INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Interprocedural constant propagation: A study of jump function implementations

Grove, Daniel Dwight, M.S.

Rice University, 1993
RICE UNIVERSITY

Interprocedural Constant Propagation: A Study of Jump Function Implementations

by

Daniel D. Grove

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Linda Torczon, Faculty Fellow, Chair
Computer Science

John K. Bennett, Assistant Professor
Electrical and Computer Engineering

Keith D. Cooper, Associate Professor
Computer Science

Houston, Texas

April, 1993
Interprocedural Constant Propagation: A Study of Jump Function Implementations

Daniel D. Grove

Abstract
Procedure calls have long been recognized as an impediment to performance in compiled code. This happens because procedure calls hide information from the compiler. Interprocedural constant propagation attempts to discover the formal parameters and global variables that are constant on every invocation of a procedure. An implementation of interprocedural constant propagation must model the transmission of values through each procedure in the program. In the framework proposed by Callahan, Cooper, Kennedy, and Torczon, this transmission is modeled with jump functions. While Callahan et al. propose several jump functions, they give no data to help choose among them. This thesis describes the results obtained by employing several jump functions. Our study examined scientific FORTRAN codes. It shows that different jump functions find different numbers of constants, and suggests a particular function, the pass-through parameter jump function, as the most cost-effective in practice. The importance of interprocedural MOD information is also discussed.
Acknowledgments

I wish to thank my committee: John Bennett, Keith Cooper, and Linda Torczon. The contributions of both Keith Cooper and Linda Torczon throughout my undergraduate and graduate years at Rice are too broad to fully recognize here. Their friendship and advice helped make my career at Rice a truly enjoyable and rewarding experience. John Bennett’s interest and criticisms were quite helpful in preparing this work.

I would also like to thank Paul Havlak for providing the SSA infrastructure which I used in this study and for much advice and support along the way. Mary Hall and René Rodriguez provided the ParaScope call-graph infrastructure which supported this project. Preston Briggs also provided wisdom and guidance for which I am deeply grateful. In addition, I would like to thank all those who worked on the ParaScope project.

I would also like to thank my friends who gave me the support, consolation, and guidance that I needed in this endeavor: Christine Logar, Dave Schwartz, and Bill Mark. Without their help, this would not have been possible.

My family certainly deserves a great deal of thanks – without their help, I would never have made it this far. Special thanks to my sister, Elisabeth Grove, for encouraging me when I needed it the most.
Contents

Abstract ...................................................... ii
Acknowledgments ............................................. iii

1 Introduction .............................................. 1
1.1 Motivation ............................................. 1
1.2 Study .................................................. 2
1.3 Overview .............................................. 2

2 Background .............................................. 4
2.1 Motivation ............................................. 4
2.2 Proposed Solutions .................................... 5

3 Interprocedural Constant Propagation .................. 9
3.1 Theoretical Framework ................................. 9
3.2 Jump Functions ....................................... 10

4 Jump Functions .......................................... 12
4.1 Forward Jump Functions ............................... 12
  4.1.1 The Literal Constant Jump Function .......... 13
  4.1.2 The Intraprocedural Constant Jump Function .. 13
  4.1.3 The Pass-Through Parameter Jump Function .... 14
  4.1.4 The Polynomial Parameter Jump Function .... 15
4.2 The Return Jump Function ............................. 15
4.3 Jump Function Summary ............................... 16

5 Implementation .......................................... 18
5.1 Generating Return Jump Functions .................. 19
5.2 Generating Forward Jump Functions ................ 19
5.3 Interprocedural Propagation of Constants .......... 19
5.4 Recording the Results ........................................... 20
5.5 Alternative Implementation Strategy .............................. 20

6 Study ..................................................................... 22

7 Conclusion ................................................................ 27
  7.1 Contributions ......................................................... 27
Chapter 1

Introduction

1.1 Motivation

The use of a modular programming style eliminates information which is available to the compiler when compiling monolithic programs. Procedure calls generally prevent the compiler from making use of information about the environment that the procedure inherits and the side effects of procedures that are invoked. Interprocedural data-flow analysis attempts to overcome this problem by propagating information about data usage and creation among the procedures of the program. The compiler can then take advantage of contextual information to generate better code for any single procedure. In particular, a compiler can generate better code for a procedure if it knows at compile time which parameters and global variables will have constant values, and what those constant values will be at the procedure's invocation. These constants are called interprocedural constants. Interprocedural constants are used to improve the results of optimization in a variety of different ways, including:

- Dependence analysis – Interprocedural constants affecting loop bounds, loop strides, and control flow can often eliminate dependences [14].

- Automatic parallelization – Knowledge of interprocedural constants can be useful for proving that portions of code may execute in parallel. Loop bounds and strides can help determine when parallel execution is profitable by exposing the available parallelism per processor [9, 15].

- Procedure cloning – Interprocedural constants can be used to guide procedure cloning [6, 13].

- Intraprocedural constant propagation – Interprocedural constants can substantially increase the numbers of constants detected by intraprocedural constant propagation [13].
Several techniques for discovering interprocedural constants have been proposed [3, 4, 13, 16] (See Section 2, “Other Work”).

1.2 Study

The study presented in this thesis is based on the work of Callahan, Cooper, Kennedy, and Torczon [4]. They view interprocedural constant propagation as a problem with three key parts: modeling the flow of values through procedures, interprocedural constant propagation, and intraprocedural constant propagation. Callahan et al. suggest that the transmission of values within a procedure can be modeled using jump functions. A jump function describes the value of a variable at some point within a procedure as a function of a subset of intraprocedural constants, incoming parameters and global variables, and variables whose values are returned from procedures called by the procedure. If a variable’s value cannot be determined at compile time from this information, its value is considered to be indeterminate. After jump functions are constructed, interprocedural constant propagation is performed to transmit these values around the call graph. This problem can be solved by using any of the methods that are typically employed to solve interprocedural data-flow problems. After the values obtained from jump functions are propagated around the call graph, intraprocedural constant propagation is used to propagate the values of constant parameters to their uses within each procedure. Decomposing this problem in this way allowed us to examine the element which had not yet been fully considered: modeling the flow of values from procedure entries to call sites.

In this study, we examine the results obtained by several jump function implementations. The amount of time and the complexity of the data structures required to generate different jump functions vary with the jump function’s complexity. Our goal is to understand the relationship between the number of constants found using different jump functions and the complexity of the jump functions.

1.3 Overview

This thesis presents the results of a study of the number of constants found when several different jump functions were used to model the transmission of values through each procedure in a program. Chapter 2 illustrates the motivations for interprocedural constant propagation and methods which have been proposed for the solution of
the problem. Chapter 3 presents the jump function framework employed for inter-procedural constant propagation. Chapter 4 describes the jump functions which we examined. Chapter 5 presents the implementation we used for our study. Chapter 6 describes the results of our study. Chapter 7 presents our conclusions.
Chapter 2

Background

2.1 Motivation

Interprocedural constant propagation has been suggested as one way to improve the effectiveness of many tests, including dependence analysis [14]. In addition, information about interprocedural constants can be used to assist program transformations, such as automatic parallelization and procedure cloning [6, 9, 13, 15]. The availability of these constants can also improve the effectiveness of a variety of intraprocedural optimizations through improved intraprocedural constant propagation [13].

Interprocedural constant propagation has been proposed as a way of significantly improving the precision of dependence tests. Shen, Li, and Yew examined the improvement in dependence information that is obtained when the values of interprocedural constants are known [14]. In their experiment, they performed the interprocedural constant propagation manually and added assertions to the program text about constant variable values. In their examination of FORTRAN library routines, they found that knowledge of interprocedural constants significantly decreased the number of nonlinear subscript references. They found that approximately 50 percent of the subscripts which had previously been considered nonlinear were found to be linear in the presence of interprocedural constant information. Because many dependence analyzers are incapable of analyzing nonlinear subscripts, this is an important application of interprocedural constants.

Program parallelization algorithms can also be enhanced by the availability of interprocedural constants. Eigenmann and Blume show an example where knowledge of an induction variable increment allows a loop to be parallelized [9]. In their example, the value must be determined through interprocedural constant propagation. They suggest that loop bounds and strides are often constructed from interprocedural constants. This information helps determine whether a loop is worth parallelizing. Hennessy and Singh also examined the problem of program parallelization and found that knowledge of interprocedural constants that are used as loop bounds or incre-
ments can improve the degree to which a program can be parallelized [15]. The importance of parallelization will rise as the use of multiprocessing expands.

In addition to helping with program parallelization, interprocedural constants can be used to improve single-processor code by guiding procedure cloning. Cooper, Hall and Kennedy investigated this possibility when they developed their procedure cloning algorithm [6]. In *goal-directed cloning*, all expressions in dimension statements, control flow statements, and subscript expressions are examined. In their scheme, formals that appear in these expressions are marked as "important". These formals must be constant in order for these expressions to be constant. Based on this information, they are able to determine which parameters and globals could, if constant, improve dependence information in each procedure. This information helps limit the number of clones that are created. In this way, interprocedural constant propagation can be used to drive the cloning process by examining the flow of constants through various control flow paths within the program.

Metzger and Stroud show an empirical study which demonstrates that goal-directed cloning based on interprocedural constants can substantially increase the number of interprocedural constants available for use by later analysis and optimization passes in a compiler [13]. Experience with the CONVEX Application Compiler has shown that interprocedural constants can be used to increase the effectiveness of intraprocedural constant propagation, dependence analysis, dead code elimination, strip mining, loop interchange, and interprocedural optimizations.

Thus, the literature suggests that interprocedural constant propagation has a positive impact on several program restructuring techniques. Enlarging the set of known constants can improve parallelization. As multiprocessing becomes more common, the ability to perform this sort of transformation will become increasingly valuable. In addition, interprocedural constant propagation can be used to control procedure cloning. Because cloning can also improve dependence analysis, this will further improve the effectiveness of program parallelization techniques. In addition, interprocedural constants can be used to sharpen the results of a variety of intraprocedural optimizations.

### 2.2 Proposed Solutions

Several different solutions to the interprocedural constant propagation problem have been proposed. Burke and Cytron examined this problem as part of a framework
for performing interprocedural dependence analysis [3]. Wegman and Zadeck proposed another solution that involves inlining procedure calls to allow the interprocedural problem to be solved using intraprocedural techniques [16]. Callahan, Cooper, Kennedy, and Torczon propose summarizing information about actual parameters in terms of values which may be known on entry to each procedure [4]. Metzger and Stroud suggest that combining simple jump functions with controlled cloning algorithms allows compile times to remain reasonable while obtaining many of the benefits which can be obtained through inlining [13].

Burke and Cytron propose that the interprocedural constant propagation problem can be solved by repeated forward passes over the call graph [3]. They suggest that intraprocedural constant propagation be employed to determine which constants are available for propagation at call sites. Whenever a formal or global parameter to a procedure is discovered to be constant, the value of the constant can be propagated from the procedure entry to its uses via DEF-USE chains\(^1\). Because these DEF-USE chains do not change between passes over the call graph, they are only computed once. Once the values of the actual parameters at each call site have stabilized, intraprocedural constant propagation may be used to introduce the values of constant formal parameters into the procedure bodies. As described, the method does not handle returned constants or make paths through the call graph explicit.

Wegman and Zadeck propose a solution to this problem after demonstrating several methods for intraprocedural constant propagation [16]. They suggest that procedure integration combined with intraprocedural constant propagation can solve the interprocedural constant propagation problem. Because procedure integration makes paths through the program's call graph explicit, the interprocedural information computed along a particular path may be improved (this improvement is not limited to constants, but may also include the MOD, REF, and ALIAS sets). By combining the results of the improved analysis with dead code elimination, it may be possible to eliminate some paths through the program (See Chapter 6). Data is not yet available to indicate whether or not the proposed algorithm would perform efficiently in practice. However, code growth under this scheme is potentially very large. In addition, this method suggests that interprocedural constants alone are a good basis for guiding inlining decisions. Other work in this area suggests that interprocedural constants do not provide adequate information for guiding these decisions [7].

\(^1\)This proposal is very similar to our pass-through parameter jump function. See Section 4.1.3.
Callahan et al. present an interprocedural constant propagation framework that consists of an algorithm for propagating constants across call boundaries and a family of methods, called jump functions, used to propagate constants through procedure bodies [4]. Jump functions are calculated as initial information prior to the solution of the interprocedural constant propagation data-flow problem. Both interprocedural constants passed to called procedures and those returned by called procedures can be detected using this technique. Because this technique does not make paths through the call graph explicit, it potentially detects fewer constants than the method proposed by Wegman and Zadeck. This technique has been implemented in several compilation systems and has proved to be efficient in practice. This thesis examines the tradeoff between the complexity of jump functions and the classes of constants that they find.

Metzger and Stroud implemented an interprocedural constant propagation algorithm in the CONVEX Application Compiler based on the ideas presented in Callahan et al. [13]. They use interprocedural constants to guide procedure cloning. Empirical results presented by Metzger and Stroud indicate that goal-directing cloning of procedures based on interprocedural constants can substantially increase the number of interprocedural constants available for use by later analysis and optimization passes in a compiler. This improvement is found partly because cloning makes some paths through the call graph explicit. The results of this study show that a substantial number of constants can be found when a simple jump function is used to represent the flow of values through procedures.

The different approaches to the solution of this problem fall into two groups: those that treat it as an interprocedural data-flow problem, and the approach that converts it into an intraprocedural problem. The interprocedural data-flow solutions differ in their methods of modeling the flow of the values of parameters and globals from a procedure's entry to the call sites it contains. In addition, these algorithms differ in their ability to process constant values returned by procedures and functions. The solution proposed by Wegman and Zadeck may produce more precise data about interprocedural constants, but data about the efficiency of this technique in practice is not yet available. Both Wegman and Zadeck and Metzger and Stroud modify the call graph in order to make paths through the graph explicit. Therefore, they may find more constants than the other methods. The other methods, if tested on these modified programs, would probably also discover the additional constants. This thesis
attempts to determine the effectiveness of the technique proposed by Callahan et al. using a variety of different jump functions.
Chapter 3

Interprocedural Constant Propagation

3.1 Theoretical Framework

The goal of interprocedural constant propagation is to discover, for each procedure \( p \) in the program, the set of constants that always hold on entry to \( p \). In data-flow terms, we would like to annotate \( p \) with a set \( \text{CONSTANTS}(p) \) containing \( \langle \text{name}, \text{value} \rangle \) pairs. A pair \( \langle x, v \rangle \in \text{CONSTANTS}(p) \) indicates that \( x \) always has value \( v \) when \( p \) is invoked. This set conservatively approximates reality; each pair in \( \text{CONSTANTS}(p) \) denotes a run-time constant, but not all run-time constants can be found (for example, values read from a file may be used to form constants that propagate through the program).

For the purpose of computing \( \text{CONSTANTS}(p) \), we formulate the problem in a lattice theoretic framework. With each entry point \( p \), we associate a function that maps formal parameters\(^2\) of \( p \) into elements of the constant propagation lattice \( L \). Each element of \( L \) is one of three types: the lattice top element (\( T \)), a constant value \( c \), or the lattice bottom element (\( \perp \)). The lattice's structure is shown at the top of Figure 3.1. It is defined by the rules for the lattice meet operation (\( \wedge \)). These rules are shown at the bottom of Figure 3.1. Here, "any" denotes a valid element of \( L \). The lattice, although infinite, has bounded depth. In particular, the value associated with some formal parameter \( x \) can be lowered at most twice. For any formal parameter \( x \), let \( \text{VAL}(x) \) represent the best current approximation to the value of \( x \) on entry to the procedure. After the analysis is complete, if \( \text{VAL}(x) = \perp \), the analysis was not able to determine that \( x \) is constant. (\( x \) may or may not be constant. If \( x \) is constant, the analyzer was unable to discover it.) If \( \text{VAL}(x) = c \), \( x \) has the constant value \( c \). The value \( T \) is used as an initial approximation for all parameters; a parameter retains that value only if the procedure containing it is never called. Given \( \text{VAL}(x) \) for each

\(^2\)To simplify the presentation, we extend the definition of a parameter to include global variables. Recall that formal parameter refers to the name used inside the called procedure. The reference in the parameter list at the call site is an actual parameter.
of \( p \)'s formal parameters, \( \text{CONSTANTS}(p) \) is simply the set of formal parameters for which \( \text{VAL} \) is equal to some constant value.

Interprocedural constant propagation is performed on a call graph, \( G \). Each node in \( G \) represents a procedure \( p \) and each edge \( (p, q) \) represents a particular call from \( p \) to \( q \). Interprocedural constant propagation has two parts: modeling the transmission of values through each procedure and propagating information in \( \text{VAL} \) sets around the call graph.

Discovering the flow of values through each procedure can be expensive. Typically, this has been done through \textit{intraprocedural} constant propagation passes performed on each routine in the program. Given the time required for this propagation, it is important to minimize the number of times it is performed. Large-scale scientific programs are too large to have all their procedures in memory on most machines. Therefore, it is important to avoid holding representations of all procedures in RAM at the same time. The construction of \textit{jump functions} to represent this flow of values allows us to achieve these goals.

### 3.2 Jump Functions

A \textit{jump function} is a method of describing the value of an actual parameter in terms of the values of formal parameters to the procedure in which it is used. In order to
model the flow of constant values through a procedure $p$, we associate with each call site $s$ in $p$ a set of forward jump functions, $J^y_s$. For an actual parameter $y$, $J^y_s$ gives the value of $y$ at $s$ as a function of $p$'s formal parameters, where $s$ is a call site in $p$. The support of $J^y_s$ is the exact set of $p$'s formal parameters whose values on entry to $p$ are used in the computation of $J^y_s$. We denote the support set of $J^y_s$ as $\text{support}(J^y_s)$.

Procedures can also return constant values. To account for this effect, we must model the transmission of constant values from a procedure back to the call site that invoked it. We associate a set of return jump functions with each procedure. For each formal parameter $x \in \text{MOD}(p)$, a return jump function $R^x_p$ is created to compute the best approximation to $x$'s value on return from $p$. The support of the function $R^x_p$ is the exact set of formal parameters of $p$ whose values on entry to $p$ are used in the computation of $R^x_p$. As before, we denote the support set of $R^x_p$ as $\text{support}(R^x_p)$.

Given the call graph $G$ and the forward and return jump functions for each procedure, we can propagate the VAL sets around the graph using the standard iterative technique [11, 12]. The results presented in this paper were computed using a simple worklist iterative scheme. Alternative formulations based on the binding multi-graph are also possible. The method presented by Callahan et al. essentially models the binding graph computation on the call graph [4].

Callahan et al. showed a single simple example to demonstrate that different jump function techniques produced different results. They fail, however, to provide any experimental evidence that suggests where the tradeoff lies between jump function complexity and the precision of the CONSTANTS sets. This thesis examines the effectiveness of several jump functions. From this comparative data, we can assess the value of spending additional compile time in jump function evaluation.
Chapter 4

Jump Functions

In order to study the tradeoffs between compile time and precision that arise in an implementation, we built an analyzer that can use one of several different jump functions. Our choice of implementation was driven by the goals set for our study: understanding the tradeoff between more complex jump functions and the number of interprocedural constants discovered. Thus, we chose an implementation strategy that differs from the one described by Callahan et al. We built a set of jump functions on top of an existing framework for global (intraprocedural) value numbering [1]. Callahan et al. envisioned constructing jump functions before any interprocedural analysis is performed. We found it convenient to generate the jump functions on top of global value numbering after the computation of interprocedural summary sets (MOD and REF). While this order of operations was convenient for our experiments, practical considerations might dictate that a production compiler generate jump functions earlier and parameterize MOD and REF information that has yet to be determined. Section 5.5 contains more details on this possibility. Section 4.1 describes the four forward jump functions that we considered. Section 4.2 describes the return jump function that we implemented.

4.1 Forward Jump Functions

The four forward jump functions vary both in the complexity of their construction and in the classes of interprocedural constants that they can detect. In increasing order of complexity, they are: the literal constant jump function, the intraprocedural constant jump function, the pass-through parameter jump function and the polynomial parameter jump function. The set of constants propagated by one of these forward jump functions is a subset of the constants propagated by the more complex models that follow it.

Each of the forward jump functions selected for this study can be constructed as initial information before the interprocedural propagation phase begins. It is not necessary to reconstruct the jump functions on each iteration over $G$. For simplicity in defining these forward jump functions, let $gcp(y,s)$ represent a function, computed prior to the interprocedural propagation phase, that returns the constant value of parameter $y$ at call site
s when that value can be determined with intraprocedural constant propagation or value numbering coupled with interprocedural MOD information.

4.1.1 The Literal Constant Jump Function

The literal constant jump function is defined as follows:

\[
J^y_s = \begin{cases} 
\text{c} & \text{if } y \text{ is the literal constant } c \text{ at call site } s \\
\bot & \text{otherwise}
\end{cases}
\]

It does not take into account the values of incoming formal parameters at all. Thus, the literal constant jump function does not propagate interprocedural constants through the procedure body. As a result, interprocedural constants can only be propagated over a single edge in \( G \). The only interprocedural constants propagated by this method are those that appear as a literal constant at a call site. The construction of these jump functions is trivial: no MOD information is needed and no intraprocedural constant propagation must be performed. The generation of jump functions is easily accomplished by performing a textual scan of the arguments that are present at each call site. This requires significantly less time than the other forward jump functions, which need an intraprocedural constant propagation phase. In addition, because each actual parameter value will only be transmitted once across each edge, the solution to the data flow problem will require \( O(\sum_s \sum_y \text{cost}(J^y_s)) \) time, where \( s \) ranges over all call sites, \( y \) ranges over all parameters at each call site, and \( \text{cost}(J^y_s) \) is the cost of evaluating \( J^y_s \). This method will not detect any constant global variables that are passed implicitly at the call site.

4.1.2 The Intraprocedural Constant Jump Function

The intraprocedural constant jump function is defined in the following manner:

\[
J^y_s = \begin{cases} 
\text{c} & \text{if } \text{gcp}(y, s) = c \\
\bot & \text{otherwise}
\end{cases}
\]

Like the literal constant jump function, this forward jump function ignores any values known on entry to the procedure for incoming formal parameters. Thus, it will only propagate an interprocedural constant along a path of length one in \( G \). However, since intraprocedural constant propagation is used to produce the constants propagated at the call sites, this approach can potentially detect more constants than the literal constant jump function. Unlike the literal constant jump function, this jump function requires a phase of intraprocedural constant propagation prior to the interprocedural phase. In our SSA-based implementation, this typically requires \( O(N) \) steps in each procedure, where \( N \) is the size of the procedure
(generally the maximum of the number of nodes, edges, variable assignments, or variable references in the procedure) [8]. Like the literal constant jump function, the solution to the interprocedural data-flow problem requires \( O(\sum_s \sum_y \text{cost}(J_y^s)) \) time, where \( s \) ranges over all call sites in the program, \( y \) ranges over all actual parameters and globals at each call site, and \( \text{cost}(J_y^s) \) is the cost of evaluating \( J_y^s \). Unlike the literal constant jump function, this jump function will detect constant global variables which are passed implicitly at call sites.

### 4.1.3 The Pass-Through Parameter Jump Function

The pass-through parameter jump function is defined as the following:

\[
J^s_y = \begin{cases} 
  c & \text{if } gcp(y, s) = c \\
  x & \text{if } y = x \text{ at } s \text{ and } x \text{ is a formal parameter of the calling procedure} \\
  \bot & \text{otherwise}
\end{cases}
\]

This jump function detects situations where the value of a formal parameter is passed through the procedure body, without modification, to a call site where it is used as an actual parameter. The generation of this jump function also requires an intraprocedural constant propagation phase before interprocedural analysis is performed. This intraprocedural analysis is somewhat more complex than that used in the intraprocedural constant jump function because paths from incoming formal and global parameters to their uses must be found and stored by the intraprocedural analyzer. This forward jump function will propagate interprocedural constants through a procedure body. Thus, interprocedural constants can be propagated along paths of length greater than one in \( G \). Because this analysis is similar to that used for the intraprocedural constant jump function, the time bound for this step is generally \( O(N) \), where \( N \) is the size of the procedure. The number of steps needed for this analysis should not be significantly greater than that needed for the intraprocedural constant jump function. The solution to the interprocedural data-flow problem may require more steps than the previous two jump functions. Because the constant lattice is shallow, each value can be lowered at most twice – from \( \top \) to a constant to \( \bot \). Each actual parameter depends on at most one formal parameter. Taken together, these facts suggests that the interprocedural constant propagation can be completed in \( O(\sum_s \sum_y \text{cost}(J_y^s)) \) time. The added complexity of this jump function causes \( \text{cost}(J_y^s) \) to be greater than it is for the intraprocedural constant and literal constant jump function. Callahan et al. presented an algorithm that achieves this time bound [4]. The implementation used in our experiment was less efficient. Global variables that are implicitly passed at call sites are detected by this jump function.
4.1.4 The Polynomial Parameter Jump Function

The polynomial parameter jump function is defined in the following manner:

\[
J^y_x = \begin{cases} 
  c & \text{if } gcp(y, s) = c \\
  f(\text{support}(J^y_x)) & \text{if } y \text{ at } s \text{ can be represented as a polynomial function } f \text{ of } \text{support}(J^y_x) \\
  \bot & \text{otherwise}
\end{cases}
\]

This jump function takes a more comprehensive approach towards propagating interprocedural constants through a procedure body than the pass-through parameter jump function. Actual parameters are represented as polynomial functions of the incoming values of the formal parameters of \( p \) when such a polynomial function can be constructed. All standard integer operations are supported by the routines which generate and evaluate these jump functions. This requires a more complex intraprocedural analysis than any of the previous jump functions. The intraprocedural propagation must be able to store and evaluate symbolic expressions. The data structures required for this are significantly more difficult to handle than those for the other forward jump functions. However, the time bound for this process, typically \( O(N) \), is the same as that for the pass-through parameter and intraprocedural constant jump functions [16]. The interprocedural analysis phase requires more steps than it does for the previous three jump functions. Each element in \( \text{support}(J^y_x) \) can be lowered at most twice, resulting in the re-evaluation of \( J^y_x \). Thus, the time required for the interprocedural propagation is [4]:

\[
O(\sum_s \sum_y \text{cost}(J^y_x) \cdot |\text{support}(J^y_x)|)
\]

In practice, we found that the number of complex polynomial jump functions actually constructed is small. Taken over the program, \( \text{cost}(J^y_x) \) approaches the cost of the pass-through parameter jump functions and \( |\text{support}(J^y_x)| \) approaches 1. These effects allow the solution time for the interprocedural constant propagations to approach that of the simpler forward jump functions, \( O(\sum_s \sum_y \text{cost}(J^y_x)) \).

4.2 The Return Jump Function

To understand the value of return jump functions, we implemented a single type of return jump function. It is similar to the forward polynomial parameter jump function. The values computed for return jump functions are calculated during an initial bottom-up pass
through the call graph. Interprocedural MOD information, return jump functions from routines already analyzed, and intraprocedural constants are used in the construction of each return jump function polynomial.

For simplicity in defining this return jump function, let \( rgcp(x, p) \) represent a function, computed prior to the interprocedural propagation phase, that returns the constant value of formal parameter \( x \) after invocation of \( p \) when that value can be determined with intraprocedural constant propagation coupled with interprocedural MOD information and information provided by return jump functions that have already been computed for call sites within \( p \). Formally, the return jump function is defined as follows:

\[
R_p^x = \begin{cases} 
  c & \text{if } rgcp(x, p) = c \\
  g(\text{support}(R_p^x)) & \text{if } x \text{ can be represented as a polynomial function } g \text{ of } \text{support}(R_p^x) \\
  \perp & \text{otherwise}
\end{cases}
\]

Return jump functions are generated by performing a separate intraprocedural constant propagation phase. The complexity of the data structures needed to perform this construction is the same as that of the polynomial parameter jump function. Again, the intraprocedural constant propagation typically requires \( O(N) \) time for each procedure [8]. Because these return values are only passed upwards to calling procedures, there is no real data-flow problem to be solved. Each return jump function is evaluated exactly twice at each call site. It is first evaluated during the generation of return jump functions for the calling procedure. This is done in order to expose as many return jump functions as possible in the calling procedure. The second evaluation of the return jump function takes place during the generation of forward jump functions. During this phase, any return jump function which cannot be evaluated as constant using intraprocedural information coupled with other return jump function values is set to \( \perp \). The effect of this limitation is that return jump functions which depend on parameters to the calling procedure can never be evaluated as constant. During the process of testing this program, this limitation was never evident. We found no cases where allowing return jump functions to depend on parameters to the called procedure would have discovered more constants.

### 4.3 Jump Function Summary

The four forward jump functions which were used for this study vary widely in the cost of their generation. In addition, the cost of solving the interprocedural constant propagation problem varies to some extent with the different forward jump functions. The return jump function that we implemented is capable of handling complex expressions of formal parameters of each procedure. These jump functions provide a good testbed for analyzing the
tradeoff between the complexity of generating different jump functions and the precision of the CONSTANTS sets that they find.
Chapter 5

Implementation

To study the relative effectiveness of the jump functions described in Section 4, we built an analyzer that implemented each of them. We limited the implementation in two ways:

1. The implementation only propagates integer constants. There are two principal reasons for this. First, we were most interested in discovering constants that affect control flow. These include loop bounds, loop strides, and the conditions that control if-then-else statements. Second, compile-time evaluation of floating-point numbers is problematic, particularly so in an environment like ours where the compiler and application may run on different architectures.

2. The analyzer does not try to track constants in and out of arrays. Dependence analysis might allow the analyzer to track some constant-valued array elements. We elected not to attempt this because the information from interprocedural constant propagation is used as input to the dependence analyzer. Thus, using dependence information inside a jump function would either introduce a phase-ordering problem or require iterative application of dependence analysis and interprocedural constant propagation.³

These limitations are in line with the goals of our study – to compare the sets of integer constants discovered with the jump functions described in Section 4.

The interprocedural constant propagation system used in this study was built on top of an SSA-based value number graph [8] and the call graph abstraction from ParaScope, a set of tools designed to aid in interactive parallelization of FORTRAN programs [5]. Its execution proceeds in four stages: generation of return jump functions, generation of forward jump functions, interprocedural propagation of constants, and recording the results. This implementation strategy was chosen for the purposes of this experiment only. A production implementation would probably want to consider more efficient implementations. Section 5.5 discusses alternate implementation strategies.

³We believe that tracking flow of values in and out of FORTRAN arrays would yield few additional interprocedural integer constants.
5.1 Generating Return Jump Functions

In the first phase, a return jump function is calculated for each variable modified by each procedure in a bottom-up walk over the call graph. At each node in the call graph, the analyzer builds an SSA graph for the corresponding procedure. To determine which parameters can have constant values on return from a routine, we value number its SSA graph. This value numbering pass incorporates both MOD information and values provided by already-computed return jump functions for subroutines invoked by the current routine. Building these graphs generally requires time which is linear in the size of each procedure [8]. After the return jump functions for a procedure have been built, the SSA graph, the value number graph, and the intermediate representation of the procedure are discarded in order to conserve memory. The data structures which hold the expressions representing the return jump functions are retained for later use.

5.2 Generating Forward Jump Functions

In the second phase, forward jump functions are generated at every call site which passes parameters or globals. In a top-down pass over the call graph, the analyzer builds the SSA graph and the value number graph for each routine. This analysis makes use of the return jump functions built in the first stage. Each parameter (including globals which are passed implicitly) associated with a call site receives a value number on entry. The value numbering proceeds in the standard way; it can build an arbitrarily complex representation for an arithmetic expression. The analyzer then constructs an expression tree to represent the parameter's jump function from the expression produced by value numbering. In our implementation, the value numbering builds the same representation for a given variable regardless of the jump function that is being used. The construction of the expression tree for the jump function is then limited by the type of jump function being tested. While this does not allow the user to benefit from the faster analysis times of the simpler jump functions, it provides a great deal of flexibility for testing the different jump functions. The final step in this phase is converting the expression trees into context-independent representations and storing these in the list of jump functions for the parameters of the call site.

5.3 Interprocedural Propagation of Constants

Once forward jump functions have been built, ParaScope's interprocedural data-flow solver is called to perform the actual propagation. This is a rather simple process. Meet and transfer functions were written; these evaluate the various jump functions. The data-flow
solver uses an iterative solution method. The running time for this phase is quite short – the use of an iterative data-flow solution does not lead to a long interprocedural phase. In our experimental implementation, the time spent in intraprocedural analysis is much greater than that spent in interprocedural analysis.

This meet function is called, in turn, for every parameter along each edge entering a procedure. The interprocedural data-flow solver uses an “optimistic” algorithm – initially, the values of all parameters in each procedure are initialized to T. The meet function implements the lattice meet function shown in Figure 3.1. It is evaluated using the values of parameters entering a procedure along an edge and the current approximations of these values for the procedure. If a global variable is used by the called procedure but is not declared in the common blocks of the calling procedure, its value is set to ⊥. For the other parameters and globals, an item-by-item meet is then performed, and the new approximations to the values of these variables are stored with the procedure’s node in the call graph. Any non-constant jump function is evaluated as ⊥.

Once the meet has been performed, the values of any constant parameters must be transmitted to edges leaving the node. If an incoming parameter which was previously evaluated as non-constant is now found to be constant, all non-constant parameters at each call site within that procedure are re-evaluated. This involves recomputing the values of all jump functions in the procedure; no analysis of the procedure itself is necessary. This process uses a symbolic expression evaluator to interpret the expression trees that represent each jump function. This evaluator uses all available information about incoming parameter values.

5.4 Recording the Results

Finally, the CONSTANTS sets are written to a single file. Optionally, the analyzer can produce a transformed version of the original source in the interprocedural constants are textually substituted into the code. The numbers reported in the next section count the number of constants that this option substituted into each program. Metzger and Stroud suggest that this is the right measurement of effectiveness [13]. In their experience, FORTRAN procedures often have constant-valued global variables that are known but irrelevant – that is, they are not referenced inside the procedure. This measure relates more directly to code improvement; it also factors out issues of procedure length and modularity.

5.5 Alternative Implementation Strategy

In a production compiler, where speed of compilation is an important factor, the framework outlined here may be unacceptably slow. Because we were more concerned with the
CONSTANTS sets that were found than with running time, we did not use the most efficient interprocedural data-flow techniques. While the iterative technique was fast enough for our experimental framework, a compiler that was more concerned with speed might use an interprocedural technique that converges to a solution more quickly. The method employed in this experiment is also slow at program compile time because it must calculate the jump functions at every call site after MOD and REF information has been calculated. It is possible to reverse the order of these steps. In this case, the jump functions would be built as initial information. In ParaScope, this means that the jump functions for a file would be generated when it is saved in the editor. These jump functions would include the necessary tests for MOD and REF information as part of their definitions. This would make the representations of these jump functions substantially more complex than they are in our experimental model. However, it would eliminate the need to perform the expensive jump function generation steps at program compile time. In our experience, the generation of jump functions requires much more time than the interprocedural constant propagation. Generating jump functions as initial information fits in well with the ParaScope system — local MOD, REF, and ALIAS sets are already calculated in this manner. Under this scheme, significant time savings occur when routines are used in more than one program. Additional time savings would be realized when a program is recompiled often — in our experimental implementation, we must recompute all jump functions each time a compilation takes place. Using this alternate implementation strategy, only the jump functions for modified procedures would need to be recalculated at compile time. A production compiler might find this implementation strategy more reasonable (although more complex) than that employed in this experiment.
Chapter 6

Study

In order to evaluate the relative effectiveness of the jump functions, we chose a set of scientific FORTRAN programs. We elected to evaluate the SPEC and PERFECT benchmark suites. Because ParaScope does not currently support multiple entry points, some of the PERFECT suite had to be removed from consideration.⁴

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Procedures</th>
<th>Median Lines/Procedure</th>
<th>Mean Lines/Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>adm</td>
<td>3809</td>
<td>98</td>
<td>29</td>
<td>39</td>
</tr>
<tr>
<td>doduc</td>
<td>4514</td>
<td>41</td>
<td>78</td>
<td>110</td>
</tr>
<tr>
<td>fpppp</td>
<td>1835</td>
<td>38</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>linpackd</td>
<td>392</td>
<td>11</td>
<td>25</td>
<td>36</td>
</tr>
<tr>
<td>matrix300</td>
<td>153</td>
<td>6</td>
<td>23</td>
<td>26</td>
</tr>
<tr>
<td>mdg</td>
<td>856</td>
<td>18</td>
<td>25</td>
<td>48</td>
</tr>
<tr>
<td>ocean</td>
<td>1728</td>
<td>37</td>
<td>22</td>
<td>46</td>
</tr>
<tr>
<td>qcd</td>
<td>1664</td>
<td>36</td>
<td>32</td>
<td>46</td>
</tr>
<tr>
<td>simple</td>
<td>805</td>
<td>8</td>
<td>49</td>
<td>101</td>
</tr>
<tr>
<td>snasa7</td>
<td>696</td>
<td>17</td>
<td>34</td>
<td>41</td>
</tr>
<tr>
<td>spec77</td>
<td>2904</td>
<td>65</td>
<td>31</td>
<td>45</td>
</tr>
<tr>
<td>trfd</td>
<td>401</td>
<td>8</td>
<td>40</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 6.1 Characteristics of program test suite.

Some characteristics of the programs that were tested are shown in Table 6.1. In general, the programs are of small or medium size, with a relatively high degree of modularity. The line counts exclude comments and blank lines. With the exceptions of fpppp and simple, a fairly even distribution of code throughout the procedures in each program was found (this is shown by the closeness of the mean lines per procedure and the median lines per procedure.

⁴Because the multiple entry points in ocean were never called, they were eliminated in order to include ocean in the study. We eliminated tomcatv from the test suite because the entire program is a single procedure.
in each program). In these two programs, the distribution of the lines per procedure was skewed – a single routine made up a large part of the code in fpmp and simple. While these programs may not be representative of all styles of scientific computing, the two suites of programs are well known and widely used in studies like this one.

<table>
<thead>
<tr>
<th>Program</th>
<th>Using Return Jump Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Polynomial</td>
</tr>
<tr>
<td>adm</td>
<td>110</td>
</tr>
<tr>
<td>doduc</td>
<td>289</td>
</tr>
<tr>
<td>fpmp</td>
<td>60</td>
</tr>
<tr>
<td>linpackd</td>
<td>170</td>
</tr>
<tr>
<td>matrix300</td>
<td>138</td>
</tr>
<tr>
<td>mdg</td>
<td>41</td>
</tr>
<tr>
<td>ocean</td>
<td>194</td>
</tr>
<tr>
<td>qcd</td>
<td>180</td>
</tr>
<tr>
<td>simple</td>
<td>183</td>
</tr>
<tr>
<td>snasa7</td>
<td>336</td>
</tr>
<tr>
<td>spec77</td>
<td>137</td>
</tr>
<tr>
<td>trfd</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6.2 Constants found through use of forward and return jump functions.

Table 6.2 shows the number of constants discovered and actually substituted into the code by the analyzer using each of the forward jump functions. The table shows the number of constants obtained by using both forward and return jump functions. In our test suite, the polynomial and pass-through parameter jump functions found the same set of constants. Both outperformed the intraprocedural constant jump function on four of the twelve programs. The intraprocedural constant jump function performed better than the literal constant jump function on nine of the twelve programs.

Table 6.3 compares the number of constants found in both the presence and absence of return jump functions. The polynomial and pass-through parameter functions always found the same number of constants, regardless of the presence of return jump functions. The first column shows the number of constants found by the pass-through parameter jump function when return jump functions are used (this column is identical to the second column in Table 6.2). The second column shows the results for this forward jump function without return jump functions. Return jump functions made no difference in eight of the twelve programs. In doduc, fpmp, and mdg, return jump functions let the analyzer find a few more
constants. In ocean, the return jump functions more than tripled the number of constants that the analyzer found. By recognizing that the initialization routine at the start of ocean resulted in the assignment of constant values to many variables, the analyzer was able to propagate additional constants to routines throughout the program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Using Return Jump Functions</th>
<th>No Return Jump Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pass-through</td>
<td>Pass-through</td>
</tr>
<tr>
<td>adm</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>doduc</td>
<td>289</td>
<td>287</td>
</tr>
<tr>
<td>fpppp</td>
<td>60</td>
<td>56</td>
</tr>
<tr>
<td>linpackd</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td>matrix300</td>
<td>138</td>
<td>138</td>
</tr>
<tr>
<td>mdg</td>
<td>41</td>
<td>40</td>
</tr>
<tr>
<td>ocean</td>
<td>194</td>
<td>62</td>
</tr>
<tr>
<td>qcd</td>
<td>180</td>
<td>180</td>
</tr>
<tr>
<td>simple</td>
<td>183</td>
<td>183</td>
</tr>
<tr>
<td>snasa7</td>
<td>336</td>
<td>336</td>
</tr>
<tr>
<td>spec77</td>
<td>137</td>
<td>137</td>
</tr>
<tr>
<td>trfd</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

**Table 6.3** Constants found with and without return jump functions.

Table 6.4 shows the results of several different constant propagation techniques. The first column shows results from a propagation using the polynomial parameter forward and return jump functions and no MOD information. The second column is identical to the first column in Table 6.2. The third column shows the results of a "complete" propagation. To obtain these numbers, the analyzer was run with polynomial parameter forward and return jump functions, using MOD information. After each run, dead code elimination was performed. If any dead code was found, the propagation was performed again from scratch - all of the values in the CONSTANTS sets were reset to T.\(^5\) The final column shows the results of performing a purely intraprocedural constant propagation on each procedure in the program. No constants were propagated between procedures, but interprocedural MOD information was used during the intraprocedural constant propagation phases.

In order to assess the importance of MOD information, we eliminated the MOD information from both the initial information and the jump functions. The value numbering

\(^5\)In each case, only one pass of dead code elimination was needed. The second propagation did not expose any newly dead code.
<table>
<thead>
<tr>
<th>Program</th>
<th>Polynomial without MOD</th>
<th>Polynomial with MOD</th>
<th>Complete Propagation</th>
<th>Intraprocedural Propagation with MOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>adm</td>
<td>25</td>
<td>110</td>
<td>110</td>
<td>105</td>
</tr>
<tr>
<td>doduc</td>
<td>288</td>
<td>289</td>
<td>289</td>
<td>3</td>
</tr>
<tr>
<td>fpppp</td>
<td>34</td>
<td>60</td>
<td>60</td>
<td>38</td>
</tr>
<tr>
<td>linpackd</td>
<td>33</td>
<td>170</td>
<td>170</td>
<td>74</td>
</tr>
<tr>
<td>matrix300</td>
<td>18</td>
<td>138</td>
<td>138</td>
<td>69</td>
</tr>
<tr>
<td>mdg</td>
<td>31</td>
<td>41</td>
<td>41</td>
<td>31</td>
</tr>
<tr>
<td>ocean</td>
<td>79</td>
<td>194</td>
<td>204</td>
<td>56</td>
</tr>
<tr>
<td>qcd</td>
<td>169</td>
<td>180</td>
<td>180</td>
<td>179</td>
</tr>
<tr>
<td>simple</td>
<td>2</td>
<td>183</td>
<td>183</td>
<td>174</td>
</tr>
<tr>
<td>snasa7</td>
<td>303</td>
<td>336</td>
<td>336</td>
<td>254</td>
</tr>
<tr>
<td>spec77</td>
<td>76</td>
<td>137</td>
<td>141</td>
<td>83</td>
</tr>
<tr>
<td>trfd</td>
<td>10</td>
<td>16</td>
<td>16</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 6.4 Comparison of most precise jump function with other propagation techniques.

phase was forced to make the conservative assumption that all parameters at each call site were modified in the called procedure. Comparing these results (see column 1) against those obtained with MOD information (see column 2), we see a substantial difference. In every program, using MOD information exposed additional constants. The numbers are particularly striking in adm, linpackd, matrix300, ocean, simple, and spec77. In the main routine of matrix300, for example, a series of calls are made using a number of the same constant-valued parameters at each call site. At each call after the first, the value numbering pass must assume, in the absence of MOD information, that all of these parameters were modified and now are evaluated as 1. The presence of any call in a routine eliminated potential constants along paths leaving the call site, even when no modification occurred. The data indicates that incorporating MOD information significantly increases the number of constants that can be detected in scientific FORTRAN codes.

Incorporating interprocedural constants into a routine can expose segments of code that can never be executed [16]. Removing this dead code can potentially eliminate conflicting definitions of variables and expose additional constants. Combining dead code elimination with interprocedural constant propagation in the “complete” propagation (see column 3) exposed few additional constants. A relatively small amount of code was eliminated in this way – most programs were completely unaffected. In linpackd, however, a substantial amount of code was eliminated. This is due to the fact that it contains a great deal of code which is tailored to specific matrix dimensions that are not used in the program. The
additional effort of a "complete" propagation does not appear justified by the small number of additional constants which were found.

In fact, the results that were obtained in this study by the "complete" propagation can be achieved by basing the jump function generator on a gated single-assignment form [2, 10]. An analyzer based on gated single-assignment form would never consider the dead assignments which were eliminated in the "complete" propagation. This would let the polynomial or pass-through parameter jump functions produce the same results as the "complete" propagation in this study.

Finally, the results of an intraprocedural constant propagation are presented to show how many of the constants substituted were already available within the procedures. For fair comparison, MOD information was used in the intraprocedural constant propagation. The intraprocedural constant propagation would have found fewer constants without MOD information. In each program, the interprocedural constant propagation (see column 2) detected more constants than the strictly intraprocedural constant propagation (see column 4). In some cases, the difference was substantial. Interprocedural constant propagation exposed many constants in scientific FORTRAN codes that were not found by the intraprocedural constant propagation coupled with MOD information.

The pass-through parameter jump function yielded exactly the same set of constants as the polynomial parameter jump function for this set of programs. Therefore, the additional expense and effort involved in generating the polynomial parameter jump function does not appear to be worthwhile. The use of a return jump function is necessary to ensure that interprocedural constant propagation finds values that are set in initialization routines. The use of MOD information greatly enlarges the constant sets found. When relatively precise forward and return jump functions are used with MOD information, interprocedural constant propagation finds many constants that can not be found through strictly intraprocedural analysis.

---

6Note that the information from return jump functions is used during the construction of the gated single-assignment graph. Therefore, a limited number of interprocedural constants may be available for this construction.
Chapter 7

Conclusion

A variety of solutions have been proposed to the interprocedural constant propagation problem. Callahan et al. formulate this as an interprocedural data-flow problem. They use jump functions to model the flow of values from the entry point of a procedure to the uses of those variables at call sites. We have examined the viability of this model and the results that can be obtained by using a variety of different jump functions.

7.1 Contributions

This thesis presents the results obtained by employing several different jump functions within the interprocedural constant propagation framework described by Callahan et al. Four forward jump functions were examined. We have found that the use of the most complex forward jump function, the polynomial parameter jump function, does not provide better results than the relatively simple pass-through parameter jump function in our test suite of scientific FORTRAN programs. Our work supports the use of the pass-through parameter jump function as the best balance between complexity and the number of constants found. We have also examined the importance of return jump functions. In most cases, we found that return jump functions did not improve the precision of the CONSTANTS sets that were found. In programs which use an initialization routine, however, the number of constants found when return jump functions are used can be significantly greater than the number found in their absence. This shows that there are constants defined in this way and that return jump functions must be used in order to ensure that these constants are found when using a framework such as that described by Callahan et al. We also examined the importance of using MOD information when jump functions are generated. Our data show that the absence of MOD information severely degraded the CONSTANTS sets that were found, even when our most complex forward and return jump functions were used.

Scientific FORTRAN programs use some constants that cannot be detected with intraprocedural constant propagation, even when interprocedural MOD information is used. Our study showed that interprocedural constant propagation can discover a subset of these constants, even with simple jump functions. The differences in sets of constants discovered by different jump functions are significant.
Bibliography

Detecting equality of variables in programs.

The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages.

Interprocedural dependence analysis and parallelization.
Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.

Interprocedural constant propagation.
Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.

The ParaScope parallel programming environment.

Procedure cloning.

An experiment with inline substitution.

Efficiently computing static single assignment form and the control dependence graph.

An effectiveness study of parallelizing compiler techniques.

*Interprocedural Symbolic Analysis.*  

*Flow Analysis of Computer Programs.*  

A unified approach to global program optimization.  

Interprocedural constant propagation: An empirical study.  
(To appear in *ACM Letters on Programming Languages and Systems*).

An empirical study of FORTRAN programs for parallelizing compilers.  

An empirical investigation of the effectiveness of and limitations of automatic parallelization.  

Constant propagation with conditional branches.  