+(j-1)*ij+(i-1)*ii,1)`. This subscript expression is complex enough to defy the *dependence analysis* [Kuc78] on which scalar replacement and unroll and jam rely.

The value of ii depends on the parameter job so we trace backwards through the program to locate the value of job. In *dgemm*, the value of job passed into *dgemv*

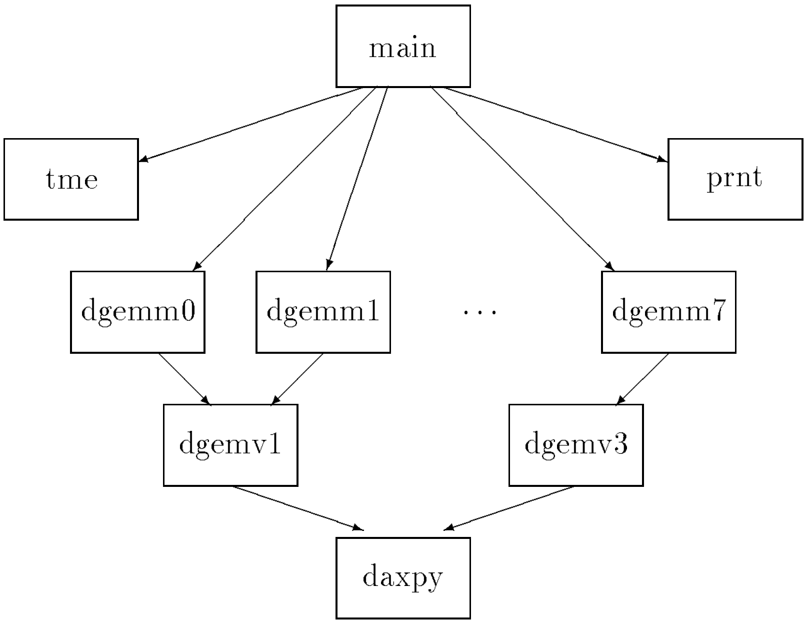


Figure 4.5 Call multigraph for matrix300 after cloning.

depends on the value of the parameter *jtrpos*. Each of the eight call sites in the main program binds a different constant value to *jtrpos* when calling *dgemm*. By cloning the calls to *dgemm* from *main*, the constant value of *jtrpos* is exposed for each clone.

After cloning, we perform constant propagation within each clone to determine the value of *job* at each call to *dgemv*. Unfortunately, in four of the clones, *dgemv* is called with a value of 1 for *job*; in the other four clones, *job* has the value 3. Again, we can use procedure cloning – this time two copies of *dgemv* are created (*dgemv1* and *dgemv3*), for the two values of *job*. The final call multigraph is illustrated in Figure 4.5.

Performing constant propagation on the clones of *dgemv* determines the value of *ii* and *ij* and allows *daxpy* to be inlined cleanly. With some simplification, we arrive at the following loop nest for *dgemv3*:

```
do j = 1, n
    do i = 1, m
        Y(1, i) = Y(1, i) + X(1, j) * A(j, i)
    enddo
enddo
```

For *dgemv1*, the results are similar.

At this point we are finally in a position to make use of the transformations described in [CCK88]. We perform *loop interchange, loop fusion,*[14] *unroll and jam,* and *scalar replacement.* The transformed source of *dgemv3* is shown below:

```
do i=1,m,10
    y0 = zero
    y1 = zero
    ...
    y9 = zero
    do j = 1, n
        x0 = X(1, j)
        y0 = y0 + x0 * A(j, i+0)
        y1 = y1 + x0 * A(j, i+1)
        ...
        y9 = y9 + x0 * A(j, i+9)
    enddo
    Y(1, i+0) = y0
    Y(1, i+1) = y1
    ...
    Y(1, i+9) = y9
enddo
```

4.2.3 Experimental Results

The table below summarizes the results of our experiment. In addition to the relevant timings, we have included the object code sizes for both versions of the program.

	compiler	*original*		*modified*		*execution*
machine	*options*	(bytes)	(seconds)	(bytes)	(seconds)	*improvement*
Sparcstation 1	`-O4`	122,880	466	122,880	229	2.0×
MIPS M-120	`-O3`	114,464	755	112,976	229	3.3×

The improvements on each machine are quite significant. There are several sources of improvement:

Memory: In the original version of *dgemv*, there were mn fetches from A, n fetches from X, mn fetches from Y, and $mn + m$ stores into Y. The improved code avoids much memory traffic by holding values in registers for reuse (the variables $y0$, $\ldots$, $y9$, and $x0$). Overall, *dgemv3* requires mn fetches from A, $mn/10$ fetches from X, and m stores into Y; there are *no* fetches from Y. Briefly, the improvements avoid nearly 100% of the stores and 50% of the loads. In *matrix300*, we save about 216 million stores and nearly as many loads.[15]

---

[14] Fusion is performed with an earlier loop that initializes the first row of Y to 0.

[15] *dgemm* is invoked 8 times, on 300×300 matrices.

Scheduling: In *dgemv3*, each of the multiply-accumulate statements can potentially be executed in parallel. Both compilers take advantage of this freedom when scheduling. The original code in *daxpy* is less amenable to scheduling improvements, even after unrolling, because of the store to Y. Without dependence analysis, the optimizers do not recognize the lack of dependence between successive stores and loads to Y.

Inlining and unrolling: The improved code has fewer procedure calls and less loop overhead. However, it should be noted that the call overhead for 720,000 calls is insignificant when compared against the improvements due to the elimination of almost 500,000,000 memory accesses.

Further improvements seem possible. In particular, further inlining would provide additional opportunities for loop restructuring to exploit reuse, possibly leading to better cache behavior.

4.3 A Strategy for Interprocedural Optimization

Our previous experience has shown that, in many cases, secondary effects in the compiler and optimizer can obscure any improvement derived from cloning or inlining. The results of the experiment with matrix300 demonstrate that cloning and inlining are worthwhile when they enable high-payoff optimizations. This suggests a new strategy for performing these interprocedural transformations: use cloning and inlining only when they can enable application of a high-payoff transformation.

The high-payoff memory-management transformations are characteristically different from the types of optimizations performed by typical global optimizers (like the commercial compilers in the inlining study and the cloning experiment from Section 4.2). For one thing, the high-payoff optimizations require a great deal of surrounding context to be applicable. Additionally, significant improvements are possible from a single application of a high-payoff optimization.

These two properties are in contrast with the low-level optimizations performed by typical global optimizers. Low-level optimizations are often applicable and require little surrounding context, but they achieve only small improvements for each application. Thus, a characterization of our optimization strategy is to use interprocedural transformations in a restricted way only for high returns, rather than widespread use aimed at accumulating small improvements. Inlining and cloning are used in the following ways:

- **Clone to expose additional interprocedural constants.**
  Exposing additional constants can simplify control flow and subscript expressions. It also can provide bindings for parameters used as array dimension sizes. As a consequence, cloning can improve the results of inlining.
- **Inline to expose loop nests.**
  The loop nests provide the adequate context to perform high-payoff optimizations.

In this section, we develop the insight from the matrix300 experiment into strategies for inlining and cloning. At the end of the section, we present an algorithm for goal-directed interprocedural optimization. This section only provides an overview of the algorithm, with details deferred until later in the chapter.

4.3.1 Cloning Strategy

By creating copies of a procedure for distinct sets of contant parameter values, cloning exposed constants that enabled optimization. The program was improved in three ways. First, control flow was simplified. Second, constants in subscript expressions simplified dependence analysis. Finally, constants passed as array dimensions resulted in less complicated subscript expressions. This made inlining possible and improved the results of dependence analysis.

Based on our experience, certain types of constants are important to optimization. If we want to avoid unnecessary cloning, we can just form clones from calls that expose "important" constants. Also, since cloning exposes constants used as array dimensions, it should be performed before any inlining. These insights led to the following approach for cloning:

1. discover "important" constants.
2. in topological order, visit each procedure p:
   (a) partition calls to p by values of the important constants.
   (b) propagate newly exposed constants through procedure bodies.

There are a couple of important points in this approach. First, we visit the procedures in topological order. This is because cloning a procedure can change the optimization of its descendants in the call multigraph. Recall from matrix300 the impact of cloning *dgemm*. Once the value of its parameter *jtrpos* became constant, we were able to determine the value of *job* in *dgemv*. As a result, it became possible to clone *dgemv*. By cloning *dgemv*, the value of the variable *ii* became known, which in turn made possible inlining the call to *daxpy*. This suggests a topological order for cloning, so that a procedure is visited prior to its descendants. Further, it suggests that interprocedural constants be incrementally updated to take advantage of newly exposed constants coming from ancestor procedures.

Newly exposed constants are also propagated through the procedure bodies for the cloned versions. In matrix300, this was needed when cloning *dgemm*. After cloning, the constant value of *jtrpos* was applied to expressions that determine the control flow within *dgemm* and the value of *job*. This, in turn, affected cloning of *dgemv*.[16]

Given the above algorithm, the only problem remaining is how to determine which constants are important. In our experiment with matrix300, the critical constants fit into one of the following three categories:

1. They determine control flow.
2. They determine the value of a parameter used as an array dimension size.
3. They contribute to the value of a subscript expression.

There are other constants that are not useful in optimization. An excellent example is the set of string constants passed to an error message routine. Even constants used as terms in an expression (such as a multiplier) do not usually improve execution time.

We gained some of the insight about what constants are important from the experiment in Section 4.1. As a result of cloning, many of the programs gained constants that appeared as terms in expressions. These turned out to have insignificant effects on execution time. However, on cedeta and linpackd, where execution time was noticeably improved after cloning, the constants exposed fit into one of the above three categories. On linpackd, the only constant exposed was an array dimension size. After cloning, the address calculations for this particular array were greatly simplified and required fewer registers. On cedeta, constants were exposed that determined control flow through two critical procedures. The results from the experiment described in 4.2 give further evidence to the importance of these types of constants, and demonstrate their usefulness for a variety of optimizations.

[16]We would like to avoid making this extra pass in which newly exposed constants are propagated over a cloned procedure body. This would require that enough information be recorded during local analysis of a procedure to be able to determine values of expressions dependent upon incoming constants. Recorded functions that are based on unknown parameter values are called *jump functions* [CCKT86]. An efficient and useful representation of jump functions for a particular problem is difficult and is not addressed in this dissertation.

To be able to partition based on "important" constants, we first need to identify all formal parameters and global variables in a procedure that, if constant, would perform one of the above three functions. For a procedure p, we call this set of variables *CloningVars(p)*. Calculating *CloningVars(p)* for each procedure is formulated as an interprocedural problem. Within p, globals and formals are located that are used either directly or through a sequence of assignments to determine control flow or subscript information within a procedure. After the preliminary local analysis, this information is propagated backward through the call multigraph, mapping formal parameters in *CloningVars(p)* to the variables that determine their values in callers of p. The final set *CloningVars(p)* contains the formals and globals that affect control flow and subscripts somewhere in the program — either in p or in one of its descendants. The calculation of $CloningVars$ is presented in Section 4.4.

4.3.2 Inlining Strategy

Once constants used as a array dimension sizes have been exposed by cloning, the compiler can proceed to select call sites for inlining. The goal for inlining is simple — we want to inline call sites in a way that enables application of the memory-management optimizations. Memory-management optimizations are used to adjust *balance* [CCK88]. A machine's balance β_M is the number of floating point operations that can be performed in the time it takes for a single memory access. The balance of a loop β_l is roughly the ratio of memory accesses to floating point operations in the loop. If for some loop l, $\beta_l > \beta_M$, then during loop execution, there are waits on memory accesses (a *memory-bound loop*).

The memory-management transformations can improve memory-bound loops by exposing reuse. In this way, fewer of the variable references actually require accesses to memory. For memory-bound loops, exposing reuse increases the extent to which memory accesses and floating point operations are overlapped, thus reducing overall program execution time. (For compute-bound loops, the memory-management transformations will effect little execution-time improvement, so these loops will be ignored.)

The inlining strategy is to produce loop nests of two or more loops. This is necessary only when we have an inner loop that is memory-bound. The inlining strategy is summarized as follows:

1. for each inner loop l, compute β_l.
2. select loop l in reverse topological order:
   (a) if l is memory-bound and the only loop in a leaf procedure, attempt to inline into caller.
   (b) otherwise, if l is memory-bound, and contains call sites, consider inlining to eliminate all calls in loop.
      i. assuming all inlining is legal, perform inlining.
      ii. recalculate β_l and goto $2a$.

The algorithm produces loop nests around memory-bound inner loops in two ways. First, it inlines leaf procedures containing a single loop into their callers. If the call site appears inside a loop, the result is a loop nest of two or more loops. For inner loops in non-leaf procedures, the algorithm attempts to inline all the call sites in the inner loop. This creates an inner loop with no calls, and possibly a loop nest with no calls. Neither approach necessarily produces a loop nest, but both tactics enable inlining other call sites invoking the inlined version. Even if an inlining decision does not produce a loop nest to which we can apply the memory-management transformations, inlining may expose reuse as a result of renaming from formal to actual parameters.

The algorithm specifically targets leaf procedures with single loops and inlines inner loops in callers because these are the most likely to benefit from memory optimization. For the best results with the memory-management transformations, we want loops with no procedure calls and would prefer *perfect* loop nests (i.e., loops made of a loop body surrounded by multiple loop headers).

Call sites are considered for inlining in reverse topological order. Recall from Chapter 3 that reverse topological order guarantees that the minimal amount of inlining is performed by the compiler. It also makes it possible to inline a single-loop procedure into its caller, and subsequently evaluate the caller as a target for inlining based on a new value for the loop's balance.

From an implementation perspective, this algorithm requires a formulation of loop balance. Because balance is considered for loops containing procedure calls, an interprocedural formulation is required. These topics are covered in Section 4.5.

4.3.3 Algorithm for Goal-Directed Interprocedural Optimization

Combining the techniques in the previous two sections gives us the algorithm shown in Figure 4.6. To complete the algorithm description, a few clarifications are needed. The cloning phase examines constants at both call sites and procedures. Interprocedural

Phase 1: Cloning

1. Compute $CloningVars(p)$ for each procedure.
2. In topological order, visit each procedure p:
   (a) Partition calls c_i to p by $(CloningVars(p) \cap \textsc{constant}(c_i))$
   (b) Create a cloned version for each partition.
   (c) For each cloned version cv:
      i. Propagate $\textsc{constant}(cv)$ through cv.
      ii. Update $\textsc{constant}(c_i)$ for call sites c_i in cv.

Phase 2: Inlining

1. For each inner loop l, compute β_l.
2. Visit call sites c in reverse topological order:
   (a) If c invokes a leaf procedure that contains a single memory-bound inner loop l, attempt to inline c.
   (b) If c invokes a non-leaf procedure containing a memory-bound inner loop l,
      i. Evaluate inlining of all call sites in l.
      ii. If all inlining legal, perform inlining.
      iii. Reevaluate β_l and goto $2a$.

Figure 4.6 Goal-directed interprocedural optimization strategy.

---

constants, as defined in Chapter 6, exist only for procedures. The constants at call sites in procedure p include $\textsc{constant}(p)$ and any constants initiated in the procedure, as long as they are not modified on any path to the call site. In both cases, the constants appearing as actuals at the call site must be mapped to the corresponding formals in the called procedure.

As a second point, the intersection in step 2a of the cloning phase should be explained. The set elements of $\textsc{constant}(c_i)$ are $\langle variable, value\rangle$ pairs. When the intersection is performed, the variables from these pairs are extracted and compared with the variables in $CloningVars(p)$.

4.4 Calculating *CloningVars*

The set *CloningVars(p)* for some procedure p contains the formals and global variables that may either affect control flow or be used for dimensions or subscript expressions. These variables are used in one of these ways either in the procedure, or in one of its descendants in the call multigraph. Once interprocedural constants have been calculated, the set of *CloningVars* can be compared against the variables with constant values to determine if cloning a particular call will expose a constant value for a variable in *CloningVars*.

Calculating *CloningVars* can be formulated as a backward interprocedural problem. As with previous interprocedural analysis problems, calculation of *CloningVars* occurs in two phases. The local analysis builds a list of the globals and formals that affect control flow, are used in subscript expressions, or are passed as actual parameters at some call site in the procedure. The ones used as actual parameters are associated with the corresponding formal parameters of the called procedure. This information is required by the propagation phase, which maps variables in $CloningVars(p)$ to variables that determine their values at call sites invoking p.

4.4.1 Phase 1: Local Analysis

The approach for local analysis is inspired by Ball's work [Bal79]. To determine the benefits of constant propagation on a procedure (as a result of inlining), he calculates the *strong dependency set* for each statement. For a given statement, the strong dependency set consists of those variables that, if constant, would determine the value of the statement. We would like something similar to the strong dependency sets for the important program points – control flow decisions, subscript expressions and actual parameters. However, Ball's formulation is too pessimistic. Whenever paths merge in the control flow graph and two definitions of a variable reach a statement, the strong dependency set for the statement is assumed to be $\perp$.

Even when control flow paths merge, a variable can still be constant. The following code fragment illustrates this point:

```
S1: x = ...
S2: if (c) then
S3:     x = ...
    endif
    ...
S4: if (x) then
    ...
```

In the example, x is a local variable of the procedure. Since x is used in the conditional at S4, we are interested in what globals and formals determine the value of x. There are two definitions of x that reach S4: at S1 and S3. There are several ways in which constant propagation could determine the value of x.

If S1 and S3 assign the same constant value to x, then x is constant. Otherwise, if we can determine that the condition at S2 will evaluate to *false*, then x is constant if constant at S1. Similarly, if the branch at S2 always evaluates to *true*, then x is constant if constant at S3. Knowledge about constants at any two of S1, S2 and S3 may be adequate to determine the value of the branch at S4. Thus, the value of a variable is dependent both on its definitions and on the control flow decisions that guard the definitions.

This motivates a simple formulation of local analysis for $CloningVars$, which may perform cloning when unnecessary but does not miss any important opportunities. The basic idea is to propagate globals and formals to their uses to derive the list of globals and formal parameters that determine the values of expressions at important program points. To perform this propagation, we take the union of the sets for each definition reaching a program point. This provides all of the possible variables contributing to the value of the definitions of an expression. Variables determining the value of the guards on the assignments will also be discovered since all guards are considered important program points.

Local analysis is based on a *static single assignment* (SSA) representation of the source [CFR+89]. To represent a program in SSA form, it is transformed so that only one definition reaches a use. Definitions are given unique names, and each use of a variable is renamed to correspond to the definition that reaches it. Wherever two definitions reach a use, a new statement is inserted where the control flow paths merge. The new statements, called ϕ-functions, are definitions. The left-hand side is a unique name for the definition, and the right-hand side is essentially a list of the names for the reaching definitions. Edges in the SSA graph connect the definition of a variable to its uses, to allow direct propagation of information from definition to use.

SSA form is used in constant propagation because the space requirement is much less than traditional data-flow techniques on the control flow graph, and it has a better expected time bound [WZ89] [CFR+89]. We use the SSA representation for this reason, but also because SSA form is already constructed in ParaScope to perform local analysis for constant propagation.

Algorithm. The primary goal of the local analysis algorithm is to compute the portion of *CloningVars* used directly in this procedure. For each node n in the SSA graph, the algorithm constructs a variable set V_n. This is the set of variables that, if constant, could determine the value of the node. Since nodes correspond to definitions and uses of variables, a variable set will exist for assignment statements, conditionals and subscript expressions. Upon termination, the variable sets for important expressions determine the value of *CloningVars*.

The algorithm also builds annotations A_c^i for the i^{th} actual parameter at call site c, which provides the set of variables determining the value of the actual parameter. These are used in translating formal parameter members of *CloningVars(p)*, for some procedure p, to the appropriate variables in p's callers during the propagation phase.

The algorithm propagates variables that could be constant on entry — formal parameters and globals that appear in the procedure body — to their uses in the procedure. It begins by propagating these variables to their uses along paths where they are not modified. Whenever variable sets for all operands on the right-hand side of an assignment become available, the union of the variable sets is propagated to uses of the defined variable. When multiple definitions of a variable reach a use (i.e., at ϕ nodes), the use inherits the union of variable sets for its definitions.

To simplify the presentation of the algorithm, assume that constants have already been propagated and variables proven constant have been replaced by their constant values. Assume also that formals and globals that appear in the procedure body are represented by dummy assignments at the beginning of the code. (Note that we are only interested in scalar variables.) These assignments are treated as definitions of the formals and globals, and edges from these nodes to uses of initial values of the variables are integrated into the SSA representation.

For a dummy node n_d, the set V_{n_d} contains only the formal or global that the dummy assignment represents. Nodes are also created for constant-valued expressions n_c, with variable sets V_{n_c} equal to $\emptyset$. All other nodes n have V_n initialized to $\top$, indicating that the node's variable set has not yet been determined. The value of *CloningVars* for this procedure is initially $\emptyset$. All annotations for actuals at call sites in the procedure are also initialized to $\emptyset$. The $Worklist$ is initialized to contain outgoing edges from the dummy statements.

Until the $Worklist$ becomes empty, the following steps are performed:

1. Select an SSA edge from the $Worklist$. The target for the edge is node n appearing in some sort of expression e, possibly a ϕ-node. The variable set V_e can be determined as follows:
   (a) if e is a ϕ-node, we form the union of the sets V_{n_e} for all the definitions making up the ϕ-node, including n.
   (b) if e is some other expression, we form the union of the sets V_{n_e} for the definitions of each of the operands in the expression, including n. However, in this case, all operands in the expression must have some set value defined for them other than $\top$. An expression is not processed until all of its operands have defined set values.
2. If the e is the right-hand side of an assignment statement, as soon as V_e has a value other than $\top$, V_e can be propagated to uses of the defined variable. If V_e has changed, all SSA edges emanating from the defined variable are added to $Worklist$.
3. If e determines control flow, or is part of a subscript expression (including step sizes), we union the set V_e with *CloningVars*.
4. If e is an actual parameter, we add V_e to the annotation for the actual at the call.

The difference between the treatment of ϕ-nodes and other expressions is simple. To be able to determine the value of an expression, there must be at least one definition of each of its operands for which we can determine the value of the operand. In the case of ϕ-nodes, there is a chance that the value will be constant if the value of only one of the definitions can be determined.

Time Complexity. Constant propagation on the SSA graph has a time bound of O(EV), where E is the number of edges in the control flow graph and V is the number of variables in the procedure. However, the expected time bound is linear in the size of the SSA graph [WZ89].

The algorithm for calculating initial *CloningVars* is more expensive than this because the set values associated with a node can change more than twice. The number of elements in a set V_s is bounded by the number of scalar formal parameters and global variables appearing in the procedure. Thus, the set values can change once for every possible element in the set. We expect this number to be small, and to not grow with the procedure size. This is consistent with assumptions made in other interprocedural analysis algorithms [CK88b] [CK89].

4.4.2 Phase 2: Propagation

The propagation phase adds to *CloningVars(p)* any variables of p that can be used at important program points in p's successors in the call multigraph. This requires a function which maps variables in *CloningVars(s)* to variables in p, for some successor s of p. This mapping relies on the annotations A_c^i for the actual parameters at the call to s, calculated in the local analysis phase. In the mapping function, assume p invokes s at call site c. Then for a variable v in s, the mapping function is the following:

$$map_c(v) = \begin{cases} \{v\} & \text{if } v \text{ is a global variable} \\ A_c^i & \text{if } v \text{ is the } ith \text{ formal parameter of } p \end{cases}$$

Assume that the definition of $map_c(S)$ for some set of variables S is just the union of the sets $map_c(v_i)$ for each variable $v_i \in S$. Then the value of $CloningVars(p)$ can be determined by the following set of simultaneous equations:

$$\begin{aligned} &CloningVars(n) = \\ &\qquad CloningVars(p) \;\cup\; \textstyle\bigcup_{c=(p\rightarrow s)} map_c(CloningVars(s)) \end{aligned}$$

4.5 Estimating Loop Balance

As a simple measure of loop balance, we can count the number of floating point computations and memory accesses in the loop. Any variable reference is considered a memory access, thus implicitly assuming that no variables remain in registers or cache across accesses. This measure is not very accurate in a compiler that attempts to expose reuse. However, if the measure indicates a loop is compute-bound, it would definitely be compute-bound when variables were reused in registers or cache. Thus, it allows us to ignore the loops that it considers compute-bound.

To take reuse into account in measuring loop balance requires dependence information [CCK90]. A dependence suggests the occurrence of two accesses to the same memory location. Certain types of dependences indicate that two accesses refer to the same value – two reads, or a write followed by a read. Dependence analysis has a quadratic time bound, and because it is potentially expensive, it is rarely found in scalar optimizing compilers. Nevertheless, dependence analysis is critical to the memory-management transformations. Because of the potential for dramatic performance improvements – speedups of 2 and 3 in our experiment – the expense of dependence analysis is warranted. The issue of incorporating dependence analysis into the program compiler is addressed in Chapter 7.

Assuming we can use dependence analysis to locate opportunities for reuse, we have one remaining problem: our inlining algorithm requires that we estimate balance in loops with call sites. The precision of dependence analysis makes it intractable across procedure boundaries [Mye81]. As a compromise, reuse is only considered within a procedure. Memory accesses crossing procedure boundaries are considered to be independent.

This approach is reasonable since reuse across procedure boundaries, even if detected, would be difficult for a compiler to exploit. Inlining alone may expose reuse across procedure boundaries, without even applying the memory management transformations. After inlining, a better estimate of loop balance can be made when the code representing a call site is in place.

The balance calculation is formulated as a backward interprocedural problem. In a leaf procedure, the number of floating point operations and memory accesses are counted, taking reuse into account. These counts are also determined for any loops contained in the procedure. For a non-leaf procedure, the estimate of floating point operations is the sum of operations occurring within the procedure and the estimate of floating point operations for each procedure it invokes. The estimate of memory accesses begins with accesses within the procedure, taking reuse into account. Added to this are estimates of memory accesses for each procedure it invokes.

4.6 Related Work

For the most part, the primary goal of the inlining strategies presented in Chapter 3 is to eliminate procedure call overhead. In general, these inlining strategies are driven by heuristics weighted toward frequently executed procedures, while attempting to control program growth:

- Hecht's SIMPL-T compiler only inlines procedures called once with a single entry and exit [Hec77].
- Scheifler substitutes any procedure not resulting in an increase in code size, and then procedures with the highest ratio of expected number of executions to net increase in code size [Sch77]. He restricts the final program size to twice its original size. Execution frequency estimates are obtained from run-time measurements of the program on sample input data.

- Hwu and Chang use a similar approach, considering not only the increased code size as the cost of inlining, but also the size of the control stack during execution [HC89].
- Davidson and Holler substitute called procedures when their caller appears in the same source file [DH88]. They avoid inlining for recursion and when register declarations would exceed the number available to the compiler. In her dissertation, a model for predicting the improvement to a call site due to inlining is presented [Hol91]. The model considers the following costs: saving and restoring registers, passing parameters, local stack adjustment, parameter stack adjustment and the call/return sequence.

Heuristics focusing on eliminating call overhead work reasonably well for non-optimizing compilers [Sch77] [DH88]. However, when compilers perform optimization and register allocation, the interaction with inlining can reduce the importance of call overhead. What is really significant is the amount of optimization enabled by inlining. Only Ball and Cooper select call sites for inlining based on the optimizations that will result [Bal79] [Coo83].

Ball formulates improvement estimates resulting from inline substitution as a data-flow problem [Bal79]. The strong dependency sets, as described in section 4.4, provide the set of formal parameters whose values determine a statement's value. Given a set of constant parameters, the compiler can examine the strong dependency sets for the statements in the procedure and evaluate the impact that inlining would have on the procedure body. This method estimates decreases in code size and execution time as a result of constant propagation and test elision.

Building on Ball's work, Cooper presents an algorithm for *linkage tailoring*, assigning linkages to call sites [Coo83]. The possible linkage styles include a default linkage where separate compilation is improved by interprocedural information, inlining, cloning and some variants. He uses a similar estimate of code improvement to determine how the constant value of a global or parameter exposed by inlining or cloning can affect optimization. (He also suggests that these estimates could be used to determine what constants might make cloning worthwhile.)

For each procedure, the benefits of optimization and the costs of each linkage choice (in terms of code growth) are calculated. The linkage tailoring algorithm assigns linkages to call sites in order of choices with the highest *benefit/cost* ratio. At call sites where costs are negative, inlining is always performed. The algorithm stops when program growth exceeds some constraint, and remaining call sites are assigned the default linkage.

Both Cooper and Ball developed estimates that attempt to capture the improvements due to inlining. In light of the results of the inlining study, it seems unlikely that such estimates can predict good choices for inlining. This is because these simplified models do not capture the complex interactions inlining has with other aspects of compilation.

4.7 Chapter Summary

Our previous experience with inlining and cloning demonstrated only small improvements. There was little motivation to include these optimizations in a FORTRAN compiler, given their substantial compile-time costs. However, the experiment with matrix300 showed that inlining and cloning could yield significant improvements if used to enable high-payoff transformations. The high-payoff optimizations described in this chapter were memory-management optimizations, but a similar strategy could be used to enable parallelizing transformations. To enable the memory-management optimizations, cloning is performed only to expose certain important constants and inlining is used to produce loop nests of 2 or more loops around memory-bound inner loops.

This chapter has ignored the problem of code growth and compilation dependences associated with inlining and cloning. In the next chapter, controls on the amount of cloning are described, and an algorithm is given for merging clones whenever they produce the same effects on optimization. In Chapter 6, constraints on inlining are suggested to avoid code growth and extensive compilation dependences.

Chapter 5

Procedure Cloning

A major goal of this dissertation research was to explore the potential of inline substitution and procedure cloning in enhancing aggressive code optimization. For cloning, there were two main subgoals. The first was to determine when cloning is effective, and the second was to develop a general algorithm for cloning which could be used to partition calls based on solutions to many interprocedural problems.

The work in the previous chapter on goal-directed optimization provided insights into the answer for both of these. We learned when cloning based on interprocedural constants could be effective. In addition, issues arose in the exercise of optimizing matrix300 that led to a general cloning algorithm.

This chapter presents a cloning algorithm that partitions calls based on the solution to any forward interprocedural problem. Because cloning can theoretically result in exponential program growth and an exponential increase in the number of procedures in the program, the algorithm has certain restrictions on the amount of cloning and program growth tolerated. After the algorithm has determined the maximal set of clones that meet the restrictions, clones are merged whenever they produce the same effect on optimization. The final phase of the algorithm applies cloning based on these decisions until program growth reaches a certain size constraint.

The next section develops the insights that led to the cloning algorithm. The second section presents the cloning algorithm, with separate subsections devoted to each of algorithm's three phases. Section 5.3 presents related work on procedure cloning, and section 5.4 concludes the chapter.

5.1 Motivation

Procedure cloning was introduced by Cooper as a method of partitioning calls to a procedure based on its interprocedural constants information [Coo83]. Previous research, particularly Ball's work, used inlining to accomplish the same result [Bal79]. Cooper observed that cloning had an advantage over inlining in refining constants information, since multiple calls to a procedure can share a cloned version. A further advantage of cloning is that it does not introduce as many compilation dependences among procedures as compared to inlining. Procedures invoking or invoked by a cloned version do not necessarily require recompilation when the cloned procedure is edited but can just be relinked

This chapter provides a general approach to procedure cloning. The algorithm can be used to refine a variety of interprocedural solutions. The algorithm attempts to use cloning in the most effective way, while controlling the costs of cloning due to additional analysis and compile time. It was motivated by the following four observations:

1. Cloning can be used to partition calls based on the solution to any *forward* interprocedural data-flow problem.
2. Cloning a procedure can affect the interprocedural information at its descendant procedures.
3. Creating unnecessary clones can be avoided by merging clones that create the same effect on optimization.
4. Cloning is bounded by exponential time and program growth, so considerations must be made to avoid this worst-case behavior.

Each of these points is developed in the paragraphs below.

Cloning on any forward interprocedural problem. Although previous work has only considered cloning based on interprocedural constants, cloning can be used to refine any information coming into a procedure from its predecessors. Thus, cloning can be used to partition calls to a procedure based on the solution to any forward interprocedural data-flow analysis problem. A forward data-flow problem determines information at a node based on its predecessors in the graph. Call multigraph construction, interprocedural CONSTANT and ALIAS analyses are forward interprocedural problems.

Propagation. Cloning refines information coming into a procedure from its callers by avoiding approximating the interprocedural solution to match all callers. By improving interprocedural information at a procedure, we may also be able to refine information at its descendants. This is because the solution of a forward interprocedural problem at a procedure is propagated during analysis to its descendants. This suggests that a cloning algorithm should propagate cloning opportunities top-down in the call multigraph. The impact of these options on descendant procedures can be evaluated after all cloning opportunities have been located.

The decision to clone a procedure may in turn enable opportunities for creating a chain of cloned procedures down the call multigraph. In some cases, as with matrix300, this cloning can result in dramatic improvements in program performance. Consider that using inling to accomplish the same effect would require constructing a single procedure that compresses a large chain of calls. Such a large procedure would be expensive to optimize, could have adverse performance behavior, and would introduce compilation dependences requiring all procedures in the chain of calls to be recompiled if any one of them changed.

Merging equivalent clones. After determining all possible clones for a program, it may be the case that some of them can be merged. Each clone will have different interprocedural information. However, in some cases, the differences in interprocedural information will have little effect on the optimizations that can be performed on the procedure. If two clones of a procedure produce the same effects on important optimizations, they are good candidates for merging.

This relates to the goal-directed strategy of the previous chapter. We selected for cloning only those call sites that exposed constant values for important variables. During the merging step, we can test the effects of these constant values on optimization, since we have the set of possible constant values from the analysis of possible cloning decisions. The notion of equivalence of clones is explored in this chapter. The second phase of the cloning algorithm merges equivalent clones, producing the minimal number of clones required for specific effects on optimization.

Restricting cloning. The amount of cloning that can be performed on a program has a worst-case exponential time bound. Based on our experimentation, the amount of cloning performed on a program, especially after filtering out uninteresting cases, is likely to be be very small. Nevertheless, a cloning algorithm should impose restrictions on the amount of cloning in the event of pathological behavior.

In the first phase of the algorithm, we restrict the number of clones being considered to a polynomial of the number of procedures. Based on our experience, we expect the restrictions on the number of clones being considered to rarely be necessary. For this reason, when restrictions are imposed, they do not attempt to produce the best cloning solution within the constraints. This polynomial number of clones is only tolerated during the analysis phase.

When cloning is actually performed in the third phase, the growth of the program size is also restricted. The restriction on program growth, which is more likely to be required, does in fact attempt to perform the most important cloning. The next chapter defines *PathFrequency*, an ordering on nodes determined by execution frequency estimates of a procedure's descendants. The idea of *PathFrequency* is to give priority to paths in the call multigraph leading to frequently executed calls. Cloning is performed on procedures in *PathFrequency* order, and halted when program growth exceeds its constraints. As defined, *PathFrequency* also preserves topological order.

5.2 Cloning Algorithm

The algorithm has three phases. First, we propagate vectors of interprocedural information describing the possible cloning that can be performed on the program. In the second phase, we merge vectors that represent clones with "equivalent" effects. In third phase, the cloning is actually performed.

The rest of this section is divided into five parts. The next three parts describe the three phases of the algorithm. The other two parts present how to use the algorithm to clone based on multiple interprocedural problems, and how to actually perform the cloning as a source-level transformation.

5.2.1 Phase 1: Calculate *CloningVectors*

In the first phase, cloning information is propagated down the call multigraph to determine the maximal number of clones that should be created for the program. The idea behind the algorithm is to retain interesting interprocedural information along all paths through the program, rather than conservatively approximating information when multiple paths join.

Each unique procedure clone can be represented by a *CloningVector*. This can be thought of as a vector of information representing the value of the interprocedural set used as the basis for cloning. For example, it could be the list of $\langle variable, constant\rangle$ pairs from constant propagation or the list of variable pairs from alias analysis. (This definition will be altered slightly in the discussion below.)

Given a forward interprocedural problem and an input program, the cloning algorithm calculates all values for the interprocedural set of each procedure that are possible through execution of the program. Essentially, the algorithm is tracing the flow of interprocedural sets on all paths through the program. It can be thought of as performing interprocedural analysis of the set used as the basis for cloning but propagating this information in a way that is flow sensitive on the call multigraph.

Algorithm

The algorithm for calculating *CloningVectors* is given in Figure 5.1. It operates by propagating all values for some forward interprocedural set S that can be created during program execution. Since a procedure can inherit information exposed by cloning from its callers, *CloningVectors* are propagated in topological order. This makes it possible for the algorithm to make only a single forward pass over the call multigraph. Procedures involved in recursive cycles are handled specially. Their *CloningVectors* set contains only the set value for that procedure. In other words, no cloning will be performed within recursive cycles. This restriction can be relaxed with some modifications to the algorithm. An explanation of the restriction and discussion of how to relax it are given in Section 5.2.4. Time considerations did not permit support for cloning in recursive cycles to be incorporated into the algorithm.

Before presentation of the algorithm, a few definitions are needed:

- S identifies the interprocedural set being used as the basis for cloning. It is also used in the algorithm to give the value for that interprocedural set at a particular procedure or at a call site.
- The set *CloningVectors(S, p)* gives *all* possible values for interprocedural set S that can reach procedure p.
- The function *Translate(c, cv)*, for some call site c with caller p and callee q, maps elements in the *Cloning Vector cv* of p to the corresponding variables in q based on parameter passing at c. The result is the creation of a new *CloningVector* for q. This mapping function is similar to what is used in interprocedural propagation to map variables in the caller to variables in the callee.

The *CloningVectors* for each procedure are initialized to $\emptyset$. Possible values for the interprocedural set S are then propagated in topological order down the call multigraph. It is necessary to initialize *CloningVectors* for a recursive procedure p in case p invokes some other procedure q that is not contained in a cycle. In this way, when calculating *CloningVectors(q)*, q inherits a value from p.

```
/* Initialization */
foreach procedure p
    CloningVectors(S, p) ← ∅

/* Propagation */
foreach procedure p in topological order
    if p is part of a recursive cycle then
      CloningVectors(S, p) ← {S(p)}
    else
      foreach call site c invoking p
         let n be the procedure invoking p at c
         foreach vector v in CloningVectors(S, n)
             CloningVectors(S, p) ← CloningVectors(S, p) ∪ Translate(c, v)
    endif
end
```

Figure 5.1 Algorithm for calculating *CloningVectors*.

Time Complexity

The time required by the algorithm is bounded by the number of procedures and by the number of unique *CloningVectors* generated at each call site, since the outer loop iterates over procedures, and the inner loop iterates over *CloningVectors* at a call site. Assume the maximum number of elements in a *CloningVector* is L, and the maximum number of values for each element is V. N is the number of procedures in the program, and E is the number of call sites. Then, the algorithm is bounded by $O((N+E)V^L)$ time.

The actual sizes of V and L are dependent on the interprocedural set being used and the possible values of the set elements. Since we are dealing with interprocedural information, the size of L is related to the number of externally accessible variables in the scope of the procedure. This is the number of global variables and formal parameters of a procedure. Let us assume that this number is bounded by $clogN$. This assumption is based on the idea that the number of variables in a program grows logarithmically with the program size.

Let v_i be the number of distinct values that the ith element in a *CloningVector* can have. For each v_i, there is a k_i such that $2^{k_i-1} < v_i \leq 2^{k_i}$. For a given procedure, an upper bound on the number of unique *CloningVectors* is defined by the following equation:

$$\prod_{i=1}^{L} 2^{k_i} = 2^{\sum_{i=1}^{L} k_i}.$$

Taking the average of k_i over its L possible values, we arrive at some value k_p. 2^{k_p} gives an average number of values for each element. Assuming $L \leq clogN$, we know that the number of *CloningVectors* for a procedure $p \leq 2^{k_p clogN}$. The total number of *CloningVectors* is as follows:

$$\sum_{i=1}^{N} 2^{k_i clogN} \leq N * 2^{clogN * \max_i k_i} = N^{c*\max_i k_i}.$$

This shows that, given reasonable values for L and the k_i, the time complexity is polynomial.

These bounds are plausible for most programs and most interprocedural problems. However, even if the bounds are too low, they can easily be enforced. If the vector length exceeds $clogN$ for some c, worst-case assumptions are made about the variables in the positions after the $clogNth$. Also, if 2^k unique values for an element have been found, additional values are not tracked. (There is a boundary condition here. If $\perp$ is not one of the 2^k unique values, then only $2^k - 1$ unique values are tracked, and remaining values are assumed $\perp$.) Ideally, the restriction should merge *CloningVectors* that are similar so that little is lost by the merge.

The restriction on V may be somewhat limiting, especially for problems like interprocedural constants where V can be quite large. So instead of imposing both restrictions in all cases, we restrict the overall number of *CloningVectors* to be bounded by $O(N^{ck})$. Only when the number of *CloningVectors* exceeds this bound is it necessary to impose either of the restrictions. Extensive filtering of *CloningVectors* can occur by propagating values for interesting variables only, such as the *CloningVars* of the previous chapter. This further reduces the likelihood that the restrictions on cloning will need to be imposed.

Based on the experiments with cloning, it is unlikely that any restrictions on the number of *CloningVectors* will be necessary in practice. The amount of cloning possible in a program has not been that large. For these reasons, the restrictions do not make an attempt to produce the optimal cloning while obeying the constraints. Instead, they are simply a guard on the potentially exponential behavior theoretically possible with cloning.

Examples

Let us consider as examples cloning based on interprocedural constants and on aliases. For cloning based on constants, we will have vectors of $\langle constant, value \rangle$ pairs. In this case, the number of elements in the vector will likely be small. However, the number of values for each element in the vector can be relatively large. In the case of constants, restricting the number of values tracked for each element could inhibit optimization. However, if we only restrict the total number of vectors, it is unlikely the need will arise to restrict the number of values for an element.

For aliases, we will have vectors representing each pair of variables, with a value of true or false indicating whether they are aliased. If all pairs of variables were represented, then the vector length could get quite large, although the number of values for each element is at most 2. However, the only elements that will have multiple values are those that the interprocedural information indicates are aliases, but which are not aliases under some caller. This number will probably be small.

5.2.2 Phase 2: Merge Equivalent *CloningVectors*

Assuming none of the restrictions on cloning are imposed, the previous algorithm produces *CloningVectors* describing all interesting cloning in the program. Some of these clones may be equivalent in the sense that, even though their interprocedural information is different, they have the same effect on important optimizations. As an example, in the previous chapter describing the matrix300 experiment, eight copies of the procedure *dgemm* were made, but only two of those were needed to create the appropriate two clones of the called procedure *dgemv*.

The second phase of the cloning algorithm locates equivalent *CloningVectors* and merges them. This step reduces the amount of cloning required by the program, while not affecting the important optimizations. This second phase is completely unnecessary for the correctness or the effectiveness of the cloning algorithm. Nevertheless, it is useful because it reduces the significant costs of cloning without an appreciable effect on optimization.

Determining when two *CloningVectors* result in the same important optimizations requires a goal-directed strategy. It is necessary to locate specific targets of optimization, so that the effects of a particular *CloningVector* on the targets of optimization can be ascertained. If two different *CloningVectors* have the same effect on these targets of optimization, then they can be merged. The importance of a cloning decision is represented by a *StateVector*.

Defining a *StateVector*

For each cloning problem, it is necessary to determine what contributions from a *CloningVector* are significant. As an example, this can be based directly on the important program points used in a goal-directed strategy to calculate the *CloningVectors*. For interprocedural constants, the *StateVector* would then be the values of important constants appearing in the procedure: the values of each control flow test, subscript expression and formal parameter used as an array dimension. For each one of these, we construct a *jump function* that describes the expression value as a function of potential interprocedural constants [CCKT86]. To reduce the number of jump functions, they are only provided for program points whose values could become known through interprocedural constants, as determined during the calculation of *CloningVars*. With this information and a *CloningVector* describing constant interprocedural values, the value of a *StateVector* can be determined. The example in Figure 5.2 illustrates these points.

If each cloning problem was formulated in a goal-directed way, and the *StateVectors* were directly generated from the interesting program points targeted by the goal-directed strategy, then the merging of *CloningVectors* would not hurt the optimizations targeted by the goal-directed strategy. However, it should be noted that the information propagated to locate worthwhile cloning may be more general than that required by the important optimizations.

To see this point, let us return to the matrix300 example. Eight copies of *dgemm* were created, but this could have been reduced to two copies without affecting the important optimizations. However, each copy has a different value for a variable which

procedure $p(f_1, f_2)$
S_1: **dimension** $A(f_1, 1)$
S_2: **if** $(f_2 \ mod\ 2)$ **then** $\ldots$
S_3: $\quad A(f_1 + 2, 1) = \ldots$

Jump functions:
S_1: f_1
S_2: $f_2 \ mod\ 2$
S_3: $f_1 + 2$

$$CloningVector(c_1) = \langle (f_1 = 10), (f_2 = 7) \rangle \quad StateVector(c_1) = \langle 10, 1, 12 \rangle$$
$$CloningVector(c_2) = \langle (f_1 = 10), (f_2 = 5) \rangle \quad StateVector(c_2) = \langle 10, 1, 12 \rangle$$

Figure 5.2 Example illustrating calculation of *StateVectors*.

determines control flow within *dgemm*. Thus, by defining *StateVector* to consider the same program points as what was used in the goal-directed cloning strategy, we would still create the eight copies of *dgemm*.

It turns out that what is really important for memory-management optimizations is improvement in the precision of dependence information. The constants used to determine control flow are not interesting by themselves, but only to the extent that they affect dependence analysis. In the example, once all possible values for the control flow variable were known, the effects could be determined, with certain procedures producing equivalent results. By only tracking subscript expressions and certain types of control flow (such as loop bounds and control flow within inner loops), we can reduce the size of *StateVector* and merge more clones. This would have allowed us to only generate two copies of *dgemm*.

Any goal-directed strategy, because it involves heuristics, should be driven by the results of experimentation. Reducing the contents of *StateVector* from that prescribed by the goal-directed strategy should also be driven by experimentation. Additionally, experimentation might indicate that the amount of cloning resulting from the goal-directed strategy is so small that minimization of cloning is not an important issue.

Partitioning Algorithm

The algorithm for merging equivalent *CloningVectors* is related to the algorithm for minimizing the number of states in a Deterministic Finite Automaton (DFA) [Hop71]. It is very similar to an algorithm used in ParaScope to minimize the number of implementations of a procedure required when multiple definitions of the procedure occur in the program composition [CKT+86c].

The partitioning algorithm is presented in Figure 5.3. Initially, all clones of a procedure are placed in the same partition. The algorithm distinguishes between cloned versions, based on their *StateVector* and the partitioning of procedures they invoke. Upon termination of the algorithm, clones remaining in the same partition can be merged and represented by a single clone. Two clones can be merged if they have the same *StateVector*, and for corresponding call sites in the cloned versions, the invoked procedures are in the same partition of *CloningVectors*.

Procedures are visited in reverse topological order. Since cloning is not considered for procedures involved in recursive cycles, this ordering always exists for the procedures with multiple *CloningVectors*. With reverse topological order, the clones of a

1. Partition the *CloningVectors* such that all *CloningVectors* for a particular procedure are in the same partition.
2. In reverse topological order, visit the partition π representing each procedure p:
   (a) Partition elements p_i of π based on the value of *StateVector(p_i)*.
   (b) For each partition π_i of π consisting of multiple elements:
   Form partitions of elements of π_i such that if two *CloningVectors* a and b in π_i result in invocations at some call site c with *CloningVectors* x and y of the called procedure, then a and b are in different partitions if x and y are in different partitions.

Figure 5.3 Algorithm for minimizing the number of *CloningVectors*.

---

procedure have been partitioned before any of its callers are considered. This means that only one pass over the procedures is necessary.

Time Complexity

The expected time required by the algorithm is linear in the number of *CloningVectors*. The *CloningVectors* for a procedure are only partitioned once. With an appropriate representation for *StateVector* and for *CloningVectors* resulting from call sites, the expected time required for partitioning can be done in time linear in the number of elements being partitioned. (A different representation would yield $O(nlogn)$ time, even for worst-case performance [Hop71].)

As a possibility, the set representations can be treated as strings, with some canonical order imposed on the set elements. Then, partitioning can be performed by hashing to a location matching the string representation of the set. If two sets hash to the same location and have the same set value, they belong in the same partition.

5.2.3 Phase 3: Perform Cloning

After minimization of the *CloningVectors*, a potentially polynomial number of them remain. If all of this cloning were performed, the final program size could be a polynomial of its original size. The polynomial bound on the number of *CloningVectors* is acceptable during analysis, but a polynomial growth in program size is intolerable due to its effects on compile time. Thus, as an additional safeguard to the costs of cloning, we only clone until program growth exceeds some threshold. As with the restrictions on the number of *CloningVectors*, we expect the need for this will be rare.

```
originalSize ← program.size
foreach procedure p in PathFrequency order
    tempPF ← p.PathFrequency
    performCloning(p)

/* If size constraint exceeded, clone all remaining procedures with same PathFrequency */
    if (program.size > originalSize * threshold) then
        foreach remaining procedure p' such that p'.PathFrequency = tempPF
            performCloning (p')
        exit
    endif
endfor

performCloning (p)
    foreach partition π_p of p
        – create a copy newp of p
        program.size ← program.size + newp.size
        – annotate representation of newp with StateVector and set of CloningVectors in π_p
        – update other interprocedural information for newp and descendants
    end
endfor /* performCloning */
```

Figure 5.4 Algorithm for performing cloning.

Algorithm

The algorithm for performing cloning is given in Figure 5.4. The algorithm performs the cloning indicated by the partitions of *CloningVectors* produced in the previous step. Since the cloning decisions at a procedure are affected by cloning of its ancestors in the call multigraph, it is critical that the cloning be performed in topological order.

The algorithm clones in *PathFrequency* order, first cloning procedures in portions of the call multigraph leading to frequently executed procedures. The calculation of *PathFrequency*, as presented in Chapter 6, also preserves topological order. An ideal ordering of cloning decisions would take into account not only execution frequency but also an estimate of the benefits expected by cloning.

The algorithm performs cloning until the program size reaches some threshold factor of its original size. A good rule of thumb is to allow the program size to double. Once the program has passed its size constraint, a little more cloning may be performed. The algorithm will continue to create the clones for the current procedure. It will also create the clones for any remaining procedures with equal values of *PathFrequency*. This is because the entire path leading to a frequently executed procedure will have the same value for *PathFrequency*. Thus, cloning is completed on the path that is currently being optimized so that the cloning that was performed above will not have been wasted.

After performing the cloning, the program representation needs to be updated to reflect changes to the interprocedural information. Because the program structure is changed, some of the interprocedural solutions may observe refinements to their information, even though the purpose of the cloning was not to refine these solutions. For interprocedural problems other than the one that was used as the basis for cloning, the information must be incrementally updated. Incremental updates to interprocedural information will be addressed in the next chapter.

For the interprocedural problem being used as the basis for cloning, we need to associate with a clone its set of *CloningVectors* and its *StateVector*. Since multiple *CloningVectors* may have been merged into a partition, it is not sufficient to just update the interprocedural information to reflect the new values. This is because the conservative approximation of multiple *CloningVectors* will not reflect the optimizations that can be performed on the procedure. The *StateVector* will be needed by the optimizer to enable reconstructing the important effects that initiated the cloning decisions. The *StateVector* and the set of *CloningVectors* will be needed for recompilation analysis, to ensure on a subsequent compile that the optimizations are still valid. The problem of recompilation analysis in the presence of this cloning algorithm will be presented in Chapter 6.

5.2.4 Cloning in Recursive Cycles

If cloning were performed in recursive cycles during calculation of new *CloningVectors*, and interprocedural sets were incrementally updated, the effect of cloning could be to unroll the recursive cycle. Potentially, the algorithm would not terminate. This was our concern when disallowing cloning within recursive cycles.

We now understand how to support recursion. First, strongly-connected regions are located in the call multigraph, and each cycle is replaced with a representative node [Zad84]. When the algorithm reaches a node representing a cycle, it must take each incoming *CloningVector* and propagate it within nodes in the recursive cycle until the *CloningVector* information stabilizes. The *CloningVectors* resulting from the propagation, which may contain less information than the original incoming *CloningVectors*, determine both cloning of the cycle and the *CloningVectors* that are propagated to successors of the cycle in the call multigraph. Whenever the final cloning algorithm determines that cloning should occur at a representative node, each procedure involved in the cycle is cloned.

5.2.5 Cloning Based on Multiple Interprocedural Problems

If multiple interprocedural problems are being used as the basis for cloning, then the entire cloning process should be iterated for each interprocedural problem. The three phase process of constructing *CloningVectors*, merging them and performing cloning should first be performed based on the most important interprocedural problem. If the size constraints have not been exceeded when this process is complete, it can be repeated on the second most important problem. Cloning for the second interprocedural problem should be performed on the altered call multigraph after edges have been reassociated and interprocedural information has been updated.

The reason for the iterative process is that cloning based on one interprocedural problem can affect the solution to other interprocedural problems. Since cloning changes the structure of the call multigraph, it may result in refinements to other interprocedural solutions.

5.2.6 Implementing Cloning

A final issue in cloning is how to represent it in the source code. This issue came up in the cloning experiment, described in the previous chapter. During the experiment, a procedure was cloned by making a copy and renaming the copy. Call sites invoked the clone by referencing the new name. We first considered that the constant parameters of a clone be eliminated from the parameter list and each reference in the source code be replaced by the corresponding constant value. An unfortunate consequence of this approach is that any call sites invoking a clone must have their actual parameter lists modified to eliminate the constant-valued actuals passed at the call. If callers of cloned procedures are modified, this introduces an unnecessary compilation dependence from a cloned procedure to its callers.

We decided that an automatic system should leave call sites invoking clones intact. The assignment of call sites to clones can then be done at link time. This makes it possible for a call site invoking a clone to be mapped to a different cloned version on later compilations of the program. This requires that the compiler have some control over program linking. It would be impossible to do this strictly at the source level.

5.3 Related Work

5.3.1 Procedure Cloning

Procedure cloning was introduced in Cooper's dissertation as part of his linkage tailoring algorithm [Coo83]. At that time, the technique was referred to as *node splitting*; it was renamed *cloning* in a later publication [CKT86a]. Cooper performed cloning based on interprocedural constants, and did not propagate improvement in information due to cloning down the call multigraph. Thus, a cloning decision was based on the direct effects it would have on a procedure. Because cloning was only performed at one level in the call multigraph, the final number of procedures in the program was bounded by the number of call sites. For this reason, Cooper was not concerned about restricting cloning to ensure efficiency.

5.3.2 Similar Techniques

Cloning bears some relationship to node splitting as used in interval-based data-flow analysis techniques [AC76]. Two copies of a control flow node are made, and edges into the node are reassociated with the copies, to break up unstructured program portions. In this case, the copies are made to enable the analysis, not to improve optimization.

In his dissertation, Wegman describes a technique called *node distinction*, which uses node splitting *intraprocedurally* on the control flow graph to improve optimization [Weg81]. His technique makes multiple copies of a control flow node if its data-flow information differs on incoming edges. Thus, based on the value of some forward data-flow set, he constructs the maximal node distinction control flow graph. As with the *CloningVectors* algorithm, this can potentially produce an exponential number of nodes in the flow graph.

Wegman uses a goal-directed strategy to reduce the amount of copies being made. For a given data-flow problem, he considers if the refinement of information provided by making a copy of a node will actually have an effect on optimization. Some heuristics for restructuring the control flow graph also reduce the amount of copying that occurs. As a final restriction, he suggests only performing this optimization in innermost loops. Nevertheless, the amount of copying retains an exponential bound on the nodes in innermost loops since the heuristics do not guarantee that copying is eliminated.

Also, the algorithm deals with intraprocedural data-flow sets, which may have profoundly different properties from interprocedural sets. It may be the case that intraprocedural sets are more likely to differ than intraprocedural sets when paths in the graph merge.

5.4 Chapter Summary

This chapter has presented a general algorithm for procedure cloning. The algorithm represents significant progress over previous work. It can be used for partitioning calls to a procedure based on any forward interprocedural problem. Time complexity for the algorithm will be polynomial for most programs and cloning problems. Restrictions are provided to maintain a polynomial time bound if needed. After locating all opportunities for cloning, clones are merged whenever they result in equivalent effects on optimization. When cloning is actually performed, further restrictions are imposed to limit program growth. The algorithm has restrictions to avoid the worst-case exponential behavior of cloning. Even so, based on experimental evidence, we expect the restrictions will rarely be necessary and the full amount of cloning will be performed.

Chapter 6

Interprocedural Compilation System

This chapter completes the treatment of interprocedural optimization for scalar optimizing compilers. A general system is described for supporting interprocedural optimization. This system performs interprocedural optimizations using some combination of three techniques: 1) inline substitution, 2) procedure cloning, and 3) global optimization improved by interprocedural information. These transformations interact with each other and with interprocedural analysis. This chapter addresses the interaction between the techniques and the implication this interaction has on the design of the compilation system.

The previous design of the ParaScope compilation system did not consider how to incorporate inlining and cloning while avoiding unnecessary recompilation. The recompilation problem affects all aspects of the design of the compilation system. When deciding whether an inlining or cloning decision is worthwhile, the compilation dependences resulting from this decision are considered. Also, the batch system for interprocedural optimization is formulated to enable efficient recompilation. Both the batch and recompilation systems for interprocedural optimization are presented in this chapter.

In the next section, the effect of procedure cloning on other optimization techniques and on program growth is reviewed. Section 6.2 briefly presents how inlining can be used to enable optimizations other than memory-management transformations while avoiding performance degradation and excessive compile-time costs. The third section describes interprocedural analysis in ParaScope and the additional interprocedural analysis required for inlining and cloning. In section 6.4 we present an algorithm for applying interprocedural optimization that evaluates call sites for cloning and inlining. Section 6.5 extends this algorithm to manage recompilation when interprocedural optimizations have been applied, attempting to reuse as many of the procedure components from the previous compilation as possible. Section 6.6 presents related work in the area of compilation systems for automatically managing the dependences among procedures.

6.1 Procedure Cloning

The previous chapter thoroughly described the cloning algorithm. This section presents two remaining issues on cloning as it relates to system support. First of all, cloning can affect the results of inlining and optimization based on interprocedural information. The relationship of cloning to these other problems is presented here. Secondly, to avoid excessive program growth, the amount of cloning is constrained. At the end of the section, we briefly describe how to constrain program growth.

6.1.1 Effects of Cloning

Cloning has two effects on interprocedural information. First, it refines information for the interprocedural problem used as a basis for cloning. Second, it changes the structure of the call multigraph, which may indirectly improve the precision of other interprocedural solutions. These two effects of cloning have some implications on how it should be used in cooperation with other interprocedural techniques.

The second effect of cloning implies that after cloning based on one interprocedural problem has been performed, the solutions of other forward interprocedural problems should be updated. In this way, the other forward problems will be correct with respect to the current call multigraph. Such updates should be performed before these other interprocedural solutions are used as a basis for further cloning. For efficiency reasons outlined in Section 6.4, updating solutions to backward problems is deferred until after cloning.

The most important effect of the cloning phase is to produce refinements in forward interprocedural solutions. As it turns out, solutions to forward interprocedural problems are very important, and refinements in the solutions can produce dramatic improvements in opportunities for optimization. Most significantly, improved constants information can greatly simplify the control flow within a procedure, which can refine the solution to any interprocedural problem. Moreover, increased information about constants and aliases can result in improvements to backward interprocedural problems, such as side-effect information. (This should become clearer when the calculation of this interprocedural information is described in section 6.3.) Also, in Chapter 4 we established that improved constants information could affect inlining decisions. Taken as a whole, it appears that cloning should be the first interprocedural technique used on a program, and cloning should be completed before any other optimization techniques are used.

6.1.2 Restricting Cloning

To avoid problems with program growth, cloning should be restricted so that program size does not increase past some threshold. A useful measure is allowing the program to grow to twice its original size. However, this may need to be adjusted if inlining will also be performed, since it also requires constraints on program growth.

Program growth can be measured in a variety of ways. Text size of the source code provides a rough measure of program size. In our system, the overriding efficiency concern is the size of the Abstract Syntax Tree representation, so we use this as the measure. In general, size of lower-level representations of the program will likely provide a more accurate measure than higher-level ones, since they more closely reflect object code size. Factoring in the effects of optimization will also improve the size estimates, especially constant propagation.

6.2 Inline Substitution

Based on the inlining experiments in Chapter 3, inlining should be avoided unless it is very likely to improve performance. This is particularly true in ParaScope, where other interprocedural optimizations are performed which achieve some of the benefits of inlining. Chapter 4 provided one motivation for inlining – to enable memory-management transformations. The brief discussion of inlining in this section gives guidelines for using inlining for other purposes. There are three main points:

- Perform inlining of call sites in reverse topological order.
- Base inlining decisions on heuristics that guarantee inlining will improve the code or at least avoid degradation.
- Stop when constraints are exceeded to avoid recompilation costs.

6.2.1 Inlining Order

To avoid unnecessary steps, inlining should be performed in reverse topological order. To see this, consider the call chains ($a \rightarrow b \rightarrow c$) and ($d \rightarrow b \rightarrow c$). Suppose that b is inlined into a and d. Then, to inline c into b, the inlining has to be performed to both a and d. If c had been inlined first, then the inlining would have only been performed once.

The next section describes an ordering *EdgeFrequency* on call sites that favors most frequently executed calls. As defined, *EdgeFrequency* preserves reverse topological order. Such an ordering is needed because inlining is restricted to avoid compilation costs. There may be some call sites which the heuristics suggest are good targets for inlining but do not meet the inlining constraints. Thus, the most important call sites are considered first.

While inlining is being performed, solutions to backward interprocedural problems can be updated incrementally using the method described in section 6.5. The backward interprocedural solutions may have improved from the cloning phase, or as a direct result of inlining. Updating the information is particularly useful if the inlining heuristics happen to be based on the solution to some backward interprocedural problem. Changes to forward interprocedural solutions following inlining are ignored for efficiency considerations.

6.2.2 Inlining Heuristics

The heuristics for inlining should either strongly suggest that inlining will improve the code or at least that no significant degradation will result. In Chapter 4, we inlined calls in memory-bound loops, with the assumption that the inlining would either reduce the number of memory accesses or would enable other optimizations that would do so. A previous approach we considered was to estimate the number of registers used by a procedure and its caller before and after inlining. Then, inlining could only be performed if the estimate indicated that the number of required registers did not increase after inlining. We selected the former approach because it targets optimizations with significant payoffs. The latter approach would have been justified if we had found that, in general, inlining alone could bring about substantial performance improvements.

6.2.3 Restricting Inlining

Inlining has a profoundly different effect on compilation costs than cloning. It not only increases program size but also increases the size of individual procedures. In addition, inlining introduces increased compilation dependences. All procedures making up an inlined version are interdependent. The inlined version requires recompilation if any one of them is edited. To avoid growth in compile time as well as extensive compilation dependences, inlining requires several restrictions.

To control compilation time, overall program growth is restricted to twice its original size. The growth of individual procedures is also restricted. As supported by the experiments in Chapter 3, some optimization techniques use nonlinear algorithms. Since an optimizer only examines one procedure at a time, the procedure size is actually more critical than program size. Procedure size should be restricted to twice the size of the largest procedure in the original program. This keeps the overall compile time cost about the same. It may also be necessary to place a fixed limit on procedure size. For example, in ParaScope manipulating large procedures causes serious performance problems. This was also the case with the MIPS compiler.

Further restrictions on inlining are needed to avoid extensive recompilation. First, the length of inlined call chains is restricted. Additionally, we restrict the number of call sites within a procedure that may be inlined. This puts a fixed limit on the number of procedures that may be interdependent as a result of inlining. To see this, suppose that the limit on the length of call chains is L_1 and the the limit on the number of call sites is L_2. Then, in a given inlined procedure, the number of procedure bodies it includes can be no greater than ${L_2}^{L_1} + 1$. Since the number of compilation dependences grows exponentially, it should be obvious how important these restrictions are to avoiding unnecessary recompilation. Note that these two restrictions to inlining mean that the algorithm can be used on recursive programs.

6.3 Optimization Using Interprocedural Information

This section describes interprocedural information that can be used to enhance global optimization and to detect opportunities for inlining and cloning. This section has three main purposes: to document the interprocedural analysis used in ParaScope, to demonstrate the extent to which interprocedural information enables many optimizations considered important consequences of inlining, and to describe the remaining interprocedural information needed to support inlining and cloning.

In describing the interprocedural analysis performed in ParaScope, the calculation of the analysis is described as well as the applications of the interprocedural sets to optimization problems. It is important to understand the benefits of interprocedural information to use as a basis of comparison against inlining. In terms of space costs and compilation dependences, it is preferable to use the default linkage and optimize based on interprocedural information if the effects on optimization are similar. (Of course, it is difficult to efficiently make this determination.)

Interprocedural information is also needed by the inlining and cloning algorithms to locate good targets for optimization. As an example, execution frequency estimates are presented in this section. Other interprocedural information used in cloning and inlining was presented in Chapter 4. Finally, interprocedural information is used to optimize the cloning and inlined procedure versions.

6.3.1 Interprocedural Analysis in ParaScope

ParaScope currently calculates the following sets of interprocedural information:

- MOD$(e) = \{x \mid x$ *may* be changed through call $e\}$
- REF$(e) = \{x \mid x$ *may* be accessed through call $e\}$
- ALIAS$(p) = \{\langle x, y\rangle \mid x$ and y *may* refer to the same memory location in $p\}$
- CONSTANT$(p) = \{\langle x, c\rangle \mid x$ *must* have value c across all calls to procedure $p\}$

Side effects. Side-effect information is comprised of MOD and REF sets. The side-effect information for a call site c contains variables affected by the procedure q invoked at c, including variables affected by the descendant procedures of q. Both MOD and REF sets are flow-insensitive; that is, they represent the union of information occurring along all paths in the program. As a result, MOD and REF sets may contain some variables that are not actually modified or used (i.e., the sets are conservative). A flow-insensitive approach is necessary in the presence of aliasing to guarantee a polynomial time bound [Mye81].

With side-effect information, a compiler can avoid reading in a variable from memory after a procedure call if the variable is not modified by the call. The compiler can also avoid writing a variable to memory before a call if it is neither referenced nor modified by the called procedure. These uses of interprocedural information reduce call overhead, which is considered an important benefit of inlining.

Side-effect information is also useful in a variety of global data-flow problems. Modification information can be used in determining available expressions and reaching definitions. Reference information can be used for live variable analysis and reachable uses.

Aliases. The set ALIAS(p) contains all pairs of variables $\langle x, y\rangle$ such that x and y may refer to the same memory location along some chain of calls in the program leading to procedure p. In FORTRAN, aliases result from two mechanisms: passing by reference the same variable to two formal parameters at a single call site, and passing by reference global variables as parameters. In other languages, pointer variables also contribute to aliasing.

Once aliasing information is known, MOD and REF sets are updated to reflect this knowledge. This is because a variable is modified (referenced) if any of its aliases are modified (referenced). Side-effect analysis and alias analysis are performed separately, with alias information added to the side-effect information [Ban79]. Recent algorithms have been proposed to calculate aliasing and interprocedural side-effect information in time effectively linear in the size of the call multigraph [CK88b] [CK89]. However, currently in ParaScope, we use an iterative technique based on Banning's equations [Ban79].

In addition to making side-effect information correct, alias information is useful to the register allocator. Compilers cannot place variables in registers that might be aliased, so the absence of a variable in the ALIAS set enables allocating it to a register. Also, any optimization that involves moving computation from one place to another in the program (e.g., code motion) requires that the order of loads and stores be preserved, which is difficult to determine if the variables involved may have aliases.

Interprocedural constants. The set CONSTANT(p) gives variables and their constant values only when the constant values exist for all calls to p. Since the problem of finding all variables that are constant at run-time is undecidable [KU77], the set calculated is an approximation.

The algorithm used for interprocedural constant propagation is given in [CCKT86]. It relies on a jump function J_c at the call site c which gives the values of the actual parameters of c as functions of the formal parameters of the called procedure. These jump functions are determined by the module editor on completion of an editing session. There are several alternative methods for generating jump functions in the editor, with each improvement to the information requiring additional analysis.

In the current implementation in ParaScope, we use a modification of the *pass through* scheme described in [CCKT86]. This detects constants when they are directly passed at a call, or when they are passed as parameters through a chain of calls without modification before being passed. As defined, the pass-through method uses analysis within the procedure to determine whether a formal is passed to a call before modification. In order to avoid analysis within a procedure, the implementation instead uses the interprocedural MOD information to detect whether the formal can be modified anywhere in the procedure. While not as precise as the method described in the paper, it was easily added to the implementation, and it catches many constants used to declare bounds of arrays, and loop bounds.

The benefits of constant propagation have been touted throughout this dissertation: for simplifying control flow, improving the results of dependence analysis and enabling inlining by providing array dimension sizes. Overall, the most important benefit of interprocedural constants is the simplification of control flow within a procedure, which can improve the results of all analysis techniques. Constants are also particularly useful in calculating estimates of execution frequency, described later in this section. Additionally, CONSTANT information is useful as input to supplement the results from an intraprocedural constant propagator.

6.3.2 Information Required by Cloning and Inlining

Execution frequency estimates are used to target inlining and cloning for frequently executed portions of the call multigraph. For inlining, *EdgeFrequency(c)* is calculated, which provides an estimate of the number of times call site c is executed. For cloning, *PathFrequency(n)* is calculated for each procedure, which gives the maximum execution frequency for any descendant procedure of n in the call multigraph. Both of these rely on *NodeFrequency(n)*, which is an estimate of the number of times procedure n is invoked.

Frequently executed calls and procedures are important for several reasons. It targets places in the call multigraph with the highest call overhead. Also, the estimate favors calls in loops, an important target of many optimizations. For cloning, high execution frequency estimates only suggest that a path in the program is important. An ideal measure would weight the execution frequency estimate by an estimate of the benefits expected from cloning.

Annotations on call site edges

The orderings required for inlining and cloning are based on execution frequency estimates of call sites in the program. To obtain these estimates, an annotation *EdgeFunction* is added to each edge $(p \to q)$ that describes the number of times q is invoked through this edge whenever p is invoked. For example, if q is called from a loop in p, and the loop iterates 10 times, the estimate at $(p \to q)$ is 10. The following rules determine the annotation on an edge:

1. Edges representing calls that appear in loops in the caller are annotated with an estimate of the number of iterations of the loop.
2. Edges representing conditional calls are given an estimate of $\max(1, (call\ estimate)/2)$.
3. All other edges are given the annotation 1.

The first rule requires further explanation. Accurately estimating the number of iterations of a loop can be difficult. As a simplification, Cooper suggests that we assume that all loops perform 8 iterations [Coo83]. Then, the number of times a call site is executed is 8^d, where d is the nesting depth of the loop in the procedure. This estimate at least takes into account the multiplicative benefits of optimizing a deeply nested procedure call. To support this approximation, the module editor need only determine the loop nesting depth of each call site.

There are many cases where a more precise estimate is possible. Consider the FORTRAN DO loop. Often, the lower and upper bounds, and the step size, are constants. If not, they may be variables that interprocedural constant propagation can determine are constant. To take advantage of this information, the module editor must write out the initial and final values of the induction variable and the step size for each loop in the nest in which a call site appears.

Further improvements are possible with other techniques. For example, in loops that are not DO loops, we can look for *induction variables* [ASU86]. Then, we locate exit branches from the loop and try to determine how many iterations of the loop are required to make the exit condition evaluate to *true*. We can also improve the results of constant propagation using *range analysis* and *range propagation*, which give ranges of variable values rather than a single value [Har77a]. However, since the analysis described in this paragraph must occur in the module editor at the end of an editing session, it is doubtful that the increased precision of the execution frequency estimates will be worth the added cost.

The second point in describing edge annotations, dealing with conditional calls, is designed to make conditional calls less important than calls that are always executed. However, the minimum value of an edge annotation must be 1. This is required so that the frequency estimates preserve topological order, as explained later.

Execution frequency estimates

With the edge annotations, *EdgeFrequency* and *NodeFrequency* are calculated. First, assume that the cycles in the call multigraph have been located, and nodes appearing in a cycle are collapsed into a single node [Zad84]. Secondly, assume that a topological ordering of the nodes in the reduced graph is available.

NodeFrequency(n) is initialized to 1 for the *main* procedure. To deal with the possibility of cycles in the call multigraph (denoting recursion), we locate cycles and eliminate back edges from our consideration. Starting at *main*, the nodes in the reduced graph are visited in topological order so that a procedure is visited before any procedures it invokes. *NodeFrequency* values, once determined, are used to calculate *EdgeFrequency* values. The calculations are as follows:

$$NodeFrequency(n) = \sum_{p \in pred(n)} (NodeFrequency(p) * EdgeFunction(p, n))$$

$$EdgeFrequency(\langle p \rightarrow q \rangle) = NodeFrequency(p) * EdgeFunction(p, q)$$

Frequently executed paths

To locate paths in the call multigraph with high execution frequencies, we use a simple solution that relies on the values of $NodeFrequency$. Again, assume that cycles in the graph have been collapsed into a single node, and that a topological order exists for the nodes.

PathFrequency(l) is initialized to *NodeFrequency(l)* for all leaf procedures l. Then, *PathFrequency(n)* for each procedure is calculated in *reverse* topological order. The equations for *PathFrequency* are as follows:

$$PathFrequency(n) = \max_{s \in succ(n))} PathFrequency(s)$$

From the calculation of *PathFrequency(n)*, we know the highest *NodeFrequency* value for any descendant of n. Nodes with high *PathFrequency* values are on paths leading to frequently executed procedures.

6.4 Batch System for Interprocedural Optimization

Recall from chapter 1 the five phases of the program compiler. In this chapter, we have focused on Phase 3: Planning. This section reviews the five program compiler phases, and overviews the planning phase as discussed in this chapter.

The most important part of the planning phase is the relationship between cloning and inlining, and their relationships to interprocedural analysis. Cloning improves results of forward interprocedural problems and should be performed in topological order. Inlining is performed in reverse topological order. Since cloning can enable inlining, cloning should be performed before inlining. All of these suggests that the planning phase should consist of two phases on the call multigraph. First there is a top-down sweep where cloning decisions are made and forward interprocedural information is updated. Then a bottom-up sweep makes inlining decisions and updates backward interprocedural information. The algorithm is outlined below.

1. Build call multigraph (discussed in chapter 2).
2. Calculate interprocedural information (overviewed in section 6.3).
3. Plan transformations:
   (a) **Cloning**
      i. For each interprocedural problem used as a basis for cloning:
         A. Visit procedures in *PathFrequency* order.
         B. Partition calls to the procedures.
      ii. Incrementally update interprocedural solutions for problems used as basis for cloning.
      iii. Stop when program growth exceeds constraints.

   (b) At this point, if we have not done so incrementally, we propagate new solutions to the remaining forward interprocedural problems.
   (c) **Inlining**
      i. Visit call sites in *EdgeFrequency* order.
      ii. Apply tests to determine value of inlining the call, and mark call site for inlining if worthwhile and meets constraints.
      iii. Incrementally update backward interprocedural information used in inlining decisions.
      iv. Stop when program growth exceeds constraints.

   (d) At this point, propagate new solutions to any remaining backward interprocedural problems.
4. Perform interprocedural transformations.
5. Build executable (to be discussed in the next section).

The algorithm is based on the idea that improvements to forward interprocedural information affect each other, and the solutions to backward interprocedural information. In fact, changes to backward interprocedural solutions may also affect solutions to forward problems. In particular, MOD results can be used to improve CONSTANT information, by formulating jump functions to include values of a variable conditional on whether it is modified at a call site [CCKT86]. Although taking advantage of improved backward solutions to refine forward solutions may seem desirable, this prevents a two phase algorithm on subsequent compilations. The addition of extra passes over the call multigraph causes a significant recompilation problem when trying to evaluate whether the compiled version of a procedure has correct interprocedural information. Thus, initial backward information is used in the solution of forward problems, but forward solutions are not updated due to improvements in backward ones.

6.5 Recompilation Algorithm

In a traditional separate compilation system, only those modules which have been edited since the last compilation need to be recompiled. However, when interprocedural optimizations are performed, dependences between modules arise, potentially causing unedited modules to need to be recompiled. Modules optimized based on interprocedural facts will need to be recompiled if any of those interprocedural facts change. Understanding the impact of changes to interprocedural information is required when interprocedural information is used in optimization. *Recompilation analysis* determines procedures requiring recompilation, attempting to minimize the recompilation requirements. Interprocedural transformations such as inlining and cloning require further consideration.

6.5.1 Recompilation Analysis for Interprocedural Information

Torczon presents three different approaches for minimizing the need for recompilation after interprocedural optimization [Tor85] [CKT86b] [BCKT90]. The method currently used in the program compiler compares interprocedural information from the last compilation of a module with information obtained in the current compilation. For a procedure and its call sites, the MOD, REF, ALIAS and CONSTANT information must either be more precise than the information calculated in the previous compilation, or where it is less precise, the variables involved must not be referenced by the procedure. If these tests fail, the procedure must be recompiled.

By allowing the new interprocedural set to be more precise than its previous value, recompilation is avoided at the expense of lost optimization opportunities. The test of whether or not a variable that represents less precise information is referenced avoids recompilation when the difference could not have affected optimization.

6.5.2 Support for Cloning and Inlining

To analyze recompilation requirements after cloning and inlining, a representation of the program is needed to describe the cloning and inlining that was performed on the previous compilation. This representation must provide sufficient information to reconstruct the cloning and inlining performed on the program and the interprocedural information used to compile the transformed modules.

Inlining is represented with *call trees*. A call tree is a subtree of the call multigraph where all of the elements have been inlined into the root node of the tree. This uniquely describes a sequence of inlining operations. A node representing a call tree is added to the graph, replacing the group of nodes and edges making up the call tree. The representative node is annotated with the call tree it represents.

Special nodes are also added to the call multigraph to represent cloning. When a procedure is cloned, a new node is created to represent the cloned version. The incoming edges in the call multigraph are divided between the new node and the default procedure. A cloned version is annotated with its set of *CloningVectors* and its *StateVector*. Interprocedural data-flow sets are associated with all nodes and edges in the call multigraph, even when they occur in call trees or represent cloned versions.

An auxiliary data structure is required, representing each of the procedures in the original program. For each procedure, a list of its clones and the call trees in which it is included maps a call site in the original program to the appropriate version of the procedure. With this data structure, it is straightforward to locate relevant procedure versions when determining recompilation requirements. This structure is also useful when one version of a procedure is no longer correct with respect to its interprocedural information. If its information matches some other compiled version of the procedure, recompilation can be avoided.

6.5.3 Algorithm

The recompilation algorithm is shown below. It is based on the algorithm from the previous section. Again, a two-phase approach is used. The first phase is a top-down sweep, testing for recompilation requirements based on edits or changes to forward interprocedural problems. During this phase, the call multigraph is updated to include inlining and cloning that is still valid from the previous compilation.

There are really two recompilation tests to validate call trees. The first is, have any of the member procedures been edited? This indicates if the inlining is still valid. The second is, has the interprocedural information for the call tree changed? If so, the inlined source is still valid, but the object code is not. As long as the inlined source is still valid, the version is left in its inlined form, but the call tree node is slated for recompilation.

For call sites that map to versions in the old executable of the program, the recompilation test is applied to the forward interprocedural information. For a cloned version, this may require a test on the value of *StateVector* if the version represents multiple *CloningVectors*. If the forward interprocedural information has changed and is now less precise than on the previous compilation, the procedure is slated for recompilation. The backward interprocedural information is not tested, since the second phase of the algorithm may make the information in the new call multigraph more precise. The compiler assumes it is more likely that the information remains the same than that it becomes less precise. This is because the recompilation algorithm is intended to be used only after small changes to the program.

During the first phase, a call site invoking a procedure version that is not up to date may be mapped to a different compiled procedure version. The call site is not up to date if either it did not appear in the previous version of the program or it has less precise forward interprocedural information than on the previous compilation. It can be mapped to another compiled version if the interprocedural information at the call site is more precise than the information of some compiled version. In this case, both the forward and backward interprocedural information is tested. Since the call site was not mapped to the compiled version in the previous compilation, there is no reason to believe that the backward interprocedural information will approximate the compiled version's after the second phase of the algorithm. After locating such opportunities, any remaining call sites require recompilation. For these, cloning opportunities are considered.

The second phase is a bottom-up sweep, evaluating recompilation needs based on backward interprocedural problems. At the same time, inlining decisions are made using the same heuristics as in the batch algorithm. However, inlining is only allowed if the caller is slated for recompilation. Further inlining is permitted into already inlined source in those cases where the source is still valid but the object code is not. Finally, the procedure versions marked for recompilation are compiled.

1. **Analyze current version of program:**
   (a) Build the call multigraph based on the new version of the program.

   (b) Calculate interprocedural information for the new call multigraph.

2. **Forward pass over call multigraph:**
   For each procedure p in *PathFrequency* order:

   (a) For each call tree ct, where p is the root, and one of the members of ct has been edited since the last compilation (possibly p):

      i. Mark ct as invalid.

      ii. Delete source and compiled versions on ct from database.

   (b) If p has been edited since the last compilation, goto step 2g.

   (c) For each call site c invoking p, match c to its version of p in the old call multigraph if a valid version exists and the new forward interprocedural information for c is more precise than the old information.

   (d) For any unmatched call site c invoking a valid call tree, replace the corresponding portion of the new call multigraph with a node representing the call tree and mark as requiring recompilation.

   (e) For unmatched call sites c invoking p, match c to some version of p in the old call multigraph if *both* the forward and backward interprocedural information for c is more precise than the information for the old version.

   (f) For any call site matched to a call tree in 2c or 2e, replace the corresponding portion of the new call multigraph with a node representing the call tree.

   (g) For remaining unmatched call sites c invoking p, evaluate cloning decisions for each problem used as a basis for cloning, if program growth constraint has not been exceeded. For each cloning opportunity, add a node representing the clone to the new call multigraph and reassociate the edges representing the call sites invoking this clone. Mark the node as requiring recompilation.

(h) The rest of the unmatched call sites are grouped together, representing the default version of the procedure. The node in the new call multigraph representing the default procedure is marked as requiring recompilation.

Propagate in the new call multigraph the forward interprocedural solutions from each node representing p.

3. **Reverse pass over call multigraph:**
For each node n in *NodeFrequency* order:

(a) If the node is matched to some version in the old call multigraph, mark as requiring recompilation if the backward interprocedural information for the node is less precise than the version to which it is matched in the old call multigraph, and delete the compiled version in the database.

(b) Evaluate inlining for each caller of the node marked as needing recompilation, keeping within program and procedure growth constraints. Perform inlining where indicated, and update the new call multigraph to reflect the inlining.

Propagate in the new call multigraph the backward interprocedural solutions from the node if it has not been inlined at all calls.

4. **Compilation:**
Compile all nodes in the call multigraph marked as requiring recompilation.

6.5.4 Incremental Updates to Interprocedural Information

Incremental Updates During Planning Phase

During the forward and backward passes over the call multigraph, interprocedural information is updated to reflect refinements. To do this efficiently requires an incremental algorithm for performing the analysis. The design of the two-phase algorithm is amenable to efficient incremental updates to the interprocedural information.

The incremental updates are based on Marlowe's algorithm for incremental dataflow analysis [MR90] [Mar89]. This algorithm requires that the graph be maintained in topological order, and that the strongly connected components, or loops, be located. Topological ordering guarantees that during a single update, the information is not propagated to a particular node more than once. Locating loops is necessary to guarantee correctness of the result.

In the batch version of the algorithm, information is propagated iteratively within the strongly connected components and summarized at their head nodes.[17] Then the information is propagated in topological order on the call multigraph. The incremental algorithm is similar, first updating information within strongly connected components, and then propagating information, where needed, in topological order.

With this algorithm, the types of incremental changes that are expensive are those that change the strongly connected component structure, change the topological order or both. The types of changes during cloning and inlining are not likely to have these effects. Inlining results in edges deleted from the call multigraph, but cannot eliminate edges within a strongly connected component unless the corresponding recursive cycle is completely unrolled. With cloning, nodes and edges may be added to the graph, but cannot affect the topological ordering. However, as a result of cloning, newly exposed constants could simplify control flow sufficiently to eliminate call sites and thus break up a recursive cycle. In general, it is very unlikely that inlining or cloning will result in an expensive incremental update.

The topological ordering and strongly connected components needed by the algorithm were already being used in other aspects of the cloning and inlining process. Also, since cloning is performed in topological order, incremental updates to the information can be made in such a way that the changes at a node are accumulated until all its predecessors have been visited. Then, only a single update occurs at each node. The same is true for updates to the backward interprocedural solutions. Inlining occurs in reverse topological order, and changes can be accumulated at a node until it is visited. Note that we do not update backward interprocedural information during the cloning phase, nor do we update forward interprocedural information during the inlining phase.

Batch Analysis Between Compiles

We could also use incremental interprocedural analysis between compilations. However, there is a distinct difference between the controlled types of changes occurring during cloning and inlining and the unpredictable changes occurring from program edits.

[17]The iterative solution within the strongly connected components allows the algorithm to work even for unstructured code.

Even if we attempt to calculate the information in topological order, the call multigraph could change radically, affecting the topological order. There is no way to guarantee that the amount of work done with incremental analysis is less than doing batch analysis. So, interprocedural information between compiles is calculated in batch, especially since we have nearly linear time algorithms for the call multigraph, MOD, REF and ALIAS problems.

6.6 Related Work

This section focuses on related work in the area of managing the relationship between procedures in a program, since work related to other aspects of this chapter have already been covered. The Unix *make* facility by Feldman was perhaps the first tool that managed the relationship between procedures in a program [Fel79]. The *make* facility allows a programmer to specify the components of a program, the actions required to build the program, and the dependences among the components. Modification to a component is enough to force rebuilding of components that depend upon it.

Tichy and Baker considered a finer granularity for establishing dependences among components [TB85]. Their method determines the portions of a component that are shared by other components: definitions, declarations and constants. This is basically the interface to the outside. Then modification of a component can only force recompilation of other components if the interface changes. Specifically, a *reference* set is built for each component C that describes information used in C that is defined elsewhere. During module editing, a *change* set is generated that describes changes to the interface. Recompilation for a component C is indicated if the intersection of the *reference* set for C and the *change* set for a component upon which C depends is non-empty.

Müller developed an algorithm similar to Tichy and Baker's for the Rigi software development environment [Mul86] [MHK86]. Rigi is designed to support code sharing across programmers and projects. The *global interface analysis algorithms* are designed to detect recompilation requirements across a system of programs based on imported and exported interfaces of the components.

The research following the introduction of the *make* facility freed the programmer from specifying the dependences among procedures and decreased the amount of recompilation required as a result of an editing change. ParaScope drew from these ideas but extended them to manage optimization across procedure boundaries. This is unusual, even for compilers supporting interprocedural optimizations. Compilers such as the Modula 2 compiler and the Gnu C compiler provide inlining facilities, but they require the programmer to manage the compilation dependences of a procedure on its inlined call sites. The Cray FORTRAN compiler manages dependences when it performs automatic inlining.

The recompilation algorithm in this chapter extends the previous algorithms, accommodating incremental inlining and cloning [Tor85] [CKT86b] [BCKT90]. Torczon suggests that inlining and cloning be part of the program compiler design, but does not provide an algorithm for determining if inlined or cloned versions require recompilation. A representation of inlining and cloning similar to the one in this chapter is described in [BCKT90]. However, the recompilation algorithm does not consider how to perform inlining and cloning on a partially compiled program, only how to determine if an inlined or cloned version is up to date.

6.7 Chapter Summary

The main contribution of this chapter is to describe how inlining and cloning can be incorporated into the ParaScope compilation system, while maintaining a separate compilation system. The chapter considers the interaction among the optimization techniques and presents an algorithm designed to exploit these interactions.

The algorithm for inlining and cloning has two phases. A top-down sweep of the call multigraph occurs first, making cloning decisions and updating forward interprocedural information. A bottom-up sweep follows which makes cloning decisions and updates backward interprocedural information. Cloning and inlining are performed, and the compilation units are optimized based on the updated interprocedural information.

This chapter has also considered recompilation, and the algorithm has been designed to accommodate recompilation. A representation of the program was described which recreates the cloning and inlining applied to the program. The algorithm attempts to match call sites to their implementations in the previous compilation. Additional cloning and inlining is only performed when recompilation is needed.

Chapter 7

Interprocedural Optimization for Parallelization

Previous work on interprocedural optimization for parallelism has focused on inline substitution and interprocedural analysis of array side-effects. Even though array side-effect analysis and inlining are frequently successful [Hus82] [HK91], each of these methods has its limitations. Considerations of compilation time and space require that array side-effect analysis summarize information about accesses. In general, summary information is less precise than the analysis of inlined code. On the other hand, inlining can yield an explosion in code size while disastrously increasing compile time and seriously inhibiting separate compilation. Furthermore, inlining can sometimes cause a loss of precision in dependence analysis, due to the complexity of subscripts that result from array parameter reshapes. For example, in the SPEC benchmark matrix300 described in Chapter 4, the dimension size of a formal array parameter was also passed as a parameter. The translation of references to the formal after inlining introduced multiplications of unknown symbolic values into subscript expressions.

This chapter introduces a hybrid approach to interprocedural optimization that overcomes some of these limitations. Array side-effect information is used to locate opportunities for parallelizing transformations across procedure boundaries.[18] These transformations move a small amount of code across procedure boundaries, and the effects of the transformation are annotated in the call multigraph. This yields many of the benefits but few of the costs of inline substitution. Code growth of individual procedures is nominal. Overall program growth is moderate since multiple callers can invoke the same transformed procedure. In addition, compilation dependences among procedures are reduced since the compiler controls the small amount of code movement across procedures and can easily determine if an editing change of one procedure involved in an interprocedural transformation invalidates other procedures.

[18] Although the chapter focuses on optimizations for multiprocessors, the same transformations would be useful for vector uniprocessors or distributed memory multiprocessors.

The effects of inline substitution on parallelization are also considered. From the inlining study of Chapter 3, we observed that although inlining eliminated call sites from loops, the resulting loops were often not parallelized. We suggest some optimizations to further reduce dependences in inlined loops, and describe an experiment to determine the effectiveness of the optimizations in increasing parallelism. The chapter also describes a form of array side-effect analysis called *regular section* analysis [CK88a] [HK91].

This chapter provides an overview of interprocedural optimization for enhancing parallelization. It is organized into five sections. The next section describes interprocedural information required to perform dependence analysis of loops containing procedure calls. Section 7.2 demonstrates how a few important parallelizing transformations can be performed across procedure calls, providing a framework for interprocedural transformations. Support for the extensive analysis required by this approach necessitates modications to the ParaScope compilation system. These modifications are described. In Section 7.3, we discuss inline substitution in the context of parallelism. The bulk of the section describes some optimizations that break dependences after inlining. We present experimental results from applying these transformations to the programs from the inlining study. Section 7.4 presents related work, mostly regarding techniques for summarizing array subsections. The chapter closes with a summary of important points.

7.1 Interprocedural Analysis for Parallelization

7.1.1 Scalar Interprocedural Information

The interprocedural information described in Chapter 6 is useful for parallelization. In particular, discovering interprocedural constants used in loop bounds, array dimensions or subscript expressions may improve dependence analysis. Constants describing loop bounds also aid the code generator in determining whether parallelization is profitable. Scalar MOD can be of help in dependence testing when scalars appear in subscript expressions. If a such a scalar is not modified at any call sites within a loop, dependence tests may be able to determine that the scalar is loop-invariant. This enables symbolic dependence tests. Finally, KILL information can eliminate scalar dependences.

Information provided by scalar interprocedural analysis is useful in reducing dependences. However, by itself scalar interprocedural information is too coarse to effectively increase parallelism in loops. Scalar analysis treats arrays as single units, marking an entire array as used or modified for each reference. This deficiency with scalar interprocedural information has motivated techniques for summarizing the effects of procedure calls on array subsections.

7.1.2 Array Side-Effect Analysis

With more precise information about the portion of an array accessed in a procedure, dependences may be pruned from the dependence graph. At the end of this chapter, we describe a number of techniques designed to provide information about array side-effects. We base our transformations on *regular section descriptors* (RSDs) [CK88a] [HK91].

Regular sections describe side-effects to a few important substructures of arrays: single elements, rows, columns, grids and their higher dimensional analogs. The restriction to a few shapes makes the implementation efficient as compared to other techniques. These shapes form a lattice, which allows formulation of array section analysis as a data-flow analysis problem.

An RSD consists of a variable name and a representation of each dimension. Each dimension is described in one of three ways: an invocation invariant expression (representing a single element); a range consisting of a lower bound, an upper bound and a step size; or $\perp$, signifying that the entire dimension is affected. Like scalar side-effect information, the regular sections are separated into modified and referenced sets.

Two-Phase Analysis

As in the scalar interprocedural analysis approach, gathering regular section information is separated into two phases. The local phase, performed at the end of an editing session, locates array references. Subscript expressions are examined for each array reference. A regular section is constructed based on the loop induction variables, constants, global variables or parameters that appear in the subscript expression. For subsequent references to the same array, the sections representing the two references are merged, based on the lattice meet function. In the interprocedural phase, the regular sections are propagated over the call multigraph. Scalar MOD and CONSTANT information is calculated first in order to refine RSDs based on parameters and global variables.

An important aspect of the propagation phase is renaming from formal parameters to actual parameters across a call site. Just as with inlining, this can be difficult if the programmer has reshaped the array at the call so that dimension sizes are different. We could linearize the subscript expressions [BC86], reducing all arrays to a single dimension. However, this yields very complicated subscript expressions that may greatly hamper the ability of the dependence analyzer in disproving dependences, especially since dependence analysis techniques are more successful on simple subscript expressions.

Instead of linearization, the same rules used in Chapter 3 for inlining govern array renaming. The actual must have at least as many dimensions as the formal. Also, for a formal of k dimensions and an actual of l dimensions, the size of dimensions $l-k+1$ to $l-1$ in the actual must match dimensions 1 to $k-1$ in the formal. This makes the renaming fairly straightforward. Otherwise, linearization may be performed as a last resort.

Dependence testing with RSDs

Locating dependences on procedure calls is very much like dependence testing on ordinary statements. The RSD information for the call consists of a list of modified and referenced subsections. The modified and referenced subsections appear to the dependence analyzer like the left- and right-hand sides of an expression, respectively. For single element subsections, dependence testing is the same as it would be for any other variable access. For subsections that contain one or more dimensions with ranges, the dependence analyzer simulates DO loops for each of the range dimension. The lower bound, upper bound and step size of each loop are derived from the range of the corresponding dimension.

Information at each loop

One of the transformations described in the next section requires RSDs for each outer loop of a procedure. This information can be gathered during the first phase of analysis, in the same way as for the entire procedure.

However, propagation of these RSDs is not required. We are only interested in translating the loop information for a procedure to its callers, based on the parameters passed at the call. There is no need to propagate this information all the way up the call multigraph. Whenever a caller examines the RSDs for loops in a procedure it calls, it can then do the translation from formal parameters appearing in the RSDs to the actual parameters at the call.

7.2 Interprocedural Transformations

This section demonstrates how to use RSD information to locate opportunities for parallelizing transformations that are legal across procedure boundaries. We introduce a new transformation, *loop embedding*, that moves a loop header into a procedure that is invoked within the loop. Following loop embedding, we can perform intraprocedural transformations on the loop, since we have eliminated the call site in the loop. In this section, we consider loop embedding and *loop distribution*, in order to enable *loop permutation* of a loop nest containing a procedure call. This group of transformations is not complete, but by describing these, we develop a framework for efficient interprocedural transformation.

This section describes each of the transformations in the context of a compilation system and a transformation framework. To provide the needed background, the phases of the program compiler are described below. Following this, we describe the transformation framework.

7.2.1 Program Compiler

As described in Chapter 1, the program compiler consists of six phases: (1) building the call multigraph, (2) computing interprocedural information, (3) performing dependence analysis, (4) planning transformations, (5) performing transformations, and (6) creating the program executable. We now describe the activities of the first five phases as it relates to interprocedural parallelizing transformations.

Program Representation. We augment the call multigraph to contain information about the interprocedural loop nesting of the program. Special *loop* nodes and *nesting* edges are added to the graph. Nesting edges emanate from loop nodes. Their targets are loops nested and procedures invoked within the current loop.

Interprocedural Analysis. RSD information is constructed as described in the previous section. Regular sections are computed for each procedure and for each outer loop in a procedure. These are translated across the call to variables in the caller's name space. Nesting edges and call site edges in the augmented call multigraph are annotated with their regular section information.

Dependence Analysis. Dependence analysis is performed in procedures with the benefit of the RSD information. Analysis occurs in a separate pass from code generation so that the results of dependence analysis can be used to determine the safety and profitability of interprocedural transformations. Dependence analysis also marks parallel loops in the augmented call multigraph. We restrict dependence analysis (and optimizations requiring dependence analysis) to those procedures that require recompilation, thus limiting the extent of the analysis.

Planning and Transformation. At this point, we can determine the safety and profitability of interprocedural transformations. Targeting profitable interprocedural transformations is particularly important for parallelization since unnecessary optimization can lead to performance degradation and significant compile-time costs. By separating this phase from dependence analysis, we avoid the compilation order dependences associated with using dependence information to make decisions about multiple procedures.

The flow of information among the phases is depicted in Figure 7.1. Each step adds annotations the augmented call multigraph that are used by the next phase.

7.2.2 Transformation Framework

The methods used to perform these transformations indicate a framework for interprocedural transformations based on RSD information and dependence analysis. For each optimization, we need to answer the following questions:

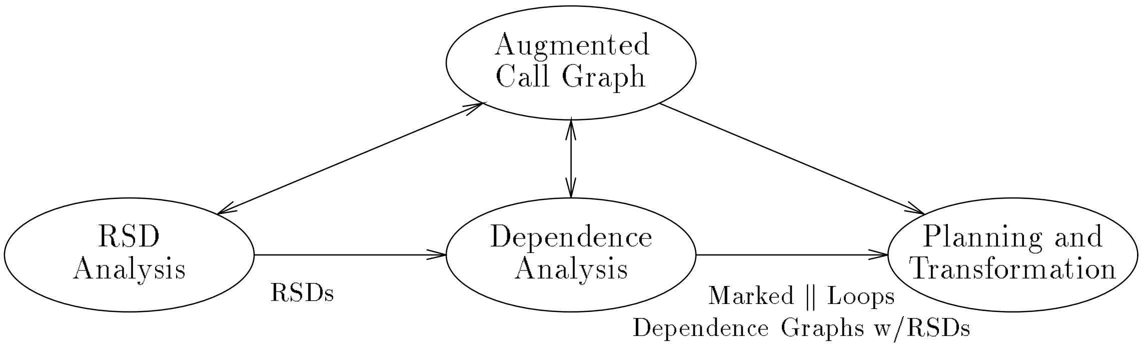


Figure 7.1 Flow of information for interprocedural transformations.

- **When is the transformation safe?**
  Determining safety usually takes place during dependence analysis. Wherever call sites occur, RSDs are used to stand in for the called procedure.

  An important aspect of safety pertains to optimizations that require dependence tests over multiple loops. We derive tests that can be performed a single loop at a time, with requisite interprocedural information saved to enable merging the individual results for each loop into a single result. By developing tests that can be applied to loops independently, we avoid any compilation order dependences.

- **How is the transformation performed?**
  The technical issues involved in performing a transformation need to be addressed. In each case, the benefits of the optimization are weighed against its costs, where costs include code growth and compilation dependences.

- **What tests need to be performed during recompilation to ensure transformations are still valid?**
  Recompilation is required if the tests that indicated safety of the transformation are no longer true. However, these tests are usually based on dependence analysis. Given that dependence analysis may be expensive, it should be avoided, especially for procedures that are not expected to require recompilation.

  For this reason, RSDs are used as an initial test before dependence analysis. If the procedures involved in a transformation have not been edited and the RSD information for relevant call sites has not changed, then there can be no additional dependences that invalidate the transformations. In fact, even if the RSD information has changed, as long as it has not become larger (i.e., as long as there are no additional sections, and existing sections do not have additional elements), no new dependences can exist. However, if this test on RSDs fails, dependence analysis must be repeated. Since RSDs are not precise, it is possible that the test on RSDs will fail, but dependence analysis will prove that no new dependences exist.

- **How can transformed procedures be used by multiple callers?**
  It is useful to describe the transformations performed on a procedure with annotations on the call multigraph. This information is important after program changes to determine whether the transformations are still valid.

This description of the procedure can also be used in the cloning process. It should provide other callers of the procedure with enough information to detect whether the transformed procedure is also valid for their call. Cloning can then be performed by matching this information across all the callers of a procedure. In this way, the annotations can be thought of as special *CloningVectors*.

We address each of these issues as we present the transformations in the rest of this section.

7.2.3 Loop Embedding

When a call site occurs within a loop, parallelization may be hampered, even with accurate knowledge available about the interprocedural side-effects of the call. For example, consider the following pair of procedures:

```
procedure p(a, b)
    dimension a(100), b(100)
    do i = 1, 100
        call q(a, b, i)
    enddo
end /* p */

procedure q(x, y, k)
    dimension x(100), y(100)
    x(k) = y(k) + k
end /* q */
```

In the above example, the loop in p containing the call to q could be vectorized if q was inlined into p. However, even with knowledge of the side effects of q, the compiler cannot vectorize the loop, because vectorizing a call does not make sense to the code generator. Also, since the vectorization of the statement in q is dependent on the calling environment, it is possible that other calls to q will exist in the program where vectorization is not appropriate.

Of course, we could inline all calls to q when inlining makes vectorization of the statement in q possible. Alternatively, we can move the loop header across the call and into q. This transformation can sometimes achieve the same benefits as inlining into a loop, but it may not increase the code size, and it induces fewer compilation dependences among procedures.

We present two methods for loop embedding. The first method is always possible, requires little analysis, but makes reuse of the resulting procedure by other calls unlikely. The second method promotes the possibility of reuse, but is not always possible, and requires more expensive analysis to determine safety. Both solutions introduce fewer compilation dependences than inlining, with the second solution preferable to the first on this point. Also, both solutions reduce call overhead, but may instead add other types of overhead.

Loop Embedding: General Solution

Safety. The most obvious way to move a loop header into a procedure is to move the entire loop body into the called procedure along with the header. Then, the order of statement execution in the loop body is preserved, and we are guaranteed that it is always possible.

However, there should be one restriction on this transformation. Suppose a loop in some procedure p contains calls to two procedures q and r. Then if the loop header is moved into q, there is now a new call $(q \rightarrow r)$. By further moving the loop header into r, this introduces compilation dependences among p, q and r, almost as if the calls to q and r were inlined into p. Moving a call into another procedure also changes the calling structure of the program, which could consequently change the solutions to interprocedural problems. For these reasons, loop embedding should not be allowed when multiple calls appear in the loop.

Mechanics. With this method, we must deal with the following problems associated with renaming variables moved from the caller to the callee:

- Eliminate actual parameters that vary in the loop body, since their values at the point of the new call will be unknown. Add statements to the callee to calculate their values. If it is the case that a subscript expression of an actual array parameter contains the loop induction variable, the entire array is passed as the parameter, and subscripts in references to the corresponding formal parameter are updated in a manner similar to inlining.
- Ensure that global variables accessed in the loop are within the scope of the called procedure.
- Pass as parameters to the called procedure any local variables of the caller that are accessed within the loop and are visible outside the loop.
- Create local variables in the called procedure corresponding to locals of the caller that are accessed within the loop but are not visible outside the loop.

Recompilation. Consider how an edit in the calling procedure affects the called procedure. Without explicit knowledge that the loop where the call appears is not changed, the code movement into the callee needs to be performed again and the callee requires recompilation. On the other hand, if the callee is edited, we only have to repeat the code movement into the callee. The caller is unaffected, and the caller does not require recompilation. This optimization is less costly than inlining because it does not induce a dependence from the callee to the caller.

Cloning. The procedure resulting from this transformation is specialized for the particular loop in which it is called. It is therefore unlikely that any of the other callers can invoke this modified version. Even if the code in the loop is identical to that in some other caller, the compiler is not able to detect this. Thus, the effect of this optimization on code growth is essentially the same as with inlining. Each time it is applied, we are creating an additional copy of the procedure body.

Additional overhead. The overhead associated with this transformation has two sources. The first is that we may be passing more parameters at the call. However, after optimization the call executes only once, rather than once per loop iteration. With a sufficient number of loop iterations, the cost of passing the extra parameters should be less than the call overhead being eliminated.

The other source of overhead is the increased name space in the callee. Local variables from the caller have been added, either as parameters or as new locals. We have also added globals from the caller. It is difficult to determine if this will have any appreciable effect on run-time.

Loop Embedding: A Solution Enabling Cloning

The second solution is designed to enable sharing of the transformed procedure by multiple callers. It also has the added advantage of reducing compilation dependence from caller to callee.

This time, the call is isolated from any other statements in the loop body. That is, if the body of the loop looks like $\langle S_1;\ \mathbf{call}\ q;\ S_2\rangle$, where S_1 and S_2 are sequences of statements, it must be possible to separate the loop into three loops: one containing $\langle S_1\rangle$, one containing $\langle \mathbf{call}\ q\rangle$ and one with $\langle S_2\rangle$. This transformation is called *loop distribution* [KKL+81].

Safety. This version of loop embeddings is safe if loop distribution around the call is legal. Loop distribution must preserve the dependences in the loop. This is true as long as statements involved in a dependence cycle remain within the same loop.

The test for loop distribution can be performed whenever an opportunity for the transformation is located. The dependence graph is traversed for the loop containing the call. During the traversal, we look for cycles in the dependence information for the loop. For each cycle, if the call is included in the cycle along with additional statements, then partitioning the call into a separate loop is not safe.

Mechanics. The goal is to isolate the call from the rest of the loop body in order to move only the loop header into the called procedure. For a FORTRAN DO loop, the loop header consists of three pieces of information: the lower bound of the induction variable, the upper bound, and a step size. If any of these are parameters or local variables of the caller, they must be passed as additional parameters at the call.

Although loop-variant variables cause cycles in the dependence graph, there is one way in which loop-variant parameters can be passed at the call site. Suppose that functions of the induction variable are passed as parameters to the call. Since the value of the function cannot be computed at the call, it must be copied into the called procedure with the loop header. Again, the call representation is annotated with information about the parameter and the function.

Recompilation. Since the caller evaluates the legality of the optimization based on the RSDs at the call site, the recompilation test must verify that the RSDs have not changed in a way that suggests that dependences have been introduced. Changes to the RSDs at the call site (caused by edits to the called procedure or to one of its descendants in the call multigraph), must not enlarge or add new sections that were not found in the previous compilation. Similarly, other call sites in the loop must not have RSDs larger than those found in the previous compilation. If the RSDs have not grown larger, there can be no new dependences on this call site, so the optimization is still valid. This is true even if the called procedure has been edited.

However, if RSD information suggests that new dependences exist, it is possible that the optimization is no longer legal. Safety analysis must then be performed in the caller, to determine whether dependences exist which prevent loop distribution. If no such dependences exist, then the optimization still is safe, and the caller does not require recompilation.

Editing changes to the caller may also invalidate the safety of the optimization. In this case, dependence analysis must be performed on the caller since it is being recompiled. At this time, the safety of the optimization can be determined. Further tests are needed to verify that the call matches the annotations describing parameters that are functions of the loop induction variable. Thus, the callee is only affected by changes in the caller that either invalidate the legality of the optimization or change the annotations.

Cloning. Two callers can invoke a transformed procedure under the following circumstances:

1. The transformation is legal in both callers.
2. They have the same annotations describing the actual parameters that are functions of the loop induction variable.

Additional overhead. This optimization takes a single loop and turns it into as many as three loops. Within this loop, a procedure call is eliminated. Thus, we are trading decreased call overhead for increased loop overhead. Their relative costs depend on a number of factors, including whether the loop is parallel or scalar, and how many iterations it executes.

7.2.4 Interprocedural Loop Permutation

The primary goal in loop-based parallelization is to parallelize an outermost loop. A large amount of code can then be executed in parallel without having to synchronize parallel processes. However, an outermost loop cannot be parallelized if it *carries* a dependence [AK87]. A particular loop carries a dependence if the references of the source and sink of the dependence occur in different iterations of the loop.

Even if the outermost loop cannot be parallelized, it may be possible to move an inner parallel loop to the outermost position. This transformation is called *loop permutation* [Ban90], a generalization of *loop interchange* [AK84] [AK87] [Wol86] [Wol89]. To perform loop permutation, the dependence analysis phase looks for loops that do not carry any dependences. These loops will be candidates for moving to the outermost position.

This section presents two versions of loop permutation. First, we address the problem of perfectly nested loops. These are loop nests in which the only statements appear in the innermost loop. Then the transformation is expanded to consider imperfectly nested loops. The technique for perfectly nested loops is much simpler. However, it is unlikely that perfectly nested loops will occur often when the loops cross procedure boundaries.[19] For this reason, it is important to consider imperfectly nested loops.

Loop Permutation for Perfectly Nested Loops

We begin with an example of interprocedural loop permutation. Consider the following pair of procedures:

```
...                          procedure q(A,i,j)
do i = 1,n                       do k = 1,n
    do j = 1,n                       A(i,j,k) = A(i-1,j,k) + A(i,j-1,k)
        call q(A,i,j)            enddo
    enddo                    end /* q */
enddo
```

Because the i and j loops carry the dependence on $C(i,j)$, neither can be run in parallel. The k loop carries no dependence, and so it can be moved to the outermost position as in the following:

```
...                              procedure q(A,i,j,k)
do k = 1,n                           A(i,j,k) = A(i-1,j,k) + A(i,j-1,k)
    do i = 1,n                   end /* q */
        do j = 1,n
            call q(A,i,j,k)
        enddo
    enddo
enddo
```

Safety. In terms of preserving the dependences in a loop, it is legal to move a loop outward if for all dependences in the loop, the source and sink of the dependence occur on the same iteration. Most loops that do not carry a dependence can be moved outward. There are also some requirements on the lower bound, upper bound and step size of the loop. It must be possible to perform the translation from variables in the original procedure to variables in the procedure to which the loop header is moved. To this end, these values must either be constants, globals in the scope of

[19] In fact, experiences from the experiment described in Section 7.3 indicate that procedures often contain initialization and testing code at the beginning.

the procedure to which the loop is moved, or parameters that can be mapped to a name in the procedure to which the loop is moved. Furthermore, the variables in the bounds must not be modified along the chain of calls from their new position to their original one. Possible exceptions include loop bounds based on induction variables of an outer loop. Permutation of these triangular and trapezoidal loops requires a slight extension to the transformation described here [Wol86].

Mechanics. During dependence analysis, loops that do not carry any dependences are marked. To determine whether a loop containing a procedure call carries a dependence, RSDs represent the called procedure.

When selecting loops for loop permutation, the compiler examines a chain of procedures making up a loop nest. Starting at the procedure containing the outermost loop, each procedure in the nest is considered, looking for a loop that has been marked by the dependence analysis phase. When such a loop is located, the loop bounds are considered to determine if it is legal to move the loop to the outermost position.[20]

With a loop that is legal to move outward, the transformation is simple. The loop is extracted from the procedure and added to the procedure containing the outermost loop in the nest. Variables in the loop header are translated as needed to names in the new scope. If needed, the induction variable of the extracted loop is passed as a parameter down the chain of calls to the original procedure.

These changes essentially only affect the two procedures representing the original and final locations of the loop. The only potential change in the intervening procedures in the call chain is the addition of an induction variable to the parameter list. However, even if no textual changes are made to the intervening procedures, they are still affected by the transformation. This is because the intervening procedures directly or indirectly invoke an altered version of the procedure formerly containing the parallel loop. If they have callers outside of this call chain, the altered version of the procedure will not be correct. For this reason, cloned versions of the intervening procedures will be required if they can be invoked by callers outside the call chain being optimized.

[20] A more general solution would attempt to move the loop to some position other than the outermost if loop bounds proved to be illegal. Then, the code generator would have to select among the possible parallel loops, locating the one which could be moved to the best position in the nest.

Recompilation. The recompilation test involves the procedure formerly containing the parallel loop, the procedure containing the outermost loop, and all intervening procedures in the call chain.

If the procedure formerly containing the parallel loop has not been edited, the loop is guaranteed to still be parallel if RSDs for all call sites within the parallel loop have not grown larger. The conditions for the loop bounds must also be verified since these can become invalid even if the procedure has not been edited. If the procedure has either been edited, or the RSDs have changed, we must perform dependence analysis on the procedure. If the same loop is still parallel and conditions for the loop bounds have not changed, then the optimization is still valid. This procedure may need to be transformed and recompiled, but the other procedures will not be affected.

If the procedure to which the loop has been moved has not been edited, it only needs to be recompiled if the transformation is no longer valid. If the transformation is still valid, but the procedure has been edited, the outer loop needs to be added and the procedure recompiled. Intervening procedures in the call chain require changes if they have been altered by the transformation, and it is no longer valid.

Cloning. A procedure containing a parallel loop that has been moved out can be reused by other callers if it is legal to move the same loop out of the procedure. This requires an annotation on the call to the procedure that specifies the loop that has been eliminated.

The modified intervening procedures in the call chain can also be shared by other callers if they lead to the same transformed procedure. To do this, annotations are added to the cloned version that describe the call chain in the call multigraph that is being optimized, of which this procedure is a part. Two callers can share the modified procedure if they are both part of an optimized call chain. However, the tails of the optimized paths, from the modified procedure to the end of the chain, must be the same. Opportunities for sharing can be located by applying the minimization phase of the cloning algorithm from Chapter 5, which recognizes and merges equivalent cloned versions.

Permutation for Imperfect Loop Nests

Consider what must be done if the loop is not perfectly nested. In the following example, the inner k loop is parallel, but the i and j loops are not:

```
...                                procedure q(A, i, j)
do i = 1, n                            do k = 1, n
    do j = 1, n                            A(i, j, k) = A(i − 1, j, k) + A(i, j − 1, k)
        A(i, j, i + j) = 0             enddo
        call q(A, i, j)            end /* q */
    enddo
enddo
```

To be able to move the k loop outward, we first have to distribute the j loop around its two statements. At this point, we can either interchange the j and k loops, or distribute the i loop around the revised j loops. The result of doing the former looks like this:

```
...                              procedure q(A, B, C, i)
do i = 1, n                          do k = 1, n
    do j = 1, n                          do j = 1, n
        A(i, j, i + j) = 0                   A(i, j, k) = A(i − 1, j, k) + A(i, j − 1, k)
    enddo                                enddo
    call q(A, i, j)                  enddo
enddo                            end /* q */
```

In the above example, the j loop is distributed around the the call site. Then the j loop is embedded in the callee. Finally, the k loop is interchanged with the j loop. Although we could have interchanged the k loop outside the i loop quite easily in this example, it would not have been so simple if q contained statements other than the single loop.

The discussion of loop permutation on an imperfectly nested loop only deals with a subset of the possible combinations of caller and callee. We restrict consideration to distributing a single inner loop in the caller, and interchanging the resulting loops with outermost loops in the callee. (Another approach to imperfect loop interchange avoids this distribution step. The class of loop nests for which interchange is applicable is different than the one for this version [Wol86].) This transformation is not performed along arbitrary chains in the call multigraph, but only across a single call. Following the presentation of this transformation, we explain why these restrictions are necessary.

Safety. On imperfect loop nests, the test for safety is divided between analysis in the caller and analysis in the called procedure. In the called procedure, just as with perfect loop nests, parallel loops are marked during dependence analysis. The loop bounds and step size must be non-varying within the caller's loop. However, if these were varying in the caller's loop, it would cause a dependence cycle, which would be caught in a subsequent test for safety of loop distribution.

In the calling procedure, the safety of loop distribution is considered. It must be legal to create a partition of the loop containing only the call site. If the called procedure contains multiple loops, it must also be legal to form partitions surrounding each one of the callee's loops. Testing for legality of loop distribution could also performed during dependence analysis. However, since traversing the dependence graph looking for cycles can be expensive, it is preferable to only consider this transformation when the loop in the caller is not parallel, but the caller invokes a procedure with parallel loops.

Test the safety of loop distribution within the callee requires RSDs for each of the loops in the callee, rather than a single RSD set for the call site. If distribution is legal for any loops in the callee, the representation of the call is annotated with the list of loops where distribution is legal.

Mechanics. This transformation is worthwhile when the only parallel loops in the nest are inner loops. If this is the case, the safety of loop distribution in the caller is evaluated. If such a combination is found, the loop header containing the call is moved out of the caller. Prior to this step, it may be necessary to distribute the loop in the caller around the call site. The parameter list in the call site may need to be adjusted, as in Section 7.1 when moving the loop header across the call.

Within the callee, the loop header from the caller is added. Then, we distribute the loop around inner loops wherever distribution is legal and the inner loop is parallel. Finally, for each portion of the distributed loop, the parallel inner loop is interchanged with the outer one.

Recompilation. This validity of this optimization relies on three factors: (1) parallel loops must exist in the called procedure; (2) loop embedding into the called procedure must be legal; and, (1) distribution of the loop in the caller around loops in the callee must be legal. The test for recompilation must verify all three criteria.

If RSDs for any call sites within the inner loop of the caller have grown larger, it is possible that loop distribution around the call site and around the loops in the callee is no longer legal. The test for safety of loop distribution must be applied to determine whether the changes in RSDs affect the validity of the transformation. The callee only requires recompilation if the distribution of the inner loop of the caller is no longer legal.

If any of the RSDs for the loops in the callee grow larger, then dependence analysis must be repeated on the callee to verify that new dependences have not invalidated the optimizations. The safety of loop distribution of the caller's inner loop must also be considered. If the same loops in the callee are parallel, and the same partitioning of the caller's inner loop is still legal, no recompilation is required. If either of these tests fails, the callee must be recompiled. The caller only requires recompilation if it is no longer safe to move the loop header into the callee, even if the callee has been edited.

Cloning. Two calls can use the same transformed procedure if the inner loop of the caller can be partitioned in the same way. There is also a second test. From the caller's perspective, this optimization only moves the loop header across the call and into the called procedure. As a result, the same tests that applied to cloning for loop embedding (Section 7.1.2) also apply here.

Requirements for general permutation of imperfect loop nests. The discussion above restricts permutation to an inner loop in the caller being interchanged with the outermost loops in the callee. All other possibilities fall into at least one of three categories. The other possibilities are presented here, accompanied by a discussion of why they are more difficult to support.

- *Moving a loop from the called procedure outside an outer loop in the caller.*
  For this case, the transformation described in this section is performed. If this is successful, RSDs must be constructed for the new partitioned loops. Moving to the enclosing loop, the transformation is repeated. This continues until either the outermost loop is reached, or a loop is reached for which the transformation is not legal.

  This analysis is extensive, and the representation of the procedure becomes quite involved. Even if we are willing to perform the analysis, rather than attempting to retain the separate procedures, inlining the call may be more appropriate. Analysis for the other two possibilities is similar.

- *Moving a loop to another ancestor in the call multigraph.*
  If a loop is moved out to a procedure other than the immediate caller, evaluating the safety of distribution must be performed at each procedure in the chain of calls. First, distribution is tested for each loop in the caller. Each time a distribution is found to be legal, RSDs are constructed for the new distributed loop. Then, the safety of distribution is considered in the caller's caller, and so on through the procedures in the call multigraph.

- *Moving an inner loop in the callee outside of the loop in the caller.*
  First, the safety of distribution of outer loops in the callee is considered. If this is legal, RSDs are constructed to represent the distributed loops. This information must be available for analysis in the caller.

7.3 Inline Substitution to Enhance Parallelism

In this chapter, inlining is considered in a new light. Previously, inlining was recommended as the only transformation technique which allowed code to be moved across the call boundary. The transformations in the previous section actually move code across the call without inlining, as long as changes to the code can be represented concisely with annotations to the call multigraph. The transformed procedures can be thought of as special clones, which may be shared by other callers.

Inlining can still be useful for enhancing parallelism. For some transformations that move code across call boundaries, inlining may be more appropriate than any other transformation. As an example, general loop permutation for imperfectly nested loops can be so complex to describe that even if the analysis was feasible, the program representation required would still be too complex. Inlining may also succeed in exposing parallelism when other optimization techniques fail, since the RSD information is not always precise. Additionally, certain transformations designed to break dependences (such as scalar expansion and scalar renaming) are not evaluated across procedure boundaries.

A goal-directed inlining strategy should be followed during evaluation of possible optimizations on a loop nest. As a possible strategy, inlining could be performed to produce loop nests of two or more loops. This is because many transformations to break dependences require or work better with loop nests. The goal-directed strategy for parallelization is not considered further. This section instead considers how inlining affects parallelism.

The results of the inlining study from Chapter 3 indicated that in most cases inlining was not particularly beneficial to enhancing parallelism To gain insight into the cause, we examined loops containing call sites that had been inlined. As Figure 3.11 showed, often too many dependences remain in these loops to gain improvement in parallelism. However, the nature of these dependences in many cases seemed to be related to inlining. Loops with inlined call sites exhibit some similar properties that are unlike human-generated code. We observed instances where dependences could be broken by certain optimizations. These optimizations are designed to improve inlined code and probably not as profitable for code written by humans.

The rest of this section describes some unique properties of loops containing inlined procedure calls and presents optimizations that promote parallelism in loops with such properties. Most of the examples used to illustrate these optimizations arose in the inlining study. A compiler that supports automatic or programmer-specified inline substitution can more effectively parallelize a program by performing these optimizations either during inlining or, in cases where the optimizations can possibly cause execution-time performance degradation, during dependence analysis.

7.3.1 Properties of Inlined Programs

After inlining, often loops exhibit properties inhibiting optimization that would not ordinarily appear in human-generated code. These properties can be categorized in the following way:

1. Unreachable code.
2. Loop-invariant code.
3. Bounds checking.
4. Partial parallelism.

Unreachable code

Unreachable code can result from inline substitution when constant actual parameters appear at the inlined procedure call. As a result of replacing constant parameters in the procedure body, tests based on the value of these parameters can be evaluated at compile time. Thus, the tests themselves can be eliminated, and if they evaluate to *false*, code conditionally executed based on such tests can also be eliminated.

One obvious benefit of eliminating unreachable code is that less of the program needs to be examined during analysis and optimization. However, a more important reason to eliminate unreachable code is because it can contribute dependences that inhibit parallelism. For example, code guarded by a conditional such as $if\ (1 \neq 1)$ may be contributing dependences, even though the condition is always *false*.

Unreachable code elimination occurs before dependence analysis in many compilers. Since a certain amount of semantic analysis is required to perform inlining, it is beneficial to eliminate unreachable code *during inlining*, so that the code growth associated with inlining is reduced.

Loop-invariant code

Because our goal is to parallelize more loops after inlining, call sites selected for inlining are usually those that appear within loops. When a procedure is called repeatedly within a loop, it is often the case that there is some initialization code appearing at the beginning of the procedure body that only needs to be executed once within the loop. Inlining exposes the opportunity to move this code outside of the loop body.

Moving loop-invariant code out of loops is profitable within a scalar optimizing compiler because doing so can greatly reduce the amount of computation performed at run time. However, there is an additional reason why it is profitable in a parallelizing compiler. Just like unreachable code, loop-invariant code can contribute dependences that inhibit parallelization. Consider the following example:

do $i = 1, 10$
    if $(x \neq 1)$ $j = 1$
    y(i) = y(i) + j
enddo

A dependence analyzer would detect the dependence of the assignment to $y(i)$ upon the conditional assignment to j in the previous statement. However, the value of the condition $(x \neq 1)$ is loop-invariant, so j will have the same value on each iteration of the loop. The conditional assignment to j can thus be moved outside the loop, and the loop can then be parallelized.

The traditional loop-invariant code motion algorithm is used to eliminate a single expression at a time. To eliminate dependences and improve parallelism, compound statements such as loops and conditionals must be located and moved outside of the loop. Such extensions to the traditional algorithm have been described [CLZ86] [FOW87].

Loop unswitching. A similar technique, *loop unswitching* [AC72], can be applied when a condition in the loop is loop-invariant, but the code guarded by the condition is not. For an *if-then-else* clause within a loop, two copies of the loop are created. One is guarded by the *if* condition, eliminating the portion of the loop that the *else* condition guards. The other copy is guarded by the *else* condition, eliminating the part of the loop guarded by the *if* condition.

Bounds checking

When call sites are inlined within a loop, many optimization opportunities involving loop induction variables can arise. When calls appear in loops, the current problem size, a function of the loop induction variable, may be passed as a parameter. Then, within the called procedure, there may be a test of the value of the parameter to ensure that it is within the range of the array bounds or within some other suitable bounds (e.g., greater than 0). Because the bounds of the induction variable and variables whose values are based on the induction variable can be determined directly from the bounds of the condition for loop execution, tests involving such variables can often be eliminated, as in the following example:

```
do k = 1, n − 1
    /* before inlining, call p(n − k, ...) appeared here */
    t = n − k
    if (t ≤ 0) then
       ⋮
enddo
```

Because the loop induction variable k ranges from 1 to $n-1$, the value of t ranges from $n-1$ on the first iteration, to $n-(n-1)=1$ on its final iteration. Thus, the test for $t \leq 0$ will always evaluate to *false*. The test and the code guarded by the test can be eliminated.

A similar opportunity arises when the test only evaluates to *true* on the first or last iteration of the loop (or the first few or the last few). By peeling off the first or last iteration and removing the test and its accompanying code within the loop, control flow is simpler, possibly exposing parallelism within the loop.

Locating variables whose values are based on the induction variable. Some variables that are functions of the induction variable are auxiliary induction variables and can be located using a variant of *induction variable elimination* [ASU86]. Once these variables are located, their possible ranges must be determined. If we know that i is a loop induction variable ranging from lb to ub, and j is an induction variable expressed in terms of i, we can substitute lb for i in the expression for j's value to determine the lower bound of j. Similarly, ub can be substituted for i to get the upper bound of j. Note that if $-i$ appears in the expression for j's value, substituting lb for i instead gives the upper bound of j, and substituting ub gives the lower bound of j.

This is a simplification of the techniques *range analysis* and *range propagation* [Har77a]. These techniques track the range of values for all variables in a program. Although much more precise, tracking ranges of all variables is too expensive for most practical compilers. Here, we have limited ourselves to only tracking ranges of induction variables because often their ranges are explicitly declared in the loop body. This knowledge has proven to be especially useful in the context of optimization after inline substitution.

Partial parallelism

After inlining call sites appearing in loops, the loops are often long and complicated. With such loops dependences are more likely to exist that defy the dependence analyzer. Even after applying the optimizations described above, a loop may contain dependences that make it inherently sequential. To parallelize such loops, it is necessary to locate parallel portions of the loop and distribute the loop among the parallel and sequential portions of the loop.

We observed that many of the dependences remaining in the loops after inlining were on scalar variables. The technique commonly used in vectorization to eliminate dependences on scalar variables is *scalar expansion* [KKL+81]. A similar technique is used in parallelization, where such variables are made *private* to each processor. A scalar r is expanded in a loop with induction variable i by replacing accesses to r with accesses to an array element $r(i)$, where the new array r has length at least as great as the number of iterations of the loop. This optimization is always safe, but the compiler must ensure that uses of the expanded scalar are translated to correspond to either the previous or the current iteration, whichever is appropriate. However, due to the increased memory requirements resulting from scalar expansion, it is not usually performed unless doing so allows the loop to be parallelized.

Scalar expansion can also be used even when it does not automatically allow the loop to be parallelized. The combination of scalar expansion and loop distribution may allow portions of the loop to run in parallel. Two such opportunities arose in our sample programs.

First, it may be possible to expand a scalar and calculate its value for all iterations in parallel, even if the rest of the loop is sequential. Another possibility is to calculate the scalar values sequentially and expand them into array values for every iteration so that the rest of the loop can be run in parallel. Both of these cases are illustrated in this example from wave:

$$
\begin{array}{l}
\textbf{do } i = 2, ny \\
\quad yn = (i - 1.5) * hy \\
\quad \textbf{do } j = lb, ub \\
\quad\quad r = 0. \\
\quad\quad \textbf{if } (yn = y(j))\ r = dy(j) \\
\quad\quad \textbf{if } (yn > y(j-1) \textbf{ and } yn < y(j)) \textbf{ then} \\
\quad\quad\quad r = (dy(j) - dy(j-1)) * (yn - y(j)) \\
\quad\quad\quad y(j+1) = yn \\
\quad\quad \textbf{endif} \\
\quad \textbf{enddo} \\
\quad d = d + r * hy \\
\textbf{enddo}
\end{array}
$$

Scalar expansion permits all of the values for yn to be calculated in parallel or vector. However, the inner loop is inherently sequential. Thus, yn is an example of scalar expansion to permit the scalar to be calculated in parallel. Now, since the inner loop is sequential, values for the variable r are calculated sequentially. However, scalar expansion of r using an array element for each outer loop iteration allows the value of d to be calculated in parallel using a *sum reduction*.[21] This expansion of r is an example of scalar expansion to permit use of the scalar in a vector loop. Here is the parallelized version of the above loop:

[21] A vector reduction operation uses special hardware to accumulate the results of certain operations applied across an array of values, even though there is a dependence. To compute the same result as the scalar version, reduction operations should be associative.

```
doall i = 2, ny
    yn(i) = (i - 1.5) * hy
enddo
do i = 2, ny
    do j = lb, ub
        r(i) = 0.
        if (yn(i) = y(j)) r(i) = dy(j)
        if (yn(i) > y(j - 1) and yn(i) < y(j)) then
            r(i) = (dy(j) - dy(j - 1)) * (yn(i) - y(j))
            y(j + 1) = yn(i)
        endif
    enddo
enddo
doall i = 2, ny
    d = d + sum_reduction(r(i) * hy)
enddo
```

This particular pair of optimizations should be applied with great care since they may carry with them a substantial overhead. The overhead of loop distribution arises from the cost of duplicating loop control structures combined with the overhead of parallelizing a loop. Thus, the number of loop iterations and the number of statements appearing in the distributed loops must be sufficiently large to justify the increased overhead. Scalar expansion increases the memory requirements of a program, which can adversely affect performance of the memory hierarchy. These issues are further discussed below.

7.3.2 Experimental Results

The optimizations in this section were applied by hand to the eight programs from the inlining study. On three of the programs, these optimizations yielded improvements in parallelism. After optimization, we executed versions of the inlined program with and without optimization on the Stardent Titan, configured with four processors. The code was instrumented to measure the time spent in the optimized portions of the programs. The results are summarized in Figure 7.2 and discussed in the remainder of this section.

Explanation of Results

In Figure 7.2, line 1 displays the number of candidate loops where inlining could be applied to eliminate calls within loops. Since we used a heuristic to determine when

	efie304	wave	cedeta
1. DO loops with procedure calls	15	20	23
2. loops with no calls after inlining	14	15	22
3. loops improved after inlining	1	1	1
4. additional loops improved by opts	4	10	1
5. execution time improvement in optimized loops	51%	77%	18%
6. execution time improvement for program	6%	4%	4%

Figure 7.2 Results of applying optimizations to inlined programs.

to inline a call site, not every call site appearing in a loop was inlined. Line 2 gives the number of loops from line 1 in which all calls were inlined. The calls remaining in loops in wave and cedeta either are to large procedures, or are calls where the types of parameters do not match. Line 3 shows the number of loops from line 2 that are fully or partially parallelized after inlining. This represents the improvement in optimization gained from inlining. Line 4 gives the number of additional loops from line 2 with partial parallelism after applying the optimizations described in this section. Lines 5 and 6 represent the percentage decreases in execution time for the optimized portion of the code and for the entire program, respectively.

While the number of loops improved and execution time improvements for optimized loops are significant, overall execution time improvements are not as impressive. Comparing the inlined versions of these three programs before and after optimization, the overall execution time improvements have been no greater than 6 percent. This is because the time spent in the optimized loops is a very small percentage of the program execution time. This may be because FORTRAN programmers are concerned about the inefficiency of procedure calls, and therefore avoid placement of calls in frequently executed parts of their programs.

Problems

Throughout these experiments, there were problems that interfered with program performance after optimization. These were as follows:

- When loop distribution is needed to make a loop parallel, the overhead can be enough to make the parallelization not worthwhile. This problem is exacerbated by the code generation phase of the Titan compiler, which distributes loops into the smallest number of statements that still preserve all of the dependences.

These loops are then fused together when possible into parallel and sequential loops. Unfortunately, loops containing conditionals are not fused with any other loops [All90]. On cedeta, the optimized loop was distributed into four separate loops, two of which consisted of only a single statement. Parallelizing the short loops caused execution time of the loop to increase by 25 percent. By making the two parallel loops run in vector mode instead, we obtained the 18 percent improvement in Figure 7.2.

- Scalar expansion increases the memory requirements of a program, and as a result, can hurt performance of the components of the memory hierarchy. After the combination of scalar expansion and loop distribution, executing an optimized loop in wave increased the execution time in the optimized portion of the code by a factor of three. The combination of variable renaming and scalar expansion affected the compiler's analysis of accesses and introduced a pipeline interlock in the parallel loop. By avoiding the renaming, the interlock was eliminated, resulting in the significant improvement in Figure 7.2. Although this problem was in some sense a mistake in our hand optimization, it points out how sensitive performance can be to changes in memory accesses. A code generator that incorporates scalar expansion must consider these effects.

7.4 Related Work

7.4.1 Approaches to Summarizing Array Side Effects

A number of approaches to array side effects have been suggested in the literature. A comparison of these techniques can be found in [HK91]. In this comparison, the precision of the techniques is weighed against their costs. The costs are broken down into a number of categories. The two most important costs are the cost of merging two pieces of information about the same array, and the cost of intersecting two pieces of information to determine if there exists a dependence between them.

- Two techniques avoid summarizing accesses to the same array by building a list of all accesses. The first of these, by Burke and Cytron, linearizes all subscript expressions, resulting in a list of accesses to 1-dimensional objects [BC86]. The effect of linearization can be a significant loss of information for dependence testing. Li and Yew developed a more precise representation of array accesses, called *atom images* [LY88a] [LY88b]. These precisely represent linear subscript expressions in rectangular and triangular iteration spaces. In both cases, testing whether two lists have a non-empty intersection requires a traversal of the lists of accesses. Thus, the cost of these techniques can be significant.

- Triolet summarizes array accesses using a more precise representation of the summary than RSDs[TIF86]. This technique locates the convex hull surrounding two regions representing accesses. Dependence testing requires an assymptotically exponential linear inequality solver.
- Balasundaram proposed an approach to summarizing array accesses designed to locate opportunities for task-level parallelism [Bal89]. It can also be used to provide programmers with a visual description of affected regions of an array. For this purpose, *Data Access Descriptors* also include a traversal order and a reference template.

The implementation of RSD analysis described in [HK91] gives up some precision in favor of efficiency. By describing only simple access patterns and by summarizing multiple accesses to an array, all aspects of the technique are efficient. Experiments with regular section analysis on the LINPACK library demonstrated a 33 percent redution in parallelism-inhibiting dependences, allowing 31 loops containing calls to be parallelized. Comparing these numbers against published results of more precise techniques, [LY88a] [LY88b] [TIF86], there was no benefit to the increased precision of the other techniques.

7.4.2 Interprocedural Transformations

Prior to this research, array side-effect information has been employed only in deciding whether loops containing calls can be parallelized. In contrast, this chapter has described how array side-effect information can also be utilized in guiding interprocedural transformations. In fact, we know of only one other author who has addressed the issue of interprocedural transformations. Huson's implementation of inline substitution actually performs loop embedding [Hus82], as described in Section 7.2.1. No interprocedural information is required to determine the safety of this transformation.

The Convex Applications Compiler analyzes array side effects using a technique similar to regular section analysis [Met91]. This information is combined with inlining and cloning to parallelize loops with procedure calls. Inlining is used to eliminate call overhead. Cloning is used to expose constants in order to improve dependence analysis. Experience using this compiler has demonstrated that interprocedural analysis and optimization typically execution-time improvement of 10 to 25 percent over standard compilation. In a few cases, the compiler compiler yields a speedup of 5 over separate compilation [MS91].

7.5 Chapter Summary

The ideas in this chapter provide a framework for interprocedural optimization supporting parallelization. The chapter makes two main contributions: how to use array subsections to guide interprocedural transformations, and how to eliminate dependences in loops after inlining.

Transformation framework. In the framework for interprocedural transformation, the legality of transformations is established with tests that can be applied to a single procedure at a time. These tests can be applied during or following dependence analysis, so that we can take advantage of the detailed information gathered during this phase. When tests are performed a single procedure at a time, no ordering is imposed on the analysis of procedures.

After transformations are performed, their effects are captured with annotations on the call multigraph. This provides a natural mechanism for locating opportunities for cloning, as well as making decisions about recompilation. As an initial test for recompilation, RSDs are tested to see if they have changed. This prevents the need for dependence analysis in cases where recompilation is not required.

Optimizations after inlining. The optimizations on inlined loops can be incorporated into a compiler supporting automatic or programmer-specified inline substitution. Because the optimizations are motivated by properties of inlined programs, they can enhance parallelism beyond what is possible with inlining alone.

The optimizations were validated with experimentation. We observed that the secondary effects caused by loop distribution and scalar expansion can cause performance degradation. A compiler that performs these optimizations should consider such problems during code generation.

The experiments measured execution time improvements in the optimized portions of the programs, as well as in overall program execution time. The execution time improvements within the optimized portions of the programs were significant.

However, improvements in the overall program execution times were only moderate, since the time spent in optimized portions of most of the programs was only a small percentage of program execution time. Similarly, parallelism results after inlining in Chapter 3 suggested that inlining did not often open up opportunities for parallelization. This is perhaps due to a perception by the programmers that procedure calls are expensive. However, the improvements on just the optimized loops are an indication that significant execution time improvements may be possible on code written in a more modular style.

Chapter 8

Conclusion

This dissertation has dealt with some important issues in interprocedural optimization. It has presented a comprehensive approach to interprocedural optimization, balancing the effectiveness with costs in code growth and compilation dependences. A system was described which manages interprocedural optimization while attempting to reduce the amount of recompilation required after minor changes.

This chapter concludes the dissertation by summarizing its contributions, describing a prototype implementation and outlining future work planned for ParaScope. Each one of these topics is given a separate section. The final section presents the implications of this work on compiler design and programming practices.

8.1 Contributions of the Dissertation

This research has included extensive experimentation leading to a much improved understanding of interprocedural optimization. In particular, the inlining study described in Chapter 3 spanned over 2 years. The data from the study led to a variety of unexpected conclusions about the benefits of interprocedural optimization. Other experiments – on cloning, constant propagation and goal-directed interprocedural optimization in Chapter 4, and improving parallelism after inlining in Chapter 7 – though less extensive, also contributed to knowledge about the effects of interprocedural optimization.

Much of the compilation system described in this dissertation has been implemented in the ParaScope Programming Environment. As a result of the prototype implementation, the rest of the design was developed based on what was feasible to incorporate into the existing framework. While the experimentation suggested when interprocedural optimization would be effective, the implementation guided how to use interprocedural optimization without absorbing significant costs.

Several important algorithms resulted from this work. In Chapter 2, we presented an algorithm for constructing the call multigraph that is nearly linear for most programs, and is quadratic in the worst case. These are much better time bounds than previous algorithms. This algorithm is important because the call multigraph provides the program representation used in all interprocedural analysis and optimization.

Chapter 4 describes a goal-directed strategy for interprocedural optimization, using interprocedural transformations only to enable high-payoff memory management optimizations. This represents a significant departure from previous work on interprocedural optimization, where interprocedural optimizations are widely used to enable low-level optimizations. This strategy was a direct application of the experimental results. It includes an algorithm to recognize important variables, only allowing procedure cloning when doing so exposes constant values for these important variables. It also includes an estimate of the relationship between computation and memory accesses within a loop, so that only loops that are memory-bound are optimized.

Chapter 5 presents a general algorithm for procedure cloning, which in the worst case requires polynomial time and a doubling of program size. The algorithm restricts cloning to avoid its potentially exponential behavior. The amount of cloning is reduced by performing cloning only when it is worthwhile, and by merging clones that cause equivalent effects on optimization. If the amount of cloning would still cause intolerable code growth, then cloning is performed only on the most frequently executed portions of the call multigraph. The work on cloning in this dissertation significantly expands previous knowledge.

Chapter 6 provides a system for combining inlining, cloning and optimization based on interprocedural information. Included in this chapter is a presentation of the interprocedural analysis performed in ParaScope and general guidelines for supporting inlining and cloning. Most importantly, this chapter describes the necessary support for recompilation analyis when inlining and cloning are performed.

In Chapter 7, we develop a strategy for interprocedural optimization supporting parallelization. This chapter presents some interprocedural transformations designed to enhance parallelization when a loop containing call sites cannot be directly parallelized. It also describes transformations to apply following inlining to further enhance parallelization.

8.2 Implementation Status

8.2.1 Program Compiler

Large portions of the program compiler have already been implemented, and further implementation is planned. The current program compiler first builds the call multigraph, using the precise algorithm [CCHK90]. Then interprocedural information is calculated on the call multigraph. Currently, the program compiler builds scalar MOD, REF, ALIAS and CONSTANT sets. RSD information is also available in the environment, although it is calculated outside of ParaScope.

The program compiler performs inline substitution at the source level, and this part of the implementation was briefly described in Chapter 3. At this time, inlining requires programmer selection, made possible by a tool which acts as an interface to the program compiler. The programmer selects individual call sites for inlining, and the program compiler inlines these call sites whenever it is legal.

Recompilation analysis is performed to determine what procedures require recompilation due to changes in interprocedural information. This analysis also deals with inlining, determining if changes have invalidated an inlined version of a procedure. After recompilation analysis, the program compiler executes the appropriate commands to build a program executable. It also saves a program representation describing the interprocedural information and inlining assertions used in this compilation. This information will be used by the program compiler during recompilation analysis on a future compilation, and by other tools interested in knowledge of interprocedural information.

8.2.2 Program Compiler Display

Through the course of this research, it became necessary to build an interface to the program compiler. This was particularly important for the inlining study. A display from the program compiler interface is shown in Figure 8.1.

The display window has 3 panes. The top pane shows the composition, which is the description of the program. The middle pane, the *entry display*, allows browsing of individual procedures in the program, annotated with their interprocedural information. All statements other than the procedure declarations and call sites are filtered out of the entry display. The bottom pane displays output from the compiler. In its verbose mode, the compiler describes its activities at each phase of the program compiler. Included in this description is the list of procedures to which interproce-

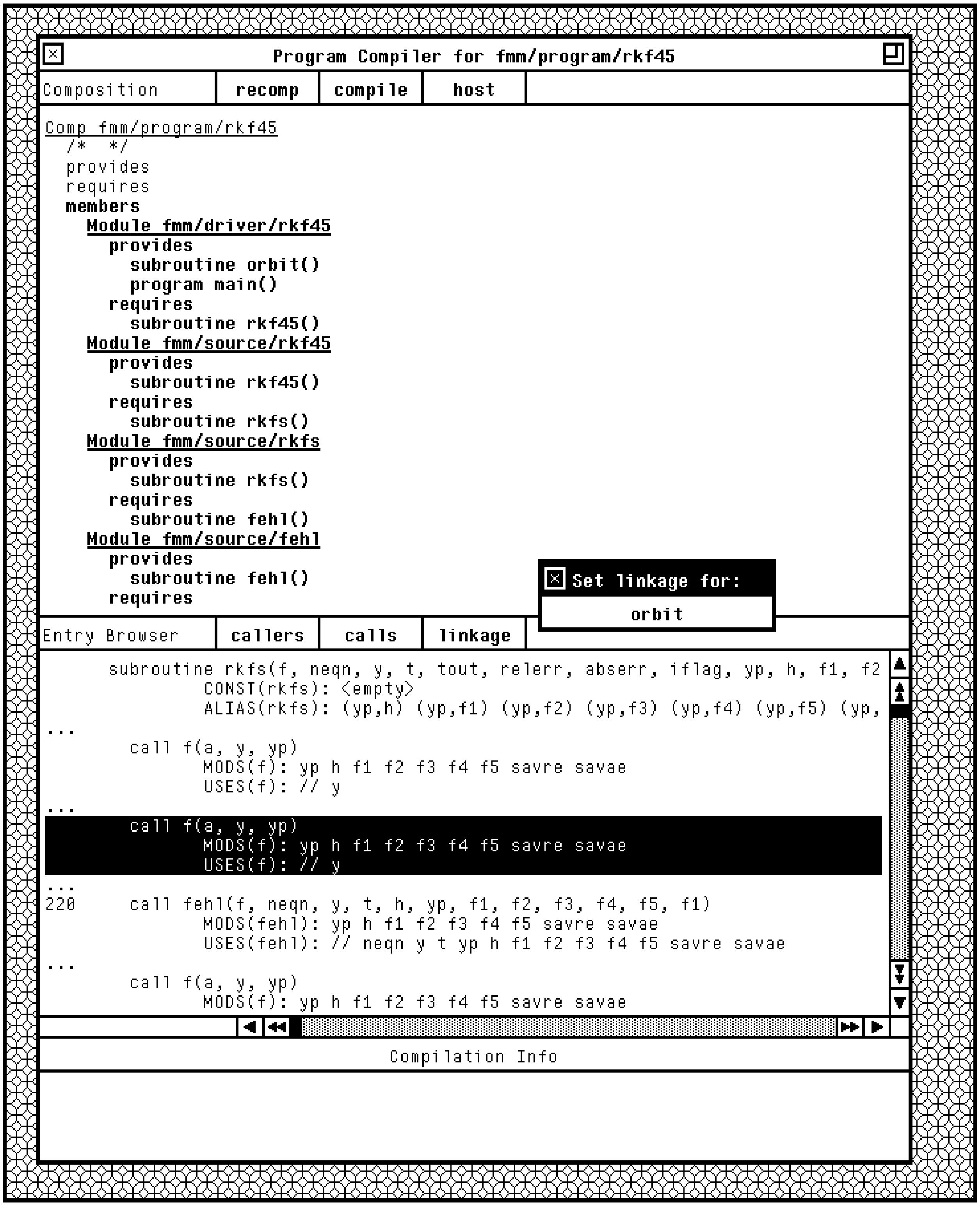


Figure 8.1 Program compiler display in ParaScope.

dural transformations are applied. The compiler explains why the transformation was applied and whether it was successful. The compiler also provides the list of procedures requiring recompilation and explains why recompilation was necessary.

To illustrate the utility of such a display, we use the example program **rkf45**, part of the Forsythe, Malcolm and Moler library package [FMM77]. In the entry display from Figure 8.1, the procedure definition of *rkfs* is annotated with its interprocedural constants and aliases. In this case, there are no constants, but a significant number of alias pairs. The call sites invoking *fehl* and procedure-valued formal f are annotated with their MOD and REF sets. The display of interprocedural information to the programmer can be useful in debugging [CS85]. For example, from the main procedure it is possible to locate variables that are modified but never referenced.

The buttons on the title bar of this pane are *callers*, *calls* and *linkage*. Selection of the *callers* button results in a menu of the procedures invoking the subroutine currently being displayed. Selection of one of the menu items changes the focus of the entry display to the information for the *caller*. Similarly, the *calls* button allows changing the focus in the display to the called procedure. In this way, the entry pane can be used to walk the call multigraph. The focus of the entry display can also be changed through a mechanism provided by the composition pane, enabling direct selection of procedures to be the focus. The *linkage* button allows the programmer to select call sites for inline substitution. This mechanism was used in the inlining study of Chapter 3.

The main subroutine of this program passes the subroutine *orbit* as a parameter at a call site and is passed down a chain of calls before it is invoked. The program compiler determines that the procedure-valued formal f invoked by *rkfs* can have only the binding *orbit*. In the snapshot of the display from Figure 8.1, we have selected a call site invoking f and the *linkage* button in order to set the linkage style for the call site. The small box to the right of the *linkage* button is a pop-up menu providing the list of procedures invoked at the selected statement. The menu shows that the program compiler has resolved the binding of f.

8.3 Future Work

8.3.1 Complete Implementation

Work continues on the program compiler implementation. First, we plan to incorporate the interprocedural optimization to enable memory-management transformations, as described in Chapter 4. This includes interprocedural analysis of *CloningVars* and loop balance, as well as automating inlining and cloning and the recompilation analysis required to support them. Analysis of RSDs will also be added to ParaScope. This will allow incorporating the parallelizing transformations described in Chapter 7. Finally, analysis will be added to the program compiler to be used by other tools, as described at the end of this section.

8.3.2 More Experimentation

We have studied inlining extensively. Experiments were also performed to assess the effectiveness of constant propagation, and cloning based on constants, although more work could be done in these areas. Another study at Rice has provided insight into the value of array side-effect analysis.

A number of additional experiments would provide a better understanding of the effectiveness of interprocedural optimization, particularly the impact of CONSTANT, MOD, REF and ALIAS information on optimization. Studies for other programming languages have shown mixed results. Conradi estimated that a 5 to 20 percent improvement in execution time was possible with a combination of inlining and interprocedural information, based on empirical results on program characteristics for the PQCC multi-language compiler backend [Con83]. Richardson and Ganapathi observed an average of 1.5 percent using only MOD and REF information to optimize Pascal programs [RG89b]. Richardson and Ganapathi's results may not reflect what can be expected from scientific FORTRAN code. Since Pascal allows the programmer to declare a parameter as call-by-value or call-by-reference, the programmer can convey a certain amount of the interprocedural information to the compiler that is not possible with FORTRAN. A study of the effectiveness of interprocedural information for compiling FORTRAN programs is needed to determine its usefulness.

8.3.3 New Uses for Interprocedural Analysis and Optimization

The original design of the program compiler was meant to support improved scalar optimization. Over time, we extended the design to include support for improved parallelization. As the implementation has matured, new applications of interprocedural knowledge have arisen. Because of the unique design of ParaScope, extending the analysis to solve new interprocedural problems is not difficult. For this reason, we are planning to apply interprocedural knowledge to some new areas. Two of these are briefly described in this section.

Distributed memory compilation

Major challenges in compiling for distributed memory multiprocessors include determining the processor on which to locate a data item, and managing the communication of the data when accessed by processors other than the one on which it is located. Most compilers for distributed memory machines require the programmer to determine how the data will be assigned to the processors [ZBG88] [RP89] [KMV90]. The compiler is responsible for adding the message passing to move the data to processors that use it and from processors that define it. The compiler also adds the necessary synchronization to guarantee that data dependences are preserved.

Distributed memory compilers typically assume that procedure calls do not occur in parallel loops, with the exception of the SUPERB compiler [Ger89]. If this assumption is relaxed, the compiler must have knowledge of decompositions across procedure boundaries. This adds a host of problems requiring interprocedural solutions [HKT91].

The compiler must propagate decompositions on the call multigraph, with formal parameters inheriting the decomposition of the actual parameters passed at the call, and globals retaining their decomposition across the call. If a variable inherits multiple decompositions, the compiler clones the called procedure, creating unique copies for each unique set of decompositions.[22] The procedure body can be tailored to the decomposition of its data, since this is known at compile time. As an example, the code generated could compute values only for variables that are located on the processor being used.

Access anomaly detection

Accesses to the same memory location by multiple parallel threads, with at least one write, are a common source of bugs in parallel programs. These types of bugs

[22]For this problem, SUPERB adds extra parameters to a call site specifying decompositions for variables of the called procedure. The code for the called procedure tests these extra parameters for each possible decomposition, and separate code is executed for each possibility. This looks a lot like procedure cloning, but may cause more code expansion, and much of the work must be done at run time.

are difficult to locate because they may only occur when concurrent threads are interleaved in a specific way. Thus, they may be difficult to repeat with subsequent executions of the program.

Access anomaly detection is a technique used in debugging on shared memory multiprocessors to locate potential race conditions by tracing memory accesses [Sch89]. To avoid significant overhead, Schonberg only traces memory locations in executing concurrent threads. When all concurrent threads that may access the same variable have completed execution, trace information about the variable is discarded.

To further reduce the overhead of access anomaly detection, researchers at Rice are taking advantage of dependence analysis [HKMC90]. By assuming that concurrent threads are only created by parallel loops, dependence analysis at the loop-level can eliminate memory locations from consideration. If there are no dependences on a variable in a parallel loop, there is no need to trace accesses to it. For parallel loops containing procedure calls, interprocedural analysis is needed to aid in the dependence analysis.

First, we need to determine if dependences exist on call sites within a parallel loop. This is done by constructing RSDs for each parallel loop, and intersecting them with RSDs for the call sites within the parallel loop. A non-empty intersection describes the set of possible dependences, and any accesses which are in the intersection must be traced in the procedure containing the parallel loop. Accesses to variables in the called procedures also need to be traced if they can fall within the intersection. To determine this, the intersection is translated to variables in the scope of the called procedures (i.e., from actual parameters at the call site to formal parameters in the called procedure). Then, accesses in the called procedure can be compared to the intersection, yielding the accesses that must be traced. Thus, the intersection is propagated throughout the portion of the call multigraph making up the parallel loop. In some cases, a procedure may inherit from its callers significantly different sets of variables that need to be traced. This situation may provide a good opportunity for cloning in order to reduce the amount of instrumentation needed.

8.4 Final Remarks

Modern architectural features – such as instruction pipelines, long instruction words and multiprocessors – place a significant burden on the compiler to take advantage of the performance potential. Moreover, effective instruction scheduling for these types of architectures requires that the compiler understand the effects of multiple statements at a time. Thus, to produce efficient object code for these architectures, a compiler must have surrounding context when performing optimization.

When a compiler can only optimize procedures as single units, the restricted context may cause it to miss some important opportunities for optimization. As a consequence, a programmer concerned about efficiency may rewrite a program to avoid procedure calls in places where optimization is important. The programmer may even attempt to hand-optimize their program to take advantage of the architectural features of a machine.

The role of interprocedural optimization is to provide the compiler with adequate context to perform optimization. We have shown in this dissertation that sometimes this can make dramatic improvements in program performance. The underlying motivation to this work is to avoid or reduce the performance penalty suffered when procedure calls occur frequently in code. Then programmers can make use of procedure calls without concern of hurting performance. This allows them to write modular, machine-independent programs, resulting in debuggable, portable and maintainable programs.

Bibliography

[AC72] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[AC76] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[ACF+80] F.E. Allen, J.L. Carter, J. Fabri, J. Ferrante, W.H. Harrison, P.G. Loewner, and L.H. Trevillyan. The experimental compiling system. *IBM Journal of Research and Development*, 24(6):695–715, November 1980.

[AK84] R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*. ACM, June 1984.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[All90] R. Allen. Private communication, March 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[Bal79] J. E. Ball. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction*. ACM, August 1979.

[Bal89] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Rice University, Houston, TX, July 1989.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*. ACM, January 1979.

[Ban90] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, chapter 4. The MIT Press, 1990.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, June 1986.

[BCKT90] M. Burke, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. Technical Report TR90-126, Dept. of Computer Science, Rice University, July 1990.

[Bur87] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. Research Report RC 12702, IBM Yorktown Heights, September 1987.

[CCH+87] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren. A practical environment for scientific programming. *IEEE Computer*, 20(11):75–89, November 1987.

[CCH+88] D. Callahan, K. Cooper, R. T. Hood, K. Kennedy, and L. M. Torczon. ParaScope: a parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–89, December 1988.

[CCHK90] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation*. ACM, June 1990.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, June 1986.

[CFR+89] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages*, June 1989.

[Cho88] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation.* ACM, June 1988.

[CHT90a] K. D. Cooper, M. Hall, and L. Torczon. The perils of interprocedural knowledge. Technical Report TR90-132, Dept. of Computer Science, Rice University, September 1990.

[CHT90b] K.D. Cooper, M.W. Hall, and L. Torczon. An experiment with inline substitution. Technical Report TR90-128, Dept. of Computer Science, Rice University, July 1990. To appear in *Software—Practice and Experience.*

[CK84] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN 84 Sym. on Compiler Construction.* ACM, June 1984.

[CK88a] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CK88b] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN 88 Conference on Programming Languages Design and Implementation.* ACM, June 1988.

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages.* ACM, January 1989.

[CKT85] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In *Proceedings of the SIGPLAN 85 Symposium on Compiler Construction*, June 1985.

[CKT86a] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbf{R}^n$ environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[CKT86b] K. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, June 1986.

[CKT+86c] K. Cooper, K. Kennedy, L. Torczon, A. Weingarten, and M. Wolcott. Editing and compiling whole programs. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986.

[CLZ86] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual Symposium on Principles of Programming Languages*. ACM, January 1986.

[Con83] R. Conradi. Inter-procedural optimization of object code. Technical Report 25/83, University of Trondheim, Norway, 1983.

[Coo83] K. D. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*. PhD thesis, Rice University, Houston,TX, April 1983.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages*. ACM, January 1985.

[CS85] R. Conradi and D. Svanaes. FORTVER — a tool for documentation and error diagnosis of FORTRAN-77 programs. Technical Report 1/85, University of Trondheim, Norway, January 1985.

[DH88] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[Fel79] S. Feldman. Make – a program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, 1979.

[FMM77] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Ger89] M. Gerndt. Array distribution in SUPERB. In *Proceedings of the 1989 International Conference on Supercomputing*, June 1989.

[Har77a] W. Harrison. Compiler analysis for the value ranges of variables. *IEEE Transactions on Software Engineering*, SE-3(5):243–250, May 1977.

[Har77b] W. Harrison. A new strategy for code generation–the general purpose optimizing compiler. *IEEE Transactions on Software Engineering*, SE-5(7):367–373, July 1977.

[HC89] W. W. Hwu and P. P. Chang. Inline function expansion for inlining C programs. In *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation.* ACM, June 1989.

[Hec77] M. Hecht. *Flow Analysis of Computer Programs.* American Elsevier, North Holland, 1977.

[HK83] R. T. Hood and K. Kennedy. A programming environment for Fortran. Technical Report MASC TR 83-22, Dept. of Mathematical Sciences, Rice University, 1983.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[HKMC90] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing 90*, November 1990.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report TR90-149, Dept. of Computer Science, Rice University, February

1991. To appear in J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*, Elsevier, 1991.

[Hol91] A. M. Holler. *A Study of the Effects of Subprogram Inlining*. PhD thesis, Univ. of Virginia, Charlottesville, VA, May 1991.

[Hop71] J. Hopcroft. An *nlogn* algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, NY, 1971.

[Hus82] C. A. Huson. An inline subroutine expander for Parafrase. Masters Thesis UIUCDCS-R-82-1118, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1982.

[KKL+81] D. J. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual Symposium on Principles of Programming Languages*. ACM, January 1981.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, March 1990.

[KU77] J. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[LY88a] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the SIGPLAN Symposium on Parallel Programs: Experience with Applications, Languages and Systems*, July 1988.

[LY88b] Z. Li and P.-C. Yew. Interprocedural analysis and program restructuring for parallel programs. Technical Report CSRD-720, University of Illinois, Urbana-Champaign, January 1988.

[Mar89] T. J. Marlowe. *Incremental Iteration and Data Flow*. PhD thesis, Rutgers University, New Brunswick, NJ, October 1989.

[Met91] R. Metzger. Private communication, January 1991.

[MHK86] H. Muller, R. Hood, and K. Kennedy. Efficient recompilation of module interfaces in a software development environment. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual Symposium on Principles of Programming Languages*. ACM, January 1990.

[MS91] R. Metzger and P. Smith. The CONVEX application compiler. *Fortran Journal*, 3(1):8–10, 1991.

[Mul86] H. Muller. *Rigi – A Model for Software System Construction, Integration, and Evolution Based on Module Interface Specifications*. PhD thesis, Rice University, Houston, TX, August 1986.

[Mye81] E. Myers. A precise inter-procedural data flow algorithm. In *Conference of the Eighth Annual Symposium on Principles of Programming Languages*. ACM, January 1981.

[RG89a] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.

[RG89b] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. *Software—Practice and Experience*, 19(2):149–169, February 1989.

[Ros79] B. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, 1979.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN 89 Conference on Program Language Design and Implementation*, June 1989.

[Ryd79] B. Ryder. Constructing the call graph of a program. *IEEE Trans. on Software Engineering*, SE-5:216–225, May 1979.

[Sch77] R. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977.

[Sch89] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation*. ACM, June 1989.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*. ACM, June 1988.

[Spi71] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the IFIP Congress 1971*, pages 376–381, Amsterdam, 1971. North Holland.

[TB85] W. F. Tichy and M. C. Baker. Smart recompilation. In *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages*. ACM, January 1985.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, June 1986.

[Tor85] L. Torczon. *Compilation Dependencies in an Ambitious Optimizing Compiler*. PhD thesis, Rice University, Houston, TX, May 1985.

[Wal76] K. Walter. Recursion analysis for compiler optimization. *Communications of the ACM*, 19(9):514–516, September 1976.

[Weg81] M. Wegman. *General and Efficient Methods for Global Code Improvement*. PhD thesis, University of California, Berkeley, CA, December 1981.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the*

Seventh Symposium on Principles of Programming Languages, January 1980.

[Wol86] M. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[WZ85] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages.* ACM, January 1985.

[WZ89] M. Wegman and K. Zadeck. Constant propagation with conditional branches. Technical Report CS-89-36, Dept. of Computer Science, Brown University, May 1989.

[Yer66] A. P. Yershov. Alpha - an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, June 1984.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.