Intermediate Compiler Analysis via Reference Chaining

Eric James Stoltz
B.S., Mathematics, Willamette University, 1976
M.S.T., Mathematics, Portland State University, 1982

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

January 1995

The dissertation "Intermediate Compiler Analysis via Reference Chaining" by Eric James Stoltz has been examined and approved by the following Examination Committee:

Michael Wolfe
Associate Professor
Thesis Research Advisor

David Maier
Professor

Steve Otto
Assistant Professor

Jonathon Walpole
Associate Professor

Paul Havlak
Research Associate
University of Maryland

Dedication

This dissertation is dedicated to my family, who has provided the necessary support and encouragement to allow me to quit my job five years ago and pursue my dream of a doctoral degree.

This exciting journey would not be possible without the unswerving encouragement of my wife, Luanne, and my stepchildren, Neil, Kara, and Krisanna. Luanne has helped me realize that almost anything is possible by applying oneself tenaciously, and never losing sight of the ultimate goal.

There is a special dedication for my parents, Jack and Cathy, who have never tried to direct my life, but have always been happy in anything I do, as long as I am happy doing it.

Thank you, all.

Acknowledgements

First, and foremost, I must thank my advisor Dr. Michael Wolfe, who has provided me with the opportunity to learn compiler analysis through research, experience (both his and mine), and writing papers, many papers. Michael, you have been such an extraordinary friend and mentor; I shall always be eternally grateful.

Second, I owe thanks to my thesis committee, who provided comments and suggestions which definitely improved the readability and clarity of my dissertation. I wish to particularly thank Dr. David Maier, without whose careful reading and many suggestions I would still be unsure of the precise distinction between restrictive and nonrestrictive clauses. Dr. Steve Otto deserves special acknowledgement for also providing guidance and direction in the use of his MetaMP distributed-memory programming environment, which I employed as a basis for my student research proficiency project during my second year of study at OGI.

Third, research is seldom performed in a vacuum, and the members of the Sparse research group have provided many hours of stimulating discussion (as well as many hours of not-so-stimulating discussion), all of which combines to produce a final product. Thanks are in order to Michael P. Gerlek, Priyadarshan Kolte, and Tito Autrey.

Finally, I must thank all the members of the class who started the same year as I at OGI. We have a special bond, and the friendships developed during the last four-plus years will remain with me always. Thanks and best of luck to: Dan Burnett, Brian Hansen, David Hansen, Nanda Kambhatla, Jeff Lewis, Jenny Orr.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Intermediate Compiler Analysis via Reference Chaining

Eric James Stoltz, Ph.D.
Oregon Graduate Institute of Science & Technology, 1995

Supervising Professor: Michael Wolfe

When performing data-flow analysis on a compiler's intermediate form of a program, sparse representations have proven their value by propagating information only to those points that affect or use such information. Static Single Assignment form (where each variable use has exactly one reaching definition) is a translation that linearizes reaching definitions in a sparse manner. This dissertation extends the concept of Static Single Assignment to include information flow other than just reaching definitions, such as reaching uses, upward-exposed references, and definition-to-definition links. Information is coalesced at confluence points via *merge operators*. The general process of providing pointers (links) between arbitrary pairs of definition and usage sites of a variable is called *reference chaining*.

A general reference chaining algorithm is developed and presented that allows parameters to be set that control the types of information to be propagated throughout the intermediate form. By providing this general algorithm, data-flow information (upward- or downward-exposed uses or definitions) is shown to be readily accessible in a compiler's intermediate representation. Many of the problems solved with reference chains utilize

a *demand-driven* technique, where classification of any node is often dependent upon its data-flow predecessors. Calls are made to classify the predecessors in a recursive manner.

The information provided by reference chains has led to the development of efficient intermediate analysis techniques, including demand-driven constant propagation, fast scalar dependence analysis, and live variable analysis, all within a unified sparse representation. Reference chaining has also been extended to parallel constructs, so that many of the same methods used to analyze sequential programs can be adapted to parallel programs.

Complete algorithms both of a general nature and specifically tailored to address the applications mentioned above are provided. Experiments have been performed on a wide variety of scientific benchmarks to determine the effectiveness of reference chaining, and comparative results are given where possible. The results of these experiments have shown that the demand-driven approach is both fast and effective, and that reference chaining is generally applicable and useful for many data-flow analysis problems.

Chapter 1

Introduction

With the advent of powerful, high performance workstations, coupled with the continuing pressure to develop parallel computing systems, powerful techniques for analyzing programs in their intermediate form is as important as ever. In order to perform aggressive and profitable (yet safe) program optimization, fast, efficient, and effective analysis techniques are required. Once a compiler has transformed a computer program into an intermediate form, analysis of its content can begin. At this point, many important questions can be posed:

- What type of intermediate form lends itself to a wide variety of analysis techniques?
- Is the intermediate form malleable to the extent of allowing adjustments that permit specialized analysis and transformation techniques to be applied?
- How can we represent information within the intermediate form that minimizes the cost of building data structures and propagating desired properties?

Clearly, these questions have no set, absolute answers. In some cases, language models may dictate the general structure of the intermediate form, such as lambda notation employed for functional languages [FH88, Chapter 8][KH89]. In other cases, specialization may require a particular form upon which to perform transformations, such as machine-dependent optimizations that operate on low-level forms during the end of the code generation phase. This dissertation, however, is concerned primarily with compiling scientific programs, which are mostly written in a procedural, imperative, programming style.

A general intermediate form can support a great many machine-independent analyses and optimizations, such as redundancy elimination, constant propagation, code floating, etc. Additionally, an intermediate form can be used for a variety of programming languages and can target multiple architectures [BCD+92]. Thus, we shall focus on an intermediate form that possesses properties of widespread acceptance, ease of portability, and flexibility for enabling analysis and optimization techniques.

1.1 Static Single Assignment Form

When performing data-flow analysis on a program's intermediate form, *dense* representations maintain all available information at each point in the program. *Sparse* representations, on the other hand, propagate information only to those points that affect or use such information. In recent years, use of sparse graph intermediate representations of programs, combined with methods to eliminate quadratic growth of data structures, have demonstrated that intermediate program analysis can be significantly simplified and streamlined [ABC+88, CCF91, FOW87, JP93, WCES94]. Since the late 1980's Static Single Assignment (SSA) [CFR+89] has become a popular intermediate representation with which to analyze programs. Utilizing SSA, many types of analyses and optimizations have already been developed, including global value numbering [RWZ88], partial redundancy elimination [BC94], constant propagation [WZ91], induction variable detection [Wol92b], code optimization [MJ92], and alias analysis [CG93], among others.

Partial answers as to why SSA has become popular, and why we began using SSA in our own research effort, are provided by these points:

- SSA is based upon the abstraction of basic blocks and flow graphs, the most common intermediate representation for program flow analysis [ASU86, Hec77, MJ81].
- SSA is well-understood by the compiler community in general, as evidenced from the many papers that refer to SSA structure.
- SSA has been demonstrated, via numerous independent implementations, to be a viable approach.

	(a)	(b)
S_1:	x = 0	x_0 = 0
S_2:	y = 0	y_0 = 0
S_3:	z = 0	z_0 = 0
S_4:	if (P) then	if (P) then
S_5:	y = y + 1	y_1 = y_0 + 1
S_6:	endif	endif
S_7:		$y_2 = \phi$ (y_0, y_1)
S_8:	x = y	$x_1 = y_2$
S_9:	z = 2 * y − 1	$z_1 = 2 * y_2 - 1$

Figure 1.1 Program in (a) standard form and (b) SSA form.

- Many methods have been developed that exploit the properties of SSA, as was noted above.

SSA is a solution to the quadratic growth of data structures and imprecision of data-flow solutions found with general reaching definitions. After a program has been converted into SSA form, it has two key properties (additional details are provided in Chapters 2 and 3):

1. Every use of a variable in the program has exactly one reaching definition, and

2. At confluence points in the control flow graph, pseudo-assignments are introduced called *ϕ-functions*. A ϕ-function for a variable merges the values of the variable from distinct incoming control flow paths (in which a definition occurs along at least one of these paths), and has one argument for each control flow predecessor. The ϕ-function is itself considered a new definition of the variable.

As an example, examine Figure 1.1. In (a) multiple definitions of y (at S_2 and S_5) reach its uses at S_8 and S_9, while in (b) exactly one definition reaches any use. Note that at S_7 in (b) a pseudo-definition of y is created that merges downward-exposed definitions. This coalescing of reaching definitions is one of the reasons that gives SSA form its appeal. One of the major tenets of this dissertation is that the concept of merging reaching

information can be extended – such as merging upward-exposed references (definitions or uses), an idea expounded in Chapter 7.

1.2 Thesis

In this work we show that SSA form is a specific instance of a more general method, which we call *reference chaining.* We demonstrate that reference chaining can improve upon traditional techniques, can be applied to solve new problems, and can be extended into the arena of parallel languages.

Our thesis is that *reference chaining for performing intermediate program analysis:*

- *can be efficiently implemented*
- *can lead to alternative solutions that can improve upon established methods*
- *can be used to develop solutions to problems previously neglected*
- *can extend its semantics into the area of parallel languages*

1.3 Contributions of This Work

The contributions of this work include:

Using FUD chains to analyze constants. Factored Use and Definition chains (FUD chains) are our implementation and extension of SSA form. We demonstrate the effectiveness of FUD chains coupled with demand-driven analysis for constant propagation, and evaluate this technique compared to the previously published methods that employ SSA form for constant propagation. By eliminating extra expression evaluation, our technique shows a 20% time performance improvement over existing methods that utilize SSA form.

Presenting new techniques for detecting scalar data dependence. We provide algorithms specifically tailored for detecting dependences of scalar variables. The methods employed are a straightforward application of reference chaining, and remove

much of the overhead associated with detecting dependences in arrays, since the need to perform subscript analysis is eliminated.

Providing a study of the types of scalar data dependences that occur in loops. We have performed experiments on common scientific benchmark codes, and have classified the basic types of scalar data dependences that can occur when the source and sink of the dependence are in the same loop.

Developing a general reference chaining algorithm. We have generalized the SSA algorithm, originally developed for reaching definitions, for many types of monotone data-flow problems. We show the applicability of our method to numerous problems other than reaching definitions.

Presenting a general method for a sparse solution to live variable analysis. Other techniques have been presented to solve the live variable problem on sparse graphs, but the solutions are based upon a separate sparse graph per variable. We provide a unified sparse graph that is an application of reference chaining based on upward-exposed definitions and uses. This method also aids in a fast dead code elimination algorithm, which dynamically updates which variables are live when eliminating useless code.

Extending reference chaining to parallel languages. The effectiveness of reference chaining on traditional, sequential code, and the current push to develop parallel languages has led us to extend the semantics into the parallel realm. Task parallelism (explicit parallel sections incorporating the *cobegin-coend* construct) is examined and shown to offer opportunities for reference chaining.

1.4 Experimental Methodology

To gather evidence and provide statistics with which to evaluate the methods and algorithms presented in this work, we have implemented all techniques in our research compiler and collected data on numerous scientific benchmarks.

Program	Lines	Routines	Statements	Description
PERFECT club				
adm	6106	97	3810	Hydrodynamics via finite differences
arc2d	3965	39	2137	2-D fluid flow solver
bdna	3978	43	3280	Finite difference solver for airfoils
dyfesm	7609	78	2435	Finite element structural analysis
flo52	1987	28	1795	Transonic airfoil flow solver
mdg	1239	16	836	Molecular water molecules
mg3d	2813	28	2280	Depth migration
ocean	4344	36	1729	Oceanic 2-D current flow
qcd	2277	35	1665	Monte Carlo lattice gauge theory
spec77	3886	65	2905	Weather simulation
spice	18522	128	13079	Circuit simulations
track	3785	32	1484	Projectile tracking
trfd	486	7	402	Electron integral transform
RiCEPS				
boast	8068	58	6355	Oil reservoir recovery
ccm	23557	145	10130	Atmospheric climatic model
hydro	13050	36	3824	2-D Lagrangian hydrodynamics
linpackd	798	11	393	Linear algebra routines
simple	1314	8	809	Hydrodynamics code
sphot	1145	7	650	Monte Carlo photon transport code
wanal1	2110	11	1182	Boundary control of wave equation
wave	7521	92	5321	Electromagnetic particle code
Mendez				
baro	984	7	553	2-D shallow water model
euler	1202	14	975	2-D shockwave model
mhd2d	928	14	511	Hydrodynamics code using FFT
shear	916	16	772	3-D turbulence code
vortex	711	20	515	Vortex sheet motion
Total	123301	1071	69827	

Table 1.1 Summary information on the programs in the scientific benchmark suites used

Nascent, our restructuring research compiler [WGS93], accepts Fortran source code and converts it into an intermediate form similar to the abstract model outlined in Chapter 2. Since all the techniques given in this dissertation have been implemented in Nascent, and our implementation provides valuable feedback on techniques and suggests useful coding methods, we shall often report on our efforts with a "Notes on Implementation" section within a chapter.

We have collected a significant number of scientific benchmarks written in Fortran to use for experiments in subsequent chapters. The programs come from the following three sets of benchmarks:

Perfect Club The Perfect Club benchmark suite [CKPK90] is a collection of programs designed to evaluate and test modern compilers and advanced architectures.

RiCEPS The Rice Compiler Evaluation Suite (RiCEPS) is a suite of numerical programs collected to test Fortran compilers, chosen to stress compiler analysis and transformations for advanced architectures. The RiCEPS suite is available via ftp from `cs.rice.edu`.

Mendez This collection of programs was compiled by Dr. Raoul Mendez of the Naval Postgraduate School in Monterey, California. Its original intent was to benchmark various American and Japanese vector computers. We obtained our copy from the National Institute of Standards and Technology (NIST).

Altogether, the 26 programs above contain in excess of 123,000 lines of code, and comprise over 1000 subroutines and 69,000 statements. Details on these codes are presented in Table 1.1.

1.5 Organization

This dissertation describes general reference chaining algorithms, together with their applications, extensions, and implementations. Chapter 2 presents the groundwork for understanding how data-flow problems can be characterized and solved within this framework. We also provide a detailed examination of related work in that chapter; we examine

other intermediate representations that are similar to SSA form, including an evaluation of their strengths and weaknesses. Chapter 3 presents the details of the most useful extension of SSA to date – Factored Use and Definition Chains (FUD chains). We have found many applications of FUD chains that admirably fit the demand-driven style of analysis described in that chapter.

Chapter 4 studies a particular analysis, constant propagation, within the FUD chain framework, and contrasts this method with both traditional constant propagation and a previously published constant propagation algorithm that also uses the SSA form. Chapter 5 presents the development of the general reference chaining algorithm, which can be used for forward or backward problems, chaining any combination of reference types. Chapter 6 presents a solution to the problem of detecting scalar data dependences, a problem that has not before been addressed explicitly. We present algorithms and data to support the contention that reference chaining can be used to construct new solution methods to problems.

Chapter 7 looks at utilizing reference chains to solve backward data-flow problems, notably live variable analysis and anticipatability of expressions. These methods provide solutions to problems that before have been considered insoluble on unified sparse data-flow representations. The algorithms presented are based upon the general reference chaining procedure of Chapter 5. We examine extending the analysis power of reference chaining into parallel languages in Chapter 8. Explicitly parallel sections are examined in detail, and we show how to provide reaching definitions within these sections in a sparse manner. By constructing this extension, we provide a coherent and sound method by which to reason sensibly about programs written using parallel syntax.

Finally, Chapter 9 looks at future work and extensions of the ideas and techniques presented in this dissertation. We draw conclusions in this chapter, and give an overall assessment of the usefulness and applicability of reference chaining as a method of choice for intermediate compiler analysis.

Chapter 2

Foundations

2.1 Internal Program Representation

Most compiler analysis techniques operate on an intermediate representation (IR). When the front end of the compiler processes a program, it must translate each statement into a form that eventually allows for later code generation. Before emitting code, however, transformations to increase speed and improve overall performance of the executable program may be desired. There are many possible intermediate forms, some fairly well known, such as postfix notation, three-address code representation, and abstract syntax trees [ASU86, FL88, TS85]. Some local optimizations (such as constant folding) can be performed within these forms, but in order to globally optimize a program, general program flow analysis must be employed. To accomplish this task, statements are grouped together in *basic blocks*, straight-line execution sequences with one entry and one exit point. The common way to express control flow through a program is to connect basic blocks together by constructing a *control flow graph* (CFG), sometimes just referred to as a flowgraph [Hec77]. Other methods will be examined later in this chapter.

2.1.1 Our Intermediate Representation

For the purposes of this thesis, our analysis of programs begins within an intermediate framework consisting of the control flow graph and a data-flow graph. Precisely, the *Control Flow Graph* (CFG) is a graph G = $\langle V,E,Entry,Exit\rangle$, where V is a set of nodes representing basic blocks in the program, and E is a set of edges representing sequential control flow in the program, where $v_1 \rightarrow v_2 \in E$ means that v_2 is a successor of v_1 in G

Tuple Operation Types		
Type	**Class**	**Description**
source	initialize	default definition at *Entry*
constant	arith	standard arithmetic constants
arithmetic	arith	standard arithmetic operations
test	boolean	used to compare quantities
proc	call	procedure call
proc_param	call arg	one for each argument of a procedure
merge	pseudo_ref	used to collect information at splits or joins
merge_arg	pseudo_ref	typically one for each predecessor or successor
fetch	use	standard load in IR
in_param	use	indicates a read-only for this procedure parameter
write	use	outputs values
store	def	standard store in IR
out_param	def	indicates a write-only for this procedure parameter
read	def	inputs values
formal	def	defined at *Entry* for formal arguments
global	def	defined at *Entry* for referenced globals
in_out_param	use & def	indicates procedure parameter is read/write

Table 2.1 Basic tuple types used in our IR

(and v_1 is a predecessor of v_2), with v_2 being the *head* of the edge and v_1 the *tail.* The set of edges entering and exiting a basic block are sometimes referred to as *inedges* and *outedges*, respectively. *Entry* and *Exit* are nodes representing the unique entry point into the program and the unique exit point from the program, respectively. *Branch* nodes have their outedges determined by a *predicate.* A *path* in G is a list of nodes in V, $\langle v_0, v_1, \ldots, v_n \rangle$, such that $v_j \rightarrow v_{j+1} \in E \; \forall j \in [0, \cdots, n-1]$. A path is *simple* if vertices $v_0, v_1, \ldots, v_n$ are distinct. We write $v_1 \stackrel{*}{\rightarrow} v_2$ to mean some, perhaps trivial, path from v_1 to v_2, while $v_1 \stackrel{+}{\rightarrow} v_2$ is a nontrivial path. We assume every node in G has a path from *Entry* and a path to *Exit.*

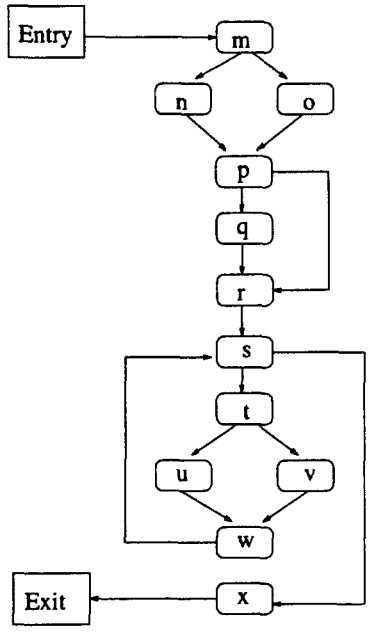
Each basic block contains a list of intermediate code *tuples*, which are linked together to form the *data-flow graph.* Tuples take the form ⟨ *type,left,right,link,symbol* ⟩, where *type* is the operation code and *left* and *right* are the two operands (both are not always required, *e.g.*, a unary minus). The tuple types available include those given in Table 2.1. The *left*, *right*, and *link* fields are pointers: they are all essentially use-def pointers but only some types use the *link* field. The *symbol* field is optional, and refers to the symbol-table name associated with the tuple, if applicable. The data-flow graph (which we will conveniently represent as an abstract syntax tree) comprises tuples together with their *left*, *right*, and *link* pointers.

A sample program and its CFG are given in Figure 2.1. Sometimes branch nodes will have each outgoing edge labeled according to the predicate value that selects the edge.

Description of Intermediate Code Tuples. Although when actually building a compiler there are a large variety of necessary tuple types (to cover special situations, such as `FORMAT` statements in Fortran or the wide variety of supported arithmetic operations), we will restrict ourselves to the set contained in Table 2.1 for two reasons. First, this allows us to describe clearly the manipulations and transformations that we wish to perform on the IR. Second, a restricted set is easily expanded in a way that loses no validity or precision; essentially it is easy to generalize. Most of the tuple types in Table 2.1 are self-explanatory. The *source*, *formal*, and *global* types are necessary

```
S1:    if (P) then          (m)
S2:       a = 2             (n)
S3:    else
S4:       a = 3             (o)
S5:    endif                (p)
S6:    if (Q) then          (p)
S7:       b = a + 1         (q)
S8:    endif                (r)
S9:    repeat while (R)     (s)
S10:      if (S) then       (t)
S11:         b = 3          (u)
S12:         c = c + 1      (u)
S13:      else
S14:         b = c          (v)
S15:      endif             (w)
S16:   end repeat
S17:   output( a,b,c )      (x)
```

(a)



(b)

Figure 2.1 A sample program (a), and its Control Flow Graph, (b)

to correctly provide the *link* field in SSA form (and, more generally, any *chaining* representation, which is covered later in this chapter) where initialization of a variable is required. In particular, *source* represents the initial definition of all variables at *Entry*, which will be required by the conversion into SSA form. The *formal* and *global* types are used as initial definitions when a variable is passed in as a subroutine argument or is a referenced global variable, respectively. The *param* tuples refer to arguments of a procedure or function. As more information about those arguments becomes available, the default of *in_out_param* may be changed (for example, as a result of interprocedural mod/ref analysis). Finally, the *merge* tuples are used to represent pseudo references, such as ϕ-functions (*merge*) and ϕ-arguments (*merge_args*).

The distinction between definitions (a statement that modifies, or potentially modifies, a variable, and typically includes the variables occurring on the left-hand side (LHS) of an assignment) and uses (a statement that fetches the value of a variable, typically found as variables occurring on the right-hand side (RHS) of an assignment) is usually clear. However, sometimes a variable occurrence is not so well-defined, such as a simple

call-by-reference procedure parameter:

```
call foo(x)
```

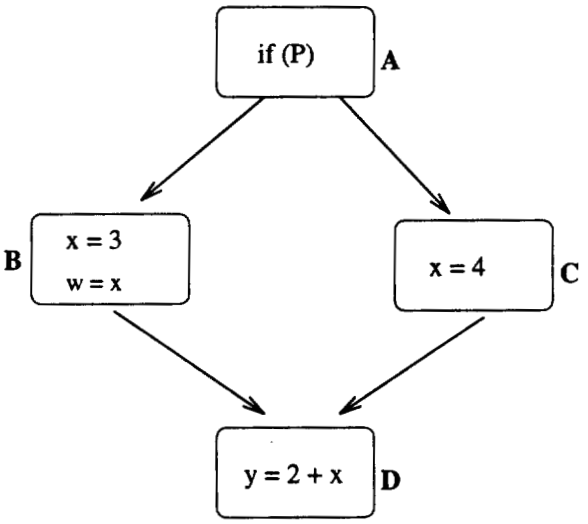
where no other analysis has been performed on the way x interacts with the program as a whole. In this case, we must assume that x *may* be modified. But x is also (potentially) fetched when foo is called. Since x may be defined, but not certainly, we call this an instance of a *nonkilling* definition. Nonkilling definitions can occur in several other contexts, such as indexed variables (arrays), and we shall look at how they are fully represented in Chapter 3.

Tuple Order Within Basic Blocks. The compiler front-end parses the program, placing tuples within basic blocks. However, the order of tuples in each basic block is not random. Some lexical order must be maintained. For example, in our IR the statement x = x + 1 will be broken down into four tuples: a *store* (x), an *arithmetic* (+), a *fetch* (x) , and a *constant* (1). Since the right hand side of the expression will fetch the value of x before the left-hand side stores it, we rely on the lexical ordering of the tuples to maintain this property. Thus, the tuples representing the right-hand side of the expression (addition, fetch, and constant) will occur (lexically) before the store within the basic block. This ordering is handled by standard parsing techniques, and we will refer to this important property again when more complicated cases arise, such as the order of representing procedure parameters when identifying nonkilling definitions.

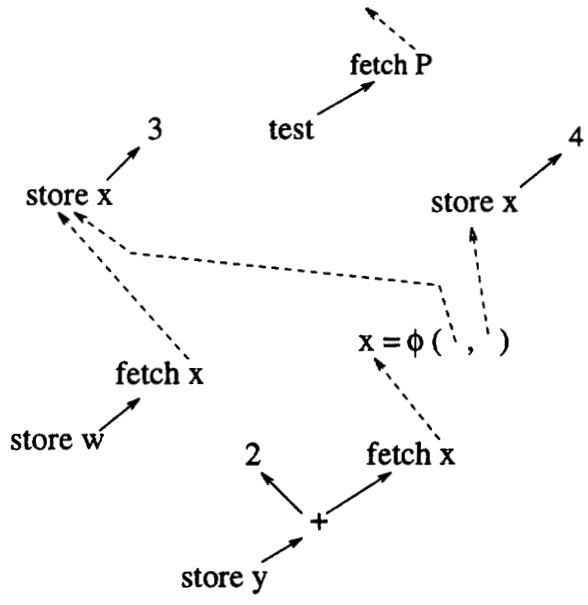
Example. We show in Figure 2.2 how this small program fragment:

```
S1: if (P) then
S2:     x = 3
S3:     w = x
S4: else
S5:     x = 4
S6: endif
S7: y = 2 + x
```

is transformed into its CFG basic block components, (a), and its data–flow graph representation, (b). The solid lines in (b) represent *left* and *right* pointers while the dashed



(a)



(b)

Figure 2.2 Program fragment shown at (a) basic block level (b) data-flow graph level

lines represent *links*. Note that a *fetch* operation has only a single *link* field, so in basic block D, to maintain the single–assignment property, the *fetch* points to the pseudo–assignment for x at the confluence point, which is a ϕ-function.

In this example, the *fetch*, *store*, and *ϕ-merge* utilize the *symbol* attribute of tuples.

2.1.2 Basic Control Flow Analysis

After constructing the CFG, several initial types of analyses are necessary that are relied upon heavily for all subsequent phases of the compiler. First, basic blocks are related by the *dominance relation*. Then, using this information, loops within the CFG are identified. As we shall see, any cycle within the CFG will not necessarily constitute a loop, although this will be the case in structured code.

Dominance. Within the CFG, node X *dominates* node Y (X *dom* Y) if all paths from *Entry* to Y must pass through X. X *strictly dominates* Y if X dominates Y and $X \neq Y$. X is the *immediate dominator* of Y if X strictly dominates Y and all other nodes which strictly dominate Y also dominate X. If X does not dominate Y, we denote this relationship as X $\overline{dom}$ Y. We show the dominator relation (which is reflexive, antisymmetric, and transitive, thus imposing a partial order upon the nodes of a CFG) for Figure 2.1 in Table 2.2. Here, DOM(Node) is the set of nodes dominated by Node (where m...x represents all nodes, alphabetically, from m to x), while DOM^{-1}(Node) is the set of nodes that dominate Node. The *immediate dominator* (idom) relationship forms a tree of all the nodes within the CFG. Dominance can be represented in tree format (referred to as the *dom tree*) since the dominators of any node form a linear ordering [Hec77], which implies that a node's immediate dominator is unique, if it exists. We show the dom tree for Figure 2.1 in Figure 2.3.

Computing Dominance. We present here an algorithm for finding the dominators of a node within a CFG. This method relies on the property that A *dom* B if A dominates all predecessors of B. Essentially, it iterates the following data-flow equation around the

Dominator Relationship for sample CFG			
Node	**idom(Node)**	**DOM(Node)**	**DOM^{-1}(Node)**
Entry	$\emptyset$	{*Entry*, m...x, *Exit*}	{*Entry*}
m	*Entry*	{m...x,*Exit*}	{*Entry*,m}
n	m	{n}	{*Entry*,m,n}
o	m	{o}	{*Entry*,m,o}
p	m	{p...x,*Exit*}	{*Entry*,m,p}
q	p	{q}	{*Entry*,m,p,q}
r	p	{r...x,*Exit*}	{*Entry*,m,p,r}
s	r	{s...x,*Exit*}	{*Entry*,m,p,r,s}
t	s	{t,u,v,w}	{*Entry*,m,p,r,s,t}
u	t	{u}	{*Entry*,m,p,r,s,t,u}
v	t	{v}	{*Entry*,m,p,r,s,t,v}
w	t	{w}	{*Entry*,m,p,r,s,t,w}
x	s	{x,*Exit*}	{*Entry*,m,p,r,s,x}
Exit	x	{*Exit*}	{*Entry*,m,p,r,s,x,*Exit*}

Table 2.2 The dominator relation from Figure 2.1

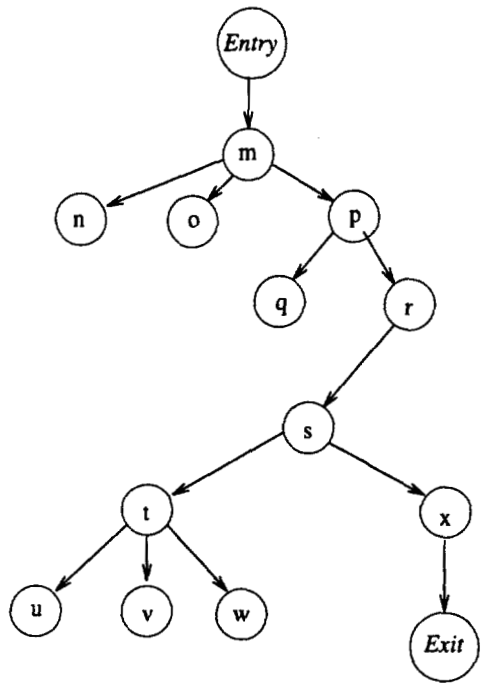


Figure 2.3 Dominator tree for nodes in the CFG from Figure 2.1

Given: CFG
Do: Initialize with lines 1 - 4.
Result: The set of dominators for each node in the CFG

```
1: Dom( Entry ) = { Entry }
2: for all N ∈ V - { Entry } do
3:         Dom(N) = V
4: endfor

5: while changes to any Dom(N) occur do
6:         for all N ∈ V - { Entry } do
7:             Dom(N) = {N} ⋃ ⋂_{p∈pred(N)} Dom(p)
8:         endfor
9: endwhile
```

Algorithm 2.1 Finding dominators of nodes in the CFG

CFG until a fixed point is reached (where *pred* represents all predecessors of a node):

$$Dom(N) = \{N\} \bigcup \bigcap_{p \in pred(N)} Dom(p)$$

Algorithm 2.1 presents the details of this method. The algorithm's running time is $O(V^2E)$ when performed as bit vector operations, although Lengauer and Tarjan note [LT79] that this asymptotic time bound is pessimistic since the constant term associated with these operations is small and the number of passes is usually no more than 2 in order to reach a fixed point.

Algorithm 2.1 does not provide the immediate dominator for each node, important for building the dominator tree and used in numerous other analysis techniques. However, identification of each node's immediate dominator can be accomplished by keeping track of the depth-first numbering (or reverse postordering number) of each node in a spanning tree of the CFG – the idom will be the element in the dominator set with highest depth-first number.

Dominance Frontier. An property related to dominance is the dominance frontier of a node, important for placing merge operators such as ϕ-functions. Z is in the *dominance frontier* of Y if Y does not strictly dominate Z, but Y does dominate some predecessor of Z. We show the dominance frontier for all nodes in Figure 2.1(b) in Table 2.3.

We compute the dominance frontier of all nodes by adapting a method developed by Cytron, Ferrante, and Sarkar [CFS90]. We demonstrate the method by referring to the dominator tree in Figure 2.3. Informally, this algorithm works due to several observations concerning a CFG and its associated dominator tree. First, only a confluence node in the CFG can be in the dominance frontier of any other node. If N has only one predecessor, then any node that dominates its predecessor must dominate N. By definition of dominance frontier, a single predecessor excludes N from being in the dominance frontier of any node.

Second, assume Z is a confluence node. Then, *idom(Z)* will dominate all predecessors P of Z. This relationship is true because all paths from *Entry* to Z must pass through *idom(Z)*, and each P has a direct edge to Z. Thus, all paths from *Entry* to P must pass through *idom(Z)*.

Third, if node Z is in the dominance frontier of node Y, then Y is not an ancestor of Z in the dominator tree. Thus, starting at each predecessor of Z and moving up the dominator tree until reaching *idom(Z)* we will find nodes that have Z in their dominance frontier. Algorithm 2.2 gives the details for computing the set of nodes in the dominator frontier for each node in a CFG.

Postdominance. We may also need the concepts of *postdominator* and *postdominance frontier*. Node X *postdominates* node Y (X *pdom* Y) if all paths from Y to *Exit* must pass through X. *Strict postdominance*, *immediate postdominator* and *postdominance frontier* are analogously defined as for dominance. In fact, postdominance properties are exactly the corresponding dominance properties on the Reverse Control Flow Graph (RCFG), the CFG with all edges turned around and *Entry* and *Exit* interchanged.

Given: CFG and DomTree
Do: Initialize dominance frontier of each node as empty
Result: List of dominance frontier nodes for each basic block

```
1:  for all nodes N ≠ Entry do
2:      I = idom(N)
3:      for each predecessor P of N in CFG do
4:          M = P
5:          while M ≠ I do
6:              DF(M) = DF(M) ∪ {N}
7:              M = idom(M)
8:          endwhile
9:      endfor
10: endfor
```

Algorithm 2.2 Computing dominance frontier sets

Dominance Frontier & Control Dependence Sets for sample CFG			
Node	**DF(Node)**	**CD_Pred(Node)**	**CD_Succ(Node)**
Entry	∅	∅	∅
m	∅	∅	{n,o}
n	{p}	{m}	∅
o	{p}	{m}	∅
p	∅	∅	{q}
q	{r}	{p}	∅
r	∅	∅	∅
s	{s}	{s}	{s,t,w}
t	{s}	{s}	{u,v}
u	{w}	{t}	∅
v	{w}	{t}	∅
w	{s}	{s}	∅
x	∅	∅	∅
Exit	∅	∅	∅

Table 2.3 The dominance frontier and control dependence for nodes in the CFG from Figure 2.1

Control Dependence. A useful concept for representing possible execution paths within the CFG is that of *control dependence.* Intuitively, node X is control dependent upon node Y if some outedge of Y guarantees that X will be reached, while another outedge from Y may avoid X. Clearly, if X is control dependent upon Y, then Y must be a branch node. Formally, CFG node X is control dependent upon CFG node Y if X postdominates every node along some path from a successor of Y to X, and X does not strictly postdominate Y. It has been demonstrated that control dependence in a CFG is equivalent to the dominance frontier of the RCFG [CFR+91], which also implies that control dependence for node N is equal to the postdominance frontier of N.

The control dependence predecessors of node N, CD_Pred(N), are those nodes that N is control dependent upon (and that above was simply referred to as control dependence on N), while the control dependence successors of N, CD_Succ(N), are those nodes control dependent upon N. We show the control dependence predecessor and successor sets for our sample CFG (Figure 2.1) in Table 2.3.

We note that a node may be control dependent upon itself, such as node s. One outedge of s (the edge s→t) guarantees that s will be visited again, while another outedge (the edge s→x) definitely will not visit s before reaching *Exit.* It is also important to note that x is not control dependent upon s, although it appears that if the edge s→t was always taken when leaving s, x would never execute. However, we assume for general program flow analysis that all programs terminate, thus an infinite loop is not a consideration.

Loop Identification. For scientific, high performance codes, loops offer fertile opportunities for optimization. This opportunity is because much of the computationally intensive code is within these loops, which often iterate over the kernel of the routine, especially when there exist nested loops. Thus, correct identification of loops is critical in a high performance compiler.

We classify loops based upon the definition of a *natural loop* [ASU86], which identifies an edge in the CFG (the *backedge* of the loop) such that the head dominates the tail. We visit each basic block, *bb*, within the CFG, checking to see if *bb* dominates any

predecessor, p. If so, a natural loop is found, and we can label the *header* of the loop as *bb* and start adding p and all the predecessors of p as members of the loop until we reach *bb*. For example, in Figure 2.1, s dominates predecessor w, so a natural loop is found with header s, and loop members w,u,v, and t.

We mark each node of the CFG as belonging to the innermost loop of which it is a member. By traversing the dominator tree bottom-up, we are able to find inner loops first. Identifying loops inner to outer is convenient when adding members of a loop, since if a basic block is already a member of a loop it must be in an inner loop, and we can immediately go to its header node and continue the process of adding nodes to the outer loop.

The technique used for finding natural loops is provided by Algorithm 2.3. The recursive calls in lines 2 - 4 insure that the dominator tree is walked from the bottom up. Loops not identified by this algorithm are due to *irreducible* flow graphs, discussed in §4.4.3.

2.1.3 Control Flow Adjustments

There are also several modifications that we routinely make to the standard CFG. These modifications enable analysis to proceed more easily, with a more uniform applicability of techniques.

An Additional Edge Within the CFG. For each control flow graph we insert an extra edge from *Entry* to *Exit*, called the *slice* edge.[†] This edge is important for several reasons:

1. To have the *control dependence* tree rooted at *Entry*, this edge is added for technical reasons, as described by Cytron *et al.* [CFR+89]. It also allows the control dependence graphs of *while* and *repeat* loops to be distinguishable.

2. The *Exit* node will be in the *iterated dominance frontier* of a CFG for variable v if and only if there is a definition of v within the CFG (which is explained further and

[†]This extra edge is also called the *technical* edge.

Given: CFG and its DomTree
Do: **forall** basic blocks bb **do** bb.loop = NULL **enddo**
call Find_Loops(*Entry*)
Result:All natural loops with node membership

```
1:  Find_Loops (basic block bb )
2:        for all  children c of bb in DomTree do
3:             Find_Loops( c )
4:        endfor
5:        for all predecessors p of bb in CFG do
6:             if bb dom p then
7:                  mark bb as header of a natural loop L
8:                  bb.loop = L
9:                  Loop_WorkList = NULL
10:                 add p to Loop_WorkList
11:                 while  Loop_WorkList ≠ NULL do
12:                      remove n from Loop_WorkList
13:                      if n.loop = NULL do
14:                           n.loop = L
15:                           for all predecessors q of n do
16:                                add q to Loop_WorkList
17:                           endfor
18:                      else if n.loop ≠ L do /* n.loop is nested in L */
19:                           h = header of n.loop
20:                           for all predecessors r of h do
21:                                if r.loop ≠ n.loop then
22:                                     add r to Loop_WorkList
23:                                endif
24:                           endfor
25:                      endif
26:                 endwhile
27:            endif
28:       enddo
29: end Find_Loops
```

Algorithm 2.3 Finding natural loops in the CFG

is proved in Chapter 3). This property of CFGs is extremely useful for propagating definitions outside of a local CFG in the case of nested control flow graphs, which may occur when translating parallel syntax into intermediate form (see Chapter 8).

3. Due to item 2, we can find the last definition of a variable within a CFG by following the correct *link* from the ϕ-function at the *Exit* node. This property has proven invaluable for work on interprocedural analysis [Aut94], specifically for interprocedural mod/ref analysis.

Loop Preheaders. It is convenient for analysis purposes that each loop header has exactly two inedges. We insure this property by adding two new nodes to the CFG. The first is a loop *preheader*, a node outside the loop, which collects all inedges to the header from nodes not within the loop [ASU86]. We would like to simply adjust the information gathered as a result of analysis performed in §2.1.2 incrementally, such as dominator computation, without having to repeat those procedures a second time. This procedure is accomplished by creating a new node for each natural loop and making the following adjustments to the CFG and immediate dominator tree:

- label the new node as the preheader of the loop.
- all nodes not inside the loop that have the header as a successor now instead have the preheader as their successor.
- the successor of the preheader is the header.
- idom(preheader) = idom(header)
- idom(header) = preheader

It is trivial to prove the properties above regarding the idom of the header and preheader.

Loop Postbodies. The second node we need to add to each loop is the *postbody*. The postbody collects edges from any node within the loop that has the header as a successor

Given: A loop in the CFG
Do: Link *postbody* node to its successors and predecessors
Result: The unique immediate dominator of *postbody*

```
1: pb_idom = any predecessor of postbody
2: for all predecessors P of postbody
3:         while pb_idom dom P
4:             pb_idom = idom( pb_idom )
5:         endwhile
6: endfor
```

Algorithm 2.4 Finding the immediate dominator of a loop's postbody node

(the postbody is not strictly necessary if only one node within the loop has the header as a successor). We create a new node within the loop and make the following adjustments:

- all nodes inside the loop which have the header as a successor now have the postbody as a successor.
- the successor of the postbody is the header.
- use Algorithm 2.4 to find pb_idom, the immediate dominator of the postbody.
- idom(postbody) = pb_idom

Theorem 2.1 *Algorithm 2.4 correctly identifies the unique immediate dominator of the postbody node.*

Proof:

Following the chain of immediate dominators from any predecessor of node N in a CFG, one must eventually reach idom(N). If not, then there would be a path from *Entry* to N avoiding idom(N), an obvious contradiction. Thus, starting with any predecessor of the postbody, successive application of line 4 will eventually reach idom(postbody). Algorithm 2.4 identifies idom(postbody) by relying on the property that X *dom* Y if X dominates all the predecessors of Y. ∎

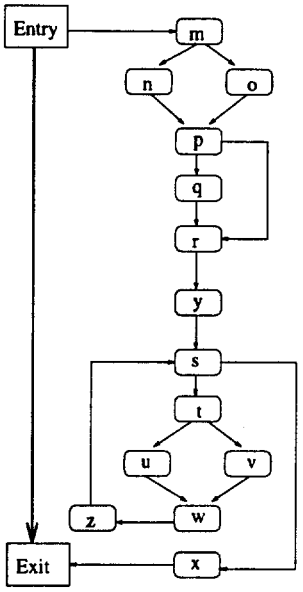


Figure 2.4 CFG from Figure 2.1 after modifications: y is the loop preheader and z is the loop postbody. There is now also a slice edge from *Entry* to *Exit*.

CFG Effects. Making the preceding modifications to the CFG will affect analysis such as the dominator relationship (now only *Entry* dominates *Exit*), and the set of basic blocks contained within a loop (the postbody is a new member of each loop). Although dominator information can be incrementally updated when adding preheader and postbody nodes, simple updates are not sufficient when adding the slice edge, since this additional edge fundamentally changes the possible paths through the CFG. Thus, it is customary to add the slice edge before any analysis begins. Loop preheaders and postbodies obviously cannot be inserted before the loop analysis phase commences.

The CFG from Figure 2.1 looks like Figure 2.4 after the modifications above have been made. The inserted preheader node, y, was not strictly necessary in this case, since loop header node s only had a single predecessor to start with (similarly with postbody node z). However, the preheader provides a convenient location to float code out of a loop in all cases, and adding these two nodes assures us that the loop header will always have exactly two predecessors, a property which we shall exploit frequently.

2.1.4 Notes on Implementation

Computing Dominators. The method presented earlier for computing dominators is nonoptimal, albeit easy to understand. In production-quality compilers, however, one desires the most efficient method available. Thus, we actually compute the immediate dominators of all nodes using an algorithm by Lengauer and Tarjan [LT79]. The asymptotic complexity of this algorithm is $O(E\alpha(E,V))$ (where α is the functional inverse of Ackermann's function [HS78]), although it is close to linear in practice. Another advantage of this algorithm is that it provides the immediate dominator (*idom*) for each node, thus allowing a compact representation of the dominator tree to be stored (each node just stores its immediate dominator). Although a linear time algorithm for computing dominators has been published [Har85], to our knowledge it has not been implemented.

Computing Dominance Frontier. Algorithm 2.2 provides the details of building dominance frontier sets for each node in the CFG. When implementing this algorithm, one needs to decide how to maintain the set of dominance frontier nodes. Since the size of the dominance frontier set for any CFG node from a structured program containing only `if-then-else` and `while-do` constructs is no larger than two [CFR+91, Theorem 4] (although unstructured code can result in quadratic space size), we store the dominance frontier for each node as a list, since bit vectors in this case would be space inefficient.

Line 6 from Algorithm 2.2 requires a union operation. This operation is important, since just appending N to the dominance frontier list of M could result in duplicate entries. To see why, examine Figure 2.5. When considering node E, idom(E) = A, and if we visit the predecessors of E in the order C followed by D, we first find that E is in the dominance frontier of C and B. When D is considered, we find that E is in the dominance frontier of D and B. Since we do not want to add E to the dominance frontier list of B twice, we use a constant time test for membership. Although a linear search would usually not be slow, "ladder graphs" create dominance frontier sets that are quadratic in size [CF93, SG93]. Since each node in the CFG is analyzed and added to the dominance frontier in turn, each node has a pointer, *point*, which is set to the last node added to its dominance frontier set. Thus line 6, for implementation purposes,

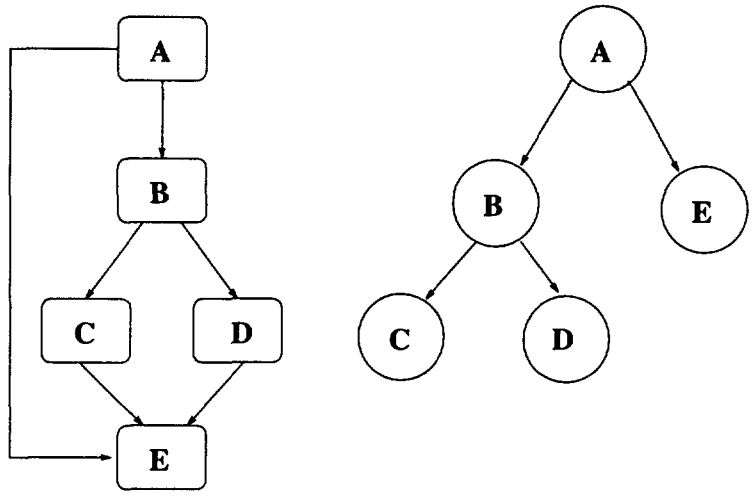


Figure 2.5 Simple CFG and its dominator tree

gets expanded to:

```
6a:  if M.point ≠ N then
6b:          append DF_list(M) with N
6c:          M.point = N
6d:  endif
```

In this way, we never have to reset *point*, and in constant time we can check to see if N has already been added to the dominance frontier list of any node.

2.2 Other Control Representations

In this section we review alternate intermediate representations. All the alternatives start with a CFG, so while the CFG may have limitations, it clearly is recognized as a fundamental abstraction for intermediate program analysis.

2.2.1 Program Dependence Graphs

The Program Dependence Graph (PDG) [FOW87] is a unifying framework in which the nodes are individual statements, and the edges reflect control and data dependence. This representation grew out of earlier work [FO83] on program regions represented by control and data dependence. One underlying principle is that the standard CFG imposes a fixed sequencing of operations that may not be necessary. In a CFG, basic blocks contain straight-line execution statements, based upon the structure given by the programmer. However, if no dependence exists between statements, they may be executed in any order. Thus, by making a graph with one statement (or predicate expression) per node, and superimposing all constraints in the form of edges, more optimizations may be discovered. Additionally, this form is intended to allow greater extraction of parallelism, since control dependence equivalence is one of the key factors in permitting code to be executed in parallel.

An interesting feature of the CFG that influences control dependence is the idea of a *hammock* [Kas75], which essentially identifies single-entry, single-exit (SESE) regions in a directed graph. One optimization used in the PDG representation to reduce the size of the graph is to create *region* nodes, where multiple statements are in the same region if they have control dependent equivalence. These regions contain a set of hammocks and some number of exit edges. The strength of this approach is shown by the demonstration that if two programs have PDGs that are isomorphic, then they are *strongly equivalent* (for any initial state, either both programs diverge or both halt with the same final state) [HPR88]. The PDG has also been used as an intermediate representation for vectorization [BB89], instruction scheduling [BR91], and as the basis for a new register allocation routine [NP94].

Although the PDG has received much attention, proving its universal applicability is not easy [CF89, Sel89]. Essentially, it is quite a difficult task to correctly determine the exact meaning, in all instances, to assign to a program once it is represented just as a set of control and data dependences. Our feeling is that it is a useful construct, but must be built from the CFG in the first place, thus incurring some extra cost for its

construction. However, the PDG can be built in worst case time of $O(V^2)$, where V is the number of nodes in the CFG, so for some applications it may be the intermediate form of choice.

2.2.2 Dependence Flow Graphs

The Dependence Flow Graph (DFG) [JP93] is intended to be a generalization of SSA form. The DFG combines control information with def-use SSA chains by defining *control regions* that consist of SESE areas of the CFG (similar to the hammocks referred to above, except that the original definition of hammock [Kas75] allowed exit nodes of the hammock to have predecessors not in the hammock). The DFG form contains both branch and confluence edges between variables, bypassing SESE regions that contain no assignment to the variable.

A number of analysis techniques (both forward and backward) are possible using DFGs, which have recently been defined in terms of Quick Propagation Graphs [Joh94], a sparse representation of the CFG. Since the construction of a DFG is $O(E)$ per variable V, the overall complexity is $O(VE)$ time. It appears that the constant on this time bound is fairly high, since for each variable a *base-level* DFG is built by adding V dependence edges in parallel with each control flow edge. Then, edges are removed when they are determined to be unnecessary, such as when bypassing SESE regions. Quite a nice result was demonstrated in this work: that of determining control dependence regions in linear time [JPP93, JPP94]. Although this method does not determine on what nodes a region depends, it identifies which nodes have equivalent control dependence, an important component for extracting parallelism.

2.2.3 The Program Dependence Web

One problem with SSA form is that the inserted pseudo-assignment ϕ-function is not interpretable. That is, when analyzing a ϕ-function within a confluence node, no information is available as to which path was taken to reach this node. In may be that, at compile time, no information is known about the predicate that determined which branch was taken earlier. However, if the predicate is determinable, that information is

not captured by ϕ-functions.

The Program Dependence Web (PDW) [BMO90] introduces a more sophisticated method of implementing SSA, which they term Gated Single Assignment (GSA), since values that flow along control paths are *gated* by the status of predicates at branch points. Essentially, GSA separates ϕ-functions into two classes: those at the headers of loops, called μ-functions, and those at control confluence nodes, called γ-functions. In addition, there is an additional operator (the η-function) added at exit nodes of loops for each variable. This function controls values of variables computed within loops.

By inserting μ-, γ-, and η-functions, a much more interpretable intermediate form is available for analysis. In particular, using a demand-driven style of analysis, more precise information is computed for such problems as constant propagation. We shall look at how to use GSA form for the constant propagation problem in Chapter 4.

The initial work on PDW and GSA [BMO90, OBM89] required a program to be in PDG form before the algorithms could translate it into GSA form. It was also targeted at pure data-flow architectures, such as the Monsoon [ADNP88] project at MIT. The PDW attempted to bridge control-, data-, and demand-driven models within a single framework. The result was a difficult representation. We will use a simplified GSA form that is based upon the CFG and control dependence only; in Chapter 4 a method that avoids the transformation into a PDG is given. The first algorithms to directly translate a CFG into GSA form were provided by Havlak [Hav93].

2.2.4 Value Dependence Graphs

The Value Dependence Graph (VDG) [WCES94] is a functional representation that expresses the computation of a procedure solely as value flow. Selection among control paths is performed in a manner analogous to that of GSA, but looping is represented by recursive function calls. A CFG is converted to a VDG by way of a Store Dependence Graph (SDG) using SESE analysis as described above for DFGs. Optimizations that can be performed independent of the final order of execution are then performed on the VDG. For code generation, the VDG is converted to a demand-based PDG and back into the CFG.

Weise *et al.* [WCES94] point out that the VDG is very similar to a simplified GSA, with the major difference being that VDGs represent looping through a procedure call and return, while a GSA uses special functions (μ and η). They also concede that the presence of γ-functions allows conditional analysis, such as constant propagation, to performed within a GSA. Chapter 4 details just such an approach within a GSA structure.

2.3 Reference Chaining

At this point, we are able to begin analyzing the content of the intermediate form. SSA, which has already been described, is an intermediate program representation that presents a solution to the quadratic nature of the *reaching definitions problem.*[†] The reaching definitions problem can be stated as follows: at each point in the program, what is the set of all definitions that can reach this point?

Many other kinds of information flow through programs, such as reaching uses, live variables, links between potential aliases, links from definitions to definitions, etc. We would like to extend the technique employed to solve the reaching definitions problems via SSA by generalizing its method. In order to accomplish this goal, we need to define explicitly how to link information between program points and how to merge information at points where links which represent information from two or more different paths come together.

We next look more closely at some of the important concepts alluded to in the last several paragraphs regarding how information flows through a CFG.

2.3.1 Data-Flow Analysis

Data-flow analysis is a general method of collecting information about a program, such as those problems described above [ASU86, MJ81]. Some problems, such as reaching definitions or computing dominance, are *forward* data-flow problems, since the flow of

[†]In SSA form, each variable use has a single reaching definition, as opposed to the many reaching definitions possible with traditional reaching definition analysis. A detailed explanation is provided in §3.1.1.

information is in the same direction as control flow. Live variable analysis is an example of a *backward* data-flow problem, since the information at the end of each basic block is dependent upon the information at the entry of control flow successors. Thus, with backward data-flow problems information flows in the direction opposite that of control flow.

A set $\mathcal{F}$ of *transfer functions*, with an $f_N \in \mathcal{F}$ at each node N in a flow graph, summarizes the data-flow effect about the desired information at that node. $\mathcal{F}$ contains an identity function i, which is used when a node has no affect on the data-flow information that enters the node; i.e., the information that leaves the node is identical to the information that enters that node. The elements of $\mathcal{F}$ are closed under function composition. Often, the only "interesting" nodes that affect the solution to a data-flow problem are those which have a nonidentity transfer function [CCF91].

For each $f \in \mathcal{F}$, f takes the general form of $f(X) = A \cup (X - B)$. For each node N in the CFG, X represents *in(N)*, the information which reaches the beginning of N, while $f(X)$ represents *out(N)*, the information leaving N after the effect of f has been computed. A and B represent how information within N is combined with X. For example, in solving the reaching definitions problem, A represents *gen(N)*, the set of definitions defined within N, while B is the set *kill(N)*, those definitions killed within N. For the problem of computing dominators in a CFG (as discussed in §2.1.2), A is N, while B is empty.

A typical method of extracting information from the CFG is to iterate a set of data-flow equations until the information available at each point (typically the beginning and end of each control flow node) has converged, such as the algorithm we gave to find dominators. When information comes together at confluence nodes, a *meet* operator combines the information from all predecessors via operations on a semi-lattice. A semi-lattice $\mathcal{L}$ is a pair $(E,\sqcap)$, where E is a nonempty set of elements and $\sqcap$ is a binary operation (the *meet*) on E, which is idempotent, commutative, and associative. E contains a distinguished zero element, $\perp$, usually called "bottom". Although "top" ($\top$), a distinguished one element, is not theoretically required for a semi-lattice, it often is useful for defining data-flow problems. E is assumed to be finite, and the elements of E are related to each

other through a reflexive partial order, $\preceq$. Semi-lattice $\mathcal{L} = (E,\sqcap)$ (which includes $\top$) has the following properties for $a, b \in E$ [Mar89]:

$$\begin{aligned} a \preceq b &\iff a \sqcap b = a \\ a \sqcap a &= a \\ a \sqcap b &\preceq a \\ a \sqcap \top &= a \\ a \sqcap \bot &= \bot \end{aligned}$$

In general, a data-flow *framework* for forward problems consists of W, the set of values to be propagated, $\mathcal{F}$, the set of transfer functions, and a binary meet, the confluence operator. After the following initializations have been made (where, depending upon the problem, *out(Entry)* may be initialized to either $\bot$ or $\top$):

for all N $\in$ {CFG - *Entry*} **do**

$in(N) = \top$

$out(N) = f_N(\top)$

endfor

the following set of equations are iterated over the CFG until no changes occur:

$$\begin{aligned} in(N) &= \prod_{P \in pred(N)} out(P) \\ out(N) &= f_N(in(N)) \end{aligned} \tag{2.1}$$

For the reaching definitions problem, W is the powerset of the set of nodes of the CFG, and the meet operator is set union. (The partial order relationship is that of superset, so for sets $\mathtt{T}$ and $\mathtt{Q}$, $\mathtt{T} \preceq \mathtt{Q}$ if $\mathtt{T} \supseteq \mathtt{Q}$.) These data-flow equations are then iterated until convergence:

$$\begin{aligned} in(N) &= \bigcup_{P \in pred(N)} out(P) \\ out(N) &= gen(N) \bigcup (in(N) - kill(N)) \end{aligned}$$

Most data-flow problems are solved using a *monotone data-flow framework*. A framework is monotonic if:

$$(\forall f \in \mathcal{F})(\forall x, y \in \mathcal{L})[x \preceq y \Longrightarrow f(x) \preceq f(y)]$$

It has been observed [KU77] that an equivalent definition of monotonicity is:

$$(\forall f \in \mathcal{F})(\forall x, y \in \mathcal{L})[f(x \sqcap y) \preceq f(x) \sqcap f(y)]$$

Iterating over a monotone data-flow problem cast on a flow graph will always converge to a *maximum fixed point* (*mfp*) [Hec77], which is the solution to equations 2.1. The *mfp* is a conservative approximation to the "true" solution, and we will return to the precision that it provides in Chapter 4.

2.3.2 Reaching Definitions in a CFG

In terms of reaching definitions, a definition of v at node A *reaches* node B if there exists a path in the CFG from A to B in which no other definition of v occurs. For reaching definitions, the interesting nodes are those that define a particular variable. All other nodes will have a transfer function that is the identity: the information leaving the node is the same as the information entering the node. The meet operator for reaching definitions is set union. Definitions along a straight-line path in a CFG can be *killed* by another definition for the same variable. If nodes X and Y both lie along a straight-line path with X preceding Y, and both define variable v, then the definition at Y kills any definitions at X. Note that (degeneratively, in this case) every path to Y passes through X. In more complicated cases, *nonkilling* definitions of a variables can exist. This occurs when a new definition *may* occur at a particular point, but we are not certain of it. An example would be a procedure call, where a procedure parameter is passed by reference. In this case, both the definition arriving at the node which has the procedure call and the nonkilling definition of the procedure parameter must be passed as reaching definitions to the next use of the variable. Nonkilling definitions can also be used for analysis of more complicated data structures, such as arrays and records, and we will discuss this application in more detail in the next chapter.

We introduce one new concept which is useful for analyzing how information in a data-flow framework can be killed: that of *shields*. If every path from A to C must pass through B, we say that B *shields* A from C, or that $B \in$ *shield*(A,C). We note that B *dom* C is just a particular instance from the set of shields for C, where $B \in$

shield(Entry,C) $\Longrightarrow$ *B dom C.*

Shielding is important when dealing with the flow of information in a CFG, since $B \in$ *shield(A,C)* means that B can kill any information generated from A on all paths to C. In particular, any definition of v within B prevents all definitions of v in A from reaching C. It is not our intent to exhaustively investigate the properties of shielding in this work, but it will be a useful concept when closely examining the details of linearizing reaching definitions, and it will have a dual definition with respect to parallel graphs in Chapter 8.

2.3.3 Initial Specifications

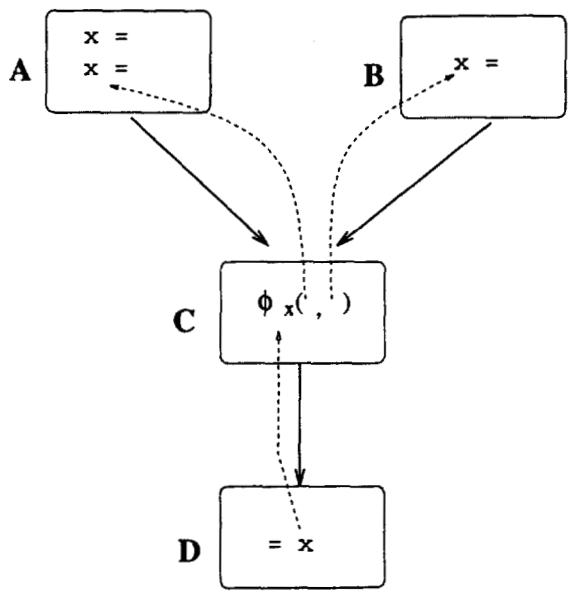


Figure 2.6 An example highlighting Definitions 2.1 – 2.4.

We now begin the investigation of how to represent abstractly the sequence of information which flows from point to point in a data-flow graph. In order to maintain such information in a sparse manner, we must understand how a particular piece of information, such as a variable definition, is propagated through the CFG. We start with several basic definitions:

Definition 2.1 *A* reference (ref) *is any definition, use, or merge operator of a given symbol.*

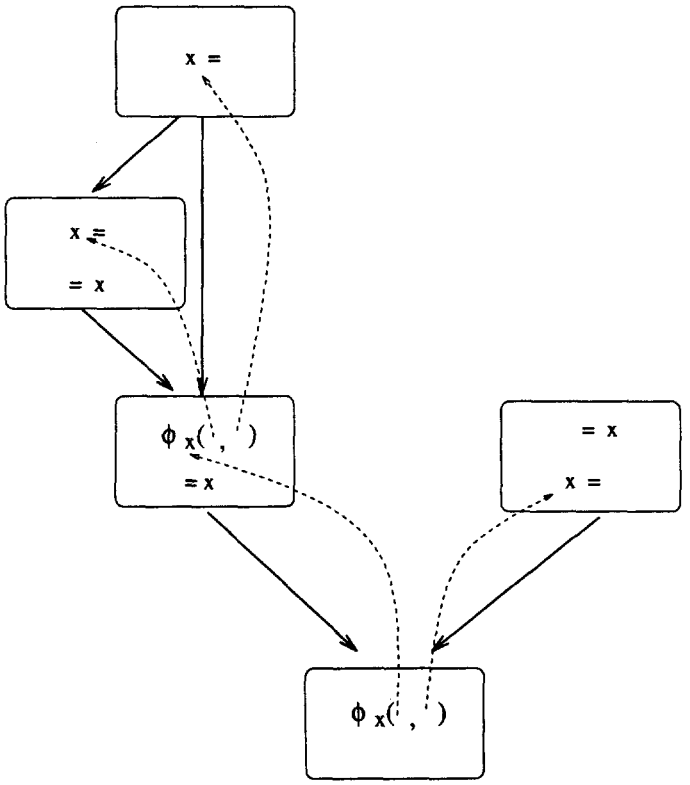


Figure 2.7 The ϕ-function merges downward-exposed definitions.

Definition 2.2 *A* link *is a pointer to the next or previous reference of a symbol.*

Definition 2.3 *A* chain *is a sequence of links that connects two or more references.*

Definition 2.4 *A* merge operator *augments the data-flow graph by collecting multiple links at branch or confluence nodes in the control flow graph.*

In terms of tuples, if *symbol(t)* $= s$ for some tuple t, then t *references* s. For example, examine Figure 2.6. In the four basic blocks depicted, there are five references: three definitions, one use, and one merge operator. An example chain would be from the use of **x** in node D, following its *link* to the merge operator (ϕ-function in this case) in node C, then following one of its *links* to either of the downward-exposed definitions in A or B.

A merge operator may point to either downward-exposed (reaching) references or upward-exposed references. We have these two cases:

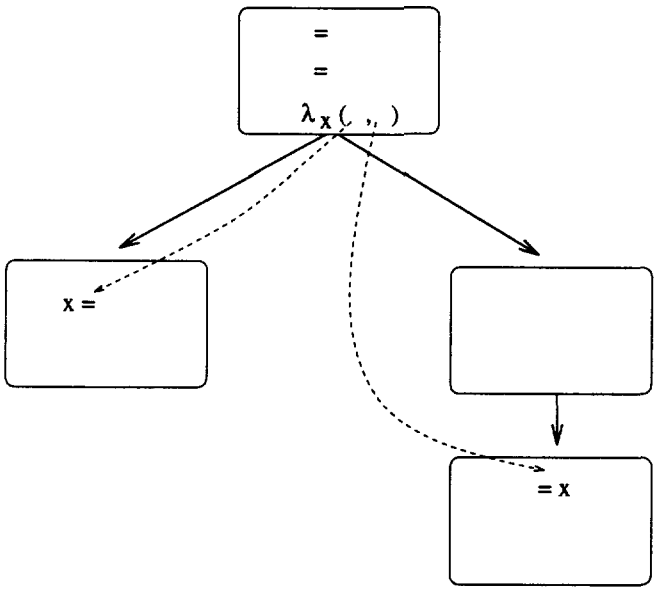


Figure 2.8 The λ-function merges upward-exposed references.

- **Downward-eXposed References (DXR)** – Operators are placed at the beginning of a confluence basic block, and merge two or more distinct downward-exposed references. The function has an argument for each control flow predecessor, which points to the most recent downward-exposed reference from the corresponding control flow path.

  An example operator which merges downward-exposed references is the ϕ-function illustrated in Figure 2.7. In this case, (SSA) references are definitions or other ϕ-functions.

- **Upward-eXposed References (UXR)** – Operators are placed at the end of a branch basic block with an argument for each outgoing control flow edge. As an example, we consider the λ-function, which collects upward-exposed references. Each argument of the λ-function points to the first upward-exposed reference (use, definition, or another λ-function) from the corresponding control flow path. An illustration of the λ-function is given in Figure 2.8.

2.3.4 Following Chains

In order to provide clarity for discussion and examples we present convenient notation with which to refer to a statement–based program in terms of its references.

Each variable site will be a reference (**ref**) for variable v in one of the following ways:

- $\mathbf{D}_n^v$ – a definition of v at statement n in the program.
- $\mathbf{N}_n^v$ – a nonkilling definition v at statement n in the program.
- $\mathbf{U}_n^v$ – a use of v at statement n in the program.
- $\mathbf{B}_n^v$ – both a definition and use v at statement n in the program.

We use the following functions to extract information from a given reference:

- $num(\mathbf{ref}_n^v) = n$
- $var(\mathbf{ref}_n^v) = v$

As an example, consider this code:

```
S1:  T1 = ...
S2:  loop
S3:     T2 = φ(T1,T7)
S4:     if TEST then
S5:        T4 = ...
S6:     endif
S7:     T6 = φ(T4,T2)
S8:     ... = T6
S9:     T7 = ...
S10: endloop
```

If we start with the use of T at S_8, U_8^T, we can follow its *link* by examining *chain(* U_8^T *)*, which is equal to D_7^T, the pseudo-definition of T (a ϕ-function) at S_7. Since the chaining function at S_7 merges two other *links*, we may choose to follow either. Following the first chain ends at the definition of T at S_5, while following the second chain results in a further ϕ-function at S_3. Traversing its *links* leads to the definitions at S_1 and S_9.

Since we were following *links* created by SSA form, we essentially discovered the definitions that could reach the use of T at S_8. This result is not surprising, since SSA is

a representation of the reaching definitions that can affect information at a given point in the program.

2.3.5 Applications

What kind of data-flow problems use reference chains? In general, DXR chains are used for forward problems, those in which the information desired at a point is in terms of behavior which occurs previously according to control flow or time. Thus, *links* will point to the reference operation of that behavior.

On the other hand, if information at a point is dependent upon what will happen later on ("forward in time with respect to control flow", known as backward problems), then the chains based upon UXR will be the reference chain of choice.

Here are some of the data-flow problems for which we have applied reference chaining:

DXR functions

- Induction variable detection [GSW]. Uses the SSA graph.
- Constant propagation. Uses SSA graph for scalars, augmented with def-def chains for arrays. Constant propagation is covered in detail in Chapter 4.
- Reaching definitions – for variables. Uses SSA graph.
- Availability – for expressions. This problem is the dual of one covered in Chapter 7: anticipatability.
- Scalar data dependence. The full solution requires many kinds of DXR functions. This topic is covered extensively in Chapter 6.

UXR functions

Chapter 7 contains details of using UXR functions to analyze backward problems. Examples of problems which take advantage of this technique are:

- Liveness – for variables. Uses UXR for uses and defs.

- Anticipatability – for expressions. In this case, references are extended to include complete expressions, rather than just arbitrary symbols.
- Useless code identification (based upon liveness information)

2.3.6 Extensions to Reference Chaining

The concepts of reference chaining, while fine for sequential constructs, need to be extended to accept the semantics of parallel constructs. We feel that a sound and coherent method needs to be provided that allows sensible reasoning about programs written using parallel constructs. We would like to use some of the same reasoning methods as employed with sequential programs. We must be careful to define the references that are exposed in light of these parallel constructs. In particular, is a new merge operator needed at a parallel confluence point? How is a parallel confluence point different than a sequential confluence point? As this issue is separate from the basic applications of reference chaining on sequential programs, we will delay the investigation of these questions until Chapter 8.

Chapter 3

FUD Chains

3.1 SSA Construction

In this chapter we provide details on an extended SSA form, which we call Factored Use and Definition Chains (FUD chains), and show how to construct them efficiently. We furnish statistics on how expensive, in terms of time and space, it is to build these graphs in practice. We also explain how to manage the semantics of FUD chains in the face of global variables, procedure calls, and generalized nonkilling definitions.

3.1.1 Merging Reaching Definitions Within a CFG

Using a traditional bit-vector method [Kil73] to keep track of data-flow information through the CFG is *dense*: it propagates information for each variable at all nodes, even those that do not use or contribute to the solution of a given problem. Alternatively, if def-use information is sufficient for a particular problem, the graph utilized is *sparse*: by following the def-use chains, nodes are bypassed that are not part of the solution.

Sparsity will not prevent multiple chains from converging at the same node. As noted in Chapter 1, one of the key properties of SSA form is that every use of a variable symbol will have exactly one reaching definition. One way this property becomes important is when we examine the general nature of reaching definitions, especially in the presence of confluence nodes in the CFG. For example, Figure 3.1 shows 3 definitions of v that merge at a confluence node, then split into 3 uses of v. General reaching definitions would require 9 def-use chains to express all these possibilities. However, we may coalesce the 3 definitions at the merge (the other key definition of SSA form) into a ϕ-function. The

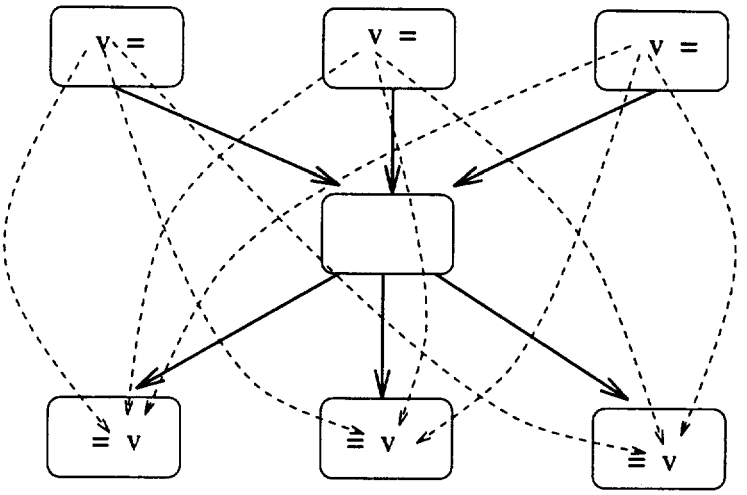


Figure 3.1 Reaching definitions can be quadratic in general.

ϕ-function collects multiple definitions of the same variable that reach a confluence node along more than one path. In this way, each use has only one chain as its destination (as depicted in Figure 3.2), and now only 6 def-use links are required to express the same information. This phenomenon has been noted previously in the literature [CCF94, CFR+91].

To generalize, when n definitions of a variable symbol reach a confluence point then split into n uses of v, the result is n^2 total links. However, if the code is transformed into SSA form, the number of def-use links to express the same information is a linear function of the number of references (we refer to this phenomenon as *linearizing* reaching definitions), cutting the total number of links from n^2 to $2n$.

At what points in the CFG do we insert ϕ-functions? Clearly, ϕ-functions are only needed at confluence nodes, since these are the only possible places where multiple definitions of the same variable can reach concurrently. However, ϕ-functions are not needed at all confluence nodes, since there may not be distinct variable definitions along more than one incoming branch.

To answer the question of ϕ-function placement, we look at the concept of the *join* of a set of nodes $\mathcal{S}$. Informally, the join of $\mathcal{S}$ is defined to be the set of all nodes Z such that there are two nonnull CFG paths that start at two distinct nodes in $\mathcal{S}$ and converge

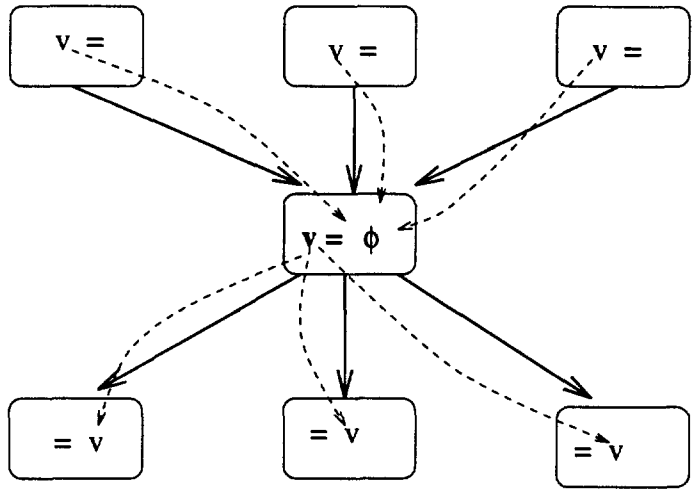


Figure 3.2 SSA form can linearize reaching definitions.

at Z [CFR+91]. Formally, we define the join as follows:[†]

Definition 3.1 *The* join *of nodes* X *and* Y, $J(X,Y) =$
$\{Z \mid \exists Z_X, Z_Y$ with $Z_X \rightarrow Z$ and $Z_Y \rightarrow Z$,
paths $p_X : X \xrightarrow{*} Z_X$ and $p_Y : Y \xrightarrow{*} Z_Y, p_X \cap p_Y = \emptyset\}$

The join of a set of nodes, $\mathcal{S}$, denoted $\mathrm{J}(\mathcal{S})$, is defined to be the union of the pairwise joins $\forall X, Y \in \mathrm{S}$, i.e., $\mathrm{J}(\mathcal{S}) = \bigcup_{X,Y \in \mathcal{S}} \mathrm{J}(X,Y)$. The iterated join, $\mathrm{J}^+(\mathcal{S})$, is defined as the limit of increasing sequences of nodes defined by:

$$\begin{aligned} \mathrm{J}^1(\mathcal{S}) &= \mathrm{J}(\mathcal{S}) \\ \mathrm{J}^2(\mathcal{S}) &= \mathrm{J}(\mathcal{S} \cup \mathrm{J}^1(\mathcal{S})) \\ \mathrm{J}^{i+1}(\mathcal{S}) &= \mathrm{J}(\mathcal{S} \cup \mathrm{J}^i(\mathcal{S})) \end{aligned}$$

For reaching definitions, $\mathrm{J}^+(\mathcal{S})$, the iterated join of the set $\mathcal{S}$ of nodes with nonidentity transfer functions for variable v (nodes with definitions of v plus *Entry*), is the correct placement for merge operators since it insures the following two properties:

1. Each join point K for v captures (directly, or indirectly via another join point) all

[†]The intersection of paths p_1 and p_2, $p_1 \cap p_2$, is the set of nodes in common to p_1 and p_2.

reaching definitions of v at K. This property was proven recently by Choi *et al.* [CCF94, Theorem 3.3].

2. Each variable use of v at CFG node A will have a single reaching definition. In the set D of nodes that contain variable definitions or join points of v, there exists exactly one element $E \in$ D such that E *dom* A and $E \in$ *shield*(d, A), $\forall d \in$ D.

   Proof of 2:

   Let F $\subseteq$ D be the set of elements of D that dominate A. We first show that if D contains any node C (which can reach A) other than *Entry*, F will contain a node other than *Entry*. Let L = J(C,*Entry*). If C *dom* A or $A \in$ L for any such C, we are done. Otherwise, choose any path p_1: $C \xrightarrow{+} P$, where $P \rightarrow A$ (see Figure 3.3). Let Q be the last node on p_1 such that $Q \in$ L. Such a Q must exist since $A \notin$ L. We first claim that all paths from C to A include Q. If not, consider path $p_2 : C \xrightarrow{+} A$ which does not contain Q. $p_2 \cap \{Q \xrightarrow{+} A\} = \emptyset$ since Q is the last node on p_1 in L. But in this case we have paths *Entry* $\xrightarrow{+} Q \xrightarrow{+} A$ and p_2 that only intersect at A, which is not possible because $A \notin$ L. We now claim that Q *dom* A. If not, some path p_3 from *Entry* to A does not pass through Q. Reasoning as above, we would then have disjoint paths from *Entry* to A and from C to A (except for A), which implies that $A \in$ L. Since we assumed otherwise, we conclude that Q *dom* A.

   Since the dominators of A form a linear ordering [†] [Hec77, Lemma 3.4], choose $E \in$ F (E is a particular Q in terms of Figure 3.3) such that all other elements of F dominate E. Then $E \in$ *shield*(f, A), $\forall f \in$ F, due to the linear ordering of F.

   Now consider B, any element of D that reaches A but does not dominate A. All paths from B to A must pass through E. If not, some path p_4 from *Entry* to A passes through E before B (this condition cannot be true for all paths from *Entry* to A, since B $\overline{dom}$ A). But then we can construct a path *Entry* $\xrightarrow{+} B$ concatenated with $B \xrightarrow{+} A$, which does not pass through E. Since this conclusion contradicts the relationship E *dom* A, all paths from B to A pass through E, which is equivalent to $E \in$ *shield*(B, A). ∎

[†]Given any nodes P,Q, and R, if P and Q dominate R, either P dominates Q or Q dominates P.

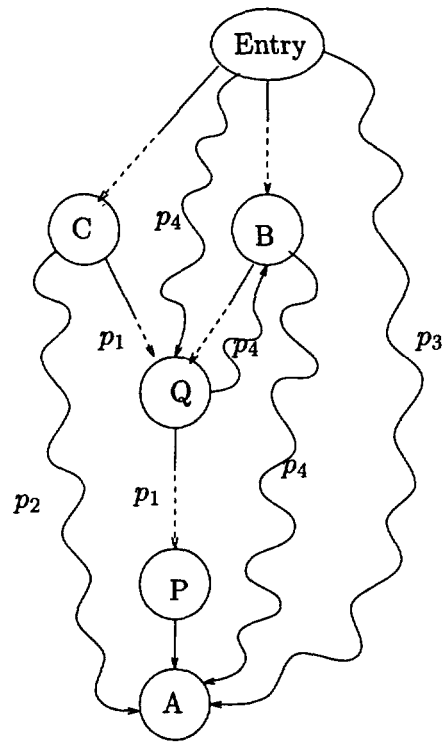


Figure 3.3 Possible paths to consider regarding the *join* property

Calculating the iterated join of $\mathcal{S}$, $J^+(\mathcal{S})$, may seem expensive, but it has been shown [CFR+91] to be equal to the *iterated dominance frontier* of $\mathcal{S}$. We define the dominance frontier of a set of nodes, $\mathcal{S}$, as follows: $DF(\mathcal{S}) = \bigcup_{X \in \mathcal{S}} DF(X)$, and DF^+ is defined similarly to J^+. The iterated DF, $DF^+(\mathcal{S})$, is defined as the limit of increasing sequences of nodes as follows:

$$\begin{aligned} DF^1(\mathcal{S}) &= DF(\mathcal{S}) \\ DF^2(\mathcal{S}) &= DF(\mathcal{S} \cup DF^1(\mathcal{S})) \\ DF^{i+1}(\mathcal{S}) &= DF(\mathcal{S} \cup DF^i(\mathcal{S})) \end{aligned}$$

For the relation $DF^+(\mathcal{S}) = J^+(\mathcal{S})$ to hold, the *Entry* node of the CFG must be in $\mathcal{S}$.

We next show how to incorporate the ideas of this section into efficient algorithms that transform the intermediate form provided by the front end of the compiler (the CFG and data-flow graph) into SSA form.

3.1.2 Building the Graph

In converting intermediate code into SSA form, we generally follow the algorithm given by Cytron *et al.* [CFR+91]. When performing this conversion, we follow four main steps:

1. **Preliminary analysis.** This step includes most of the analysis discussed in Chapter 2, such as loop identification, loop augmentation, and dominator analysis.

2. **Variable Modification List.** We make one pass through the CFG to create a linked list of variables modified anywhere in the procedure, and a list of modification sites, $A(V)$, for each variable V. This list is generated by simply examining all data-flow tuples that belong to each node of the CFG using a depth-first search, although any search algorithm that visits all nodes will suffice. We note that all variables are assumed to be initialized (hence defined) at *Entry*, a property required by the equivalence between J^+ and DF^+, as discussed in the last section.

3. ϕ **Placement.** For each variable V, we place a ϕ-function at the iterated dominance frontier of all nodes in $A(V)$. This is accomplished by a worklist algorithm, as shown in Algorithm 3.1, lines **11 - 25**. The number of arguments of each ϕ-function is equal to the number of predecessors of the CFG node in which it resides. A ϕ-function is always added to the top of a basic block – in that way, any use of the same variable within that basic block will be reached by the ϕ-function. All details are given in Algorithm 3.1. We should point out that ϕ-function placement as shown here is asymptotically quadratic in theory but usually linear in practice. A new algorithm has recently been developed to place ϕ-functions in linear time with respect to the number of variables in the procedure [SG93].

4. **Chaining.** A depth-first pass is made of the dominator tree, pushing definition sites onto a stack when encountered. Each variable use has its *link* field filled in with a pointer to the current definition of that variable. This step was originally called "renaming" [CFR+91], since each variable definition was iteratively numbered, as we have shown in examples. However, this numbering reflects the semantics of SSA form, and we present our algorithms in terms of pointer references, which

maintains the single reaching definition property of SSA.

Flow graph successors are then checked for ϕ-functions, filling in the corresponding ϕ-argument *link* field with the current reaching definition at that point. This procedure is performed recursively on the dominator tree children, logically popping definitions off a definition stack when returning. In actuality, this task is performed by saving the previous current definition at each tuple that defines a variable (in which a ϕ-function counts as another definition), then restoring the saved value when returning from the *Chaining* routine.

A traditional use-def chain would list all definitions of a variable that reach the current use. The result of the preceding procedure is the *factored* form – each use has exactly one reaching definition (see Figure 3.6b), thus preserving SSA semantics.

3.1.3 Construction Algorithms

In this section we give the details of SSA construction. The methods employed are important since we will adjust this basic algorithm to accommodate extensions to SSA as well as other reference chaining methods. Step 3 from the previous section is performed by Algorithm 3.1, while Step 4 is performed by Algorithm 3.2. We describe here the data structures used for the following algorithms:

- *A(V)* – A list of all nodes with assignments to variable V.
- *symbol(tuple)* – A function that returns the variable symbol (name) associated with this tuple, if it exists. Returns *null* otherwise.
- *V.CurrentDef* – A pointer to the current definition (tuple) of variable V. Logically points to the top of a definition stack. Initialized to *source*.
- *t.SavedDef* – A pointer to the current definition of *symbol(t)* before processing this tuple. Used to logically pop definitions off a stack when returning from recursive calls down the dominator tree.
- *DFRONT(N)* – Dominance frontier for node N.

- *WhichPred(N,Q)* – An integer indicating which predecessor of Q in the CFG is N.
- *Work_List* – An unordered set of CFG nodes. For each variable V, Work_List is initialized to A(V), all assignments to V.
- *HasFunc(*)* – A reference field to a variable for each CFG node. HasFunc(N) = V means block N already has a ϕ-function added for variable V.
- *Work(*)* – A reference field for each CFG node. Work(N) = V means that node N has already been added to the Work_List for variable V.

3.1.4 Interprocedural Links and Local CFGs

When transforming the IR into SSA form, we encounter two types of procedures: the main procedure of a program and any subroutines. The distinction is how variables are initialized, specifically how we handle formal arguments. Other issues include how to link variables correctly at call sites and how to deal with global variables. We address these issues in this section.

Formal Arguments. For any CFG, the current definition of a variable is initialized to *source*. This initialization reflects the property noted in §3.1.1 that all variables have an assumed definition at *Entry*. However, a subroutine often has variables passed in as formal arguments. Consider this case:

```
subroutine foo( a, b )
     integer i
     i = a + b - 3
end foo
```

where `a` and `b` presumably have actual values passed into `foo`. Although it may be that neither variable has a known value (which may be determinable through interprocedural constant propagation) when entering the subroutine, it may be that they do. We would like to accommodate this possibility by providing a placeholder for formal arguments, which can be assigned values provided by interprocedural analysis.

Given: $A(V), \forall\ V$.
Do: *compute DFRONT(N),* $\forall\ N \in CFG$.
Result: ϕ*-functions inserted into CFG*

```
1:      for all nodes N do
2:          HasFunc(N) ← ∅
3:          Work(N) ← ∅
4:      endfor

5:      for each symbol V do
6:          Work_List ← ∅
7:          for each N in A(V)
8:              Work(N) ← V
9:              Work_List ← Work_List ∪ { N }
10:         endfor
11:         while Work_List ≠ ∅ do
12:             remove N from Work_List
13:             for each Q ∈ DFRONT(N) do
14:                 if HasFunc(Q) ≠ V then
15:                     HasFunc(Q) ← V
16:                     i ← number of predecessors of Q
17:                     place V = φ( V1, V2, ..., Vi ) at the beginning of basic
18:                         block Q, where Vj corresponds to the j^th predecessor of Q
19:                 endif
20:                 if Work(Q) ≠ V then
21:                     Work(Q) ← V
22:                     Work_List ← Work_List ∪ { Q }
23:                 endif
24:             endfor /* each Q in DFRONT */
25:         endwhile
26:     endfor  /* each symbol V */
```

Algorithm 3.1 Placement of ϕ-functions

Given: Initialized data structures.
Do: Call *Chain(Entry)*
Result: SSA form

```
 1: Chain(N)
 2:       for all tuples t ∈ N, in forward order do
 3:           V ← symbol(t)
 4:           if t is an ordinary use of V then
 5:               link(t) ← V.CurrentDef
 6:           endif
 7:           if t defines V then
 8:               t.SaveDef ← V.CurrentDef
 9:               V.CurrentDef ← t
10:           endif
11:       endfor  /* all tuples of N */
12:       for each Q ∈ Succ(N) do /* Successors in CFG */
13:           j ← WhichPred(N,Q)
14:           for each φ-function merge tuple f in Q do
15:               V ← symbol(f)
16:               link( j^th argument of f ) ← V.CurrentDef
17:           endfor
18:       endfor
19:       for each Q ∈ Children(N) do /* children in dom tree */
20:           Chain(Q)
21:       endfor
22:       for all tuples t ∈ N, in reverse order do
23:           if t is a definition tuple do
24:               V ← symbol(t)
25:               V.CurrentDef ← t.SaveDef
26:           endif
27:       endfor
28: end Chain
```

Algorithm 3.2 Chaining: linking each use to its unique definition and correctly inserting ϕ-function arguments

The solution in this case is to create tuples ⟨ *formal*,∅,∅,∅,a ⟩ and ⟨ *formal*,∅,∅,∅,b ⟩, which are inserted into *Entry* after *source*, which is always the first tuple. In this way, lines 7 - 10 in Algorithm 3.2 sets the current definition of each variable (*V.CurrentDef*) to the *formal* tuple. The use of a and b in our example will have their *link* fields set to these *formal* tuples in lines 4 - 6 of Algorithm 3.2. The *formal* tuples thus provide a convenient location through which to pass interprocedural information.

Procedure Parameters. Special care is needed when handling procedures in SSA form. As noted in §2.3.1, procedure arguments represent references of variables that are treated as nonkilling definitions. These references are also uses of that variable, since they potentially pass values to the called procedure. In this case:

```
S1:  x = ...
S2:  call bar( x )
S3:  ...= x
```

the use of x at S_3 will have its reaching definition point to the definition (potentially nonkilling) at S_2. But the reference of x at S_2 is also a use (potentially) of x defined at S_1. So it needs its *link* field to point to the definition at S_1. The work by Cytron *et al.* [CFR+91] suggests adding another assignment statement, essentially x = x in this case, to create the proper LHS and RHS behavior for linking into SSA form. In order to accomplish this task, a procedure argument essentially needs to be split into two phases: one that represents potential use of the variable, and one that represents a potential definition of the variable. One way to accomplish this goal is described later in this chapter under Notes on Implementation, §3.1.5.

Global Variables. Global variables present a special problem since they are visible within all routines, even though they are defined only in a program's main procedure. The solution we adopt is two-fold. First, for all global variables v that are referenced within a local procedure we create the tuple ⟨ *global*,∅,∅,∅,*symbol* v ⟩. This tuple is inserted into the *Entry* node, similarly to tuples created for formal arguments. Again, lines 7 - 10 in Algorithm 3.2 will set *v.CurrentDef* to this *global* tuple.

Second, for interprocedural analysis, a particular global variable may not be referenced in some procedure (hence, it is, in some sense, invisible to that procedure), but may be referenced by both a callee procedure and caller procedure. As an example, assume we have the following call graph: $\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{C}$, where procedure **A** calls procedure **B**, which in turn calls procedure **C**. If **A** and **C** reference v, but not **B**, then analysis of global variables can be obfuscated due to the fact that there is no placeholder for v when processing **B**. We handle this situation by appending an extra argument to each parameter list at all procedure call sites. Essentially, an extra *proc_param* tuple is added to the end of each argument list, with the *symbol* field pointing to a special generic global symbol. We have denoted this special symbol as *symbol_invisible*, since it represents the set of all global variables within the called procedure that are not referenced within that procedure. In this way we provide a placeholder for any analysis that contains symbols invisible to a particular procedure.

Local CFGs. Normally, and certainly the case for the scientific codes described in Table 2.1, there are numerous procedures that together comprise the entire code. Since our intermediate form is essentially intraprocedural, each procedure has its own CFG constructed, with the methods (analysis and modifications) described in Chapter 2 applied to that CFG. Since there is a *slice* edge from *Entry* to *Exit* in each local CFG, a useful property of SSA form is that a ϕ-function for variable v will be placed in the local *Exit* node if and only if there is a definition of v within the body of the procedure. The advantages of this property were listed in §2.1.3, and we now prove the correctness of this statement.

Theorem 3.1 *Within any CFG G containing a slice edge, there exists a ϕ-function for v at the Exit of G if and only if v is defined within G - {Entry, Exit}.*

Proof:

$\Longrightarrow$ Given that a ϕ-function for v exists at *Exit*, one of its arguments must point to the last definition of v within *Entry*, since an edge exists from *Entry* to *Exit* and there exists an argument within a ϕ-function for each control flow predecessor. By definition, ϕ-functions for v are placed at $\mathrm{DF}^+(\mathcal{S})$, where $\mathcal{S}$ is the set of nodes that define v. *Entry*

always has an empty dominance frontier, since it dominates all nodes within the CFG. Thus, no definition within *Entry* can result in the creation of ϕ-functions. We also note for completeness that the dominance frontier of *Exit* is empty, since it has no outedges by definition. Thus, at least one other argument of a ϕ-function for v at *Exit* must point to a definition of v within G - {*Entry*, *Exit*}.

$\Longleftarrow$ Given that v is defined within G - {*Entry*, *Exit*}, and choosing node M from G - {*Entry*, *Exit*} (M may not be unique), we must show that *Exit* $\in \mathrm{DF}^+(M)$. We know that M $\overline{dom}$ *Exit* , since there exists a direct path from *Entry* to *Exit* via the slice edge. We also know that there exists a path from M to *Exit* by the original definitions in §2.1.1. Since the path from M to *Exit* and the slice edge converge at *Exit*, *Exit* is in the join of M and *Entry* by Definition 3.1. This result also uses the assumption that every variable used within a CFG has an initial definition at *Entry*. But, $\mathrm{J}(\{Entry\ ,\ M\}) \subseteq \mathrm{J}^+(\{Entry\ ,\ M\}) = \mathrm{DF}^+(\{Entry\ ,\ M\})$. Since ϕ-function are placed at DF^+ of nodes that define a variable, a ϕ-function for v will be placed at *Exit* . ▌

3.1.5 Notes on Implementation

Initialization. In order for $\mathrm{DF}^+(\mathcal{S}) = \mathrm{J}^+(\mathcal{S})$, *Entry* $\in \mathcal{S}$. This property is accomplished by setting *V.CurrentDef* = *source*, as noted in the description of the data structures for Algorithm 3.2. However, there is now no need to add *Entry* to the *Work_List* of Algorithm 3.1 (which computes the $\mathrm{DF}^+(V)$ for ϕ-function placement) for any V, since the dominance frontier of *Entry* is always empty.

Handling Procedure Parameters. Since procedure parameters must be regarded as nonkilling definitions and also as uses of a variable, we noted that each procedure argument must be split into two parts. Here, we describe one method to accomplish this task. We first illustrate the idea with the same simple example used before:

```
S1:  x = ...
S2:  call bar( x )
S3:  ...= x
```

Figure 3.4 shows the code above in its data-flow representation. We treat the *proc_param* tuple as a definition of `x` *if* its *right* operand (an arbitrary choice) is an

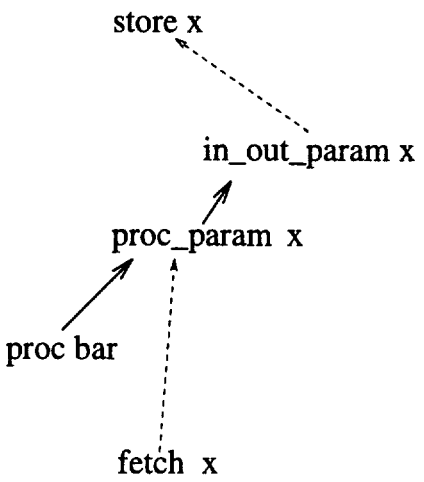


Figure 3.4 Data-flow graph for simple procedure call.

out_param or *in_out_param*; in that case we know or must assume that **x** gets modified (defined) by the call to **bar**. Similarly, the *right* operand of a *proc_param* tuple is treated as a variable use if it is of type *in_param* or *in_out_param*; in this case lines **4 - 6** of Algorithm 3.2 will fill in the *link* field with a pointer to the current definition of **x**.

The compiler front-end sets all *right* operands of *proc_params* as an *in_out_param*. This initialization is the default for Fortran, and is the conservative choice in the absence of more precise information. More information may be found by performing interprocedural mod/ref analysis prior to the translation into SSA form. In this way, we can change the type of an *in_out_param* when more precise information becomes available. For example, if mod/ref analysis provides the information that **bar** will not modify **x**, the *in_out_param* in Figure 3.4 is changed to *in_param*. Now, the *Chaining* routine will not consider the argument to **bar** a definition of **x**, so when the use of **x** at S_3 is processed by lines **4-6** of Algorithm 3.2, **x**.*CurrentDef* will be equal to the *store* at S_1. The result of the extra precision is shown in Figure 3.5.

Multiple procedure parameters that reference the same variable create another difficult situation regarding the correct generation of the V.*CurrentDef* field for V and filling in the *link* field on all uses of V. We illustrate the problem again with an example:

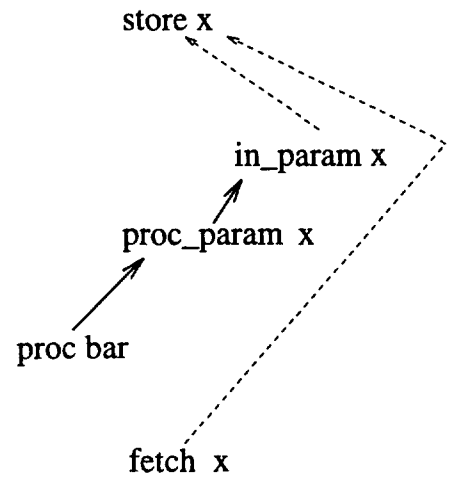


Figure 3.5 Data-flow graph with more precision about the procedure parameter.

```
S1:  X = 1
S2:  call sub1( X + 2, X, X + 3 )
S3:  Y = X
```

Here we want all the uses of `X` at S_2 to fetch the definition at S_1 and the use of `X` at S_3 to fetch the definition from the second argument in `sub1` (assuming that mod/ref analysis has not determined that the second argument is unmodified). If we are not careful, the third argument of sub could fetch the definition created from the second argument.

This problem is essentially an engineering issue. One solution (the one that we adopted) is to rely on the tuple ordering in the abstract syntax tree as constructed by the compiler front-end. The front-end builds a representation in which all the *proc_parm* tuples occur lexically after their *right* operands. For the second and third arguments from S_2, the *right* operands will be an *in_out_parm* and an *arith* (a '+' sign will be at the base of the abstract syntax tree representing this expression), respectively. Thus, all uses are processed before any definitions change *V.CurrentDef*.

While this solution works fine in this instance, a more insidious problem can occur in this case:

```
S1:  X = 1
S2:  call sub2( X + 2, X, X )
S3:  Y = X
```

Now, which definition of X in S_2 should the *link* field of the use of X in S_3 point to? We note that in some languages, notably Fortran, the call to sub2 represents nonconforming code, since such aliases are not allowed. Nonetheless, this situation would seem to impose an arbitrary order upon the modifiable arguments of a procedure call. One solution is to use lexical ordering, but this method essentially chooses a solution by fiat. Another solution is to merge the second and third arguments into a ϕ-function before the use at S_3. We believe that to some extent this question is best left to language developers, and can be answered for each language implementation. We will return to this problem in a different guise when we look at other chaining methods, notably when we solve the *reaching uses* problem in Chapter 6.

SSA Representation When translating a program into SSA form, it is convenient, as was shown in Figure 1.1, to denote each definition as a new variable. This method is useful for readability, and is the one described in the original work by Cytron *et al.* [CFR+91].

Although this traditional SSA form renames variables uniquely at every definition point, it is not really practical (and certainly not desirable) to add new names to the symbol table for all assignments. Thus, the common implementation [JP93, WZ91] actually provides def-use links [ASU86] for each new definition (see Figure 3.6). Since each use is the head of exactly one *link*, the semantics of SSA are preserved.

This def-use chain style of SSA implementation lends itself well to iterative forward data-flow problems (such as constant propagation [WZ91]) due to consistency of direction between program flow and def-use links. However, a demand-driven data-flow problem will typically request information at a program point from its data-flow predecessors. As we shall see, use-def chains admirably match the demand-driven style of data-flow analysis.

We maintain the semantics of SSA by providing *use-def links*, so that each use (*fetch* in the intermediate representation) has a *link* to its single reaching definition. The contrast in providing use-def versus def-use links is shown in Figure 3.6, and its advantages for data-flow analysis are covered in Section 3.3.

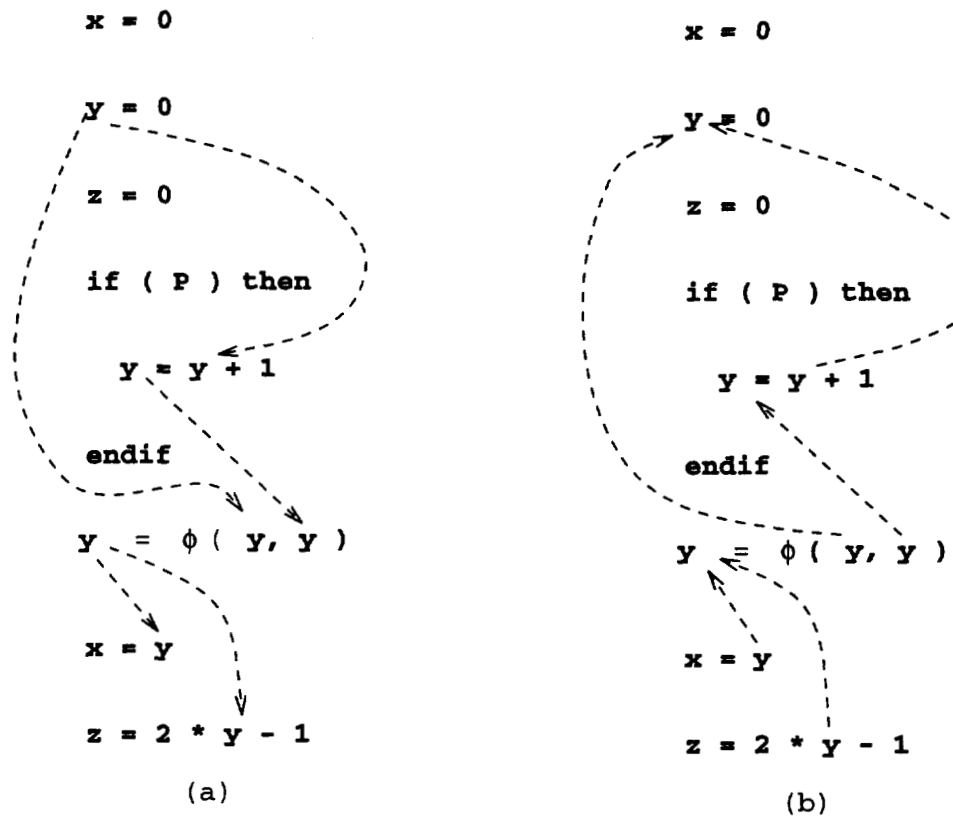


Figure 3.6 Comparison of standard SSA implementation employing (a) def-use links and (b) use-def links.

Another advantage of our approach is that it requires only constant space per node to implement. Since each variable use has exactly one reaching definition, it has only one *link* field. A traditional def-use implementation, however, must have the capability to dynamically expand its use list at each definition site, since an unbounded number of uses can be dependent upon that site. Contrast Figure 3.6(a) with 3.6(b): since the arrows in (b) are stored with the node at the tail of each *link*, at most one *link* need be stored with each tuple.

Keeping Track of ϕ-function Placement. Since ϕ-functions are constructed per variable, Algorithm 3.1 needs a mechanism to determine whether a ϕ-function for a particular variable has already been placed at any node (a node may be in the DF^+ of numerous other nodes), since it is clearly undesirable, as well as unnecessary, to place more than one ϕ-function at a node for the same variable. The original algorithm

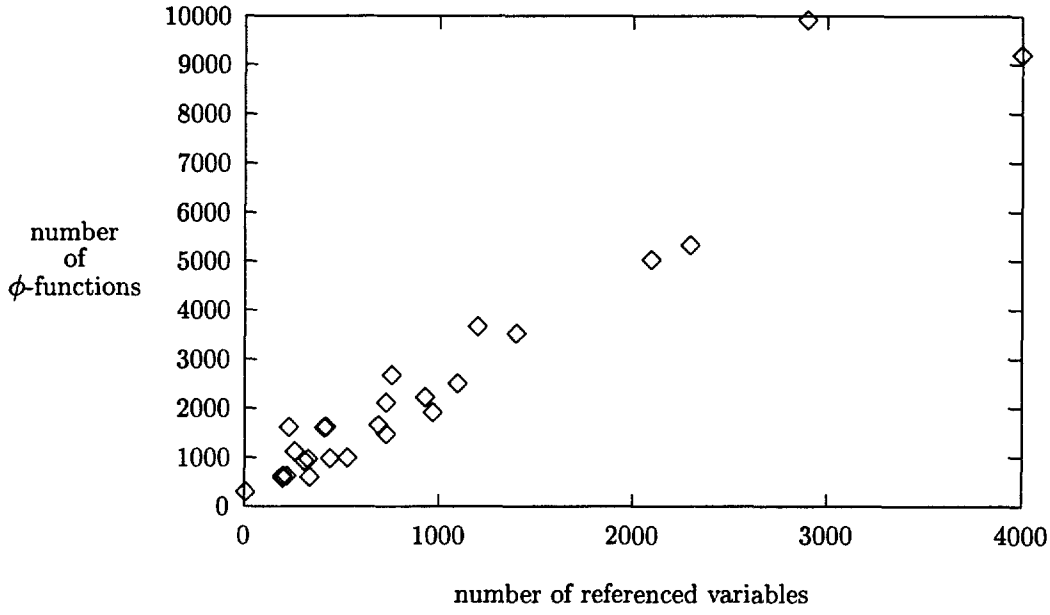


Figure 3.7 Comparing the number of ϕ-functions for each referenced variable

[CFR+91, Figure 11] renamed instances of variables by incrementing a counter. As we have already noted, new instances of a variable in our implementation are logically distinguished by keeping pointers to the most recent definition of that variable. Thus, we maintain a pointer *HasFunc* for each node which is filled in with a *link* to the last symbol (if any) which has had a ϕ-function placed at that node. In this way, lines **14 - 15** of Algorithm 3.1 do all the work that is necessary for insuring unique ϕ-function placement for every node in the CFG.

Measurements On Building the SSA Graph

We would like to compare our implementation with that of the original work by Cytron *et al.* We implemented Algorithms 3.1 and 3.2 in Nascent, running on a Sun IPX with 64 MB RAM, and upgraded with a Weitek 80 MHz clock-doubled chip. The code was compiled using GNU C++ version 2.5.8, optimization level -O2.

We counted the number of ϕ-functions generated in each of the benchmark programs from §1.4. Figure 3.7 shows the number of ϕ-functions for each of these programs as a function of the number of referenced variables, where a '◇' represents a data point for

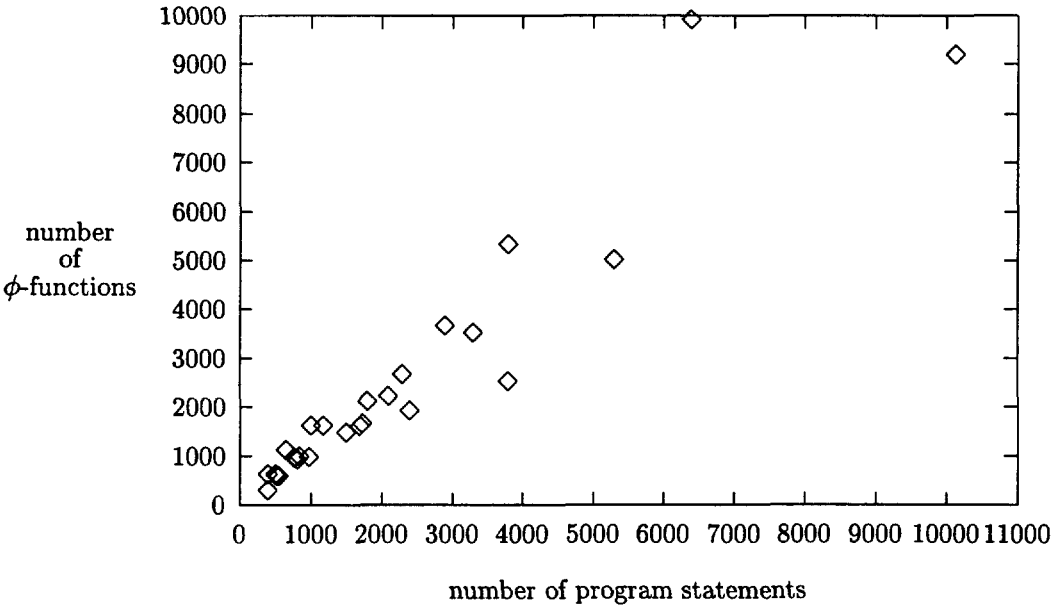


Figure 3.8 Comparing the number of ϕ-functions to program statements

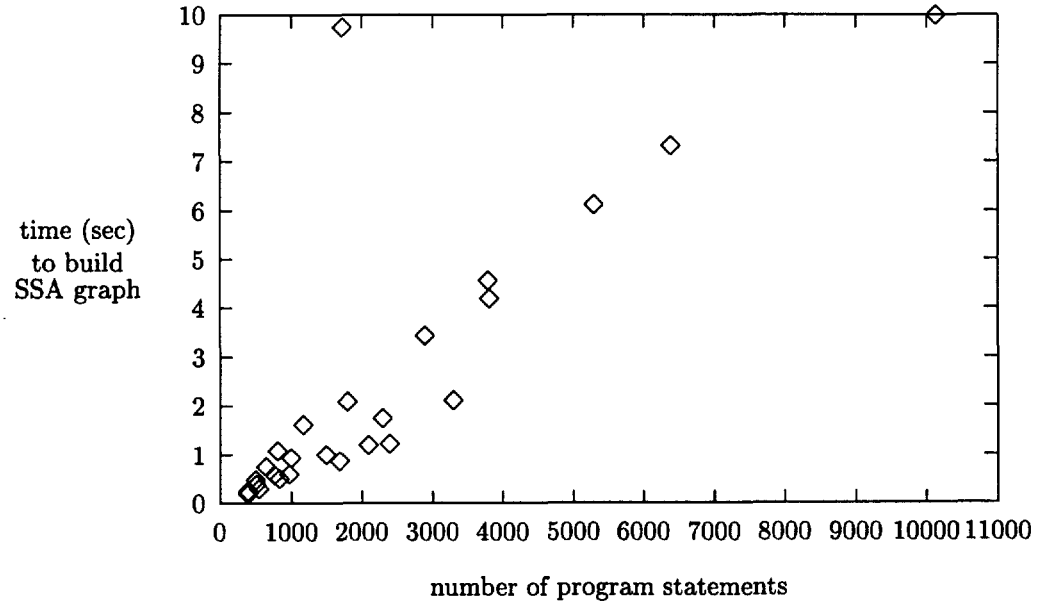


Figure 3.9 Time to build the SSA graph in terms of program statements

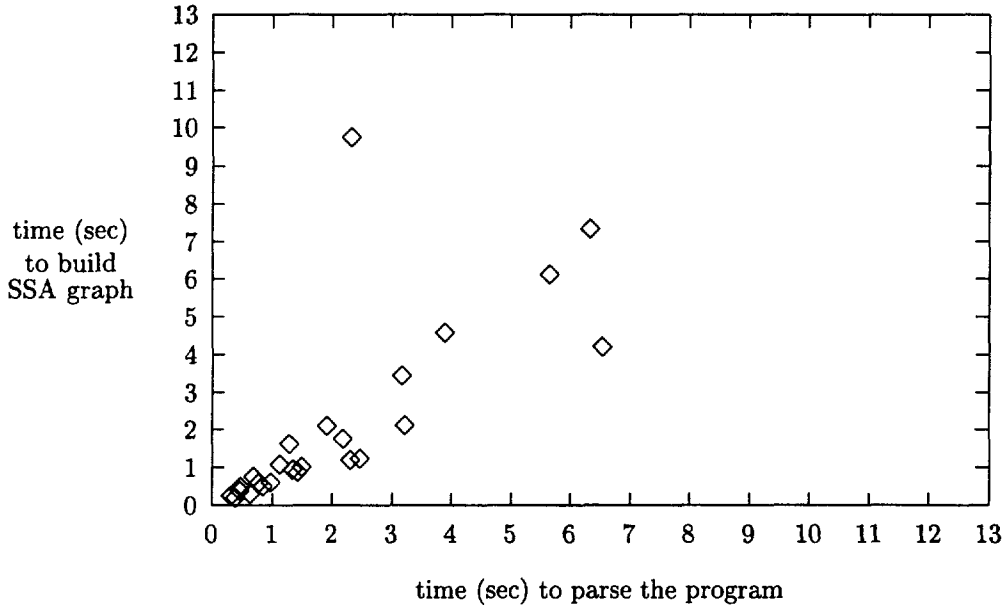


Figure 3.10 Comparing front-end compiler time with SSA build time

each of the 26 programs. We note that the relationship seems to be linear, with the ratio usually between 2 and 2.5. This result confirms earlier work that, in practice, the SSA graph is linear in the number of variables [CFR+91, Hav94]. The outlier, with a ratio of 3.4, is the `boast` program from the RiCEPS suite. Most of its behavior is due to the subroutine `master` with a ratio of 5.4, which includes one top-level loop with 9 exits, and a nest-level of 4 that contains 22 inner loops. This behavior of `master` was also noticed by Havlak [Hav94].

The original work by Cytron *et al.* built a statement-level CFG, where each basic block node contained a single statement. In our implementation, we have tried to create maximal basic blocks, maintaining a minimal increase in graph size. Thus, a straight comparison of the number of basic blocks to the number of ϕ-functions would be meaningless: multiple variables can be used and defined within a maximal basic block. Instead, we compared the number of program statements to the number of ϕ-functions, with our results shown in Figure 3.8. This graph confirms the previous result, with `boast` again being the outlier.

Next, we wanted to gather data on the amount of time necessary to build the SSA

graph structure. These results are given in Figure 3.9. Here, we compared the number of program statements to the time required to build the SSA form. It again appears linear, with the anomalous data point due to ocean in the Perfect suite. This anomaly in ocean is due primarily to our implementation, which accommodates interprocedural mod/ref analysis. We append to the parameter list of each procedure call a list of all global variables that are referenced within the local routine. This addition allows accurate determination of mod/ref usage, but may result in additional overhead when the number of global variables and call statements is large. In the main routine of ocean (941 lines and 640 statements) 135 of the 148 referenced variables are globals, with 204 subroutine calls being made.

Finally, since parsing the front end is somewhat constant for all compilers (they all must examine the entire program character by character), we compared the time taken to parse each program as compared to the time to build SSA form. The results given in Figure 3.10 are almost identical to those of Figure 3.9, which may just indicate that parsing time is directly proportional to the number of statements in a program.

3.2 Constructing FUD Chains

In this section we describe how to expand the principles of SSA form to allow greater analysis of programs. While one of the two principles of SSA is that each use of a variable has precisely one reaching definition, we extend this concept to include unique reaching definitions for definitions as well. That is, each use *or definition* of a variable will point to its last unique reaching definition. In this way we can continue an analysis phase, if desired, when a definition is encountered.

3.2.1 Definition

This expanded form of SSA will permit following definition chains through definitions themselves, if desired. Since these *links* create a sparse graph for any variable, we name these chains Factored Use and Definition Chains, or FUD chains.

Essentially, FUD chains consist of SSA form with extra *links*: those from definitions to

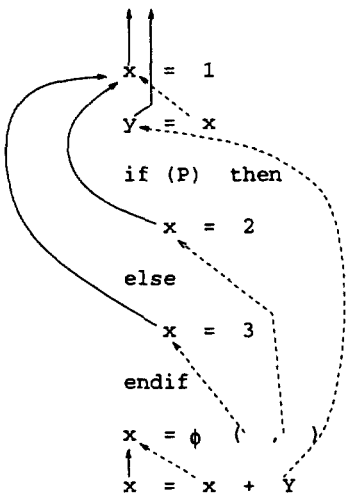


Figure 3.11 Use-def links plus def-def links make FUD chains.

definitions. How do we incorporate the extra def-def links via the construction algorithms already presented? The placement of ϕ-functions is unchanged, hence Algorithm 3.1 is unaltered. The chaining algorithm, Algorithm 3.2, needs minor adjustments. Since we already have the mechanism for linking available (each tuple contains a *link* field, which so far has only been utilized for variable uses), we simply modify line 4 of Algorithm 3.2 to read:

if *t is an ordinary use or ordinary def of V* **then**

The reason we specify that the use or def must be ordinary is that we want to exclude ϕ-functions. They are pseudo-definitions that effectively operate as both a definition (other definitions and uses may point to a ϕ-function as the unique reaching definition) and several uses (via the ϕ-arguments that collect those definitions that reach the node containing the ϕ-function). We illustrate the extra def-def *links* in Figure 3.11. The dotted arrows represent use-def *links* while the solid arrows are def-def *links*. The first definition of both x and y have empty def-def *links*, since they logically point to the assumed definition of each variable in *Entry*, the *source* tuple.[†]

[†]Note that we have omitted the *link* from P.

3.2.2 Additional Analysis

We now discuss some of the benefits of FUD chains over standard SSA form. The most notable advantage is in the additional precision offered for analyzing nonkilling definitions: arrays, structures, procedure parameters, aliases. They also are necessary for detecting scalar output dependence.

Arrays. Arrays have been a difficult data structure to incorporate into SSA form. The original work by Cytron *et al.* suggested creating new operators (*Access* and *Update*) for each array definition and use, so that they could be modeled as scalar variables [CFR+91]. For example `A[j] = k` could be transformed into `A = Update(A,j,k)`. In this way the subscript (which is actually a use of `j`) and the array name, along with the original RHS value, could be treated as uses.

Each use of an array could be transformed into a scalar access function via a scalar temporary. For example, `J = A(i)` would become:

```
Temp = Access(A,i)
J = Temp
```

After separating the components, SSA renaming proceeds as before.

This approach leaves much to be desired. It requires an extra amount of manipulation, and results in extraneous references that did not previously exist. A similar strategy was suggested to handle structures or records.

We treat definitions to arrays (and similarly, structures) as nonkilling definitions of the array name, since any definition of an array element modifies only a portion of the entire data structure. Utilizing def-def *links* provided by the FUD chain framework allows us to leave array statements unchanged. Instead, each LHS array reference points to the downward-exposed reaching definition of that same array. Analysis can proceed past any particular LHS array definition by following the def-def *links*.

As an example of following def-def *links* when analyzing an array, examine Figure 3.12. When processing the use of `A(5)` at the last statement we first follow the chain to its unique reaching definition, the definition of `A(J)`. Depending upon the analysis of

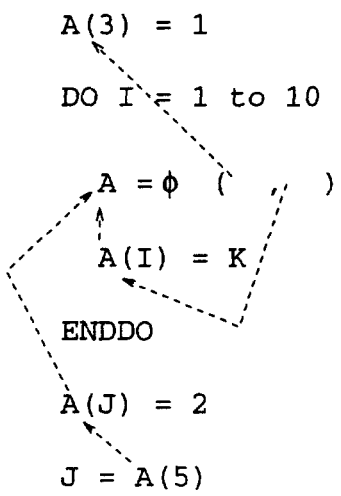


Figure 3.12 Following def-def links when analyzing an array.

the subscript J, we may choose to follow def-def *links* to the previous reaching definition, which in this case is a ϕ-function. We can continue to follow *links*, if desired, reaching both the definition of `A` inside the loop and the definition of `A` before the loop. We will see a direct application of this idea when we look at constant propagation in Chapter 4.

Procedure Parameters. As noted, the default assumption for variables that are formal procedure parameters is that they are both potentially a definition and a use of that variable. We discussed the methods for maintaining SSA properties in §3.1.4. When constructing FUD chains, this approach implies that each parameter will have both a use-def *link* and a def-def *link*. Although not obvious, both *links* may not point to the same tuple. Consider an example similar to one used previously:

```
S1:  X = 1
S2:  call sub2( X + 2, X, X )
S3:  X = Z
S4:  Y = X
```

If we follow FUD chains from the use of `X` in S_4, we will first reach the definition of `X` at S_3. Now, where does its def-def *link* point to? This problem is the same one we encountered in §3.1.4, and let us tentatively assume we choose a syntactic ordering of the arguments to **sub2**. Then $chain(D_3^X)$ will be the third argument of **sub2**. As we saw

earlier for this third argument, its use-def *link* becomes: $chain(U_2^X) = D_1^X$. However, given our arbitrary ordering for the definitions of the parameters in sub2, the def-def *link* for the third argument will point to the second argument of sub2, which in turn will point to D_1^X.

This situation can impose artificial dependencies between variables used as formal arguments to procedures, so an awareness of the problem is critical.

Scalar Output Dependence An output dependence for a variable occurs when two definitions of a variable must be sequentially ordered. It seems clear that following def-def chains is necessary to detect these dependencies. We present details of a method that detects these dependences in Chapter 6.

Alias Analysis. Due to potential aliases (*may_alias* sets), as opposed to known aliases (*must_alias* sets), a definition of one variable *may* change the value of another variable. Thus, the modification of a variable can become a nonkilling definition of another variable. By following the FUD chain of these potential aliases (starting at a definition's def-def *link* and the links of its alias sets), we hope to add more precision to the analysis of *may_alias* sets. We examine this problem in Chapter 9.

3.2.3 Notes on Implementation

We performed experiments to determine what additional overhead, if any, is incurred by adding additional *links* from every definition to the last downward-exposed definition. In other words, is there an additional time cost for constructing FUD chains instead of just SSA form? In fact, for all the benchmarks we found no extra overhead in constructing FUD chains as opposed to straight SSA form.

Upon reflection, this result is unsurprising, since all tuples are examined on lines 2 - 11 of Algorithm 3.2, and the modification of line 4 for constructing FUD chains (discussed in §3.2.1) simply replaces the *link* field with the current definition of a variable, which is a constant time operation.

3.2.4 Related Work

Factored Static Single Assignment. A recent paper [CCF94] discusses concepts similar to FUD chains, which the authors call Factored SSA (or FSSA). FSSA uses def-use chains to link up definitions to uses, while we utilize use-def links, as noted in §3.1.5. They also utilize def-def chains, but where each definition points to the next (upward-exposed) definition of that symbol, as opposed to the unique reaching definition of FUD chains (downward-exposed). However, the authors discuss a constant propagation algorithm that seems to imply that use-def links of some sort must be available for a backward traversal of the CFG. We explicitly describe how to perform constant propagation using FUD chains in the next chapter.

Also, details of FSSA construction is not provided, so it is a little unclear exactly how the semantics of their method is to be guaranteed. In particular, although they refer to the original SSA papers [CFR+91, CFR+89] (which several of them coauthored) that describe how to construct SSA form (similar to Algorithms 3.1 and 3.2), no mention is made of how to provide the def-def links. Additionally, while using def-def links to process procedure parameters and aliases is mentioned, no details are provided on how to maintain the SSA semantics in these cases.

A very nice and useful result is provided by the authors, however. They demonstrate that linearizing reaching definitions with ϕ-functions and def-def links (in the case of nonkilling definitions) is equivalent to a full-blown set of reaching definitions (which may be of quadratic size).

Sparse Evaluation Graphs. An alternate method with which to extract data-flow information from a CFG is provided by Sparse Evaluation Graphs (SEG) [CCF91]. Given a data-flow problem to solve on the CFG, the SEG method selects, for each variable, the set of nodes (N_{SG}) in the CFG that have nonidentity transfer functions. It then computes which nodes are in the iterated dominance frontier of N_{SG}, setting $N_{SG} = N_{SG} \bigcup_{X \in SG} DF^+(X)$. All nodes in the original flow graph are mapped to those nodes in N_{SG} that have the same data-flow solution.

Thus, sparsity is obtained by solving a data-flow problem on a smaller set of nodes

(either nodes that have nonidentity transfer function or nodes that may merge information) than those in the original CFG. This procedure works for both forward and backward problems. For backward problems, the RCFG is used.

SEGs differ from SSA form (and, more generally, Reference Chaining as described in Chapter 5) in two ways. First, a separate SEG needs to be built for each variable, whereas we encapsulate the information for all variables within the original CFG, but link references together in a sparse manner using chains. Although chaining functions are placed in the CFG on a per variable basis, the linking is done in one pass through the CFG, as we have seen in this chapter. Second, with Reference Chaining the solution to a problem is only computed at desired nodes, while SEGs map a solution for each variable back to all the original nodes in the CFG.

3.3 Demand-Driven Analysis

In this section we look at one of the main methods that we have employed to solve monotonic data-flow problems. In this case, abstraction merges with implementation due to the manner in which FUD chain *links* mesh with data-flow operator-operand links. That is, we are able to traverse the entire data-flow graph (all tuples, plus *left*, *right*, and FUD chain *links*) within a single framework.

Traditional iterative data-flow analysis, such as constant propagation, require multiple passes over all nodes in the CFG, recomputing information as needed for all tuples within each node until a fixed point is reached [ASU86, FL88, Ken81]. For a forward data-flow problem, after any node in the CFG is processed its successors are next examined. This behavior is due to information flowing from predecessors to successors.

In reality, however, tuples can be connected within the data-flow graph in a manner that spans basic block nodes. This phenomenon occurs when using FUD chains with the *link* field, which creates a sparse graph with respect to all variables. Often, when attempting to classify a tuple (such as the type of *induction* variable or whether a variable is a constant), its classification depends on that of its data-flow predecessors. One way to propagate information from data-flow predecessors to successors is the iterative approach

on the CFG, as discussed above. It certainly works, but may be slow since it propagates information through nodes that have no effect on that information.

Our approach is to classify each tuple in terms of its predecessors in the data-flow graph. In some sense, the CFG is of no consequence. We simply examine a tuple at any point, and make a recursive call on its data-flow predecessors when we need that information to classify the current tuple. For example, when classifying a binary operator such as '+', we typically want to examine its two operands, *left* and *right*. If one of these operands is a fetch of a variable, we follow the *link* of the fetch. In this way we follow FUD chain *links*, and we now see a major advantage of use-def *links* as a means to maintain SSA semantics. The key is that FUD chain *links* point in the same direction as operator-operand links, such as we saw in Figure 2.2. Thus, they match the demand-driven style of data-flow analysis that we would like to exploit.

We do make one pass through all basic block nodes, and look at all tuples within each node, so that we can be sure that all tuples have been processed. However, the blocks can be examined in any order, since the sole purpose is to classify each tuple. In fact, any other mechanism, such as traversing the data structure that creates tuples in the compiler front-end, would work equally well. The basic procedure, where *classify*() is the method for a particular data-flow problem, is given as Algorithm 3.3.

Algorithm 3.3 will solve all monotonic data-flow problems in one pass of the tuples, in the absence of cycles in the data-flow graph. It is naive, however, to think that no cycles will occur. Sometimes, such as the way we handle constant propagation in Chapter 4, a separate solver has already dealt with cycles. In the more general case, however, a more sophisticated method is needed. We use Tarjan's well-known algorithm [Tar72] in this case, which detects maximal strongly connected components with a linear depth-first search. It serves both as a means of cycle identification and as a mechanism to perform the traversal over all tuples. When a cycle is detected, separate solvers are typically invoked to classify all its elements [GSW]. In fact, a cycle solver can classify all its members by visiting each only once when the problem is *uniformly monotonic*. (A data-flow framework is defined as uniformly monotonic when the meet operator ($\sqcap$) is commutative and associative, and for each node N in the flow graph there is some value

Given: Data-flow graph initialized with lines 1 - 3
Do: Execute lines 4 - 8
Result: Data-flow solution at each tuple

```
 1: forall tuples t do
 2:         t.visit = false
 3: endfor

 4: forall tuples t do
 5:         if t.visit = false then
 6:             Demand( t )
 7:         endif
 8: endfor

 9: Demand ( t )
10:         t.visit = true

11:         if link( t ) ≠ ∅ & link( t ).visit = false
12:             Demand ( link( t ) )
13:         endif

14:         if left( t ) ≠ ∅ & left( t ).visit = false
15:             Demand ( left( t ) )
16:         endif

17:         if right( t ) ≠ ∅ & right( t ).visit = false
18:             Demand ( right( t ) )
19:         endif

20:         classify( t )
21: end Demand
```

Algorithm 3.3 Basic method for solving data-flow problems on demand

in the abstract domain V_N such that the transfer function is $f_N(x) = x \sqcap V_N$.)

Examples of this demand-driven analysis style include induction variable detection [GSW], constant propagation (covered in detail in Chapter 4), and bindings for global references [WGS94] (which uses the call binding graph instead of the CFG).

Chapter 4

Demand-Driven Constant Propagation

4.1 Introduction

Constant propagation is a static technique employed by the compiler to determine values of variables that do not change regardless of the program path taken. In fact, it is a generalization of *constant folding* [TS85], the deduction at compile time that the value of an expression is constant. Constant propagation is frequently used preliminary to other optimizations. The results can often be propagated to other expressions, enabling further applications of the technique. This recursive nature of the data-flow problem suggests using a *demand-driven* method instead of the more usual iterative techniques.

In the following example, the compiler substitutes the value of 5 for `x` in S_1, which is a canonical instance of constant folding. Since the value of `x` is now constant, the compiler can propagate this value into S_2, which, after applying constant folding once again, results in the determination that `y` is the constant 20. It should be noted that constant propagation for this work focuses on scalar integer values. Propagation of real-valued expressions can be performed, but special care is required since operations on real-valued expressions are often architecturally dependent and rounding methods may be dynamic. The method outlined in this work also allows for arbitrary symbolic expression propagation [GSW].

S_1: `x = 2 + 3`

S_2: `y = 4 * x`

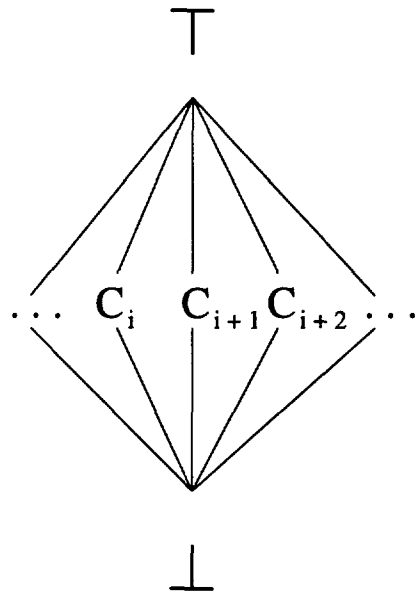


Figure 4.1 Standard constant propagation lattice $\mathcal{L}$

Although constant propagation is an undecidable problem [KU77], it is nonetheless extremely useful and profitable for a number of optimizations. These include dead code elimination [WZ91], array and loop-bound propagation, and procedure integration and inlining [GT93]. Due to these benefits, constant propagation is an integral component of modern commercial optimizing compilers [BCD+92, LFK+93, Muc88].

This chapter will first look at constant propagation within the data-flow framework. We will show how the constant propagation problem fits into the FUD chain structure, and look at the extension of FUD chains to the more interpretable Gated Single Assignment representation required to implement conditional constant propagation. A comparison of our method with other recent sparse techniques is presented, with an evaluation of the strengths and weaknesses of each approach. We also present experimental evidence that shows the number of constants found in scientific benchmark codes as well as the time needed to perform the analysis.

$$
\begin{array}{lcll}
\top & \sqcap & \text{any} & = \text{any} \\
\bot & \sqcap & \text{any} & = \bot \\
\text{constant}_i & \sqcap & \text{constant}_j & = \begin{cases} \text{constant}_i & \text{if } i = j \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Table 4.1 Rules for meet ($\sqcap$) operator.

4.2 Background for Constant Propagation

4.2.1 Iterative Solutions

As a data-flow problem, constant propagation can be cast into the monotone data-flow framework given in Chapter 2. Here, W, the set of information to be propagated, represents all the possible mappings from the set of variables in a program to the values that can be assigned to any variable. The possible values that can be assumed by any variable are represented by the semi-lattice $\mathcal{L}$ (shown in Figure 4.1), introduced by Kildall [Kil73] and standard for many constant propagation methods [CCKT86, GT93, WZ91]. This lattice is three-tiered, with distinguished $\underline{1}$ ($\top$) and $\underline{0}$ ($\bot$) elements. In the most general terms, each element of W is a set that contains all possible sets of assignments from $\mathcal{L}$ to the variables in a program. Hence, the size of W is $2^{|V \times R|}$, where V is the number of variables and R is the number of possible constants (both potentially infinite).[†] When comparing two lattice element values, the meet operator ($\sqcap$) is applied, as given in Table 4.1. Since $\preceq$ is a partial order, we notice that with this simple lattice $\mathbf{a} \prec \mathbf{b}$ is true only when $\mathbf{a}$ is $\bot$ or $\mathbf{b}$ is $\top$. The set $\mathcal{F}$ of transfer functions represent the effect each basic block node has on the information that enters at the beginning of each node. A complete description of these functions for solving the constant propagation problem using Equations 2.1 would be quite complicated, and depends on the type of operations that can be performed on W, such as assignment, copy, etc.; a "basis" for these functions is provided elsewhere [ASU86, Section 10.11].

[†]In this general case $V \times R$ is a cross-product, with $2^{|V \times R|}$ being the size of the powerset of this cross-product.

The reason that we do not go into the details here is that solving constant propagation using a standard iterative technique is quite inefficient. The general method of Equations 2.1 carries around the information of all variables within a program at both the beginning and end of each node N (*in(N)* and *out(N)*) in a CFG. Consider Figure 4.2. If information on both x and y is retained and passed through each node, all 8 nodes (with *in* and *out* sets for each) would need to maintain information on these two variables. However, half the nodes for each variable have identity transfer functions. We will instead use the method given in Chapter 3, FUD chains, which provides a sparse graph upon which to perform analysis, as well as linearization of the chains. In this example, if we wanted to evaluate z, we could just follow a total of 6 links as opposed to propagating information to nodes that are not involved in the data-flow solution. We conclude that the complete iterative solution is overly consumptive of space (and is clearly no faster) when compared to a sparse graph solution.

4.2.2 Relative Precision of Solutions

Although constant propagation is a monotonic data-flow problem, the solution obtained may not be as precise as other data-flow problems. To understand why, we define what is known as the *meet-over-paths* (*mop*) solution. Given a node M, let P_M represent the set of all paths from *Entry* to M. Then, *mop(M)* is defined as:

$$mop(M) = \bigsqcap_{\forall p \in P_M} f_p(\top),$$

where f_p is the composition of the transfer functions along each path p.[†]

One obvious problem with the *mop* solution is that when cycles are involved there are an infinite number of paths from *Entry* to some of the basic blocks in the CFG. The *true* data-flow solution is the meet over all $f_p(\top)$ of paths that actually are taken in some program execution. Since the set of paths in the *true* solution is a subset of all possible paths, the meet operator assures us that the *mop* solution is lower in a lattice theoretic sense than the *true* solution, *i.e.*, $mop \preceq true$. Thus, *mop* is a conservative approximation

[†]In fact, according to the data-flow problem, it may be right to compute $f_p(\bot)$.

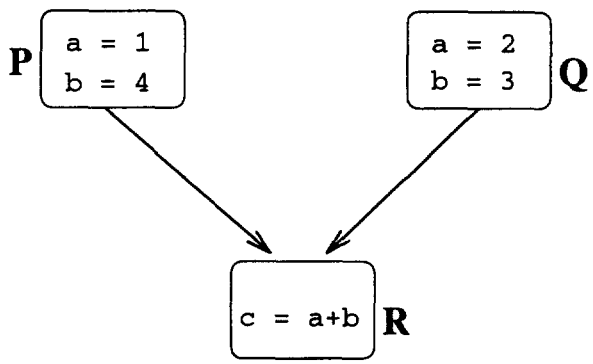


Figure 4.3 Example showing that constant propagation is not distributive

to the *true* solution, and is referred to as *safe*.[†] How does the *mfp* (maximum fixed point, discussed in §2.3.1) solution, which we know how to compute using Equations 2.1, compare to the *mop* solution? If the monotone data-flow problem is *distributive*, then it has been established that $mfp = mop$, where a distributive system obeys this property:

$$(\forall f \in \mathcal{F})(\forall x, y \in \mathcal{L})[f(x \sqcap y) = f(x) \sqcap f(y)] \tag{4.1}$$

(It is not hard to show that distributivity implies monotonicity [Hec77].)

However, constant propagation is not distributive, which we show by counterexample. In Figure 4.3, where the meet function obeys Table 4.1, the transfer function f_R for the assignment to c takes a and b from *in(R)* and computes their sum. Here, x and y in Equation 4.1 are sets of values for nodes P and Q, respectively. Thus, $x = \{(a,1),(b,4)\}$ and $y = \{(a,2),(b,3)\}$. (Technically, x and y contain all variables in a program, including c, but for convenience we omit them here.) We now make the following calculations:

$$\begin{aligned} f_R(x \sqcap y) &= \{(a,1),(b,4)\} \sqcap \{(a,2),(b,3)\} \\ &= (1 \sqcap 2) + (4 \sqcap 3) \\ &= \bot + \bot \\ &= \bot \end{aligned}$$

[†]A solution *in(B)* is a *safe solution* if $in(B) \preceq mop(B)$ for all nodes B [GW76].

$$\begin{aligned} f_R(x) \sqcap f_R(y) &= [(a,1)+(b,4)] \sqcap [(a,2)+(b,3)] \\ &= (1+4) \sqcap (2+3) \\ &= 5 \sqcap 5 \\ &= 5 \end{aligned}$$

Since the two computations are not identical, we have shown that constant propagation is not distributive. Essentially, nondistributivity for constant propagation means that the data-flow problem cannot "remember" the resulting information after applying the transfer function. Here, the sum is constant, even though neither of its addends is constant. This problem shows that confluence operators (meet for constant propagation) can lose precision, and it follows that in a data-flow framework which is not distributive $mfp \preceq mop$. In fact, the *mop* solution to any nondistributive, monotone data-flow problem is undecidable [Hec77].

4.2.3 Optimistic vs. Pessimistic Solvers

Lattice $\mathcal{L}$ is shown in Figure 4.1. The constant propagation algorithms we shall concentrate on have the lattice value for each symbol initialized to $\top$, which indicates that it has an as yet undetermined value. After analysis is complete, all symbols will have lattice value equal to $\bot$ (it cannot be determined to be constant), a constant value, or $\top$ (unexecutable code). We note that values can only move down in the lattice, due to the meet operator.

By initializing lattice values to $\top$, an *optimistic* approach is taken, which assumes all symbols can be determined to be constant until proven otherwise. A *pessimistic* approach, on the other hand, initializes all variables as $\bot$ and never propagates a constant until it can be determined to actually be constant. Thus, if analysis of an iterative pessimistic method is halted prematurely, all variables labeled constant are provably constant. With an optimistic approach, however, a value may be propagated as constant but later get lowered to $\bot$. Thus, halting an iterative optimistic solver before completion may yield incorrect results.

A pessimistic solver will not propagate an expression as being constant until each

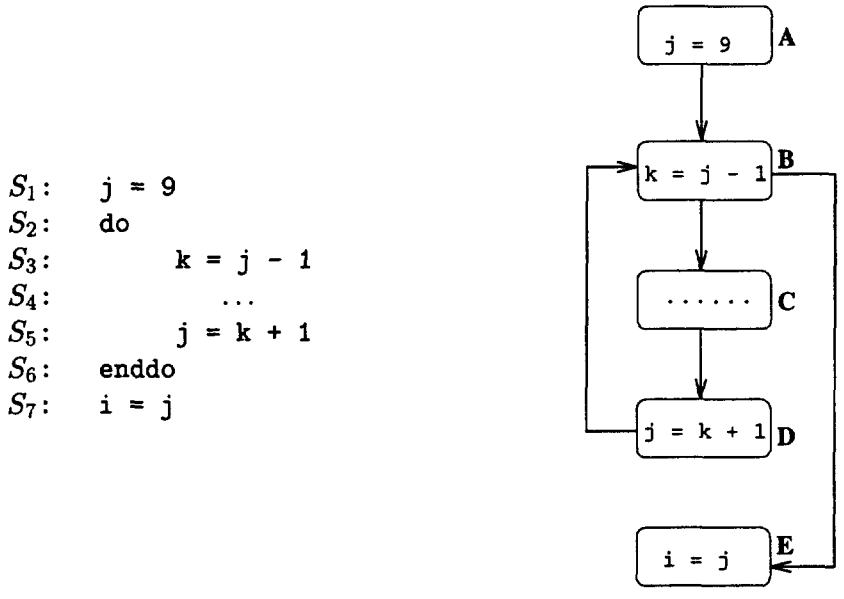


Figure 4.4 Constants can be missed with pessimistic solvers

operand has been classified as constant. This requirement can result in missing constants, as we see in the program fragment along with its CFG in Figure 4.4. In this case, the lattice value of all variables starts off as $\perp$. First, j at S_1 is determined to be the constant 9. Next, when processing k at S_3 (basic block B), its classification is dependent upon the possible values of j, which come from S_1 and S_5 (basic blocks A and D). Since all lattice values are initialized to $\perp$, $9 \sqcap \perp = \perp$, and the lattice value of k at S_3 stays at $\perp$. When processing S_5, the value of k comes from S_3, which we have seen is $\perp$, so j at S_5 stays at $\perp$. Finally, to process i at S_7, the lattice value of j comes from the top of B, which has remained at $\perp$ after applying the meet function. Thus, though all program paths result in the constant 9 for i in S_7 (which means, of course, that the *mop* solution for i at E is 9), we see that pessimistic solvers can reach a nonoptimal fixed point.

An iterative optimistic solver would begin the same way at S_1, but each tuple is initialized to $\top$. Now when processing S_3, j is $9 \sqcap \top = 9$. Thus, k at S_3 has its lattice value lowered to 8. When S_5 is now processed, the lattice value of j gets set to 9. For this example, we have reached a fixed point for the cycle B-C-D, and further iteration will not change any of the lattice values contained within these nodes. Finally, S_7 in node E inherits the lattice value of j from B, which is $9 \sqcap 9 = 9$. In this way, optimistic

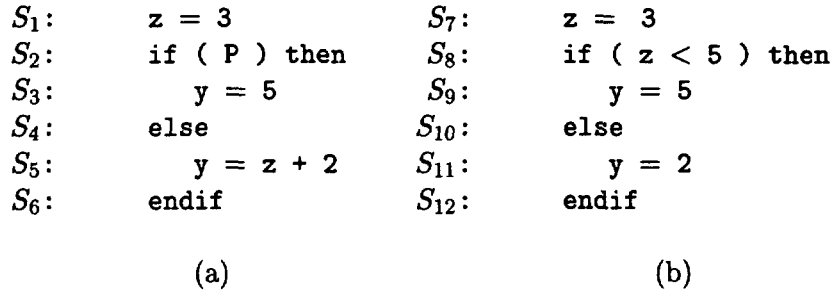


Figure 4.5 Constant propagation with (a) simple, and (b) conditional, constants

solvers find a larger class of constants than pessimistic methods.

4.3 Using FUD Chains for Simple Constants

We need to make one change to the tuple structure when performing constant propagation. A new field, *lattice*, is added to each tuple. The lattice element of each tuple can assume $\top$, $\bot$, or any of the constant values of Figure 4.1. The lattice element is initialized to $\top$ for optimistic solvers, such as the ones we shall focus upon.

4.3.1 Constants Within the FUD Chain Framework

We first show how to implement simple constant propagation within our framework. The distinction between *simple* (all paths) constants and *conditional* constants can be seen in Figure 4.5. The simple value of y is determined to be constant only if both branches which merge at S_6 are constant with identical value, as is the case in (a). A conditional constant, however, may be identified when a predicate which controls branching can be determined to be constant, since in that case only one of the branches will be executed, allowing not only y to be recognized as constant in (b), but also identifying the other path to be dead code.

The first constant propagation algorithm we present detects simple constants in a demand-driven manner. Algorithm 4.1 efficiently propagates simple constants in the SSA data-flow graph by demanding the lattice value from the unique definition point of each use. We essentially use Algorithm 3.3 where the *classify()* routine assigns constant values

to the lattice when appropriate and takes the meet of lattice values at confluence points. We visit all CFG nodes, examining each of its tuples, calling *Propagate*() recursively on any unvisited left or right tuples. Expressions are evaluated by calling *Propagate*() on all references with a nonnull *link* field. When a ϕ-function is encountered, recursive calls to the arguments are made, followed by taking the meet of those arguments. In the case of data-flow cycles, characterized by ϕ-functions at loop-header nodes, $\perp$ is returned.

We look at a simple example, with Figure 4.5(a) transformed into SSA form and slightly augmented. The resulting code and its CFG is shown in Figure 4.6. If we first call *Propagate(store* x *)* at S_8, recursive calls are made to *Propagate(fetch* y *)* at S_8 and *Propagate(merge* y *)* at S_7. Since the *merge* tuple is a ϕ-function, recursive calls are made to both its arguments, which will both eventually return 5. Applying the meet rules of Table 4.1, x at S_8 will have its lattice value assigned the constant 5.

Although we began with basic block F, we could have started with any node in the CFG. For example, had we started with node D, the store to y would get lattice value 5 after a recursive call is made to the assignment of x in A. If node E was processed next, the *left link* would get the constant 5 from C, while the *right link* would return the already computed value from D. Finally, when F is visited, only one call to *Propagate*() is made, since y has already been visited and classified as the constant 5 in E. The order of visitation of basic blocks never affects the result of the algorithm or its time complexity, since all tuples and *links* are visited exactly once.

4.3.2 Discussion of Algorithm 4.1

We discuss several important issues relating to Algorithm 4.1.

Memoization of lattice values. By storing the lattice value at each tuple, we insure that recomputation of lattice values is never needed. This property is valid since the demand-driven approach guarantees that each tuple will be visited exactly once.

Given: Data-flow graph, initialized with lines 1 - 4
Do: Execute lines 5 - 7
Result: Simple constants assigned to lattice elements

```
1:      forall tuples t do
2:          lattice( t ) ← ⊤
3:          t.visited ← false
4:      endfor

5:      Visit all basic blocks B in the program
6:          Visit all tuples t within B
7:              if t.visited = false then Propagate( t )

8:      Propagate ( t )
9:          t.visited ← true
10:         if link( t ) ≠ ∅ then
11:             if link( t ).visited = false then Propagate( link( t ) )
12:             lattice( t ) ← lattice( t ) ⊓ lattice( link( t ) )
13:         endif
14:         if left( t ).visited = false then Propagate( left( t ) )
15:         if right( t ).visited = false then Propagate( right( t ) )
16:         case on type of t
17:             constant C: lattice( t ) ← C
18:             arithmetic operation:
19:                 if all operands have constant lattice value then
20:                     lattice( t ) ← arithmetic result of
21:                                    lattice values of operands
22:                 else
23:                     lattice( t ) ← ⊥
24:                 endif
25:             store: lattice( t ) ← lattice( RHS )
26:             merge (φ-function):
27:                 if loop-header φ then
28:                     lattice( t ) ← ⊥
29:                 else
30:                     lattice( t ) ← ⊓ of φ-arguments of t
31:                 endif
32:             default: lattice( t ) ← ⊥
33:         end case
34:     end Propagate
```

Algorithm 4.1 Demand-driven propagation of simple constants

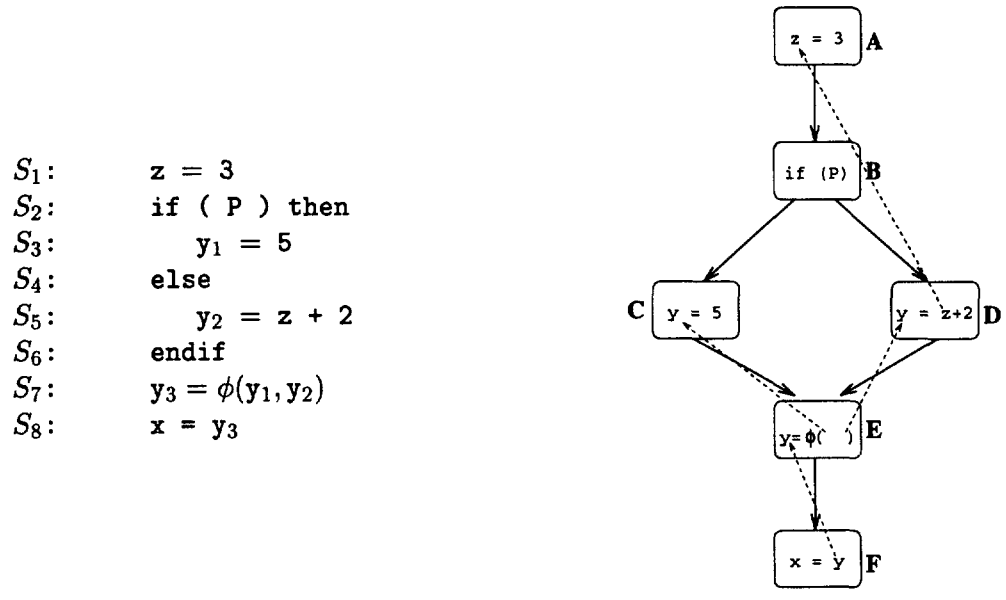


Figure 4.6 Example of demand-driven constant propagation

Only scalar integral constants are found. Since at this point in our implementation we only look for scalar constants, we have been a little imprecise on how Algorithm 4.1 is applied. If tuple t does not operate on a scalar integer object, we need consider it no further for most of the algorithms in this chapter (it effectively gets set to $\perp$).

This method is not an iterative solver. It is a recursive demand-driven technique that will completely solve the graph in the absence of cycles. The order in which basic blocks are visited is not important.

How multiple definitions are merged. When at a confluence node, we take the meet of the demanded classification of the ϕ-arguments. By Table 4.1, this will result in a constant if and only if all ϕ-arguments are constant and identical.

Class of constants found. This is an optimistic solver since all tuples are initialized to $\top$. We find the same class of simple constants as other nonconditional solvers, such as Kildall [Kil73] and Reif and Lewis [RL77], in the absence of data-flow cycles. In

the presence of data-flow cycles (due to loops in the CFG), the solver will fail to classify constant valued tuples, even if it remains constant throughout the loop. In a case such as Figure 4.4, even though the *mop* solution for i at S_7 is 9, lines 27 - 28 of Algorithm 4.1 return $\perp$. We can improve on this simple demand-driven constant propagation algorithm by identifying the data-flow cycle and calling a separate solver on that component. In fact, Algorithm 4.1 uses a simple depth-first search when traversing the data-flow graph comprised of tuples. Instead, we could use Tarjan's algorithm [Tar72], a depth-first search method with additional functionality, which identifies maximal strongly connected components of a graph with the same time complexity, $O(V + E)$, as the simple depth-first search method presented here. A separate solver for data-flow cycles may be the method of choice since it can also be used to classify induction variables for natural loops in general. A technique has been outlined in detail elsewhere [GSW] using a demand-driven SSA form. Details on how to interface between the induction variable loop solver and constant propagation are provided in §4.4.

Expression Evaluation. When requesting the lattice value of a *store* operation, line 25 of Algorithm 4.1 states that the lattice value of the *store* inherits the lattice value of its right hand side (RHS). To evaluate the expression, each RHS operand is classified just once, either immediately or after recursive calls on its *left, right,* or *link* fields. Hence, each expression is evaluated once, since the node containing the expression will only be evaluated after all referenced variables are classified. This feature of demand-driven classification will become important when we look at other sparse methods of performing constant propagation.

Complexity. The asymptotic complexity is proportional to the size of the data-flow graph, since it requires each link, left and right edges to be examined once. Hence its time complexity is $O(V + E)$ within the data-flow graph.

```
x = 0                 x0 = 0                x0 = 0
y = 0                 y0 = 0                y0 = 0
z = 0                 z0 = 0                z0 = 0
if ( P ) then         if ( P ) then         if ( P ) then
   y = y + 1             y1 = y0 + 1           y1 = y0 + 1
endif                 endif                 endif
                      y2 = φ ( y0, y1 )     y2 = γ ( P, true→ y1, false→ y0 )
x = y                 x1 = y2               x1 = y2
z = 2 * y - 1         z1 = 2 * y2 - 1       z1 = 2 * y2 - 1
```

(a) (b) (c)

Figure 4.7 Program in (a) normal form, (b) SSA form, and (c) GSA form

4.4 Constants Within Conditionals and Loops

4.4.1 Extending the Interpretability of ϕ-Functions

When demanding the classification of a variable at a confluence node, we take the meet of the demanded classification of its ϕ-arguments, as noted in the last section. However, if only one of the branches will, in fact, be taken, we would like to only propagate the value along that path.

With conditional constant propagation, if a symbol demands the value from a confluence node, we want to process the predicate that determines the path to follow. Examine Figure 4.7(b). When attempting to classify x_1, the value is demanded from the use-def SSA link of y_2, which points to the ϕ-function. However, a ϕ-function is not interpretable [BMO90]. Thus, we have no information about which path may or may not be taken. Since the predicate P in our example determines the path taken, if P is constant, we can determine which argument of the ϕ-function to evaluate. If P is not constant, the best we can do is to take the meet of the ϕ-arguments.

Augmentation of the ϕ-function is needed to include this additional information. We extend the SSA form to *Gated Single Assignment* form (GSA), introduced by Ballance *et al.* [BMO90], which allows us to evaluate conditionals based upon their predicates. Figure 4.7 shows a simple program converted to GSA form. Briefly, ϕ-functions are

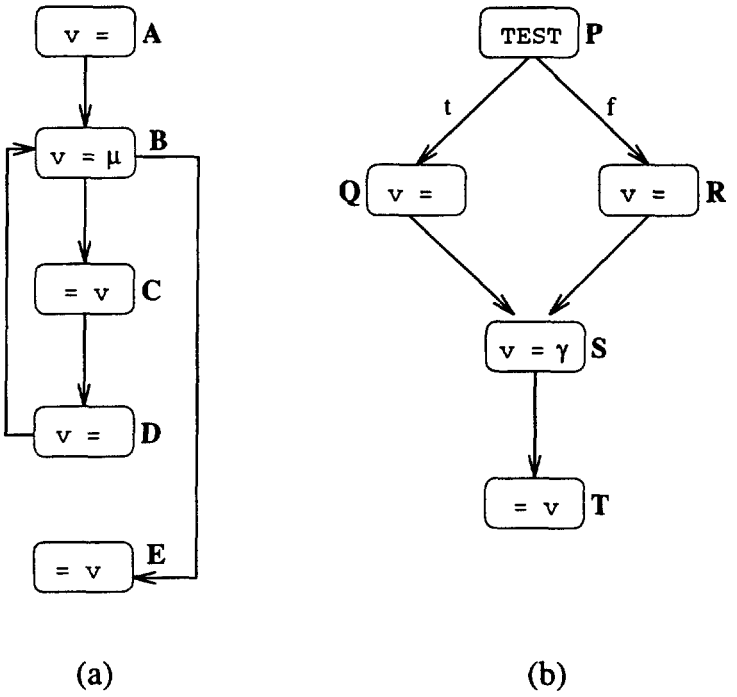


Figure 4.8 The two types of ϕ-functions: (a) μ, and (b) γ

reclassified into two types: μ- and γ-functions. All ϕ-functions contained within loop-header nodes are renamed μ-functions, while most other ϕ-functions are converted to γ-functions. Additionally, a new operator, the η-function, is introduced at loop exit points for each variable defined within the loop. We now provide the details of GSA form and its construction.

4.4.2 GSA Form

The μ- and γ-functions. Although ϕ-functions represent the merging of reaching definitions at confluence points, in truth there are two distinct types of merges in the control flow sense. We illustrate this distinction in Figure 4.8. In (a) the merge of the distinct definitions of v at nodes A and D occurs at node B, which is a loop-header node. The use of v at C has its reaching definition from A on the first iteration of the loop, and from D on all subsequent iterations. We rename ϕ-functions at loop-header basic blocks μ-functions. In (b), the definitions of v at Q and R meet at S, which is a confluence

node created by branch node P. The path taken from P is determined by some predicate within the TEST condition. The merge operator at S needs to encapsulate the predicate that determines the branch at P as well as the possible reaching definitions from Q and R. We transform ϕ-functions at these confluence points into γ-functions to reflect this additional information.

There is typically a predicate of some sort also at the loop-header that determines whether control is passed to the loop body or not. In the original work by Ballance *et al.* the μ-function was of the form $\mu = (\mathtt{P},\ v^{init}, v^{iter})$, where P is the predicate to determine whether the loop will trip, and v^{init} and v^{iter} are the values for v entering the loop and after at least one iteration, respectively. For the applications in this work, we do not need the additional overhead in determining the trip condition of the loop. Although we want a more interpretable SSA form, we do not need a complete dataflow executable model, as was the original intent of Ballance *et al.* The method of loop detection outlined in Chapter 2 (with details given by Algorithm 2.3) does not distinguish between loops created by implicit control constructs (such as IF-GOTO style) and those provided explicitly as language features (such as a traditional DO loop). The method of encapsulating the predicate condition would add complexity that we would not utilize. Thus, we distinguish ϕ-functions that occur at loop-header nodes by renaming them μ-functions, with the form for variable v being $v = \mu(v^{init}, v^{iter})$. We easily accomplish this transformation in Algorithm 3.1 by expanding lines 16 - 19 to become:

16a: **if** *Q is the header of Q.loop* **then**
16b: *place $V = \mu(\ V_{init}, V_{iter})$ at the beginning of basic block Q*
17 : **else**
18a: *$i \leftarrow$ number of predecessors of Q*
18b: *place $V = \phi(\ V_1, V_2, ..., V_i\ )$ at the beginning of basic block*
18c: *Q, where V_j corresponds to the j^{th} predecessor of Q*
19 : **endif**

We know that a μ-function will have precisely two predecessors due to the preheader and postbody nodes added to the CFG in the preliminary analysis discussed in Chapter 2.

While μ-functions are identified during the placement phase of FUD chain construction, we must translate the remaining ϕ-functions into γ-functions separately. This is due to the predicate that controls a γ-function (which we left out in μ-functions), and because γ-functions can become nested, essentially relying on a chain of predicates to determine the control flow path taken to reach a confluence node. In its most basic form, the γ-function $\mathtt{v} = \gamma(\mathtt{P}, \mathit{true} \rightarrow \mathtt{v}_1, \mathit{false} \rightarrow \mathtt{v}_2)$ means *if* P *then* $\mathtt{v}{=}\mathtt{v}_1$ *else* $\mathtt{v}{=}\mathtt{v}_2$. In this simple form, the γ-function represents an `if-then-else` construct, but it is also extended to include more complex branch conditions, such as `case` statements.

The η-function. One further operator is added to the GSA form – the η-function. The purpose of η-functions is to capture or summarize the value of each variable at exit points of a loop. Effectively, an η-function *gates* the effect of the loop from code outside the loop. The original work by Ballance *et al.* included a loop predicate that indicated under what conditions the value being gated would be used. As with the μ-function, we will not need the predicate information. Our methods use the η-function as a placeholder to summarize the exit value for each variable when the loop terminates. We discuss in §4.4.5 how the η-function interfaces with FUD chains and is used by a separate loop solver.

In order to provide η-functions, we augment the CFG for each edge exiting a loop with a *postexit* node placed outside the loop between the source of the exit edge (within the loop body) and the target (outside the loop). An η-function is then inserted into postexit nodes for each variable defined within the loop. After completing the loop identification phase of a CFG, such as given by Algorithm 2.3, a separate pass can be made through the loop structure to add the postexit nodes. The η-functions are easily inserted when creating the variable modification list, $A(V)$, as preparation for the FUD chain construction algorithms. Each η-function has a single associated *η-argument* as a placeholder for the summary information at that loop exit point.

4.4.3 Converting ϕ-Functions Into γ-Functions

Flow Graph Reducibility. We start by defining a *reducible* flow graph. Informally, a general flow graph is *reducible* if there is only one entry point for every loop. Essentially, reducibility of a flow graph guarantees that control cannot "jump" into the middle of a loop from outside the loop. There are numerous equivalent definitions of flow graph reducibility, such as those based upon *interval analysis* (where the intervals are often referred to as Allen-Cocke intervals) [AC76] or *T1-T2 analysis*, in which a flow graph graph is *collapsed* via two transformations. A flow graph is considered *collapsible* if successive applications of T1 (removing self-cycles) and T2 (turning a node with a unique predecessor into one node) result in a single node. Collapsibility has been proven equivalent to reducibility [Hec77, HU72].

Since we find natural loops (using Algorithm 2.3), and since for γ-function construction we will need a topological sort of the CFG, we use a third method of determining flow graph reducibility. Algorithm 4.2 performs a depth-first search of a CFG, pushing nodes onto a stack during their initial visit of the search.

This algorithm, using standard depth-first techniques, visits each node in the CFG exactly once (if it is a reducible graph), yielding an $O(V + E)$ time bound. Since it recursively is called on successors that have not been visited, a node is only pushed onto stack S (line 17) when all nodes in the same maximal strongly connected component (SCC) have been visited, a property used by Tarjan to classify all nodes in a directed graph into SCCs [Tar72].

Using Algorithm 4.2, we identify a flow graph as reducible if all the edges of the CFG can be divided into two classes [ASU86]:

1. Forward edges of the CFG. These are identified on line 12 in Algorithm 4.2, since the target node of a forward edge in a topological sort will not yet have been visited.

2. Back edges in which the head dominates the tail. These are precisely the same edges identified in Algorithm 2.3 as determining a natural loop. Line 9 of Algorithm 4.2 detects nodes that are the target of back edges (they have already been visited),

Given: CFG and empty stack S
Do: Initialize with lines 1 - 5.
call *TopSort(Entry)*
Result: Topological sort of nodes determined by forward edges if graph reducible

```
 1: Reducible ← true
 2: forall nodes V do
 3:         V.visited ← false
 4:         V.pushed ← false
 5: endfor

 6: TopSort( basic block bb )
 7:         bb.visited ← true
 8:         forall successors succ of bb do
 9:             if succ.visited = true & succ.pushed = false & succ dom bb then
10:                 Reducible ← false
11:                 Exit TopSort
12:             else if succ.visited = false then
13:                 TopSort( succ )
14:             endif
15:         endfor
16:         bb.pushed ← true
17:         push bb onto S
18: end TopSort
```

Algorithm 4.2 Constructing a topological sort of nodes and detecting reducibility

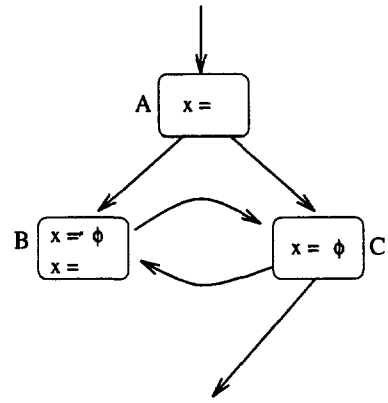


Figure 4.9 Irreducible graph that has a cycle with multiple entry points

are in the same SCC as the source node of that edge (their *pushed* field is *false*), but are headers of natural loops (they dominate the source node). Thus, these edges are ignored altogether by Algorithm 4.2.

If any edge fails to fall into one of the two categories above, the graph is classified as irreducible on line 10, and we will not be able to transform FUD chains for that graph into GSA form. However, if the graph is reducible, all its nodes are pushed onto stack S. Popping S provides a topological sort of the CFG nodes ignoring backedges.

The canonical irreducible flowgraph is shown in Figure 4.9. The cycle containing nodes B and C has two entry points from node A. If *TopSort* is called on this graph, it will identify the multiple entry loop when *TopSort* is called on B (or C) as a successor of C (or B).

Immediate Dominators and Control Dependence. To convert ϕ-functions into γ-functions we rely heavily on the concept of *control dependence*, which was defined and described in §2.1.2. In Figure 4.8(b), Q and R are control dependent upon P, since one branch from P will definitely pass control to Q, while another branch from P may (in this case, definitely will) bypass Q (and similarly for R). In translating Figure 4.8 to GSA form, we essentially want the γ-function at S to look like: $\mathtt{v} = \gamma(\mathtt{TEST}, t \rightarrow \mathtt{v_Q}, f \rightarrow \mathtt{v_R})$. After Algorithms 3.1 and 3.2 have processed the CFG and data-flow graph, S will have a ϕ-function that looks like: $\mathtt{v} = \phi(\mathtt{v_Q}, \mathtt{v_R})$. We start at each predecessor of S, and process

all its *control dependence chains* until the immediate dominator of S is reached, where a control dependence chain (CD_chain) for node $N = N_1$ is defined as a sequence of nodes $N_1, N_2, \ldots, N_i$ such that $N_j \in \text{CD\_Pred}(N_{j-1}), j \in [2 \ldots i]$. If $i > 2$, we say that N is *transitively* control dependent on N_i.

We now prove two properties of CFGs that are important to the algorithms we present. The first shows that for any confluence node, the immediate dominator of that node must be a branch node, and the second demonstrates that every CD_Chain of its predecessors includes the immediate dominator of the confluence node.

Theorem 4.1 *Given any node N in a CFG that is not the header of a natural loop and that has more than one predecessor, idom(N) must have more that one successor.*

Proof:
Let $I = \text{idom}(N)$. Then, for all paths p: $I \xrightarrow{+} Q \xrightarrow{*} N$, I *dom* Q (if not, then there is a path *Entry* $\xrightarrow{+} Q \rightarrow N$ that does not pass through I). Assume I has only one successor, S. There are two possibilities:

1. $S \neq N$. In this case, I *dom* S and S *dom* N. Thus, some node on a path $S \xrightarrow{*} P$, with $P \rightarrow N$, is the immediate dominator of N (recall the definition of immediate dominator from §2.1.2). This conclusion contradicts the assumption that $I = \text{idom}(N)$.

2. $S = N$. Consider predecessors P_1 and P_2 of N. Without loss of generality, let $P_1 \neq I$. Since I *dom* N, I *dom* P_1. Because the only successor of I is N, N *dom* P_1. But in this case the edge $P_1 \rightarrow N$ is a backedge by definition, which makes N the header of a natural loop, a contradiction of hypothesis.

Since the assumption that I has only successor leads to a contradiction in all cases, we conclude that I has more than one successor. ∎

Theorem 4.2 *In a reducible flow graph, if node N, which is not the header of a natural loop, has more than one predecessor, then given any predecessor P of N and any CD_Chain(P), idom(N) ∈ CD_chain(P).*

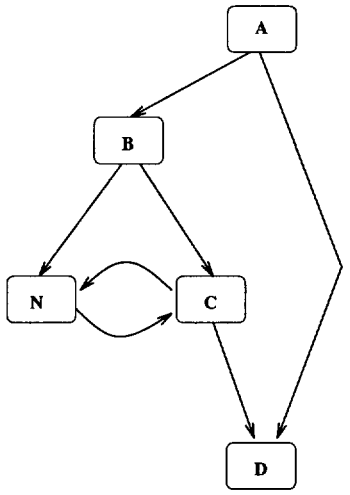


Figure 4.10 Irreducible graph does not have well-defined CD_chains

Proof:

Let I = idom(N). Since I *dom* N, we know that I *dom* P. If P *pdom* I, then all paths from I to N pass through P, and P *dom* N. (None of these paths from I can pass through N before reaching P, otherwise either (*i*) N would be the header of a natural loop with backedge $P \rightarrow N$ or (*ii*) P and N are in a cycle of an irreducible graph. Both possibilities are excluded by hypothesis.) In this case $I = P$, and the claim is trivially true.

Now consider the case where P $\overline{pdom}$ I, and any path $p_1 : \langle I = R_0, R_1, \ldots, R_{n-1}, R_n = P \rangle$. If $P = R_1$, P is control dependent on I (some path from I to *Exit* avoids P, since P $\overline{pdom}$ I). Otherwise, for the largest i on p_1, choose R_i such that P $\overline{pdom}$ R_i. Then P *pdom* R_{i+1}, and P is control dependent of R_i, which means that $R_i \in$ CD_Chain(P). Consider path $p_2 : \langle I = R_0, R_1, \ldots, R_{m-1}, R_m = R_i \rangle$. R_i will be control dependent upon some R_j, $j \in [0 \ldots m\text{-}1]$, for the largest j on p_2 such that R_i $\overline{pdom}$ R_j, which adds R_j to CD_Chain(P). Continued application of this process results in $I \in$ CD_Chain(P). If not, there will be a smallest k such that $R_k \in$ CD_Chain(P) and R_k *pdom* I. Since $R_k \neq I$, R_k *dom* N (as above, no cycles involving R_k and N can exist) and $I \neq$ idom(N). We have reached a contradiction of hypothesis, and thus conclude that $I \in$ CD_Chain(P). ▌

```
   x = 2                      x0=2
   if ( P ) goto 30           if ( P ) goto 30
   if ( Q ) goto 50           if ( Q ) goto 50
   else goto 40               else goto 40
30 x = 3                   30 x1=3
40 y = x                   40 x2 = γ( P, t→x1, f→γ( Q, t→ ⊤, f→ x0))
                              y1=x2
50 continue                50 continue
   (a)                        (b)
```

Figure 4.11 Conditional code that results in nested γ-functions

We illustrate why reducibility is necessary for Theorem 4.2 in Figure 4.10. This graph is irreducible due to the cycle consisting of N and C, which has multiple entry points. N is a confluence node with idom$(N) = B$. We note that B is a branch node as Theorem 4.1 asserts. But for node N, taking C as a predecessor of N, CD_Chain(C) = $\{C\} \cup$ CD_Pred$(C) = \{C, A\}$, which does not include B.

We use Theorems 4.1 and 4.2 to establish the correctness of the algorithms that follow. Essentially, the γ-function provides interpretability of the data-flow graph through control dependence relations. Theorem 4.1 tells us that the starting point for calculating the path that leads to a confluence node is its immediate dominator, while Theorem 4.2 assures us that by following CD_Chains of each predecessor of the confluence node we will always reach that immediate dominator node.

We can only translate reducible CFGs with merge ϕ-functions into γ-functions, since cycles not identified as loops would create infinite nested referencing within a γ-function. No unique identification of back-edges is possible within an irreducible graph, thus making a topological sort of the nodes with respect to their forward edges impossible. In Figure 4.9 neither edge $B \rightarrow C$ or $C \rightarrow B$ can be classified as a backedge since the head does not dominate the tail.

The Conversion Algorithm. We now provide the complete algorithm to translate a program from FUD form into GSA form. A topological sort of the CFG nodes with respect to forward edges is necessary, since γ-functions may refer to other γ-functions, and in this way the references will always be in terms of functions that exist. Since this

translation is only possible with reducible flow graphs, ϕ-functions at loop-header nodes have already been renamed μ-functions. The remaining ϕ-functions at confluence nodes are the result of conditional branches, and Theorem 4.1 tells us to start at the immediate dominator of that confluence node to evaluate predicates that determine which path (or paths if some predicate is not constant) are taken to reach the ϕ-function. Processing CD_chains starting at predecessors will always include the immediate dominator, as Theorem 4.2 showed. The details of translating ϕ-functions to γ-functions are given in Algorithm 4.3, and we describe here the data structures used:

- stack S – A list of all basic blocks in CFG, produced by *TopSort*
- stack T – A list of γ-arguments used by the *Reduce* function
- last_$\phi(*)$ – Previous ϕ-function processed at this basic block (Initialized to NULL)
- current_$\gamma(*)$ – The γ-function under consideration for this basic block (Initialized to NULL)
- labels – Branch values that correspond to outedges from a basic block; if there is only one successor, the branch label is *true*

Discussion of Algorithm 4.3. This algorithm is an adaptation of an earlier version by Havlak [Hav93] that converts ϕ-functions at confluence nodes to γ-functions. Prior to Havlak's algorithm, it was necessary to convert the source program into an SSA-based Program Dependence Graph form [BMO90]. Our algorithm differs from Havlak's in that we do not require a preprocessing pass over the CFG to initialize each of the branch arguments to determine when all paths that reach that branch node have been processed. Havlak's algorithm utilizes a *counter*, which increments during the preprocessing phase and decrements during processing. When *counter* = 0 during processing, all paths to a branch have been examined, and a recursive call to process each of its control predecessors is invoked. We recursively make calls to *Process()* on the first visit to any node per ϕ-function being processed, which we detect using *last_*ϕ. When each call to *Process()*

Given: Reducible CFG
Do: compute *TopSort(Entry)*
    Execute lines 1 - 12
Result: ϕ-functions converted to γ-functions

```
 1:      while S ≠ ∅ do
 2:          B ← pop S
 3:          idom ← immediate dominator of B
 4:          for each φ-function f in B do
 5:              for each CFG predecessor pred of B do
 6:                  lab ← branch label of edge from pred to B
 7:                  param ← φ-argument of f that corresponds to pred
 8:                  Process( f, pred, lab, param )
 9:              endfor
10:              replace f with Reduce( current_γ( idom ) )
11:          endfor
12:      enddo

13:      Process( function f, basic block bb, label lab, link point )
14:          if last_φ( bb ) ≠ f
15:              last_φ( bb ) ← f
16:              if bb has more than 1 successor
17:                  send ← current_γ( bb ) ← Build_Gamma( bb )
18:              else
19:                  current_γ( bb ) ← ∅
20:                  send ← point
21:              endif
22:              if bb ≠ idom then
23:                  for each cp ∈ CD_Pred( bb ) do
24:                      cp_lab ← branch label from cp that executes bb
25:                      Process( f, cp, cp_lab, send )
26:                  endfor
27:              endif
28:          endif
29:          if current_γ( bb ) ≠ ∅
30:              for argument a of current_γ( bb ) ∋ label a = lab do
31:                  link( a ) ← point
32:              endfor
33:          endif
34:      end Process
```

Algorithm 4.3 Converting ϕ-functions to γ-functions

```
35:     Build_Gamma( basic block bg )
36:         γ-predicate ← branch predicate in bg
37:         for each successor succ of bg do
38:             e ← label from bg to succ
39:             add γ-argument with label e and with link( γ-argument ) ← Top
40:         enddo
41:         return γ
42:     end Build_Gamma


43:     Reduce( tuple r )
44:         if r is not a γ-function return r
45:         predicate ← branch operator of γ-function r
46:         if predicate already on stack T
47:             arg ← γ-argument of r whose label matches the branch value of predicate
48:             return reduce( link( arg ) )
49:         endif
50:         for all γ-arguments a of r do
51:             push onto T ( predicate, label of a )
52:             link( a ) ← Reduce( link( a ) )
53:             pop off T( predicate )
54:         enddo
55:         if all γ-arguments a of r have identical link return link( a )
56:         else return r
57:     end Reduce
```

Algorithm 4.3 (cont.) *Build_γ()* and *Reduce()* routines

returns, we fill in arguments of its γ-function, if it exists. If a node has multiple successors, *current_γ* saves the corresponding γ-function for that node. Each γ-argument is initialized to $\top$, so that if not replaced it will not affect the meet operator within the constant propagation algorithms.

An Example of Algorithm 4.3 Examine Figure 4.12. *TopSort* could produce the following ordering of nodes for stack $S : A, B, C, D, E, F, G$. Lines 1 - 12 of Algorithm 4.3 will pop S and do nothing until E appears. Then, with $idom = B$, each predecessor of E will call *Process()* on line 8:

1. *Process(* v_3, C, t, v_2*)*

   Here, $last\_\phi(C) \neq f\ (v_3)$, and since C has only one successor, the following single recursive call (because CD_Pred$(C) = \{B\}$) is made on line 25:

   Process(v_3, B, t, v_2*)*

   Now, *current_γ*(B) is created with its branch predicate set to Q, and two *γ-arguments* initially set to $\top$:

   $$current\_\gamma(B) = \gamma(\mathtt{Q}, t \rightarrow \top, f \rightarrow \top)$$

   Since $B = idom$, line 22 insures us that no further calls are made to *Process()*. At this point, line 31 sets one of the *γ-arguments*:

   $$current\_\gamma(B) = \gamma(\mathtt{Q}, t \rightarrow v_2, f \rightarrow \top)$$

2. *Process(* v_3, D, f, v_1*)*

   This call is analogous to the call above. Note that even though C has a definition of v and D does not, reaching definitions have already been resolved within the ϕ-function at E during the *Chaining* routine of Algorithm 3.2. When the recursive call to *Process()* is made at B, however, $last\_\phi(B) = v_3$, so no γ-function is created, and no further calls to *Process()* are needed. (In this case $B = idom$, so no further calls would be invoked in any case. But if not, a previous invocation of *Process()* for the same ϕ-function would have made any necessary calls.) After filling in the second *γ-argument*, v_3 at E get replaced on line 10 by *current_γ*(B):

```
S1:   v1 = ...                (A)
S2:   if ( P ) then           (A)
S3:        if( Q ) then       (B)
S4:           v2 = ...        (C)
S5:        else
S6:           ...             (D)
S7:        endif
S8:        v3 = φ(v2, v1)     (E)
S9:   else
S10:       v4 = ...           (F)
S11:  endif
S12:  v5 = φ(v3, v4)          (G)
```

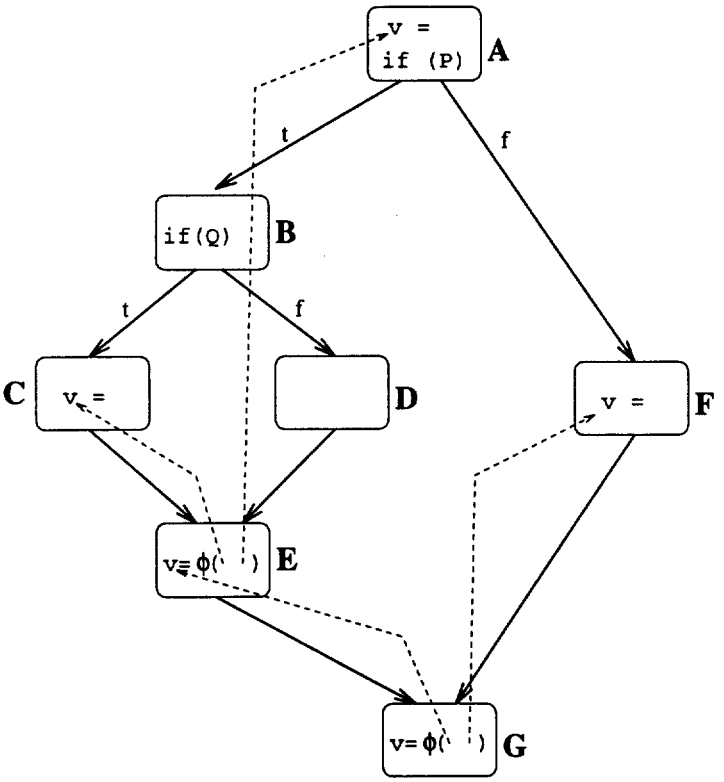


Figure 4.12 How to convert ϕ-functions to γ-functions.

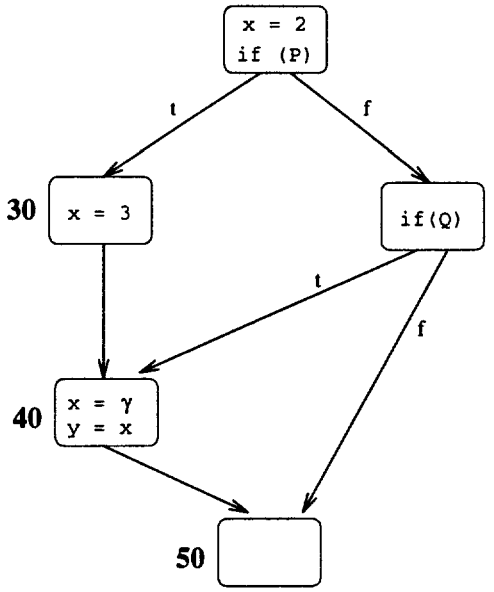


Figure 4.13 Predicates that affect constants in unstructured code

$$v_3 = \gamma(Q, t \rightarrow v_2, f \rightarrow v_1)$$

Popping F creates no work, but popping G off S will create a γ-function at A, the *idom* of G. When *Process()* is called on predecessor E, it will have v_3 as *point*, which is the γ-function built when processing E. After calling *Process()* on F, the other predecessor of G, the completed γ-function at A, which will replace v_5 at G, becomes:

$$v_5 = \gamma(P, t \rightarrow \gamma(Q, t \rightarrow v_2, f \rightarrow v_1), f \rightarrow v_4)$$

Unstructured Code. Multiple levels of conditionals result in nested γ-functions, as we just saw in an example of structured code. Unstructured code can also have this effect, as seen in Figure 4.11. Figure 4.11(b) shows the program in (a) translated into GSA form, with its CFG shown in Figure 4.13. It is quite an interesting example for constant propagation, since if we know the value of predicate P we always know what possible value of x can reach the merge at 40. However, if we do not know P, then the value of predicate Q becomes crucial:

- If Q is true, only x_1 can reach 40.

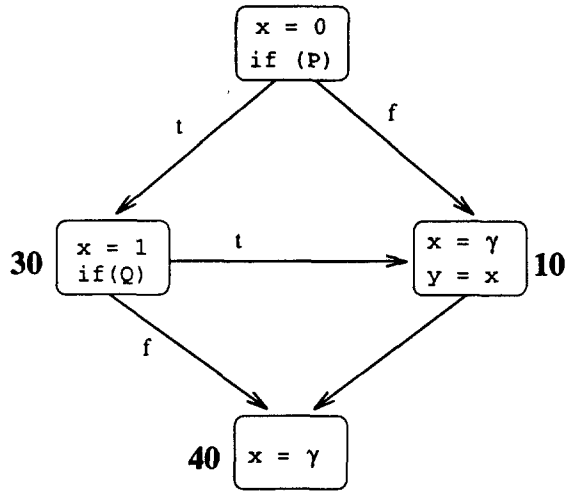


Figure 4.14 Example of how the *Reduce()* routine works

- If `Q` is false, we have no clear information on what value of `x` to propagate.

Reducing γ-functions. When nested γ-functions occur, they can often be reduced. We have noticed empirically that roughly half the γ-functions can be reduced. This reduction can occur in two ways:

1. The same predicate occurs more than once in a γ-function. In this case, the value of the first occurrence of the predicate can prune the nested predicate. The *Reduce*() function of Algorithm 4.3 accomplishes this task. This optimization is not handled by the methods of Havlak [Hav94].

2. If all γ-arguments have the same value, then the γ-function can be replaced by the value of the arguments.

The routine *Reduce*() is linear in the size of the γ-function passed in as an argument, which itself is linear in terms of the CD_Chain subtree. As an example of *Reduce*(), examine the following code fragment (whose CFG is shown in Figure 4.14):

```
        x0 = 0
        if( P ) goto 30
10      x2 = γa
        y1 = x2
        goto 40
30      x1 = 1
        if( Q ) goto 10
40      x3 = γc
```

Before reduction, the γ-function at 10 will be:

$$\mathtt{x}_2 = \gamma_a(\mathtt{P}, t \rightarrow \gamma_b(\mathtt{Q}, t \rightarrow \mathtt{x}_1, f \rightarrow \top), f \rightarrow \mathtt{x}_0)$$

And the γ-function at 40 will be:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \gamma_d(\mathtt{Q}, t \rightarrow \gamma_a, f \rightarrow \mathtt{x}_1), f \rightarrow \gamma_a)$$

After applying the first reduction rule, the γ-function at 40 (γ_c) becomes:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \gamma_d(\mathtt{Q}, t \rightarrow \mathtt{x}_1, f \rightarrow \mathtt{x}_1), f \rightarrow \mathtt{x}_0)$$

Next, the second reduction rule is applied, yielding:

$$\mathtt{x}_3 = \gamma_c(\mathtt{P}, t \rightarrow \mathtt{x}_1, f \rightarrow \mathtt{x}_0)$$

A Related GSA Form. A method employed by Havlak [Hav93], aimed at value-numbering, *thins* the γ-function to eliminate paths that cannot reach a confluence node. Essentially, if all arguments save one are $\top$, then the entire argument structure is reduced to the one non-$\top$ argument. Thinning is directed at an efficient implementation of value-numbering, but misses identifying constants in some situations, such as shown in Figure 4.11. If thinning were used, the γ-function at 40 would reduce to: $\mathtt{x}_2 = \gamma($ P, $t \rightarrow \mathtt{x}_1$, $f \rightarrow \mathtt{x}_0)$. If P is not constant, the meet of its arguments is $\bot$. However, if Q is known to be true, the constant value $\mathtt{x}_1$ will be missed using thinning, since the *false* side of predicate P is prematurely reduced to $\mathtt{x}_1$, instead of $\top$.

4.4.4 Conditional Constant Propagation

Once converted into GSA form, we can improve upon the *Propagate*() routine in Algorithm 4.1 to take advantage of predicates that can be determined to be constant at compile time. When encountering a γ-function, we first attempt to evaluate the predicate. If constant, we follow the indicated branch, propagating constant values as found. If not constant, we take the meet of its arguments. The revised routine, *CondProp()*, is given in Algorithm 4.4.

Several comments are in order regarding Algorithm 4.4:

- Simple extensions allow us to detect constants other than integers, such as logical or enumerated types.
- Special cases can detect additional constants, even when one of the operands is $\perp$. These include:
  - Zero times anything (including $\perp$) equals zero.
  - With logical types, $\mathit{true} \vee \star = \mathit{true}$, where $\star$ is any lattice value, including $\perp$.
  - Likewise, $\mathit{false} \wedge \star = \mathit{false}$.
- Reaching a μ-function returns $\perp$. This result is due to the separate solver used for loops, discussed next. We may encounter ϕ-functions in a program with irreducible loops. In this case, ϕ-functions cannot be converted to γ-functions, but we can still detect simple constants.
- The number of intraprocedural constants based upon conditionals is quite small – Section 4.5 provides experimental data.

4.4.5 Loops

Cycles in the GSA data-flow graph are the result of loops within the original program. The variables defined within these cycles are classified with a separate solver: induction variable analysis. Induction variables are traditionally detected as a precursor to strength reduction, and more recently have been used for dependence analysis with regard to subscript expressions. We have developed methods for detecting and classifying induction variables (including nonlinear induction variables [EHLP92, HP92, Wol92b], where the general class of all induction variables are referred to as *sequence* variables) based on

Given: Data-flow graph, initialize with lines 1 - 4
Do: Execute lines 5 - 7
Result: Simple and conditional constants assigned to lattice elements

```
1:      forall tuples t do
2:          lattice( t ) ← ⊤
3:          t.visited ← false
4:      endfor

5:      Visit all basic blocks B in the program
6:          Visit all tuples t within B
7:              if t.visited = false then CondProp( t )

8:      CondProp ( tuple t )
9:          t.visited ← true
10:         if link( t ) ≠ ∅ then
11:             if link( t ).visited = false then CondProp( link( t ) )
12:             lattice( t ) ← lattice( t ) ⊓ lattice( link( t ) )
13:         endif
14:         if left( t ).visited = false then CondProp( left( t ) )
15:         if right( t ).visited = false then CondProp( right( t ) )
16:         case on type (t )
17:             constant C: lattice( t ) ← C
18:             arithmetic operation:
19:                 if all operands have constant lattice value
20:                 then lattice( t ) ← arithmetic result of
21:                     lattice values of operands
22:                 else
23:                     lattice( t ) ← ⊥
24:                 endif
25:             store: lattice( t ) ← lattice( RHS )
26:             φ-function: lattice( t ) ← ⊓ of φ-arguments of t
27:             γ-function:
28:                 if lattice( γ-predicate ) = constant C then
29:                     lattice( t ) ← lattice value of
30:                     γ-argument corresponding to C
31:                 else lattice( t ) ← ⊓ of all γ-arguments of t
32:                 endif
33:             μ-function: lattice( t ) ← ⊥
34:             η-function: lattice( t ) ← lattice ( η-argument )
35:             default: lattice( t ) ← ⊥
36:         end case
37:     end CondProp
```

Algorithm 4.4 Demand-driven propagation with conditional constants

strongly connected components in the SSA data-flow graph [GSW]. Using Tarjan's algorithm [Tar72] for detecting maximal strongly connected components (SCCs) of an arbitary directed graph, we employ a variant of Algorithm 3.3 to detect an SCC within the data-flow graph. We then classify the SCC based on the number of *merge* operators and the operations that are applied to the variables involved in the SCC. For example, an SCC that involves only one variable and one μ-function is classified as *linear* if the only operations of the variable are addition or subtraction of integers within that cycle. Typically, for reducible flow graphs, the number of μ-functions is indicative of the type of induction variable which exists.

These techniques make use of an exit function, the η-function, that holds the exit value of a variable assigned within the loop. An exit expression is held by the η-argument, which if constant can be used to propagate values outside the loop. The exit value may be a function of the loop tripcount (which may itself be an expression determined to be constant), or may be invariant with respect to the loop, as this simple example shows:

i_1 = 3
loop
  $i_2 = \mu(i_1, i_3)$
  i_3 = 3
endloop
$i_4 = \eta(i_2)$
j = i_4

Since the value of i entering the loop and the last definition within the loop are identical, the value of i is easily determined to be constant and can be propagated to the store for j. This case is the easiest and simplest for resolving cycles.

If the exit value does depend on the loop tripcount, which can be determined constant at compile time, we can employ *path-sensitive* analysis, since the determination of the actual path taken by the program (in this case the precise number of times a loop will be executed) is instrumental in computing the result. In fact, path-sensitive analysis may not depend on such strong knowledge; sometimes knowing just that the tripcount is positive is sufficient to determine an exit value for a variable. Consider the loop above with a different value assigned to i within the body of the loop:

```
i1 = 3
loop
   i2 = μ(i1, i3)
   i3 = 2
endloop
i4 = η(i2)
j = i4
```

If we have no information on the tripcount of the loop, the exit value for i, stored in the η-argument, must be $\perp$ since the μ-function cannot resolve its initial and iterative values for i. But if the tripcount can be determined to be positive, the exit value for i must be 2, and j can be classified as constant.

If the exit of a loop is not at the loop-header basic block, then it may still be possible to determine a constant since some of the code of the cycle is always executed. Examine this code:

```
      i1 = 1
10    loop
         i2 = μ(i1, i5)
         if (P) then
            i3 = 2
         else
            i4 = 3
         endif
         i5 = φ(i3, i4)
         if (Q) goto 10
      endloop
      i6 = η(i5)
      j = i6
```

Regardless of Q, if P can be determined constant, the exit value of i is constant on all paths, and can be propagated to j. In this case the exit value for i points to the ϕ-function inside the loop, not the loop-header μ-function. That is because the last reaching definition of i at the exit point is i_5. Multiple exit points from a loop greatly complicate this analysis.

```
i1 = 5
while (P)
      i2 = μ(i1, i3)
      if ( i2 = 5 ) then
         j1 = 1
      else
         j2 = 2
      endif
      j3 = γ(i2 = 5, t → j1, f → j2)
      i3 = j3 + 4
endwhile
i4 = η(i2)
k = i4
```

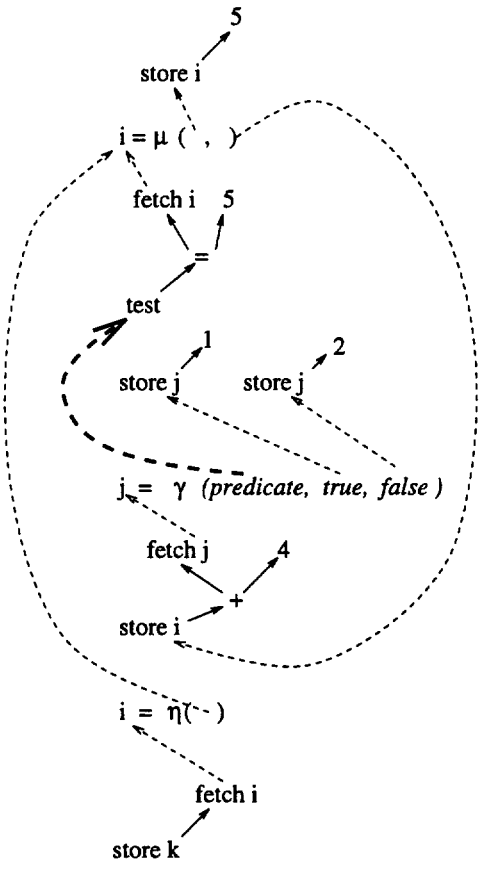


Figure 4.15 Where the conversion to GSA form is necessary to detect a data-flow cycle

A separate paper describes in detail the workings of our specialized solvers that detect and classify a large assortment of linear and nonlinear sequence variables [GSW]. Keeping in mind that these demand-driven solvers operate on the data-flow graph, we obviously only detect general sequences when a cycle is formed in the data-flow graph. Figure 4.15 shows conditional code inside of a while loop and its associated data-flow graph. Inspection reveals that the store to k is constant since all paths (including or not including the loop) maintain the value of i as 5. However, without γ-functions, there is no cycle in the data-flow graph, since the ϕ-function at the conditional merge points to two killing definitions of j. The conversion to a GSA form correctly introduces a cycle via the γ-predicate. Our current methods have not categorized the class of sequences based upon finding γ-functions in the data-flow graph. This area is a fruitful source of future research.

Another method would be to design a separate cycle solver specifically for constant propagation. By symbolically executing a loop, the pattern of its variables might be found. The number of iterations to execute the loop would vary, however. In the following example

```
i = 1
loop:  N = 1 to M
       if ( N < 7 ) then
          i = i + 1
       endif
endloop
```

i changes on the first 6 iterations, and becomes constant on the 7^{th} and succeeding iterations. This is a technique similar to that used in the Parafrase-2 compiler [HP92], where the number of iterations symbolically executed to try and detect a pattern can be arbitrarily set by the user.

4.4.6 Notes on Implementation

Irreducible Graphs. Although we have discussed some of the effects that irreducible flow graphs have on intermediate analysis, we would like to discover how often such graphs occur. Table 4.2 shows the programs from the benchmark suite that contain procedures that translate into irreducible CFGs. Only a total of 8 routines out of 1071,

Program	*Number of Routines*	*Number Irreducible*
PERFECT club		
spice	128	5
others	504	0
RiCEPS		
boast	58	1
ccm	145	1
wanal1	11	1
others	154	0
Mendez		
all	71	0
Total	1071	8

Table 4.2 Scientific codes that contain irreducible loops

less than 1 percent, contain irreducible CFGs. While certainly minimal, those scientific codes with such structure are much more difficult to analyze with techniques that rely on a reducible flow graph structure. Node-splitting [ASU86, Hec77] is a solution that can transform any irreducible graph into a reducible one by cloning some of the basic blocks and adding additional control flow.

GSA and Data Structure Size. How is the size of the data-flow graph affected by conversion to GSA form? Table 4.3 shows the increase in the number of tuples for data-flow graphs beginnning with the basic form, with no sparse representation, to FUD chain form and GSA form. Since GSA form requires one more argument for γ-functions as opposed to ϕ-functions (for the predicate), and nesting can occur in tracing control flow, we expect the GSA form to be somewhat larger than the FUD form. Table 4.3 bears this out. Severely unstructured code can result in 100% or 200% increase for the GSA data-flow graph, as was noticed in `spice`, `boast`, and `sphot`.

4.5 Experimental Results

To gauge the effectiveness of our routines, we measured the number of constants (both simple and conditional) on Fortran scientific codes found in the PERFECT, RiCEPS, and Mendez benchmark suites, as discussed in §1.4. Counting constants can be a misleading statistic. Counting *tuples* with constant *lattice* value does not indicate a constant has been propagated, since the statement `i = 1` will result in a constant *store* operation.

Program	*basic data-flow graph*	*FUD form*	*% increase over basic*	*GSA form*	*% increase over basic*
PERFECT club					
adm	51102	69598	36	80298	57
arc2d	30530	38478	26	41764	37
bdna	50204	62495	25	67005	33
dyfesm	23922	31003	30	34061	42
flo52	35630	43288	21	47480	33
mdg	16599	20197	22	22069	33
mg3d	30574	44815	47	46891	53
ocean	82278	88189	7	91751	12
qcd	16879	23334	38	28206	67
spec77	59270	72548	22	78450	32
spice	248454	316936	28	521748	110
track	18288	25300	38	34492	88
trfd	3095	5563	80	5779	87
RiCEPS					
boast	83895	123419	47	176423	110
ccm	171319	207388	21	229182	34
hydro	60992	69824	14	75008	23
linpackd	3718	4714	27	5056	36
simple	25991	29022	12	29562	14
sphot	8979	13138	46	27220	203
wanal1	14495	20953	45	21157	46
wave	79294	97667	23	111941	41
Mendez					
baro	7388	9394	27	10342	40
euler	16321	19938	22	22740	39
mhd2d	8417	10681	27	11145	32
shear	10353	13690	32	14820	43
vortex	5726	7556	32	8966	57
Total	1163713	1469248	26	1813994	57

Table 4.3 Number of tuples for the different data-flow forms

Program	*Total Fetches*	*Simple Constants*	*Conditional Constants*	*Constant Predicates*	*Nonconstant Predicates*
PERFECT club					
adm	6072	102	102	1	271
arc2d	4357	99	99	0	51
bdna	3810	48	52	0	171
dyfesm	1782	4	5	0	130
flo52	2766	77	78	0	108
mdg	771	0	0	0	39
mg3d	4774	249	255	0	118
ocean	2080	1	1	0	153
qcd	1420	19	21	2	87
spec77	4291	52	53	2	119
spice	16818	220	220	40	1841
track	1105	75	75	1	150
trfd	452	33	33	2	20
RiCEPS					
boast	10056	63	63	3	696
ccm	13301	932	958	4	537
hydro	3333	1089	1099	0	208
linpackd	499	50	53	0	32
simple	1662	635	657	1	25
sphot	591	21	21	0	32
wanal1	1789	43	43	3	28
wave	6615	171	174	1	451
Mendez					
baro	945	50	60	0	46
euler	942	172	172	4	116
mhd2d	774	412	422	0	21
shear	1398	58	58	3	39
vortex	521	34	34	0	17
Total	92924	4709	4808	67	5506

Table 4.4 Constant *fetch* tuples and predicates found using Algorithms 4.1 and 4.4

Similarly, `j = 2 + 3` is an example of constant folding, but not of constant propagation. For the purposes of this work, a constant is considered propagated if there is a fetch of a tuple whose *lattice* value is constant.

Results are shown in Table 4.4. The *Simple Constants* column shows the number of simple constants identified by Algorithm 4.1. The *Conditional Constants* column reflects constants found using Algorithm 4.4, which includes those found using Algorithm 4.1. The vast majority of constants (97%) are simple constants. Most conditional constants were as a result of loop analysis. The *Constant Predicates* and *Nonconstant Predicates* columns show the number of predicates controlling branch nodes determined to be constant using Algorithm 4.4. Although a few predicates are determined intraprocedural constants, these are mainly due to guards; with interprocedural analysis and inlining [Hal91] we expect to see many more conditional constants propagated [MS93].

It is important to note that since we have performed our experiments on benchmark programs, there may be a number of static initializations added to the codes. This condition could skew the data slightly when compared to code not targeted for benchmarking.

4.6 An Extension to Arrays

Can demand-driven constant propagation be extended to arrays? If so, is it a useful analysis technique? In this section we apply FUD chains to arrays in order to determine if a significant number of constants exist for particular array values. Essentially, FUD chains consider an array to be a monolithic data structure, even though it is composed of many parts. We need to analyze particular index values if a constant is able to be propagated. Thus, while a definition of any array value is treated as a nonkilling definition with respect to the entire structure, we can follow links of the array and check for a match of indices. By following *def-def links*, and testing whether each index value is constant, we are able to follow particular array elements until either $\perp$ is reached or another definition for that element occurs.

The idea is illustrated in this example:

```
i = 2
A(2) = 7
A(4) = 8
j = A(i)
```

When we encounter the store to `j`, its value comes from array `A`. If the index value of `A`, `i`, was not constant, $\perp$ would be assigned to the *lattice* of `j`. However, `i` is the constant 2 in this case, so we follow the *link* to the store of `A(4)`. Again, if the lattice value of this index value was nonconstant, $\perp$ would be returned. Since it is constant but not equal to 2, the *def-def link* is followed. Finally, the store to `A(2)` is encountered, which is constant. Thus, `j` is classified as the constant 7. This procedure is easily extended to accommodate multi-dimensional arrays.

We did not expect scientific programs to be written in such a way that arrays would have many constant values, since their most common use is within an iterative looping structure, resulting in nonconstant indices. This expectation was borne out in our experiments, where on all the benchmarks from §1.4 we only found a dozen propagated array constants, where the average number of *links* followed to detect these constants was 1.5.

It is possible to perform array constant propagation on array references where the index value is an arbitrary expression, but this extension may require general symbolic analysis as the indices could be quite complex.[†] The idea is explained in this code fragment:

```
S1:        ...
S2:   A(n+m) = C
S3:   {no writes to n, m, or A }
S4:   ...= A(n+m)
S5:        ...
```

If C has constant lattice value, the store to `A(n+m)` in S_2 can be assigned a constant value. When there are no writes to `n`, `m`, or `A` between S_2 and S_4, this constant can be propagated to the use of `A(n+m)` in S_4, even though `n` and `m` are not known. However, this analysis would unlikely be profitable since, as we saw experimentally with constant array indices, candidates for constant propagation are usually assigned to scalar variables, not array elements.

[†]The process of analyzing general array index expressions is covered in the work on recurrence variables [GSW].

4.7 Comparison With Other Work

Previous methods perform constant propagation analysis as an iterative data-flow problem [ASU86], in which iterations continue until a fixed point is reached [GT93, WZ91]. We will see that our demand-driven algorithm offers advantages over the traditional approach.

4.7.1 Classification of Methods

As explained by Wegman and Zadeck [WZ91], constant propagation algorithms can be grouped in two ways: (*i*) using the entire graph or a sparse graph representation, and (*ii*) detecting simple or conditional constants. This classification naturally creates four classes of algorithms. We have seen that propagating information about each symbol to every node in a graph is inefficient, since not all nodes contain references or definitions of the symbol under consideration. Sparse representations, on the other hand, such as def-use or use-def chains [ASU86], SSA [CFR+91], Dependence Flow Graphs (DFG) [JP93], or Program Dependence Graphs (PDG) [FOW87], have all shown the virtue of operating on a sparse graph for analysis.

The distinction between the four types of algorithms is explained well by Wegman and Zadeck [WZ91], and essentially shows (as one would expect) that a combination of detecting conditional constants on a sparse graph is the most efficient method with the largest class of constants detected. We will look at the algorithm that they present, since it incorporates both sparse graph representation and conditional code. The sparse graph employed in their method is based on SSA form.

4.7.2 A Closer Look at One Algorithm

The algorithm used by Wegman and Zadeck operates on CFG edges. SSA *def-use* edges are added to the graph once the program has been transformed into SSA form.

Their algorithm works by keeping two worklists, a FlowWorkList and an SSAWorkList. Flow edges are initially marked *unexecutable*. Edges are examined from either worklist until empty, with those examined from the FlowWorkList being marked *executable*. The destination node for these edges also have their ϕ-functions evaluated by taking the meet of all the arguments whose corresponding CFG predecessors are marked *executable*. Expressions are evaluated the first time a node is the destination of a flow edge, and also when the expression is the target of an SSA edge and at least one incoming

flow edge is executable. This algorithm is iterative, following CFG paths and SSA edges until both worklists are empty. More detail can be found in the original paper [WZ91].

This algorithm finds all simple constants, plus additional constants that can be discovered when the predicate controlling a switch node is determined to be constant. The time complexity is proportional to the size of the SSA graph, and each SSA edge can be processed at most twice.

Since ϕ-functions are reevaluated each time an edge with that node as a destination is examined, Wegman and Zadeck note that expressions that depend on the value of a ϕ-function may be evaluated twice for each of its operands. For example, in this program fragment:

```
          if ( P )
          then
10           y1 = 1
             z1 = 2
          else
20           y2 = 1
             z2 = 3
          endif
30        y3 = φ(y1, y2)
          z3 = φ(z1, z2)
          x1 = y3 + z3
```

if P is not constant, the expression for x_1 may be evaluated many times. If the flow edge from 10 is processed first, then x_1 equals 3, and it may stay at 3 if the SSA edges for y are examined next. Eventually, x_1 will evaluate to $\perp$, as the merge for z becomes nonconstant. It is this multiple expression evaluation that we seek to avoid by using a demand-driven technique.

4.7.3 An Evaluation of Methods

Both our demand-driven approach and the Wegman-Zadeck method have their advantages and disadvantages. We will highlight the differences, then report on performance studies of both methods.

Features of Wegman-Zadeck Technique

- Works on all CFGs, including those that are irreducible.
- Does not need additional data-structure support past the SSA graph construction.
- Automatically identifies unreachable code due to constant predicates.
- Given a loop in the CFG with the exit at the loop header, it requires that the incoming value is equal to the iterative value – essentially that the meet of the μ-function arguments is constant.
- Reevaluates ϕ-functions and expressions – this can lead to slower performance, especially when there are numerous, complex calculations involving constants. We expect this effect to be exacerbated with interprocedural constant propagation, since the number of constants certainly increases [MS93], leading to more expression evaluation.
- Needs explicit management of control flow information, and manages two worklists to achieve interpretability, one for following control flow edges, and the other to follow data-flow edges. This approach is traditional with iterative methods.
- After converting to SSA form, the Wegman-Zadeck method needs to insert both def-use SSA edges and use-def SSA edges to correctly implement the algorithm.
- Lends itself to an incremental interprocedural technique. After intraprocedural analysis, control flow and SSA data-flow edges are only added to worklists when interprocedural analysis produces additional information. Then it solves the intraprocedural problem again, but in this phase it is a pessimistic solver, since it will only detect additional constants that had previously been classified as $\perp$ [Aut94].

Features of Demand-Driven Technique

- Convert to FUD form with just use-def edges.
- Need to augment FUD chains with γ-functions. While we will consider this cost totally attributable to the demand-driven constant propagation method in our experiments, in fact the cost may be amortized in a high performance compiler that employs value-numbering. Havlak [Hav94] has shown that the GSA data-flow graph enables powerful symbolic pattern-matching and rewriting techniques.

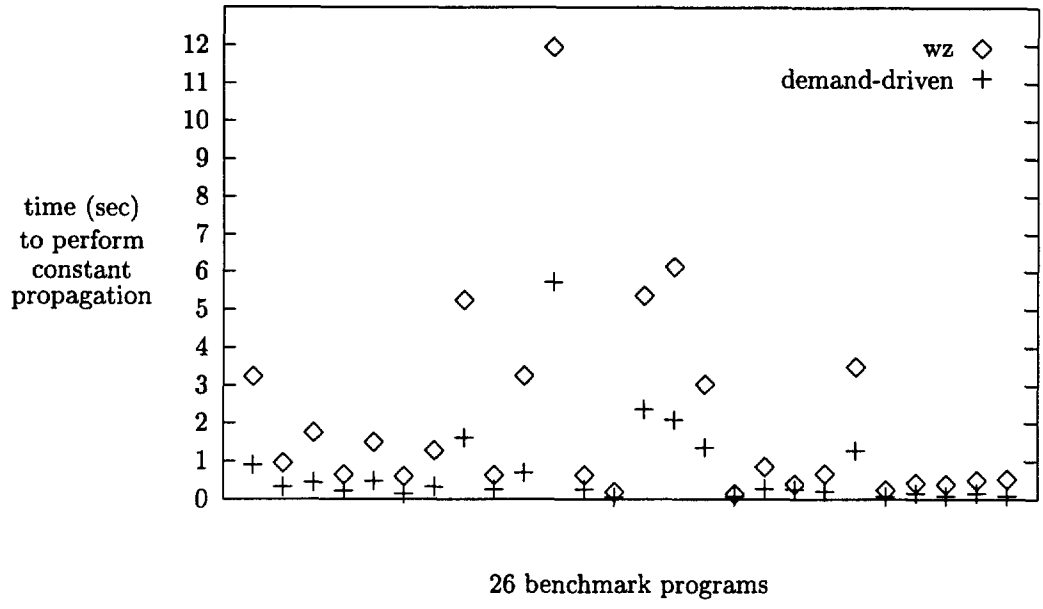


Figure 4.16 Comparison of times just for constant propagation analysis

- Each ϕ-function (or its converted form as a μ- or γ-function) and expression are evaluated exactly once.
- Demand-driven constant propagation blends well with other important analysis phases, such as induction variable detection.
- Only need to operate on the data-flow graph, since interpretability of the CFG is captured symbolically in the γ-functions.

4.7.4 An Empirical Comparison

We ran both algorithms on the scientific benchmarks discussed in Chapter 1. We compiled both algorithms using the parameters discussed in §3.1.5. A straight comparison of times to perform constant propagation is provided in Figure 4.16. The 26 benchmark programs are linearly displayed according to the order given in §1.4. If we add the total time needed to convert FUD chains into GSA form for the demand-driven approach, the results are as shown in Figure 4.17.

Even when we factor in the time for γ-function conversion to GSA form, our demand-driven method is faster than the Wegman-Zadeck approach, with 4 exceptions: `spice`, `track`, `boast`, and `sphot`. However, if GSA form is used for other applications, such

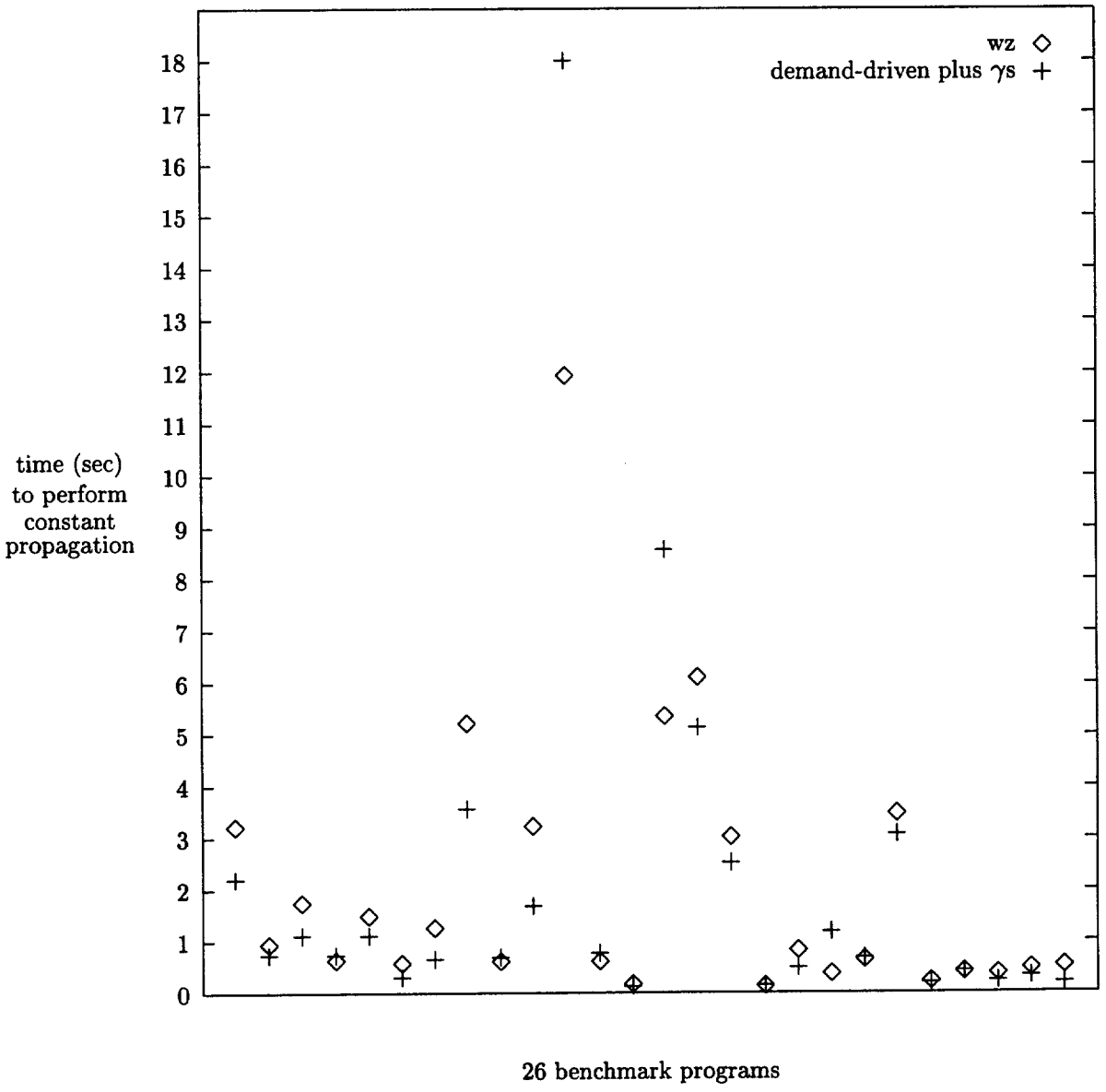


Figure 4.17 Comparison of times for constant propagation analysis with γ-functions

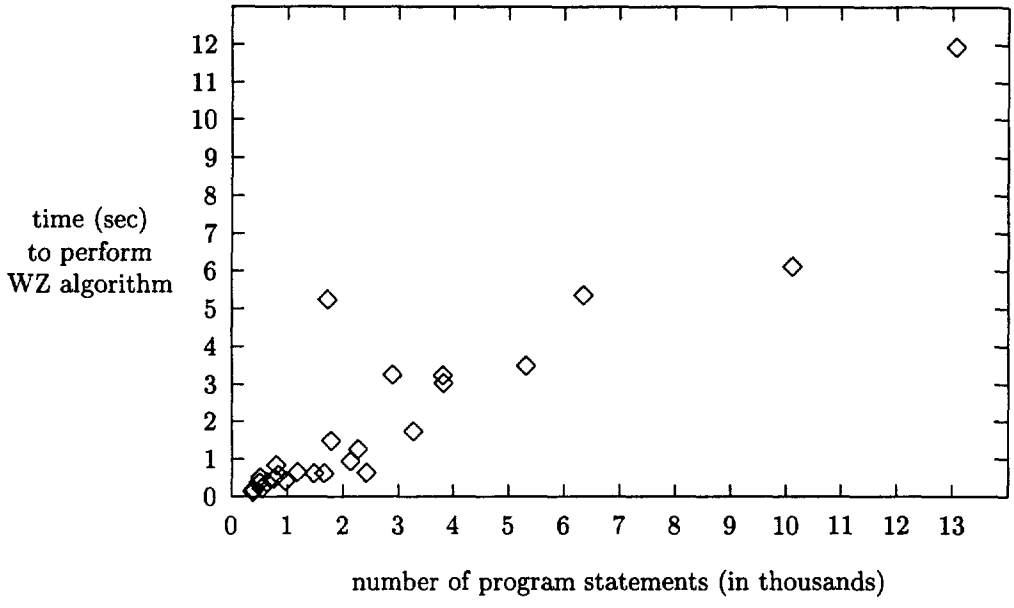


Figure 4.18 Time to perform Wegman-Zadeck algorithm as a function of the the number of statements in each program

as value-numbering, we can amortize the cost of building the GSA graph over multiple applications.

In an attempt to provide some normalization to these results, Figures 4.18 and 4.19 show the time to perform constant propagation for the two approaches as a function of the number of statements in each program. These graphs show that constant propagation processing is fairly linear in the size of the program, where the demand-driven approach has a slope that is not as steep as the Wegman-Zadeck method. The one anomalous program for both methods is `ocean`, which has complex structure due to large numbers global variables and subroutine calls. In fact, within the basic data-flow graph of `ocean`, with 82278 tuples as shown in Table 4.3, tuples involving global variables as procedure parameters comprise 61% of its entire structure.

The times displayed in these graphs include optimizations for intraprocedural analysis of the Wegman-Zadeck approach not suggested in their original work. When applying their technique incrementally (as discussed in the previous subsection) these optimizations are no longer valid. Without the additional optimizations, the Wegman-Zadeck method is two to three times slower.

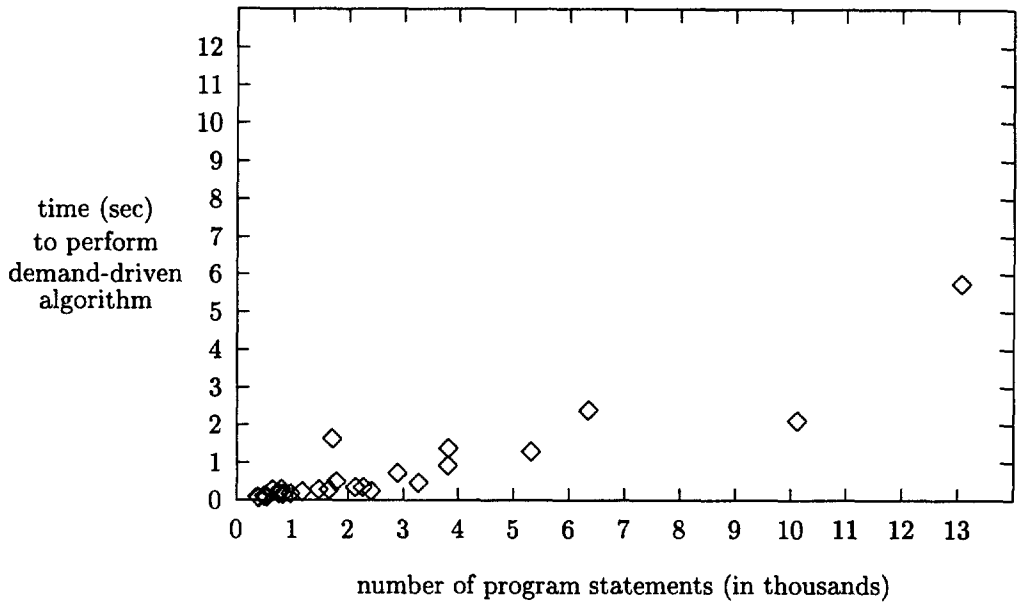


Figure 4.19 Time to perform demand-driven constant propagation as a function of the number of statements in each program

4.8 Further Extensions of Constant Propagation

Numerous extensions to this work are possible. One important topic is interprocedural analysis and procedure integration. Although some work has already been done in this area [GT93, Hal91, MS93], we would like to apply our demand-driven style to the problem. In particular, once intraprocedural constant propagation has been performed, interprocedural mod/ref analysis may provide information on procedure arguments. A reapplication of intraprocedural analysis using this information would be most efficient if the analysis can be performed incrementally. Investigation [Aut94] has suggested that a method such as the Wegman-Zadeck approach may be best suited for the second intraprocedural pass. By managing worklists of only those control flow edges and definitions affected by the interprocedural information, an efficient incremental solver may be designed. As pointed out earlier in this chapter, such an incremental solver is necessarily pessimistic.

Unreachable code can be identified with our demand-driven technique, but we have not yet developed the algorithm fully. It may well be that unreachable code is best identified using edges instead of nodes, as pointed out by Wegman and Zadeck [WZ91].

Traditional SSA form has been criticized for lacking a method to propagate constants

determined by predicate analysis [JP93]. In the following fragment:

```
if (x1 = 1) then
   i0 = x1
else
   j0 = x1
endif
```

it is desirable to be able to assign i_0 a constant value. A sophisticated compiler may analyze the guard and determine that under the range of the true side of the conditional, x_1 will always be 1. This notion of a *derived assertion* is not new [LFK+93], but to our knowledge has not yet been integrated into the SSA form. Using demand-driven FUD form, derived assertions can easily be captured by inserting dummy assignments. We propose a new reference chaining operator, the ρ-function, which serves as the new definition of its variable. By examining the right-hand side of the predicate, the fragment above becomes:

```
if (x1 = 1) then
   x2 = ρ(1)
   i0 = x2
else
   j0 = x1
endif
```

Now constant propagation may easily be performed via the argument of the ρ-function, which may be constructed of actual operations in the intermediate form.

In addition to constant propagation, the explicit representation of derived assertions may be advantageous if bounds information can be expressed. In this fragment,

```
if (n0 > 0) then
   for i=1, n0
      ...
   endfor
endif
```

if the compiler cannot determine any value for n_0, then it cannot be determined if the body of the loop will ever be executed within the range of the `if`. However, analysis of

the guard condition assures the loop will be executed at least once. If limit information can be encoded in the argument of the ρ-function, the loop may be transformed:

```
if (n0>0) then
   n1 = ρ(>0)
   for i=1, n1
      ...
   endfor
endif
```

Now it is clear from the expression describing the tripcount that the loop will be executed at least once, since the lower limit of n is known. We have seen how knowing that a loop will trip at least once can permit the detection of additional constants.

Another approach is to merge constant propagation with induction (in its most general form, sequence) variable detection and classification. When separate, each phase must be rerun to maximize effectiveness. Constant propagation can detect conditional constants, which may control loop tripcounts. A general sequence solver can find loop invariant values, which can be utilized by the constant propagation algorithms. Thus, we feel that an integrated approach holds promise for achieving greater precision of both analysis phases.

Other possible extensions include run-time analysis and value numbering (the process of finding congruent expressions by assigning unique integers to each set of congruent value graph nodes [Hav94]). It is important to obtain timing results that demonstrate how much execution time is saved for the increased analysis done at compile time. These are interesting tradeoffs, and remain an open question. Although not constant propagation *per se*, the structure of GSA lends itself particularly well to implementing value numbering, as has been shown by Havlak [Hav94]. Finally, more work can be done in the area of noninteger and symbolic expression propagation.

Chapter 5

General Reference Chaining

5.1 A Generalized Reference Chaining Algorithm

Numerous data-flow problems require information other than reaching definitions, such as reaching uses or, more generally, reaching references. While these problems imply downward-exposed references, other problems may require upward-exposed references to solve such problems as live variables. Specialized algorithms, such as those for SSA and FUD chains, could be developed for each problem. However, the method of linking arbitrary references is similar in all cases, so it makes sense to develop a generalized reference chaining algorithm, and use parameters specific to the data-flow problem to be solved as inputs to the algorithm. We will pattern the general reference chaining (GRC) algorithm after Algorithms 3.1 and 3.2, with several modifications.

Monotonic data-flow problems that utilize GRC must merge information at confluence nodes in the CFG when paths from two or more nodes with nonidentity transfer functions converge. For reaching definitions, these merge operators are called ϕ-functions. Other specific operators are used depending on the problem, but for our GRC algorithm we designate the Ω-function as the generic merge operator.

The minimal placement of merge operators for GRC is at $DF^+(S)$[†], where S is the set of nodes containing any tuples that produce a nonidentity transfer function. However, as we have seen, $DF^+(S)$ is computationally equivalent to $J^+(S)$. For this equivalence to hold, an initial reference is required at *Entry* (more generally, at the *START* node, as explained below). With FUD chains this initial reference is an assumed definition of all variables (designated as *source*), but with GRC the initial reference is set to $\oslash$, indicating that there is no initial reference.

[†]To be specific, merge operators are placed at $DF^+(S)$ for *forward* data-flow problems. For *backward* data-flow problems they will be placed at $PDF^+(S)$.

The reaching definition solution using FUD chain form has simple transfer functions based upon each tuple type. If tuple t is a definition of variable V then $V.CurrentDef = t$. If not, $V.CurrentDef$ is unchanged. This dichotomy of tuple types does not generalize to GRC, since for a given problem some tuples block all reaching references, effectively setting $V.CurrentDef = \oslash$. We call this set of tuples **BlockTuples**, where **BlockTuples** $= \emptyset$ for the problem of reaching definitions. An example of a problem with a nonempty **BlockTuples** set is reaching uses, where killing definitions block all previous uses from reaching past the definition.

Since GRC can be applied to both forward and backward problems, in some ways it is analogous to sparse evaluation graphs (SEGs) [CCF91]. However, as noted in §3.2.4, sparse evaluation graphs are constructed *per variable*, whereas GRC graphs are an augmentation of the already extant CFG and data-flow graph. Another difference is that SEGs are constructed on a per-problem basis, which solves the data-flow problem on a sparse graph and maps the solution back to each node in the CFG. The merge functions and *links* created by the GRC algorithm encapsulate information that can be extracted on demand when desired at just those points that need the data-flow solution.

As with the FUD chain algorithm, GRC is performed in two phases: function placement and chaining. Algorithm 5.1 provides the details for placing Ω-functions in the correct nodes, and Algorithm 5.2 fills in the correct *links* for a given data-flow problem.

For a given data-flow Problem input parameters to GRC are:

- **Direction**: Problem is either a forward or backward data-flow problem.
- **RefLink**: which references for each variable V, from the set {use, def}, get linked to $V.CurrentRef$
- **RefTuples**: tuple types that imply a nonidentity transfer function for the basic block node in which they reside
- **BlockTuples**: tuple types that for Problem block all previous references for V, setting $V.CurrentRef = \oslash$

Based upon **Direction**, we set the following variables:

	Direction	
	forward	backward
START	*Entry*	*Exit*
FRONT	*dom frontier*	*pdom frontier*
NEIGHBORS	*predecessors*	*successors*
NEXT	*successor*	*predecessor*
LOCATION	*beginning*	*end*

We use the following data structures for GRC:

- *R(V)* – A list of all nodes N that reference V, where N contains any tuples in RefTuples.
- *symbol(tuple)* – A function that returns the variable symbol (name) associated with this tuple, if it exists. Returns *null* otherwise.
- $V.CurrentRef$ – A pointer to the current reference (tuple) of symbol V. Logically points to the top of a reference stack. Initialized to $\oslash$.
- $t.SavedRef$ – A pointer to the current reference of *symbol(t)* before processing this tuple. Used to logically pop references off a stack when returning from recursive calls down the dominator or postdominator tree.
- *Children(N)* – A pointer to the children of N in the dominator tree if Direction = forward, and children of N in the postdominator tree if Direction = backward.
- *WhichNeighbor(N,Q)* – An integer indicating which predecessor of Q in the CFG is N if Direction = forward, and which successor in the CFG is N if Direction = backward.
- *Work_List* – An unordered list of CFG nodes. For each variable V, *Work_List* is initialized to *R(V)*.
- *HasFunc(*)* – A reference field to a variable for each CFG node. *HasFunc(N)* = V means node N already has an Ω-function added for variable V.
- *Work(*)* – A reference field for each CFG node. *Work(N)* = V means that node N has already been added to *Work_List* for variable V.

Given: $R(V)$, $\forall\ V$
Do: compute $FRONT(\ N\ )$, $\forall\ N \in$ CFG
Result: Ω-functions inserted into CFG

```
1:      for all nodes N do
2:          HasFunc(N) ← ∅
3:          Work(N) ← ∅
4:      endfor

5:      for each symbol V do
6:          Work_List ← ∅
7:          for each N in R(V)
8:              Work(N) ← V
9:              Work_List ← Work_List ∪ { N }
10:         endfor
11:         while Work_List ≠ ∅ do
12:             take N from Work_List
13:             for each Q ∈ FRONT(N) do
14:                 if HasFunc(Q) ≠ V then
15:                     HasFunc(Q) ← V
16:                     i ← number of NEIGHBORS of Q
17:                     place V = Ω( V_1, V_2, ..., V_i ) at LOCATION of basic
18:                         block Q, where V_j corresponds to the j^th NEIGHBOR of Q
19:                 endif
20:                 if Work(Q) ≠ V then
21:                     Work(Q) ← V
22:                     Work_List ← Work_List ∪ { Q }
23:                 endif
24:             endfor /* each Q in FRONT */
25:         endwhile
26:     endfor  /* each symbol V */
```

Algorithm 5.1 Placement of Ω-functions

Given: Initialized data structures
Do: Call *RefChain(START)*
Result: GRC form

```
1:  RefChain(N)
2:      for all tuples t ∈ N, in direction order do
3:          V ← symbol(t)
4:          if t ∈ RefLink of V then
5:              link( t ) ← V.CurrentRef
6:          endif
7:          if t ∈ {RefTuples ∪ Ω-functions} for V then
8:              t.SaveRef ← V.CurrentRef
9:              if t ∈ BlockTuples then
10:                 V.CurrentRef ← ⊘
11:             else
12:                 V.CurrentRef ← t
13:             endif
14:         endif
15:     endfor  /* all tuples of N */
16:     for each Q ∈ NEXT(N) do /* Neighbors in CFG */
17:         j ← WhichNeighbor(N,Q)
18:         for each Ω-function merge tuple f in Q do
19:             V ← symbol(f)
20:             link( j^th argument of f ) ← V.CurrentRef
21:         endfor
22:     endfor
23:     for each Q ∈ Children(N) do /* children in dom or pdom tree */
24:         RefChain(Q)
25:     endfor
26:     for all tuples t ∈ N, in reverse direction order do
27:         if t ∈ {RefTuples ∪ Ω-functions} for V do
28:             V ← symbol(t)
29:             V.CurrentRef ← t.SaveRef
30:         endif
31:     endfor
32: end RefChain
```

Algorithm 5.2 Reference Chaining: linking each reference to the next exposed reference and correctly inserting Ω-function arguments

5.2 Applications of GRC

5.2.1 Reaching Definitions.

One obvious example of GRC is the reaching definitions problem, solved with FUD chains. FUD chains are built with Algorithms 5.1 and 5.2 by specifying the parameters (where the merge operator is specified as ϕ-functions) for the Problem of reaching definitions:

- **Direction:** forward
- **RefLink:** {use,def}
- **RefTuples:** any killing or nonkilling definition
- **BlockTuples:** $\emptyset$

5.2.2 Live-Range Splitting

For an example of how GRC can aid in the solution of specific data-flow problems, we examine an application of live-range splitting for register allocation. The method advocated by Kolte and Harrold relies on detecting ranges (a set of tuples) from each variable use V to its downward-exposed reference, if it exists [KH93]. Their technique iterates a standard set of data-flow equations until a fixed point is reached. They compute the set load-range[a,b] for V as follows (where a is a definition or use of V and b is a use of V):

$$\text{load-range[a,b]} = \{\textit{tuples } t \mid t \in \textit{reaching}(\text{a}) \wedge t \in \textit{reachable}(\text{b})\} \tag{5.1}$$

In Equation 5.1, $t \in \textit{reaching}(\text{a})$ means that a is the downward-exposed reference of V at t and $t \in \textit{reachable}(\text{b})$ means that b is the upward-exposed use of V at t. We note that this equation contains an expensive combining operation, and Equation 5.1 is applied to all tuples for all possible pairings of a and b. Examine Figure 5.1, which has two distinct load-ranges, $A \rightarrow C \rightarrow D$ ($[A, D]$) and $B \rightarrow C \rightarrow D$ ($[B, D]$).

This situation is ideally suited for reference chaining. We invoke the GRC algorithm on the Problem of downward-exposed references with these parameters:

- **Direction:** forward
- **RefLink:** {use}

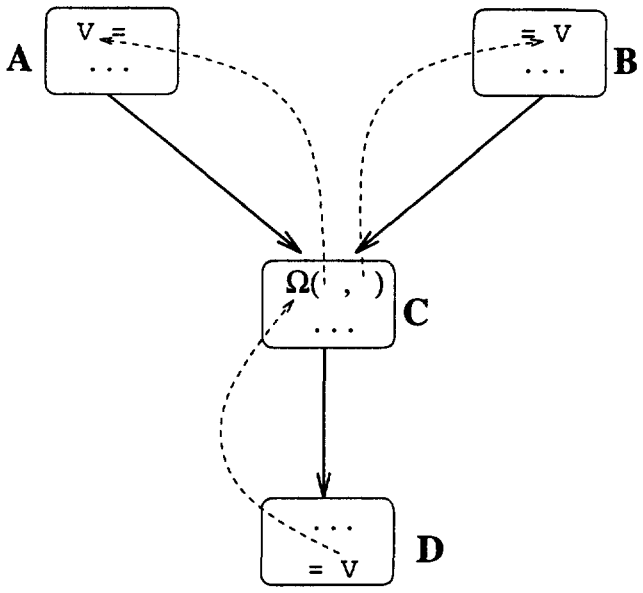


Figure 5.1 Computing load-ranges with GRC graphs

- **RefTuples:** any use or definition
- **BlockTuples:** $\emptyset$

From each use of V, tuples are processed in reverse order within a basic block node, adding it to the load-range until reaching *link(V)*. Additional ranges are generated when an Ω-function is encountered, creating a separate load-range for each argument. An undefined variable use is easy to spot since its *link* will be $\oslash$. For Figure 5.1, we start at the use of V in D and add all tuples to the load-range until we reach *link(V)*, which is an Ω-function in this case. Now, two load-ranges are created, each extending the range which already exists until its *link(V)* is reached. This results in load-ranges $[A, D]$ and $[B, D]$.

The use of GRC should improve the efficiency of load-range identification in two ways: (1) eliminating expensive operations, and (2) only considering range [a,b] that may be nonempty, instead of all possible pairings, which may often result in an a and b that have no path between them.

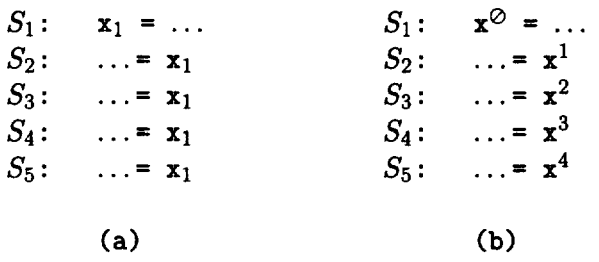


Figure 5.2 Comparing (a) FUD chains with (b) complete Reference Chaining

5.2.3 Other Applications.

For several of the algorithms we develop in the next chapter we need to solve the *reaching uses* problem: at each point what is the downward-exposed use, if it exists? Reaching uses is also a forward problem, with the details provided in Chapter 6. The examples of GRC that we have seen so far are forward problems. Chapter 7 will examine a very common backward problem, which we solve using GRC: live variable analysis.

We may inquire as to the reason for specializing reference chains. Why not, for forward problems, chain every reference to its nearest upward-exposed reference? Construction would depend upon these parameters for GRC, with the Problem being downward-exposed references in this case:

- **Direction:** forward
- **RefLink:** {use,def}
- **RefTuples:** any use or definition
- **BlockTuples:** $\emptyset$

While this may be the method of choice if the chains are used for multiple purposes, specialization creates additional sparsity. For example, in Figure 5.2(a) the use of **x** at S_5 is directly linked to the definition at S_1. However, if we chained every reference to its nearest downward-exposed reference as in (b) (where the superscript represents the line number of the next downward-exposed reference), the use of **x** at S_5 would have to follow four *links* to reach its reaching definition. Thus, each problem should be carefully studied in order to determine what information is needed. This information will result in Reference Chains which are as efficient as possible.

Chapter 6

Scalar Data Dependence

Precise value-based data dependence analysis for scalars is useful for advanced compiler optimizations. The new method presented in this chapter for flow and output dependence uses FUD chains. It is precise with respect to conditional control flow and dependence vectors. Our method detects dependences which are independent with respect to arbitrary loop nesting, as well as loop-carried dependences. If a loop-carried dependence is known to be from the previous iteration, we say its *distance* is 1 (where the distance of a dependence is the number of iterations between the source and sink of the dependence). If the distance cannot be determined exactly a dependence *direction* may be used, where direction '<' means any previous iteration. This precision cannot be achieved by traditional analysis, such as dominator information or reaching definitions.

To compute anti- and input dependence, we need to solve the reaching uses problem. A variant of reference chains is employed, Factored Redef-Use chains, in which each definition's *link* points to the closest downward-exposed reference. We are not aware of any prior work that explicitly deals with scalar data dependence utilizing a sparse graph representation.

6.1 An Introduction to Scalar Data Dependence

Data dependence analysis is usually presented in terms of array references in nested loops. A great deal of work has also been done to find dependence due to pointer aliasing. Little has been written about data dependence analysis for scalar references, except to refer to standard data-flow analysis [Tse93], to treat a scalar as a degenerate array [Fea91], or to use simple methods based on the dominator relationship or the syntactic structure of the program.

We begin with definitions of the four types of address-based dependences. A *flow*

dependence (sometimes referred to as a *true dependence*) (δ^f) appears between a definition of a variable (scalar, array element, or other memory location) and a use of that variable when the definition precedes the use in execution. An *output dependence* (δ^o) appears between two definitions of a variable when one definition precedes the other in execution. In a loop, the two definitions might each precede the other, so there can easily be a cycle of dependence. In fact, in a loop there may be an output dependence from a definition to itself. An *anti-dependence* (δ^a) appears whenever a use of a variable precedes a definition of that variable in execution. Finally, an *input dependence* (δ^i) appears between two uses of a variable. An input dependence does not represent potential memory conflicts as do the other types of dependences, but can be useful for certain optimizations.

The compiler can only approximate the actual dependence relations in a program, since it does not know the actual paths that will be taken in the control flow graph of the program. Thus, the compiler must conservatively assume that a flow dependence might exist whenever there is a path from a definition to a use, and so on for the other types of dependence. These common address-based definitions of flow, output, anti-, and input dependence can be found in many references [All83, AK87, Wol82, WB87].

Value-based dependence relations are a subset of the address-based dependence relations [Mas94]. The difference is explained by a simple example:

S_1: `A = B - 1`
S_2: `B = A + 1`
S_3: `A = B + C`
S_4: `B = B / A`

The address-based definition of dependence includes S_1 δ^f S_4 for `A` and S_1 δ^a S_4 for `B`. Since `A` is assigned a new value in S_3, statement S_4 cannot use the value assigned in S_1. Similarly, since S_2 assigns a new value to `B`, the assignment in S_4 cannot overwrite the value that was used in S_1. Thus, these two dependence relations are unnecessary. We define value-based dependence relations as follows: A flow dependence appears between a definition of a variable and a use of that variable when the definition precedes the use in execution, and the use fetches the value that was stored at the definition. An output dependence appears between two definitions of a variable when one definition precedes the other in execution, and the second overwrites the value stored at the first. An anti-dependence appears between a use and a definition of a variable whenever the

use precedes the definition in execution, and the definition overwrites the value that was fetched at the use. An input dependence appears between two uses of a variable, if there is no killing definition of that definition between the two uses. Essentially, a value-based dependence cannot reach past a killing definition of a variable. As always, the compiler can only compute an approximation to the actual value-based dependence relations.

In this chapter we present a new approach to finding data dependence for scalar variables. Our approach has several features, one of which is that it computes value-based dependence, not just address-based dependence. Value-based dependence is more precise [Mas94]; using value-based dependence reduces the number of dependence relations and may allow more optimizations. Another feature is that our method computes precise dependence distance, when precision is possible, or imprecise dependence vectors otherwise. Precise dependence distance is important for many optimizations, such as instruction scheduling, software pipelining and parallelization. For instance, knowing that the dependence distance is precisely one may simplify communication address calculation on a parallel machine, since the source and sink of the dependence are likely nearest neighbors. In other cases, *privatization* of scalar variables is possible when they are detected as being involved in only loop-independent dependences [BCFH89, Tse93]. In this example

```
loop
   if TEST then
      x = ...
   else
      x = ...
   endif
   ... = x
endloop
```

all flow dependences for x are loop-independent, so x can be privatized for each loop iteration. We note that privatizing x in this example also breaks the loop-carried anti-dependence.

Our method is precise in the presence of conditional control flow, given that the analysis is currently *path-insensitive*: it does not attempt to evaluate predicates to determine the paths that will be taken during execution. We show that simple analysis based on the dominator relationship cannot take into account conditional control flow precisely.

Since our analysis is based on the CFG, it is precise even for unstructured programs. In some sense, our technique gives the advantages of syntax-based analysis for unstructured programs, just as interval methods do for data-flow analysis.

Using value-based dependence relations is especially important within loops. Most dependence representations in loops use some abstraction to describe the iterations where the source and target of the dependence occur. As usual, we assume each iteration of a loop is identified by an integral *iteration vector* (often the index variable values). For example, vector (2,1) would refer to the iteration instance of `i = 2, j = 1` for a doubly nested loop with `i` being the outer and `j` being the inner loop index variable.

One common dependence abstraction is a distance vector, which is the vector difference between the iteration vectors of the source and target iterations; if there is a dependence from iteration $\mathbf{i}^s$ to iteration $\mathbf{i}^t$, the dependence relation has distance $\mathbf{d}$ if $\mathbf{i}^s + \mathbf{d} = \mathbf{i}^t$. Sometimes an exact distance cannot be computed; in that case, a less precise abstraction is used, called a direction vector [Wol78, Wol89]. The direction vector is a vector of relations from the set $\{<, =, >, \leq, \neq, \geq, \star\}$; if there is dependence from iteration $\mathbf{i}^s$ to iteration $\mathbf{i}^t$, the dependence relation has direction Θ if $i_k^s \ \theta_k \ i_k^t$, for loop nest level k. We allow a generalized *dependence vector*, where each element is either an integer value, if the exact distance for that loop nest level is known, and a direction relation if not. We could instead find the maximum and minimum dependence distance for each loop nest level; exact distance would be represented when the maximum equaled the minimum, and imprecise information would be represented with maximum distance of $-\infty$ or $+\infty$. However, our method suffices for the scalar analysis presented here.

The detection of scalar dependences within a loop requires careful analysis to get the precise dependence vector. We explain using the following loop:

```
S1:  T = 0
S2:  I = 1
S3:  loop
S4:     I = I + 1
S5:     C[I] = V + T
S6:     if TEST[I] then
S7:        T = B[I+1]
S8:        V = T + 1
S9:     else
S10:       V = B[I]
S11:    endif
S12: endloop
```

There is a *loop-carried* flow dependence relation $S_7\ \delta^f\ S_5$ for variable T, and two more *loop-carried* flow dependence relations $S_8\ \delta^f\ S_5$ and $S_{10}\ \delta^f\ S_5$ for variable V. However, the dependence distance for the V dependences is exactly one; since V is assigned on every iteration, any loop carried dependence relation must come from the previous iteration. We call this situation *loop-carried(1)*, to identify the dependence carried by a loop with a distance of one, and it is denoted as $S_8\ \delta^f_{(1)}\ S_5$ for this example. For the T dependence carried by the loop, however, the distance can be any positive integer, since T might not be assigned on every iteration. We call this case *loop-carried(<)*, and is denoted $S_7\ \delta^f_{(<)}\ S_5$. Finally, the flow dependence $S_1\ \delta^f\ S_5$ for T is not carried by any loop. It is *loop-independent.* A dependence within the same loop, but not carried by any loop, such as $S_7\ \delta^f\ S_8$ for T, is also loop-independent. We denote a loop-independent dependence using the common terminology δ_∞, such as $S_1\ \delta^f_\infty\ S_5$.

Simple dominator-based analysis will find precise dependence relations when the assignment dominates the use in the body of the loop (to give loop-independent dependence) or when the assignment dominates the back edge (to give loop-carried dependence with distance one). In this example, however, neither S_8 nor S_{10} dominates the loop back edge, so such simple analysis will fail. We will show this situation occurs frequently in our benchmark programs.

Our analysis is based on FUD chains from Chapter 3. Other intermediate representations, such as dependence flow graphs [JP93] or the program dependence web [BMO90], could also be used with similar algorithms; those representations contain enough information to find the actual dependences, though they do not represent the dependence

relations explicitly. We note that reaching definitions are not sufficient, since they do not take into account how definitions on conditional branches are carried by loops.

The algorithm for finding flow dependence starts at a use and follows the chain of *links* to all reaching definitions. We want more information than just the reaching definitions; we also want the most precise dependence distance information possible. This method is described in Section 6.2, along with experimental data from common scientific benchmarks. We rely on the fact that each natural loop header has exactly two predecessors: the preheader and postbody. Control flow merges at loop headers (μ-functions) are treated specially: analysis at this point lets us know whether the dependence is loop-carried, loop-entering, or loop-exiting dependence.

The algorithm to find output dependence is essentially the same as that for flow dependence. The only difference is that the initial call is from definition sites, rather than usage sites, of a variable. This algorithm is presented in Section 6.2.5.

However, the algorithm to find anti- and input dependence cannot use FUD chains. The information needed for these dependences is the set of uses that are overwritten by a definition. We generalize the FUD chain construction algorithm to create the reference chaining algorithm in Section 5.1, then use this algorithm to implement Factored Redef-Use (FRDU) chains. FRDU chains link each definition to the most recent downward-exposed use, and that use to the next most recent downward-exposed use. Their applications to input and anti-dependence is described in Section 6.3.

6.2 Flow and Output Dependence

6.2.1 Necessary Ingredients

At first glance, it may appear that scalar flow dependence information could be gathered by applying traditional data-flow techniques, *e.g.*, dominator analysis and reaching definitions. When the source of the dependence dominates the sink, or the loop postbody does not lie on any path from the definition to the use, the dependence must be in the current loop iteration, hence it would be loop-independent. If the definition dominates the postbody, the distance would be one, while remaining cases would indicate a direction of $<$ or unknown.

However, these observations are not sufficient to capture either precise distance nor correct classification. Referring to the last example presented in §6.1, and represented graphically but in simplified form for variable `V` as Figure 6.1(F), we notice that the flow

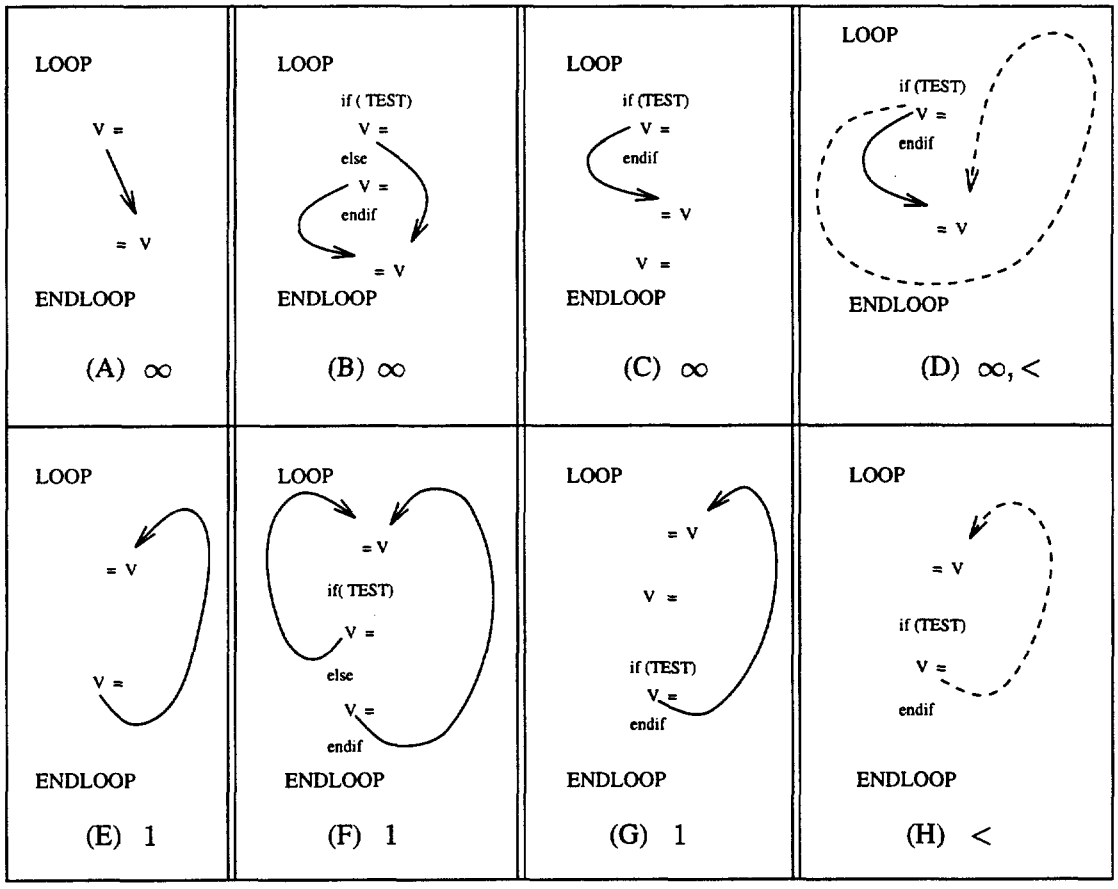


Figure 6.1 The eight kinds of scalar flow dependence that occur within a single loop, grouped by related pairs. Solid lines represent loop-independent(∞) or loop-carried(1) dependences, while a dotted line represents a loop-carried(<) dependence.

dependence distance from either definition of `V` to the use of `V` is precisely one. We also note that in this example neither definition of `V` dominates the postbody, but since all paths through the postbody contain a definition to `V`, the distance must be one.

Thus, to correctly classify flow dependences, we follow the chain from each use, which will lead to either a definition, a ϕ-function, or a μ-function. (We assume that loop-header ϕ-functions have been renamed μ-functions, as per GSA form.) When encountering a ϕ-function, we follow the chains of each argument. Intuitively, a loop-independent flow dependence will be discovered by following use-def *links* in the current loop body (or to a definition site outside the loop in which the use occurs), while loop-carried dependences must always flow through a μ-function.

When a chain reaches a μ-function, we conceptually continue to follow the *links*

around the loop. To prevent infinite cycling, each μ-function has a flag, *self* (initially set to *false*), which indicates whether following chains around the loop can reach back to the μ-function. If the *self* flag is set, a ϕ-function (or a nonkilling definition) must have been encountered while following chains for that loop instance; this condition indicates a conditional branch that may or may not be taken during any particular iteration, but where on at least one of the branches there exists a chain reaching the μ-function. Thus, while the dependence is loop-carried, we do not know its precise distance, so we denote its direction as (<). If the *self* flag is not set, then all paths must encounter a killing definition for the variable being analyzed; this condition means that the flow dependence must be to the subsequent iteration, and its distance is 1.

6.2.2 Algorithm 6.1: Precisely Detecting Scalar Flow Dependence

An algorithm for detection of scalar flow dependences within a single loop has been presented previously [SGW94]. To extend this algorithm to nested loops, several issues need to be addressed. First, we must provide a recursive routine, to allow arbitrary nesting. Second, distance and direction of the dependence must be accurate in terms of all loops containing the dependence. This second point implies that a dependence relation between two references may in fact be more than one dependence: it can be a dependence with respect to an inner loop as well as another dependence with respect to an outer loop.

Data Structures

We use the following data structures for the algorithms in this chapter (where $f[n]$ refers to the n^{th} argument of ϕ- or μ-function f):

$\Longrightarrow$ μ-functions:

- *self* – A flag representing whether a μ-function can transitively reach itself. Initialized to *false*.
- *Reaching_set* – A set of definitions which can be reached by following *chain*(μ[2]). Initialized to $\emptyset$.

$\Longrightarrow$ loops:

- *nest*(*loop*) – A function that returns the nest level of *loop*

- *loop(ref)* – A function that returns the innermost loop containing *ref*
- *nl(ref)* – A function that returns *nest(loop(ref))*
- *common(ref1,ref2)* – A function that returns the most deeply nested loop containing both *ref1* and *ref2*

$\Longrightarrow$ references:

- *marked(ref)* – A value such that *marked(ref)* $= u$ indicates *ref* has already processed use u in *Find_Dependence()*. Initialized to NULL.
- *Reached(ref)* – A set where $r \in$ Reached(*ref*) indicates *ref* has already processed μ-function r in *Find_Reaching*. Initialized to $\emptyset$

For each dependence (excluding loop-independent dependences) we build a *dependence vector* to describe the most precise information available for all loops, from the outermost loop containing the dependence to the most deeply nested loop containing both references of the dependence. A *nest level* is associated with each loop, starting at 1 for outermost loops.

When building the dependence vector, we must first find the most deeply-nested loop (*com*) containing both the source and sink of the dependence, since we do not consider loops at a nest level greater than *com*. Next, the nest level of *com* is compared to the nest level of the loop containing the μ-function being processed (*mu_loop*), since all loop-carried dependences are as a result of processing μ-functions, as noted above. If the nest levels of *com* and *mu_loop* are equal, then the *mu_loop* will carry the dependence. If the nest level of *com* is greater than the nest level of *mu_loop*, *mu_loop* still carries the dependence, but loops between *mu_loop* and *com* have unknown information, and their entry into the dependence vector is denoted $\star$. Finally, if the nest level of *mu_loop* is greater than the nest level of *com*, *mu_loop* clearly does not carry the dependence, hence the dependence is loop-independent(∞).

6.2.3 Discussion of Algorithm 6.1 for Flow Dependence

Figures 6.2 and 6.3 provide the basis for Algorithm 6.1, which detects scalar flow dependences. Algorithm 6.1 first calls *Find_Dependence()* on each scalar use, its *link*, and type f (flow dependence). Encountering a scalar definition results in a loop-independent dependence, while a ϕ-function results in a recursive call on the *links* from each argument. When a μ-function is encountered, the set of references reachable within that

Given: A program converted to FUD chain form
Auxiliary data structures initialized
Do: $\forall$ scalar uses U,
Find_Dependence(chain(U), U, f)
Result: A list of statement-based flow dependences, and a dependence vector for any dependence which is not loop independent

Algorithm 6.1 Identifying scalar flow dependences

```
1:  Find_Dependence( d, u, t )
2:      if type( d ) = source then return endif
3:      if marked( d ) = u then return endif
4:      marked( d ) ← u
5:      if d is a μ-function then
6:          Find_Dependence( Chain(d[1]), u, t )
7:          if Reaching_Set(d) ≠ ∅ then
8:              Find_Reaching( Chain(d[2]), d )
9:          endif
10:         for c ∈ Reaching_Set( d ) do
11:             if self( d ) = true then
12:                 dep_vec ← Build_Vector( d,c,u,< )
13:             else
14:                 dep_vec ← Build_Vector( d,c,u,1 )
15:             endif
16:             output S_num(c) δ^t_(dep_vec) S_num(u)
17:         endfor
18:     else if d is a φ-function then
19:         for each argument j of d do
20:             Find_Dependence( Chain(d[j]), u, t )
21:         endfor
22:     else
23:         output S_num(d) δ^t_∞ S_num(u)
24:         if d is a nonkilling definition then
25:             Find_Dependence( Chain(d), u, t )
26:         endif
27:     endif
28: end Find_Dependence
```

Figure 6.2 Procedure for identifying scalar dependences

```
29: Find_Reaching( d, f )
30:         if d = f then
31:             self( f ) ← true
32:             return
33:         endif
34:         if f ∈ Reached( d ) then return endif
35:         Reached( d ) ← Reached( d ) ∪ f
36:         if  d is a μ-function then
37:             Find_Reaching( Chain(d[1]), f )
38:             Find_Reaching( Chain(d[2]), d )
39:             add Reaching_Set( d ) to Reaching_Set( f )
40:         else if d is a φ-function then
41:             for each argument j of d do
42:                 Find_Reaching( Chain(d[j]), f )
43:             endfor
44:         else
45:             add d to Reaching_Set( f )
46:             if d is a nonkilling definition then
47:                 Find_Reaching( Chain(d), f )
48:             endif
49:         endif
50: end Find_Reaching


51: Build_Vector( func,def,ref,entry )
52:         com ← common( def, ref )
53:         if nest(com) = nl(func) then
54:             dep_vec[1 ... nl(func) - 1] ← 0
55:             dep_vec[nl(func)] ← entry
56:         else if nest(com) > nl(func) then
57:             dep_vec[1 ... nl(func) - 1] ← 0
58:             dep_vec[nl(func)] ← entry
59:             dep_vec[nl(func) + 1 ... nest(com)] ← ⋆
60:         else
61:             dep_vec ← ∞
62:         endif
63: end Build_Vector
```

Figure 6.3 Routines for *Find_Reaching* and *Build_Vector*

loop are calculated with an on-demand call to *Find_Reaching()*, which determines what definitions of a particular variable can reach around the loop. Then, each element in this set results in a dependence with a dependence vector computed by *Build_Vector()*.

Since μ-functions are distinguished from ϕ-functions occurring at other merge points in the CFG, cycles in the call graph for *Find_Dependence()* are fairly rare. However, redundant calls may occur in two ways. First, there may be an irreducible flow graph. Second, a nonkilling definition along one branch of a conditional for the variable being processed may result in repetitious calls to *Find_Dependence()* for the same use of that variable. Since *Find_Dependence()* processes one use of a variable at a time, we associate a *marked* field with each reference point. By checking this field, we eliminate extra calls due to nonkilling definitions and potential infinite loops as a result of irreducible flow graphs. We illustrate how the *marked* field is used with this example:

```
S1:   V1 = ...
S2:   if ( P ) then
S3:        call sub( V2 )
S4:   endif
S5:   V3 = φ( V2, V1 )
S6:   ...= V3
```

We call *Find_Dependence(chain(U_6^V),U_6^V,f)* to find the flow dependences for this code fragment. Since *chain(U_6^V)*, equivalent to D_5^V, is a ϕ-function, recursive calls are made on both arguments. The call to *Find_Dependence(D_1^V,U_6^V,f)* finds the dependence $S_1\ \delta^f\ S_6$, while the call to *Find_Dependence(D_3^V,U_6^V,f)* finds the dependence $S_3\ \delta^f\ S_6$. However, D_3^V is a nonkilling definition, so another call to *Find_Dependence(D_1^V,U_6^V,f)* is made from line 25 of *Find_Dependence()*. Since U_6^V has already been *marked* at D_1^V, this call will immediately return, preventing a redundant dependence from being detected.

The situation is similar for the routine *Find_Reaching()*, except that the *Reaching_Set* is associated with particular μ-functions, and not in terms of a processed usage site. In this case we keep, for each definition site, a list of μ-functions that have already processed that site.

We illustrate Algorithm 6.1 with two examples. First, we show a single loop in which the dependence distance of one cannot be ascertained by conventional analysis:

```
S1:   T1 = ...
S2:   loop
S3:        T2 = μ(T1,T5)
S4:           ...= T2
S5:        if TEST then
S6:           T3 = ...
S7:        else
S8:           T4 = ...
S9:        endif
S10:       T5 = φ(T3,T4)
S11:  endloop
```

For the use of T at S_4 a call is made to *Find_Dependence(chain(U_4^T),U_4^T,f)*, which is equivalent to *Find_Dependence(D_3^T,U_4^T,f)*. A further call to *Find_Dependence(D_1^T,U_4^T,f)* is made on the first argument of D_3^T, since it is a μ-function, resulting in this flow dependence from line 23 in *Find_Dependence()* (Figure 6.2):

$$S_1 \ \delta_{\infty}^{f} \ S_4$$

A call to *Find_Reaching(D_{10}^T,D_1^T)* is made on the second argument of D_3^T, since this is the first time this μ-function is encountered. *Find_Reaching()* recurses on the two arguments of D_{10}^T, returning when finding the killing definitions of T at S_6 and S_8. Thus, *Reaching_Set(D_3^T)* = $\{D_6^T, D_8^T\}$, and *self(D_3^T)* stays at *false*. Lines 10 and 14 of *Find_Dependence()* output these two flow dependences of distance one:

$$S_6 \ \delta_{(1)}^{f} \ S_4 \quad \text{and} \quad S_8 \ \delta_{(1)}^{f} \ S_4$$

As previously noted, simple dominator analysis will not discover these dependences of distance 1, since neither killing definition of T within the conditional construct dominates the postbody of the loop, although jointly they do so.

We now show how Algorithm 6.1 operates for scalar flow dependences in nested loops with this example (in which we assume the exit is at the top of each loop):

```
S1:  V1 = ...
S2:  loop1
S3:     V2 = μ(V1, V3)
S4:     loop2
S5:        V3 = μ(V2, V5)
S6:        if TEST then
S7:           V4 = ...
S8:        endif
S9:        V5 = φ(V3, V4)
S10:       ... = V5
S11:    endloop2
S12: endloop1
```

Within the above loop, the only use of V occurs at S_{10}. Thus, we make the call *Find_Dependence*($chain(U_{10}^V)$,U_{10}^V,f), which is equivalent to *Find_Dependence*(D_9^V,U_{10}^V,f). Since D_9^V is a ϕ-function, recursive calls are made to *Find_Dependence*(D_5^V,U_{10}^V,f) and *Find_Dependence*(D_7^V,U_{10}^V,f). From the second of these calls we get the loop-independent dependence

$$S_7 \ \delta_{\infty}^{f} \ S_{10}$$

from line 23 of Figure 6.2.

The other call, *Find_Dependence*(D_5^V,U_{10}^V,f) points to a μ-function, thus *Find_Dependence*(D_3^V,U_{10}^V,f) is called on the first argument in S_5, while the second argument invokes a call to *Find_Reaching*(D_9^V,D_5^V), setting the *self* flag for D_5^V and discovering its *Reaching_Set* = $\{D_7^V\}$. Lines 10 - 12 and 16 of Figure 6.2 give us the dependence

$$S_7 \ \delta_{(0,<)}^{f} \ S_{10}$$

Finally, the call to *Find_Dependence*(D_3^V,U_{10}^V,f) from the first argument of the μ-function at S_5 is a reference to another μ-function, thus *Find_Dependence*(D_1^V,U_{10}^V,f) is called on the first argument in S_3, and *Find_Reaching*(D_5^V,D_3^V) is called by the second argument, setting the *self* flag for D_3^V and discovering its *Reaching_Set* = $\{D_7^V\}$ (by merging with the *Reaching_Set* of D_5^V). We then output these last two dependences:

$$S_7 \ \delta_{(<,*)}^{f} \ S_{10} \quad \text{and} \quad S_1 \ \delta_{\infty}^{f} \ S_{10}$$

6.2.4 Measuring Algorithm 6.1 on Scientific Benchmarks

How often do the cases in Figure 6.1 occur? To discover the usefulness of our method, we ran our algorithm over the scientific benchmarks contained in the Perfect Club, RiCEPS, and Mendez suites. In order to keep the investigation at a level that is easy to analyze, this set of data only counted flow dependences in which the source and sink of the dependence were within the same inner loop. As shown in Figure 6.1, there are eight categories of dependences within a single loop, although cases F and G are statistically grouped together since they are semantically equivalent. We show F and G in Figure 6.1 separately because the eight cases form a coherent pattern of matching pairs.

Table 6.1 shows the result of our analysis. To be widely useful, cases other than those where the source dominates the sink (case A) need to occur with some frequency. Case F corresponds to the second example from the previous subsection, where no definition dominates the use, but all paths reaching the postbody contain a definition. This case comprises 7% of all loop-carried(1) dependences where the source and sink of the dependence lie within the same inner loop. In some codes, such as `spice` and `flo52` from the PERFECT Club suite or `sphot` from the RiCEPS suite, the percentage ranges from 17% to 68%. These results demonstrate that traditional data-flow techniques are not sufficient to achieve the precision captured with our algorithms.

Case H also includes subroutine calls where we know no information about the arguments, such as that available from interprocedural mod/ref analysis. Any particular iteration of this loop

```
loop:
   call f(x)
endloop
```

may modify `x`, hence it will be detected as a $\delta^{f}_{(<)}$ dependence. In terms of Algorithms 6.1 and 6.2, the reference to `x` represents a nonkilling definition.

Note that D is the case that will result in both a loop-independent and loop-carried(<) dependence. This case is correctly analyzed by our algorithm, since a ϕ-function for `V` will be placed immediately after the `endif` statement, resulting in recursive calls to the *Find_Dependence()* routine; one will discover the loop-independent flow dependence, and the other will discover the loop-carried(<) flow dependence. An examination of Table 6.1 reveals that some codes possess structure in which this class of dependence is quite significant.

routine	*lines*	*# loop-independent*				*# loop-carried* (<)		*# loop-carried* (1)	
		A	*B*	*C*	*D*	*D*	*H*	*E*	*F & G*
PERFECT club									
adm	6106	3439	163	1	92	92	2302	149	16
arc2d	3965	2600	6	0	2	2	193	8	0
bdna	3978	3465	27	6	6	6	57	112	8
dyfesm	7609	929	7	11	129	129	103	82	0
flo52	1987	1620	20	10	2	2	81	180	44
mdg	1239	728	4	1	1	1	106	157	2
mg3d	2813	1625	19	7	4	4	124	1420	4
ocean	4344	1421	23	3	310	310	1509	118	3
qcd	2277	570	50	0	0	0	339	191	0
spec77	3886	1807	36	5	24	24	263	780	9
spice	18522	10456	3184	38	907	907	1810	2075	435
track	3785	624	20	0	3	3	72	26	3
trfd	486	197	9	0	0	0	10	39	0
RiCEPS									
boast	8068	5499	598	21	330	330	1188	582	40
ccm	23557	5286	142	4	12	12	555	1200	11
hydro	13050	1667	167	6	0	0	221	31	0
linpackd	798	165	0	0	0	0	26	20	0
simple	1314	1065	6	1	0	0	11	23	1
sphot	1145	480	43	0	0	0	216	13	27
wanal1	2110	925	16	0	253	253	113	125	6
wave	7521	3919	57	6	24	24	162	288	8
Mendez									
baro	984	416	0	0	0	0	17	44	0
euler	1202	484	48	0	6	6	45	37	2
mhd2d	928	359	0	0	0	0	3	23	6
shear	916	561	3	6	0	0	5	34	4
vortex	711	495	14	0	7	7	116	19	0
Total	123301	50802	4662	126	2112	2112	9647	7776	620

Table 6.1 A count of the different kinds of scalar flow dependences detected in scientific codes, classified according to the type of loop structure from Figure 6.1. The source and sink of the dependence are within the same inner loop.

Given: A program converted to FUD chain form
Auxiliary data structures initialized
Do: ∀ scalar definitions D
Find_Dependence(chain(D), D, o)
Result: A list of statement-based output dependences, and a dependence vector for any dependence which is not loop-independent

Algorithm 6.2 Identifying scalar output dependences

It is interesting to wonder what percentage of the case A loop-independent dependences are uses within the loop of the loop index variable. We measured this relationship, and found a low of 6% in `spice` from the Perfect Club suite, to a high of 87% in `baro` from the Mendez suite. The mean over all benchmarks was 16%. This figure is somewhat deflated due to the fact that 18% of all scalar flow dependences came from `spice`. The median over all benchmark programs was 55%.

6.2.5 Algorithm 6.2: Output Dependence

Figures 6.2 and 6.3 also provide the method for computing scalar output dependence. The differences are that definitions are used for input, and we pass the dependence type o (output dependence) to *Find_Dependence()*. Computing output dependence for scalars is a fairly trivial modification to Algorithm 6.1, since def-def *links* have been inserted as a component of FUD chains.

To illustrate Algorithm 6.2, we present a simple example. The following loop has two definitions of `W`, at S_4 and S_7.

```
S1:  loop
S2:      W2 = μ(W0,W7)
S3:      if TEST then
S4:          W4 = ...
S5:      endif
S6:      W6 = φ(W4,W2)
S7:      W7 = ...
S8:  endloop
```

The first call is to *Find_Dependence*($chain(D_4^W)$,D_4^W,o), which is equivalent to *Find_Dependence*(D_2^W,D_4^W,o). Since D_2^W is a μ-function, a call to *Find_Dependence* is made on its first argument, which we shall consider no further here, restricting our attention to dependences contained within the loop body. The second argument invokes a call to *Find_Reaching*(D_7^W,D_2^W), discovering D_2^W's *Reaching_Set* = $\{D_7^W\}$. Since the *self* flag is not set, we get the following loop-carried dependence:

$$S_7 \ \delta^o_{(1)} \ S_4$$

The second definition generates the call to *Find_Dependence*(D_6^W,D_7^W,o), which recursively follows chains to S_4 and S_2, resulting in the two dependences (*Find_Reaching* for S_2 has already been calculated):

$$S_4 \ \delta^o_{\infty} \ S_7 \quad \text{and} \quad S_7 \ \delta^o_{(1)} \ S_7$$

6.2.6 Complexity Analysis

A useful measure of complexity for the scalar dependence algorithms is the number of *links* followed during the analysis to find the dependence. We counted the average number of *links* followed for each detected dependence on all the benchmark programs. The information is displayed in Table 6.2 for both flow and output dependence. As expected, most programs exhibit a structure where following several *links* in a chain is sufficient to detect a flow dependence. A few programs displayed a much more complex structure. The number of *links* followed on average in ocean is very high due to the combination of large numbers of subroutine calls and many global variables, as noted in §3.1.5 and §4.7.4.

When the dependence algorithm executes, all the *links* are not actually traversed as depicted in Table 6.2, since multiple uses of the same μ-function do not require a recomputation of the *Reaching_set*. The information is effectively memoized at each μ-function.

We also show the total number of dependences detected for these cases over the benchmarks. These statistics are a superset of those shown in Table 6.1, since they include all dependences, not just those where the source and sink are within the same inner loop.

Program	*Total # Flow Dependences*	*Ave. # Links Traversed*	*Total # Output Dependences*	*Ave. # Links Traversed*
PERFECT club				
adm	12055	9.0	8357	11.9
arc2d	4703	1.2	1284	3.5
bdna	4808	1.4	2089	2.7
dyfesm	2203	1.9	885	3.4
flo52	4726	3.8	2323	7.3
mdg	1494	2.0	568	3.3
mg3d	6886	1.7	2142	3.0
ocean	6511	14.4	5121	5.7
qcd	2235	3.3	1331	4.7
spec77	5412	2.1	2031	3.9
spice	23780	2.4	18485	5.0
track	1064	1.8	442	3.6
trfd	539	2.1	330	4.3
RiCEPS				
boast	17534	5.6	9421	7.7
ccm	14345	1.7	5669	4.0
hydro	2987	1.7	1264	3.5
linpackd	544	4.2	324	6.9
simple	1742	1.2	479	3.2
sphot	1633	3.5	1832	7.5
wanal1	4787	3.8	1234	4.8
wave	7023	1.5	2009	3.2
Mendez				
baro	1052	1.5	213	3.6
euler	1026	1.6	332	3.3
mhd2d	739	1.4	364	4.1
shear	1277	1.4	397	4.3
vortex	818	3.6	493	4.8
Total	131932	3.8	69419	6.0

Table 6.2 Total number of scalar flow and output dependences and the number of links traversed

6.3 Anti- and Input Dependence

6.3.1 Building Chains With ϒ-Functions

As mentioned in §6.1, FUD chains do not provide the correct information to detect scalar anti- and input dependence. FUD chains are a sparse solution to the reaching definitions problem, while anti- and input dependence require information on reaching uses: at any point, what is the closest downward-exposed use for a given variable? To obtain this information, we need *links* to uses, merging this information at confluence nodes. We therefore use GRC Algorithms 5.1 and 5.2 to construct reference chains to solve the reaching uses problem. This structure, which we call *factored redef-use chains* (FRDU chains) links each definition to the closest preceding downward-exposed use, and each use to the next downward-exposed use. We call the merge operators for this problem ϒ-functions. The GRC algorithm builds FRDU chains to solve the Problem of reaching uses with the following parameters:

- **Direction:** forward
- **RefLink:** {use,def}
- **RefTuples:** any use or definition
- **BlockTuples:** any killing definition

Since ϒ-functions are a merge of upward-exposed uses, they are themselves considered uses of a variable. With reaching definitions, we number each definition uniquely, since it logically represents a new variable instance. Multiple uses, however, can repeatedly reference the same definition instance. Hence, for reaching uses we depict each definition with a superscript to the line number of the downward-exposed use that reaches that definition. Each use is superscripted by the line number of the next most recent downward-exposed use, if another use can reach without passing through a killing definition. The superscript $\oslash$ reflects the fact that there are no reaching uses at that point. We illustrate this idea in Figure 6.4. The code on the left of Figure 6.4 has its CFG shown on the right. The definition of T at S_6 is reached by the ϒ-function at S_5, which merges reaching uses along the control paths from nodes B and A. The path from B has a reaching use at S_3, while the path from A has no reaching use, denoted by $\oslash$. Note that for FUD chain form, no ϕ-functions would be placed at node C since it is not in the dominance frontier of A.

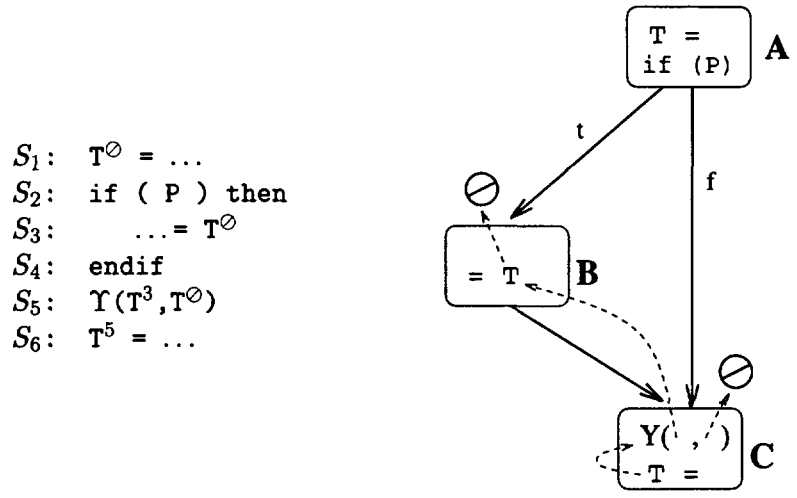


Figure 6.4 ϒ-functions merge downward-exposed reaching uses

6.3.2 Algorithms for Scalar Dependence Using ϒ-Functions

We can use the procedures in Figures 6.2 and 6.3, with several minor modifications, to construct the algorithms that detect scalar anti- and input dependence. We also need one additional data structure that identifies when an ϒ-function is in the header node of a loop, analogous to renaming loop-header ϕ-functions as μ-functions in GSA form:

lh(ϒ): returns *true* if an ϒ-function resides in the header node of a loop.

All references to μ-functions, specifically the data structures *self* and *Reaching_set* in Figures 6.2 and 6.3, now refer to *lh*(ϒ).

We also need to check if a *link* points to ⊘, in which case the routines immediately return. *Find_Dependence()* just needs to check for ⊘ instead of the tuple type *source*, while *Find_Reaching()* needs a conditional statement added. The following alterations are made to the procedures in Figures 6.2 and 6.3:

- line 2: **if** $d = \oslash$ **then** *return* **endif**
- line 5: **if** *d is a lh*(ϒ) **then**
- line 18: **else if** *d is an ϒ-function* **then**
- line 24: **if** *d is a use or nonkilling definition* **then**
- line 29a: **if** $d = \oslash$ **then** *return* **endif**
- line 36: **if** *d is a lh*(ϒ) **then**

Given: A program converted to FRDU chain form
Auxiliary data structures initialized
Do: $\forall$ scalar definitions D
Find_Dependence(chain(D), D, a)
Result: A list of statement-based anti-dependences, and a dependence vector for any dependence which is not loop-independent

Algorithm 6.3 Identifying scalar anti-dependences

```
S1:  T^⊘ = ...
S2:  loop
S3:     ϒ(T^⊘,T^9)
S4:     if (Q) then
S5:        T^3 = ...
S6:        ...= T^⊘ ...
S7:     endif
S8:     ϒ(T^6,T^3)
S9:     ...= T^8 ...
S10: endloop
```

Figure 6.5 FRDU chains for example loop.

- line 40: **else if** *d is an* Υ*-function* **then**
- line 46: **if** *d is a use or nonkilling definition* **then**

Figure 6.5 illustrates Algorithm 6.3 using FRDU chains for a simple loop. Placement of Υ-functions occurs at the loop header and at the **endif**. To find anti-dependence, Algorithm 6.3 is invoked on all scalar definitions D with their FRDU chains and parameter a for anti-dependence. In this example, there are two definitions, at S_1 and S_5. In the first case, there are no reaching uses, signified by the $\oslash$ superscript. The definition at S_5 is reached by any uses that reach the Υ-function at S_3. One of these *links* is again empty, since there are no uses outside the loop. The other *link* causes the compiler to find the set of uses in the loop that can reach the Υ-function, and a flag whether the Υ-function can reach itself. In this case, the *Reaching_Set* $= \{T^9, T^6\}$, and the *self* flag is set. Thus the algorithm will find the two loop-carried anti-dependence relations $S_6\ \delta^a_{(<)}\ S_5$ and

Given: A program converted to FRDU chain form
Auxiliary data structures initialized
Do: ∀ scalar Uses U
Find_Dependence(chain(U), U, i)
Result: A list of statement-based input dependences, and a dependence vector for any dependence which is not loop-independent

Algorithm 6.4 Identifying scalar input dependences

$S_9\ \delta^a_{(<)}\ S_5$, each with nonconstant distance.

In this example, the Υ-functions were placed in the same locations that the ϕ-functions would be placed, since there are distinct definitions and uses along each control path.

Input dependence is computed in the same way as anti-dependence, except that we start at variable uses and chain to reaching uses. Algorithm 6.4 provides the procedure to detect all statement-based input dependences, which is also based upon the modifications made to Figures 6.2 and 6.3. Detecting value-based input dependence can be useful for optimizing locality of reference, achieving better memory-hierarchy (i.e. cache) performance [Wol92a].

6.3.3 Experimental Results

We ran Algorithms 6.3 and 6.4 on all the benchmark programs. Total number of anti- and input dependences detected and the number of *links* followed is displayed in Table 6.3. The average number of *links* followed to find anti-dependences is inflated due to the nature of `ocean`: it was responsible for 25% of all anti-dependences, with an extraordinarily high number of *links* traversed. The number of input dependences and average number of *links* traversed is very large, as expected. Again, memoization at loop-header Υ-functions reduces the actual number of *links* followed by Algorithms 6.3 and 6.4 when they execute.

We experimentally compared the number of Υ-functions as a function of referenced variables (Figure 6.6) and program size (Figure 6.7). These graphs show that the growth in data structures is linear in both cases.

Since the set of nodes which contain ϕ- or μ-functions is a subset of those containing Υ-functions (all uses and definitions imply nonidentity transfer functions for reaching

Program	*Total # Anti-Dependences*	*Ave. # Links Traversed*	*Total # Input Dependences*	*Ave. # Links Traversed*
PERFECT club				
adm	13319	19.9	49907	14.6
arc2d	3368	5.7	48526	12.1
bdna	5586	10.3	36103	15.7
dyfesm	1821	11.0	13866	24.4
flo52	5339	19.9	43959	19.0
mdg	1705	8.8	11562	12.3
mg3d	6628	17.1	133018	28.3
ocean	57061	80.2	123808	88.9
qcd	2211	10.1	6460	9.9
spec77	3382	9.1	31297	10.3
spice	50222	13.2	279355	21.0
track	938	7.8	3469	7.1
trfd	553	6.3	1625	5.7
RiCEPS				
boast	18966	12.5	109327	14.6
ccm	12632	10.5	186704	19.4
hydro	10630	30.8	70622	37.9
linpackd	383	10.3	3244	16.6
simple	6694	28.6	53438	31.1
sphot	5861	18.3	11621	14.4
wanal1	7668	66.9	146104	84.7
wave	5248	7.7	72908	15.4
Mendez				
baro	368	6.6	10634	11.2
euler	423	4.9	4506	6.0
mhd2d	2477	23.8	16728	32.0
shear	804	7.7	24256	24.3
vortex	1334	11.8	3033	9.6
Total	225621	33.4	1496080	32.6

Table 6.3 Total number of scalar anti- and input dependences and the number of links traversed

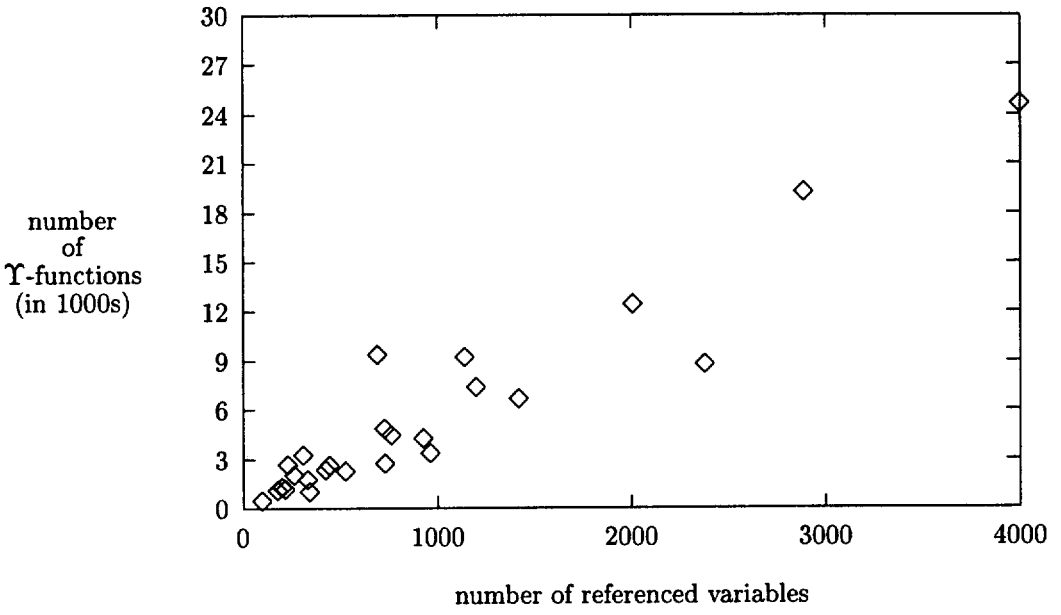


Figure 6.6 A comparison of Υ-functions to referenced variables in the benchmark programs

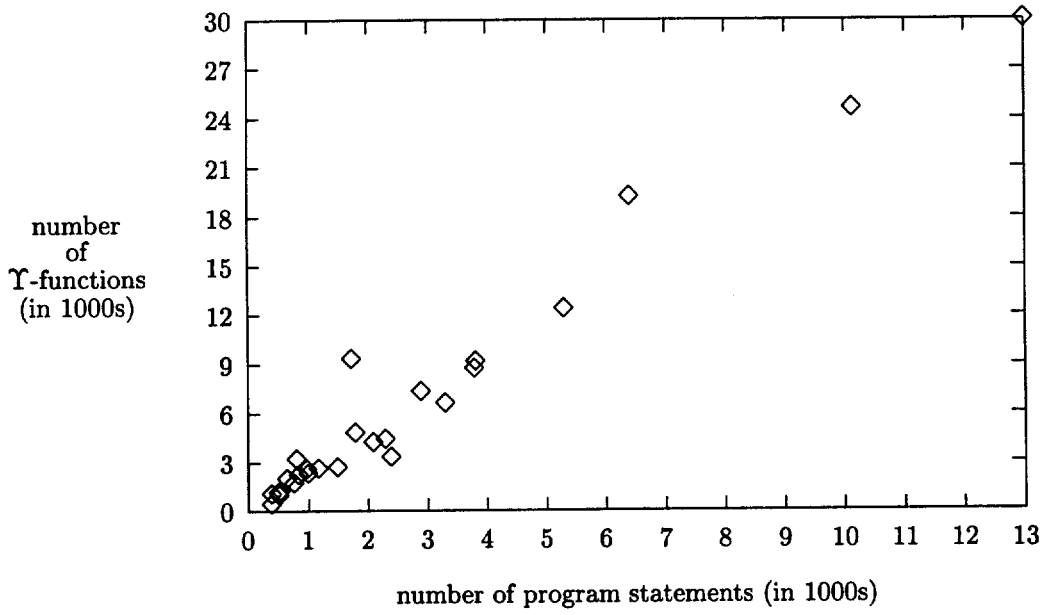


Figure 6.7 A comparison of Υ-functions to program statements in the benchmark programs

uses, but only definitions imply a nonidentity transfer function for reaching definitions), we were interested in the percentage increase from the number of ϕ-functions created with FUD chains to the number of Υ-functions created when constructing FRDU chains. The results are shown in Table 6.4, where we note that there are about 2.3 Υ-functions on average for every ϕ-function.

6.4 Extensions

There are several useful extensions to scalar dependence analysis. For example, precise information is available for dependences that only flow into the first iteration of a loop. This loop

```
S1:    W = ...
S2:    loop 1 to N
S3:       ...= W
S4:       W = ...
S5:    endloop
```

has a flow dependence from S_1 to S_3 on just the first iteration of the loop. In its current state this loop cannot be parallelized due to the loop-carried flow dependence from S_4 to S_3. Recognition of the dependence from S_1 to S_3 allows *loop rotation* to be performed, resulting in this equivalent form of the same loop:

```
S1:    W = ...
S3:       ...= W
S2:    loop 1 to N - 1
S4:       W = ...
S3:       ...= W
S5:    endloop
S4:    W =
```

By creating a prologue (the first statement of the first iteration) and epilogue (the last statement of the last iteration), the loop is transformed into an equivalent form in which all flow dependences are loop independent. This procedure allows the resultant loop to be completely parallelized, if desired. This situation can be recognized if there is a variable use whose *link* points to a μ-function, and the *self* flag of the μ-function is *false*.

Program	*number of ϕ-functions*	*number of Υ-functions*	*% increase*
PERFECT club			
adm	5355	8558	60
arc2d	2170	4193	93
bdna	3409	6721	97
dyfesm	1856	3293	77
flo52	2094	4858	131
mdg	970	2163	123
mg3d	2649	4390	66
ocean	1587	9264	484
qcd	1611	2286	42
spec77	3647	7332	101
spice	15115	41710	176
track	1531	2670	74
trfd	600	1011	69
RiCEPS			
boast	9887	19225	94
ccm	9184	24556	167
hydro	2476	9164	270
linpackd	271	411	27
simple	896	3169	254
sphot	1063	1952	84
wanal1	1599	2610	63
wave	5007	12371	147
Mendez			
baro	559	1206	116
euler	849	2564	202
mhd2d	600	1121	87
shear	936	1670	78
vortex	524	965	84
Total	76445	179433	134

Table 6.4 Comparison of data structure sizes between ϕ-functions and Υ-functions

(If *self* was *true*, the definition from outside the loop might reach more than the loop's initial iteration.)

Similarly, privatization analysis of variables is enhanced if a flow dependence exits a loop on only its last iteration, since in this case it is easy to identify which variable must copy its contents to a global variable. A definition inside a loop that is the last reaching definition for a variable within the loop may be the source of a last-iteration dependence if the μ-function for that variable has its *self* flag set to *false*.

The inclusion of path-sensitive analysis (if available at compile time) may also aid in the detection of scalar dependences. Information such as tripcounts for loops or γ-functions in GSA form provide more precise information on the behavior of loops, and may result in more efficient implementation of the algorithms, since some recursive calls at confluence nodes may be eliminated.

Again note that we compute value-based dependences. We can find all memory-based dependence relations by always tracing back to other reaching definitions (for flow or output dependence) or other reaching uses (for anti- and input dependence), regardless of the presence of a killing definition. For scalars, however, value-based dependence is easy to compute.

Chapter 7

Backward Data-Flow Problems

In this chapter we study backward problems – those problems in which data-flow information flows in the direction opposite control flow. We will look at two backward problems that may benefit from GRC, live variable analysis and expression anticipatability. For these problems, general Ω-functions are placed at the end of branching basic blocks, since they merge upward-exposed information from CFG successors. The material in this chapter on live variable analysis is an adaptation of work done with Michael P. Gerlek at Oregon Graduate Institute of Science & Technology [GWS94].

In the next section we present the algorithm for constructing reference chains used to solve the live variables problem, while Section 2 shows how to use this translation to compute liveness information. Section 3 explains how to use these chains to construct an interference graph, perform dead code elimination, and for other analysis methods. In Section 4 we analyze the performance based on our implementation of these algorithms, while Section 5 discusses how to extend analysis to anticipatability of expressions.

7.1 Live Variables and Chaining

7.1.1 Defining Live Variables

Identification of which variables at any point p in the program are *live* is known as live variable analysis. Variable v is considered *live* at p in the program if there is an upward-exposed use of v at p, *i.e.*, v is used after p with no intervening definition of v [ASU86]. Otherwise, v is *dead* at p. Since v is live at the end of a basic block if and only if it is live at the entry of any of its successors in the CFG, live variable analysis is a backward data-flow problem.

Live variable analysis is most commonly used for register allocation. It is also used for a variant of dead code elimination (if the target of a defining statement is not live

after the point of definition, that statement can be removed). Other less common uses of live variables are described later.

The traditional solution to live variables is to iterate data-flow equations. To account for the backward solution, we use the following variant of Equations 2.1:

$$\begin{aligned} out(N) &= \bigsqcap_{S \in succ(N)} in(S) \\ in(N) &= f_N(out(N)) \end{aligned} \tag{7.1}$$

For live variables, the meet operator is set union, and we iterate this particular instance of Equations 7.1 until convergence:

$$\begin{aligned} out(N) &= \bigcup_{S \in succ(N)} in(S) \\ in(N) &= use(N) \bigcup (out(N) - def(N)) \end{aligned}$$

Iterating these equations requires keeping the information on all variables available at the beginning (*in*) and end (*out*) of each basic block in the CFG. This procedure is generally accomplished using bit-vectors. Bit-vectors usually work well in practice, but can be overly consumptive of space [CCF91]. A second problem with the traditional approach is that it requires iteration for its solution. Though often convergence is achieved after several iterations, this is not always the case and the method is subject to more costly meet operations, such as the example we saw in §5.2.2.

We present a new approach to identifying live variables based upon reference chaining. Our approach requires no iteration, but instead encapsulates information at branch points in the CFG through merge operators known as λ-functions. Since live variables is a backward problem, λ-functions merge upward-exposed references when a basic block contains two or more successors. The storing of liveness information at λ-functions avoids the need for explicit *in* and *out* sets. Instead, liveness is computed at each point within a basic block on-the-fly.

7.1.2 Background

Since live variables is a backward problem, we will be using the concepts of *postdominator* and *postdominance frontier* as originally introduced in §2.1.2. We write X *pdom* Y if X postdominates Y and Z *ipdom* Y if Z is the immediate postdominator of Y. The *postdominator tree* contains the set of nodes V from the CFG, connected by edges $Z \rightarrow Y$

in the tree if and only if Z *ipdom* Y. For completeness, the *iterated postdominance frontier* of Y, $PDF^+(Y)$, is the limit of the sequence:

$$\begin{aligned} PDF^1(Y) &= PDF(Y) \\ PDF^i(Y) &= PDF(Y \cup PDF^{i-1}(Y)) \end{aligned}$$

The running example for this chapter is the following program:

```
i = 0
while (p) do
   if (q) then
      i = i + 1
   endif
endwhile
```

For this example we make the basic blocks explicit and consider only references to `i`. Since we want to refer to both definitions and uses as they relate to the basic blocks, we uniquely identify each definition or use with a new subscript number:

```
Entry:
   A:      i0 =
   B:      if (...) goto G
   C:      if (...) then
   D:      = i1
           i2 =
   E:      endif
   F:      goto B
   G:      ...
Exit:
```

The CFG (including the slice edge *Entry* $\rightarrow$ *Exit*) and postdominator tree are shown in Figure 7.1, with the immediate postdominator and postdominance frontier sets given in Table 7.1. We also show the *def*, *use*, *in*, and *out* sets.

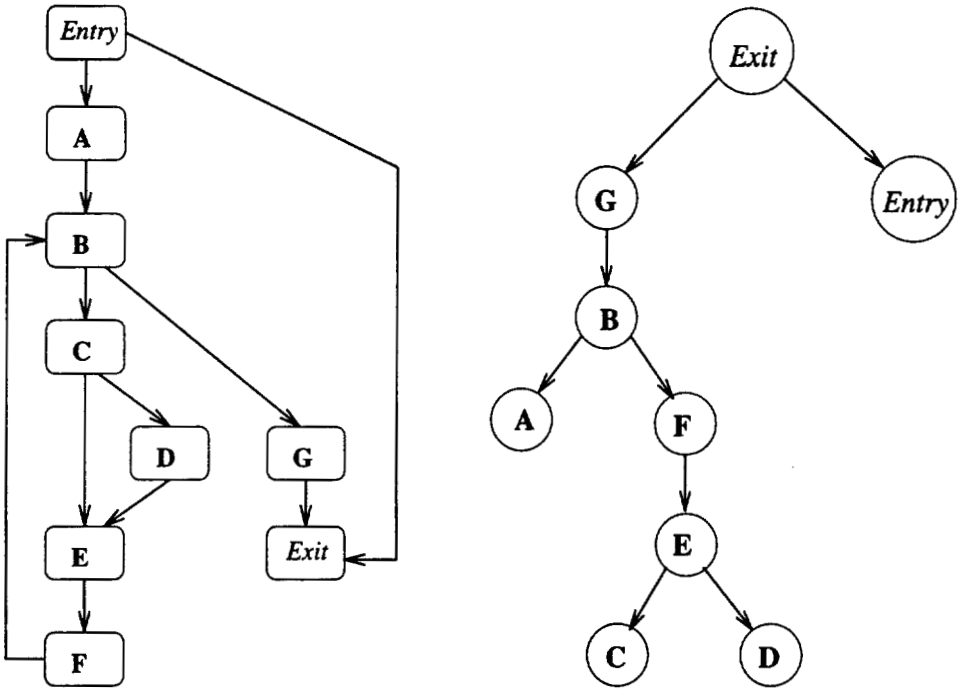


Figure 7.1 CFG and pdom tree for live variable example program

node N	*ipdom of N*	*PDF(N)*	def_N	use_N	in_N	out_N
Entry	*Exit*					
A	*B*	*Entry*	i			i
B	*G*	*Entry, B*			i	i
C	*E*	*B*			i	i
D	*E*	*C*	i	i	i	i
E	*F*	*B*			i	i
F	*B*	*B*			i	i
G	*Exit*	*Entry*				
Exit						

Table 7.1 Postdominator and liveness for example program

7.1.3 The λ-Chaining Algorithm

We use GRC to augment the CFG with λ-functions at branch points in the CFG. The GRC algorithm is used for the Problem of upward-exposed references, and invoked with the following parameters to create λ-chains:

- **Direction:** backward
- **RefLink:** $\emptyset$
- **RefTuples:** any use or definition
- **BlockTuples:** any killing definition

Notice that for this problem **RefLink** is empty, which means that the only *link* fields set are the arguments of λ-functions. The live variable problem maintains a set at point p that contains all variables live at p. This condition implies examining, for each basic block, all tuples within that node, and for each use or def of variable v, adding or deleting v from the live set, respectively. Hence, within each node no sparsity is possible, but some sparsity is preserved by capturing the merge of upward-exposed references at branch points via λ-functions. In this way we avoid the iteration associated with traditional methods.

Since this example is the first case of GRC with a backward data-flow problem, we will walk through the construction of λ-chains for our running example. The only nodes containing instances of **RefTuples** for `i` are A and D, so the GRC *Work_List* is initialized to these nodes. $PDF^+(A) \cup PDF^+(D) = \{Entry, B, C\}$, so a λ-function is placed at the end of each of these nodes:

node	*i.CurrentRef*	*t.SaveRef*	*pred λ-functions*
Exit	⊘		$i_3 = \lambda(\ ,\oslash)$
G	⊘		$i_4 = \lambda(\ ,\oslash)$
B	i_4	i_4.SaveRef=⊘	
A	⊘	i_0.SaveRef=i_4	$i_3 = \lambda(\oslash,\oslash)$
F	i_4		
E	i_4		$i_5 = \lambda(i_4,\ )$
C	i_5	i_5.SaveRef=i_1	$i_4 = \lambda(i_5, \oslash)$
D	i_1	i_2.SaveRef=i_4, i_1.SaveRef=⊘	$i_5 = \lambda(i_4,i_1)$
Entry	i_3	i_3.SaveRef=⊘	

Table 7.2 Chaining states for example program

```
Entry:    i3 = λ( , )
    A:    i0 =
    B:    if (...) goto G
          i4 = λ( , )
    C:    if (...) then
          i5 = λ( , )
    D:    = i1
          i2 =
    E:    endif
    F:    goto B
    G:    ...
 Exit:
```

For the chaining phase, *i.CurrentRef* is initially set to ⊘. We traverse the CFG in the order $\{Exit, G, B, A, F, E, C, D,\ Entry\}$. Following Algorithm 5.2, Table 7.2 shows the state at each visited node of *i.CurrentRef* (at the top of the node), *t.SaveRef*, and any λ-functions in CFG predecessors of that node. Starting with *Exit*, the second link of the λ-function in predecessor node *Entry* is set to ⊘. At node *G*, the λ-function at predecessor *B* has its second *link* set to ⊘. Visiting *B* next, *i.CurrentRef* is set to i_4. At *A*, *i.CurrentRef* is set to ⊘ (i_0 is a definition, thus an element of **BlockTuples**) and the old reference to i_4 is stored in i_0.*SaveRef*. The first *link* of the λ-function in predecessor node *Entry* is now set to ⊘. We must return down the postdominator tree:

in A *i.CurrentRef* is restored to i_4. The process continues at node B by visiting its other postdominator child, F. After all basic block nodes have been visited and all *links* have been set, the resulting program becomes:

```
Entry:    i3 = λ(⊘, ⊘)
    A:    i0 =
    B:    if (...) goto G
          i4 = λ(i5, ⊘)
    C:    if (...) then
          i5 = λ(i4, i1)
    D:       = i1
             i2 =
    E:    endif
    F:    goto B
    G:    ...
 Exit:
```

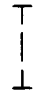
7.2 Computing Liveness

After constructing the λ-graph for a program, we can calculate liveness by traversing the CFG, adding and deleting variables from the live set at each applicable tuple.

7.2.1 Liveness Algorithm

Visiting the postdominator tree in a depth-first manner will traverse the CFG bottom-up. The set of live variables is initially empty, and variables are added or deleted based upon whether each variable reference is a use or definition. Since *Exit* is the root of the postdominator tree, it is always visited first. The tuples in a node are visited in reverse lexical order, adding a variable at a usage site to the live set, if not currently a member, and deleting a variable at a definition site, if currently a member. At the beginning of each basic block (the state of the live set after all tuples have been examined in that block) the live set corresponds to the *in* set in the traditional method.

We use the following simple lattice to represent liveness information at each tuple,



where $\top$ represents "dead" and $\bot$ represents "live". Each tuple that is the use of a variable has its lattice value set to $\bot$ and each tuple which defines a variable has its lattice value set to $\top$. Other tuples, with the exception of λ-functions, are ignored by this algorithm. The lattice value of λ-functions are initially $\top$.

All λ-functions are encountered at the end of each basic block (these are the first tuples examined for each block). Each λ-function must be evaluated according to the meet of the lattice values of its argument *links*. Since λ-functions are placeholders for liveness information, they constitute a use or definition of a variable as a function of their *links*. A λ-argument pointing to a variable use is set to $\bot$, while an argument pointing to $\oslash$ (indicating a variable definition or no reference along that path) gets set to $\top$.

A λ-argument may also point to another λ-function, whose arguments may in turn point to still another λ-function. In fact, there may be a cycle containing λ-functions, as is the case with our example program. Because the lattice value of a λ-function is dependent on other λ-functions, we use our demand-driven technique (Algorithm 3.3) for classifying λ-functions, and again employ Tarjan's algorithm [Tar72] for detecting strongly connected components (SCCs) in a directed graph. Tarjan's algorithm is applied to an abstraction of the data-flow graph, the *λ-graph*, $G_\lambda = \langle V, E \rangle$, where V is the set of λ-functions for a particular variable in the program and E is the set of *links* of each λ-function pointing to other λ-functions. When a nontrivial SCC is detected, the λ-functions within that cycle (whose members comprise a *λ-set*) are assigned a lattice value based on the meet of the lattice value of all *links* of the λ-set functions that do not point to elements of that λ-set. The result is the meet of the lattice value of all the successors of the SCC. Thus, if any component in the SCC has a *link* which points to a variable use, all members of that λ-set are classified as live. No iteration of the cycle is necessary, since liveness is an example of a *uniformly monotonic* [WGS94] problem, as described at the end of Chapter 3. After processing all λ-functions at the end of each basic block (and before processing any other tuples within that block), the live set corresponds to the *out* set in the traditional method.

In our running example, the λ-graph contains a λ-set consisting of λ-functions $\mathtt{i}_4$ and $\mathtt{i}_5$. Figure 7.2 shows this graph, with the initial value of each λ-argument shown as a superscript of that argument, and the final lattice value of each λ-function shown on the right of each node. Initially, we note that both arguments to $\mathtt{i}_4$ are $\top$, corresponding to the fact that `i` is dead after the loop and its liveness before the loop depends on liveness within the loop body.

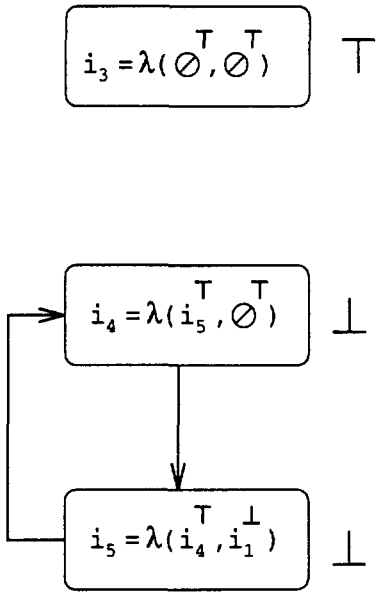


Figure 7.2 λ-graph for running example program

To find the lattice value of our example λ-set, we take the meet of all arguments that do not point to i_4 or i_5: $\top$ for the second argument of i_4 and $\bot$ for the second argument of i_5. The result is $\bot$, which indicates that i is live within the cycle, hence live after basic blocks B and C.

The details of computing live sets are given in Algorithm 7.1. We use the following data structures:

- *Live* – A set of variable symbols representing those symbols that are live at any point in the program.
- *visited* – A field used by λ-functions to indicate whether they have already been processed by Tarjan's algorithm.
- *marked* – A field used by tuples to indicate whether they affect the *Live* set.

The set *Live* is initialized to $\emptyset$ and Algorithm 7.1 is invoked as *Liveness(Exit, Live)*. The lattice element associated with each tuple is initialized as described above.

7.2.2 Correctness

We show the correctness of the liveness algorithm at each basic block in the CFG by performing induction on the depth of each node in the postdominator tree. Since live

Given: CFG with λ-chains, $Live = \emptyset$
Do: Call *Liveness(Exit, Live)*
Result: *Live sets*

```
1:  Liveness(N, Live)
2:      forall tuples t ∈ N in reverse order do
3:          if type( t ) = λ-merge function then
4:              if t.visited = false then
5:                  set lattice( t ) using Tarjan's alg on the λ-graph
6:                  t.visited ← true
7:              endif
8:          endif
9:          if type( t ) = λ-merge or t is a use or def then
10:             t.marked ← false
11:             s ← symbol( t )
12:             case lattice( t )
13:                 ⊥:  if s ∉ Live then
14:                         Live ← Live ∪ {s}
15:                         t.marked ← true
16:                     endif
17:                 ⊤:  if s ∈ Live then
18:                         Live ← Live − {s}
19:                         t.marked ← true
20:                     endif
21:             endcase
22:         endif
23:     endfor

24:     for M ∈ PDomChild(N) do
25:         Liveness(M, Live)
26:     endfor

27:     forall tuples t ∈ N in forward order do
28:         if (type( t ) = λ-merge) or (t is a use or def) then
29:             s ← symbol( t )
30:             case lattice( t )
31:                 ⊥: if t.marked then Live ← Live − {s}
32:                 ⊤: if t.marked then Live ← Live ∪ {s}
33:             endcase
34:         endif
35:     endfor
36: end Liveness
```

Algorithm 7.1 Computing *Live* sets on a CFG

variables is a backward problem, analysis starts at *Exit*, with the order of visitation of the nodes determined by a depth-first walk of the postdominator tree (lines 24 - 26 in Algorithm 7.1). Once the live *out* set is correctly computed for node N (after processing all λ-merge tuples at the end of the node, lines 3 - 8 in Algorithm 7.1), in_N is easily shown to be correct by the repeated application of lines 9 - 22 on the remaining tuples of the node in reverse lexical order.

Because *Live* is a set consisting of all variables, we show correctness for particular variable v, and note that *Live* can be constructed at any point in the program by applying the union operator to the liveness state of all variables at that point.

Base case: Since Algorithm 7.1 is invoked with *Liveness(Exit, Live)*, and initially $Live = \emptyset$, the solution for v (within out_{Exit}) is trivially correct after *Exit*.

Induction step: We consider the liveness of v within out_N, given that each prior node in the walk of the postdominator tree has correctly computed its *out* set. Two cases are presented, based upon the existence of a λ-function for v at N:

- No λ-function for v exists at N. If N has just one successor in the CFG (only *Exit* has no successors), it must be the immediate postdominator of N, and v in out_N must be correct, since in this case $out_N = in_{ipdom(N)}$ by definition.

  If N has more that one successor in the CFG, there can be no upward-exposed references of v in any intermediate node on any path from N to *ipdom(N)*, since otherwise a λ-function would exist at N for v. Thus, the liveness of v at out_N is equal to $in_{ipdom(N)}$, and out_N is inherited from *ipdom(N)* at line 25 of Algorithm 7.1. Since the liveness of v within $in_{ipdom(N)}$ is correct by hypothesis, and $out_N = in_{ipdom(N)}$ by construction, liveness of v at out_N must be correct.

- A λ-function for v exists at N. In this case N must have multiple successors in the CFG. The state of v within *Live* just after block N is dependent upon the liveness of v within the *in* set of all CFG successors of N. If there is an upward-exposed use of v within any successor of N, then v is live within out_N; else v is dead at out_N. For each successor of N, the λ-function for v in N will have one *link* to the subsequent reference of v along the path starting with that successor. If the subsequent reference is a use or $\oslash$, the lattice value corresponding to that *link* is set to live or dead, respectively. If the subsequent reference is another λ-function, the λ-graph will form a tree whose leaves are a use or $\oslash$ (once strongly

connected components are collapsed). Thus, irrespective of the CFG, the lattice values inherited by the λ-function at N will reflect the upward-exposed references of the variable; there can be no other intervening references to the variable in the program by the GRC construction in Section 7.1. If any of these references are uses, the λ-function for v at N will represent a use (implying v is live at out_N), due to the meet operator of the live variable lattice. Otherwise, the λ-function for v at N represents $\oslash$, so v is dead at out_N.

The *Live* set at N is now modified by each use, $\oslash$, and λ-function of v in N. At the top of N, the liveness for v is carried to the next block in the postdominator tree. When returning to N, only those tuples that affected v are undone (lines 27 - 35 of Algorithm 7.1), so that after N the liveness of v is precisely equal to what it started as. After visiting N and all its postdominator children, restoring *Live* to its state before processing N is important since the contribution by N to *Live* needs to be undone before processing other nodes. While the particular order of the postdominator tree traversal is immaterial, the depth-first walk allows induction to be performed on the height of that tree. ▌

7.3 Applications of Liveness

7.3.1 Interference Graph Construction

As mentioned in §7.1, the main use for live variable analysis is register allocation. This analysis involves the construction of an *interference graph*, where variable v interferes with variable w at a given point in the program if both are simultaneously live. When this condition occurs, v and w cannot share a register. Formally, the interference graph is $G_{IG} = \langle S, E \rangle$, where S is the set of symbols in the program and E is a set of edges such that $(p, q) \in E$ if and only if p *interferes* with q, *i.e.*, they are both live concurrently at some point in the program. The traditional approach computes the *in* and *out* sets at each basic block in the CFG, then performs a separate pass to construct the interference graph. Before visiting all the tuples in a basic block, a *Live* set is initialized to the live *out* set of the block. At each definition of a variable, that variable is removed from the *Live* set and marked as interfering with all other variables currently live. The interference graph is a symmetric matrix K, such that $K_{pq} = 1$ if variable p interferes with variable q and $K_{pq} = 0$ otherwise.

We make a simple modification to Algorithm 7.1 to build the interference graph

Given: CFG with λ-chains;
K, initialized to $K_{st} \leftarrow 0, \forall s, t \in S$
Do: Determine the liveness at each block and compute the interferences
Result: Interference graph for the program

Use Algorithm 7.1 with the following addition:
20.a **if** *t is a def* **then** $K_{sl} \leftarrow 1, \forall l \in Live$

Algorithm 7.2 Interference graph construction

during the same pass as liveness is computed. Algorithm 7.2 is the same as Algorithm 7.1, with the addition of a line after line 20 (which uses matrix K, as described above).

7.3.2 Useless Code Elimination

Code that never affects the final results of a program is termed *dead.* Dead code can be classified into two categories. *Unreachable* code is code that will never be visited when a program executes. Determination of constant predicates, such as the methods discussed in Chapter 4, may preclude paths in the CFG from ever executing. *Useless* code is an assignment to a variable that is not subsequently used. Useless code usually occurs as a result of compiler optimizations; for example, code motion might effectively copy but not move computations. Dead code elimination is an important optimization that is always beneficial and may be required several times during compilation. This section looks at a dynamic implementation of useless code elimination.

Typically, useless code elimination is performed by visiting the tuples in each basic block, updating the *Live* set as used by the traditional interference graph technique. When a definition is encountered and the defining variable is not in *Live*, the entire statement is removed. This method does not take into account the liveness information of the deleted right-hand side of the statement across basic blocks, however. In this case:

```
A:   y = 1
     if (cond) then
B:      x = y + 2
C:   endif
```

the live variable information is $out_A = \{\mathtt{y}\}$ and $out_B = out_C = \emptyset$. The assignment to x can be eliminated because it is not live after B, but the assignment to y cannot be eliminated: the set out_A does not reflect the elimination. Good useless code detection requires the dynamic live information that our algorithm provides.

In order to dynamically update useless code identification, *links* may need to be updated because code has been eliminated. In particular, a λ-argument may point to a variable use that is now part of a dead expression. To accurately update the argument pointer, *links* need to be provided from each use to the next upward-exposed reference.

If a λ-argument points to $\oslash$, eliminating useless code never requires an update, since the next upward-exposed reference must be $\oslash$. This reference cannot be a variable use, since in that case the expression defining the variable would not have been identified as useless.

To provide the capability for updating λ-function *links* we need to augment λ-chains with upward-exposed use-ref *links*. Thus, for dynamic useless code elimination we invoke GRC with the following parameters for the Problem of upward-exposed references:

- **Direction:** backward
- **RefLink:** any use
- **RefTuples:** any use or definition
- **BlockTuples:** any killing definition

Algorithm 7.3 is much the same as Algorithm 7.1, with line 20 changed and five lines added. It removes useless code, based upon a dynamically updated live set at each tuple. We note that this useless code algorithm does not identify all *faint* code [KRS94]: dead code plus code that is only used (perhaps transitively) by itself.

7.3.3 Other Uses of Liveness Information

Liveness information has several other uses; we include a list of several of them here for completeness.

Uninitialized variables: The use of a variable before its definition can be an incorrect program in some languages, and may result in indeterminate behavior. Such cases are easily detected if a (local) variable is live at *Entry*.

"Intent" determination: The intent of a variable may be important to such analysis as interprocedural constant propagation, where it is useful to know if arguments

Given: CFG with λ-chains augmented with use-ref links
Do: determine the liveness at each block and
    eliminate dead code
Result: Tuples identified as useless

Use Algorithm 7.1 with the following change:

```
20      else if type( t ) ≠ λ merge then
20.a        forall r on rhs of def ∋ lattice( r ) = ⊥ do
20.b            replace link( ref ) which points to r with link( r )
20.c        endfor
20.d        remove this def and its associated rhs
20.e    endif
```

Algorithm 7.3 Useless code elimination

passed to other procedures are modified or not. This type of information may be reflected in some languages, such as the INTENT attribute in Fortran 90 [ABM+92]. If a passed reference parameter is used but not defined it stays live at the call site.

Thread migration: Some methods that implement a migration of processor threads generate special procedures to handle the migration [HWW93]. The body of the procedure is the continuation of the migrating procedure at the point of migration. The arguments to this procedure are the live variables at that point.

7.4 Experimental Results

In this section we report results of our experiments to compare the classical method of determining liveness information and our λ-chain approach. We measured space and time of both approaches, and also measured the effect of performing dynamic useless code elimination during the same pass. While our λ-chain algorithm for live variable analysis is no faster than the traditional method, it is still a competitive alternative and performs useless code elimination with little additional cost.

We have implemented the algorithms for constructing λ-chains, determining liveness, and constructing the interference graph in Nascent. The programs used to collect the data are the usual benchmarks described in §1.4. To model the use of interference graph construction more accurately, a "lowering" phase in Nascent is first performed, where many compiler temporary variables are created.

7.4.1 Data for the Traditional Approach

Table 7.3 shows the amount of space needed by the iterative, bit-vector method to perform interference graph construction. This traditional approach keeps four sets, the *use*, *def*, *in* and *out* sets at each basic block. The first column shows the number of bits necessary to store each of these sets. Thus, in total, a little over a 1000K 32-bit words were necessary for all the benchmark programs. The *use* and *def* sets are extremely sparse (about 1% usage for each), while the *in* and *out* sets each use about 20% of the bit-vector.

The number of iterations required for convergence of the *in* and *out* sets is also shown in Table 7.3, averaged over all the routines within each benchmark program. On average, less than four iterations are required for computing the sets.

7.4.2 Data for the λ-Chain Approach

As we have done with other reference chain merge operators, we experimentally compared the number of λ-functions as a function of referenced variables (Figure 7.3) and program size (Figure 7.4). These graphs show that the growth in data structure size is again fairly linear in both cases, supporting the contention that in practice programs exhibit this linear behavior for all types of reference chaining.

Over all the benchmarks, a total of 309,494 λ-functions were created, with an average of one λ-function per 4.6 variable references. It is difficult to establish the precise amount of space required to construct a λ-function, but the best implementation would be one word per λ-function. This calculation results in the same order of magnitude (300K bytes) as the traditional algorithm (1000K bytes).

To determine liveness, the method we have outlined requires processing strongly connected components in the λ-graph. We have found that on average slightly more than 40 percent of the λ-functions in the benchmarks were part of a nontrivial component; the percentages ranged from a low of 7 to a high of 83, and most were clustered between 30 and 50 percent. The nontrivial component sizes ranged from 2 to 8, with an average of 4.3.

7.4.3 Comparative Performance

We now consider the relative speed of the two implementations. Both the iterative, bit-vector approach and the λ-chain approach were implemented on a Sun SPARC 80mhz

	space	% of vector used				iterations	
Program	10^3 *bits*	*use*	*def*	*in*	*out*	*avg*	*max*
PERFECT club							
adm	283	2.5	2.3	24.5	24.5	3.5	7
arc2d	174	2.4	2.0	19.5	19.8	3.9	5
bdna	317	1.2	1.2	17.4	17.5	3.5	6
dyfesm	202	1.3	1.3	13.8	14.0	3.7	6
flo52	220	1.3	1.3	19.9	20.1	3.8	5
mdg	54	2.0	2.3	27.5	27.5	3.8	6
mg3d	162	2.0	1.6	17.0	17.2	3.5	7
ocean	314	0.8	0.7	27.8	27.9	3.1	6
qcd	149	1.2	1.3	14.6	14.8	3.6	8
spec77	202	2.2	1.8	40.3	40.5	3.6	6
spice	1077	1.2	0.9	24.5	24.5	3.1	7
track	65	2.2	2.3	28.2	28.1	3.4	6
trfd	45	1.6	1.4	20.8	21.3	5.1	6
RiCEPS							
boast	176	0.7	0.5	19.4	19.5	4.4	6
ccm	989	1.2	0.9	22.6	22.7	3.0	6
hydro	177	1.4	1.2	24.6	24.7	3.2	5
linpackd	22	2.5	2.2	18.7	18.8	3.2	4
simple	127	0.8	0.8	29.2	24.4	3.1	5
sphot	178	0.5	0.5	18.7	18.7	3.7	7
wanal1	2908	0.1	0.1	18.0	18.0	4.0	8
wave	369	1.8	1.2	24.6	24.6	3.0	5
Mendez							
baro	64	1.7	1.2	27.0	27.3	4.1	5
euler	97	1.4	1.3	13.7	13.7	3.1	5
mhd2d	46	1.6	1.3	36.3	36.5	3.0	6
shear	97	1.2	0.9	32.2	32.4	3.1	6
vortex	13	5.4	4.4	35.5	35.6	3.1	6
Total	8527	1.1	1.0	20.9	21.0	3.5	8

Table 7.3 Data on solving IFG using traditional bit-vector approach

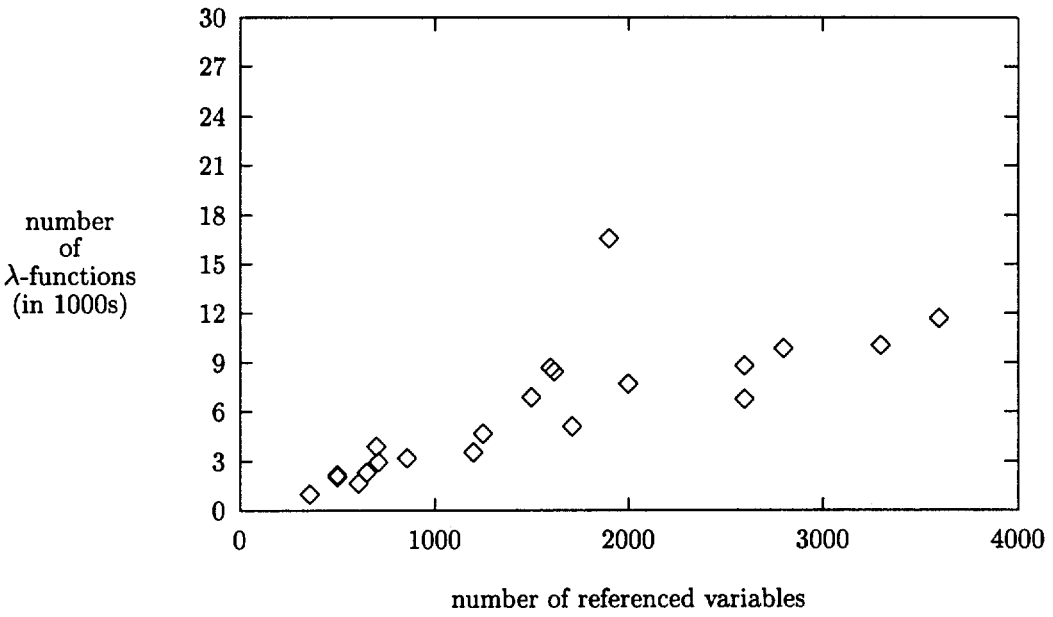


Figure 7.3 A comparison of λ-functions to referenced variables in the benchmark programs

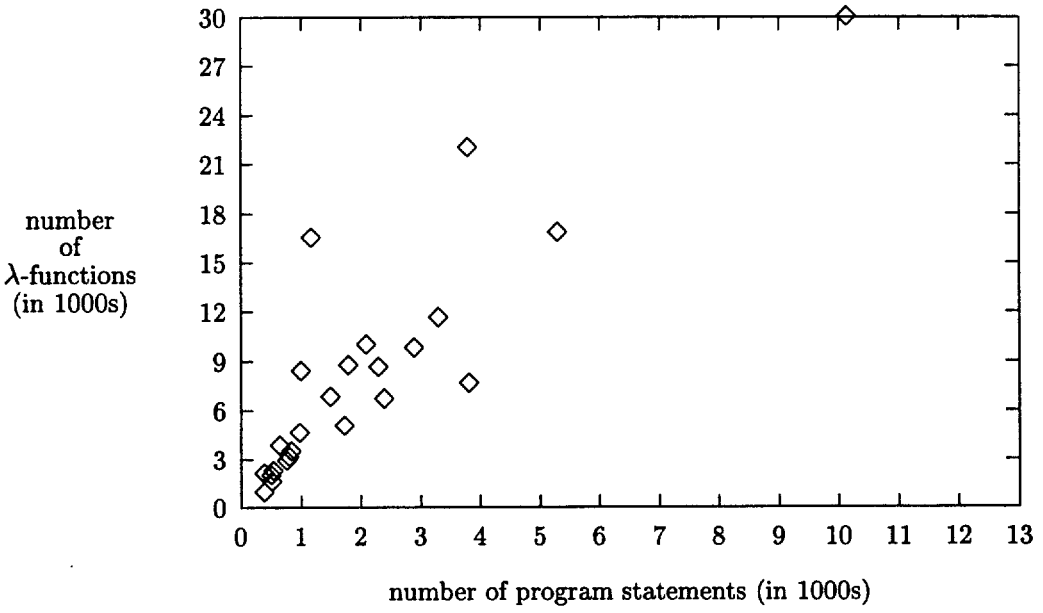


Figure 7.4 A comparison of λ-functions to program statements in the benchmark programs

IPX workstation with 64MB of memory. All optimizations were left to the compiler (gcc version 2.5.8, -02). While the traditional method uses bit-vectors to maintain its live set, a linked list was used in maintaining the live set in the λ-chain algorithm.

The traditional algorithm for interference graph construction has three steps:

- *init*: compute the *use* and *def* sets
- *live*: compute the *in* and *out* sets
- *ifg*: compute the interference graph

The λ-chain algorithm also has three steps:

- *place*: insert λ-functions
- *chain*: set the *links* for the λ-functions
- *ifg*: compute liveness and the interference graph

Both methods require overhead in our implementation to allocate storage for the data structures used. Overall, the traditional method performs faster since convergence generally was achieved within a few iterations, whereas the λ-chain approach spent considerable time computing the interference graph. This additional time is partly due to visiting all tuples within each basic block twice, once on the way down the postdominator tree and once on the way coming back up (another approach, that of storing the *Live* set before processing each merge node, may help ameliorate this effect). We also did not count the time to construct the postdominance frontier set for each node since this information is useful for other analyses such as control dependence.

Table 7.4 presents the time (in seconds) for three phases of each approach. We also give the total time for each method and the performance ratio. The data is presented for three runs, one for all of the programs in each suite (granularity precludes accurate per-program timing). The overall time required for interference graph construction using λ-chains is roughly 45% greater than the traditional algorithm, but performing useless code elimination with Algorithm 7.3 adds less than one percent to the times recorded using λ-chains. Thus, although the traditional method performs faster, we can obtain dynamic useless code elimination for "free" when using the λ-chain approach.

	PERFECT club	RiCEPS	Mendez
Traditional	144	123	7
init	3	3	1
liveness	1	2	<1
ifg	64	63	4
overhead	76	55	2
Lambda	211	175	10
placement	21	30	2
chaining	6	6	1
ifg	112	89	5
overhead	72	50	2
Ratio	1.47	1.42	1.43

Table 7.4 Times (in seconds) for traditional and λ-chain methods

7.5 Anticipatability of Expressions

An expression e is said to be *anticipatable* at point p if every path from p to *Exit* computes e before any variable of e is redefined. (Anticipatable expressions were originally termed *very busy* expressions [ASU86].) In Figure 7.5 `a+b` is anticipatable at P, but `c+d` is not since one of its operands, `c`, is defined in R before `c+d` is computed. The expression `e+f` is also not anticipatable at P since it is not computed on the path to Q. As opposed to liveness, which is an "or" problem, anticipatability is an "and" problem.

The main value of determining anticipatability for expressions lies in the ability to *hoist* the expression to an earlier point in the program. Space may be saved by computing the expression only once, although this benefit might be offset by the need to store the result for a longer period of time.

Anticipatability is a backward problem, so an iterative solution uses Equations 7.1. Let *gen(N)* be the set of expressions computed in basic block N prior to any definition of the variables in those expressions, while *kill(N)* represents the set of expressions in which any right-hand side variables have upward-exposed definitions in N. If *in(N)* and *out(N)* represent the set of anticipatable expressions at the beginning and end of N, respectively, then, after appropriate initialization, anticipatability can be solved by iterating the following equations until convergence:

$$out(N) = \bigcap_{S \in succ(N)} in(S)$$

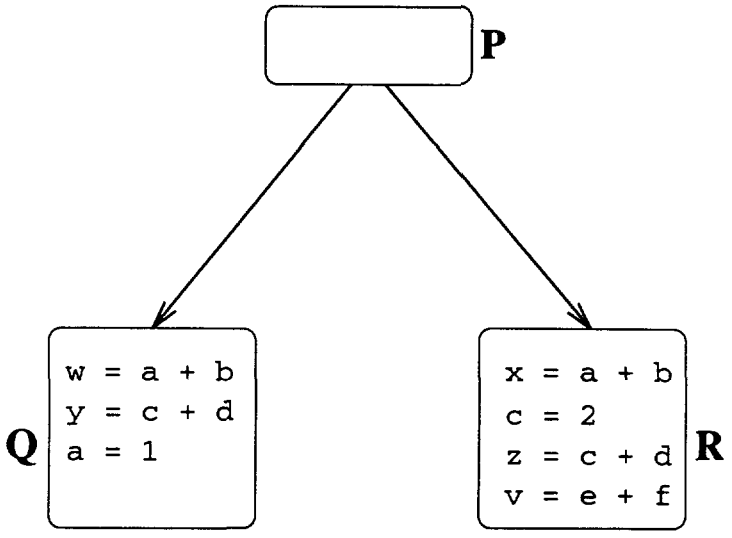


Figure 7.5 Expression anticipatability

$$in(N) \;=\; gen(N) \bigcup (out(N) - kill(N))$$

Note that for this data-flow problem, the meet operator is set intersection, since an expression is anticipatable at the end of a branch node if and only if it is anticipatable at the beginning of all successor nodes.

We can also solve this problem using GRC in the same general manner as was used for live variables. A simple lattice framework is again used:

$$\begin{array}{c} \top \\ | \\ \bot \end{array}$$

where the value $\top$ represents "anticipatable" and $\bot$ represents "not-anticipatable". All computed expressions are assigned $\top$, while all definitions which are operands of some expression (*relevant* definitions) are assigned $\bot$.

Chaining needs to be augmented for expressions in general, as opposed to just tuples. For this case, **RefExpressions** replaces **RefTuples** and **BlockExpressions** replaces **BlockTuples**. A relevant definition, however, may serve to block more than one expression. For example, in this code fragment

```
a = 1
w = a + b
x = a + c
```

the definition of a blocks both a+b and a+c from being anticipatable.

With these augmentations in mind, a few adjustments need to be made to Algorithm 5.2 in order to account for the semantics of relevant definitions. Line 3 is modified as follows:

```
3a:  e = expression t
3b:  if e is a relevant def then
3c:          F = set of all expressions with e as an operand
3d:  endif
```

We now need to loop over lines 7 - 14 for all elements of F, since each element of F refers to a different expression. A similar looping construct is also necessary when restoring current expression references in lines 27 - 30.

After accounting for the adjustments above, the GRC algorithm is invoked with the following parameters for the Problem of anticipatable expressions:

- **Direction:** backward
- **RefLink:** $\emptyset$
- **RefExpressions:** any evaluated expression or relevant definition
- **BlockExpressions:** any relevant definition

Algorithm 7.4 keeps a set of anticipatable expressions, *AntExp*, which is dynamically updated as the CFG is walked bottom-up by traversing the postdominator tree. The analysis of a cycle of Ω-functions is done similarly to that for λ-functions, except that the meet operator works on a slightly different lattice. As with live variable computation, no iteration on the CFG is required.

Anticipatability of expressions is the dual of *availability*: expression e is said to be *available* at point p if every path from *Entry* to p computes e and none of the operands of e are redefined between the last computation of e and p. Availability of expressions, a forward problem, is often used to perform common subexpression elimination, and GRC can be used to solve this problem in a manner analogous to the anticipatability problem.

Given: CFG with Ω-chains, $AntExp \leftarrow \emptyset$
Do: Call *Anticipate(Exit, AntExp)*
Result: *Anticipatable expression sets*

```
1:  Anticipate(N, AntExp)
2:      forall expressions and relevant definitions e ∈ N in reverse order do
3:          if type( e ) = Ω merge function then
4:              if e.visited = false then
5:                  set lattice( e ) using Tarjan's alg on the Ω-graph
6:                  e.visited ← true
7:              endif
8:          endif
9:          e.set ← ∅
10:         case lattice( e )
11:             ⊤:  if e ∉ AntExp then
12:                     AntExp ← AntExp ∪ {e}
13:                     e.set ← e
14:                 endif
15:             ⊥:  F ← set of all expressions with e as operand
16:                 forall f ∈ F do
17:                     if lattice( f ) = ⊤ then
18:                         AntExp ← AntExp − {f}
19:                         e.set ← e.set ∪ {f}
20:                     endif
21:                 endfor
22:         endcase
23:     endfor

24:     for M ∈ PDomChild(N) do
25:         Anticipate(M, AntExp)
26:     endfor

27:     forall expressions and relevant definitions e ∈ N in forward order do
28:         case lattice( e )
29:             ⊤:  if e.set ≠ ∅ then AntExp ← AntExp − e.set
30:             ⊥:  if e.set ≠ ∅ then AntExp ← AntExp ∪ e.set
31:                 endif
32:         endcase
33:     endfor
34: end Anticipate
```

Algorithm 7.4 Computing anticipatability of expressions

Chapter 8

Extension into Parallel Constructs

How do we extend the semantics of reference chaining to parallel programming languages? Our idea is provide a coherent and sound method by which to sensibly reason about programs written using parallel constructs. Application of these methods would enhance the ability to analyze and optimize such programs. We shall look at one parallel construct in detail – explicit parallel sections, and discuss how we extend the methods of reference chaining to it.

The original work that develops SSA-like semantics for explicit parallel sections is due to Srinivasan, Hook, and Wolfe [SHW93]. Here we will describe the details of algorithms, methods, and techniques to successfully implement these concepts. We also identify and prove correct the minimum set of merge points in parallel precedence graphs, analogous to the join set for sequential CFGs. This new set, the iterated *meet*, is a correction of earlier work that attempted to identify such merge points. Since reference chain operators are based upon *branch* and *merge* nodes in the CFG, it is important to correctly understand the abstraction of execution flow with respect to parallel constructs.

This chapter provides the theoretical basis for linearizing reaching definitions within explicit parallel sections. These foundations can be extended to other types of information flow in a manner analogous to the sequential case, where SSA form was generalized to reference chaining.

8.1 Execution Order in a Precedence Graph

A precedence graph is an abstraction that imposes order of execution among its nodes. Precedence graphs can easily be used to express DAG parallelism [CHH89] by using `Wait` clauses to enforce constraints between section nodes. A precedence graph is also

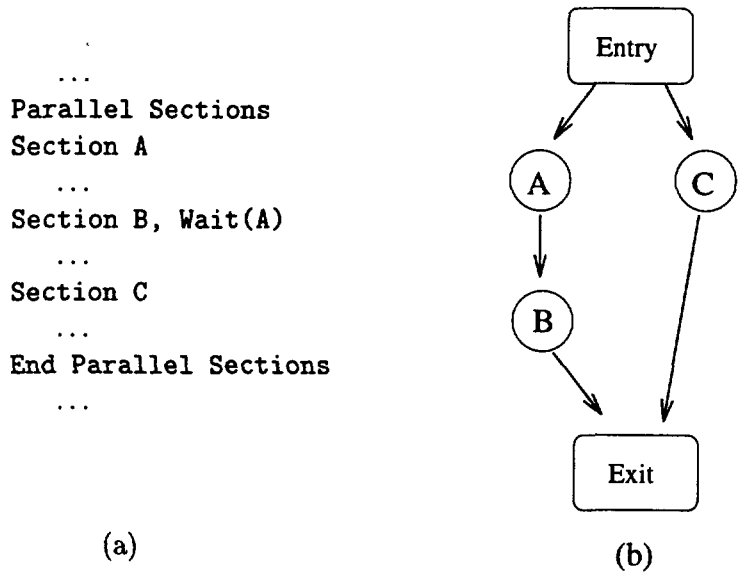


Figure 8.1 Example Precedence Graph

a simplified, special case of a Parallel Program Graph [SS93] that only contains *synchronization* edges, with the synchronization condition that all code in a node completes before beginning execution of any successor.

For the purposes of this chapter, we deal with DAG parallelism (a subset of *task* parallelism [F+93]), specifically explicit parallel sections fashioned after the `Parallel Sections` construct [Par91], which is similar to the `cobegin-coend` syntax of Brinch Hansen [BH73]. An example is shown in Figure 8.1(a), where `Section B, Wait(A)` means that all code in Section A must complete before the code in Section B may begin. Each parallel section (A, B, or C in this example) contains a local CFG with its own *Entry* and *Exit* node.

8.1.1 An Abstract Representation

The ordering of sections is arranged within a *precedence graph* (PG), an abstract representation that dictates what sections may execute in what order. Formally, a PG is a directed graph $\mathrm{P} = \langle V_P, E_P, Entry_P, Exit_P \rangle$, where V_P is a set of nodes, each representing a section in a parallel block, E_P is the set of edges that represent wait-dependence arcs (corresponding to the `Wait` syntax described above), and $Entry_P$ and $Exit_P$ are the *cobegin* and *coend* nodes, respectively. We will always show the *Entry* node in an

```
(p)   a = 2
(p)   b = 3
(p)   c = 4
(p)   if (Q) then
          Parallel Sections
          Section A
(s)           if (P) then
(t)               b = a * 5
              else
(u)               b = a + 7
(u)               f = b * a
(v)           endif
          Section B
(w)           c = c + 15
(w)           f = c * 16
          Section C, Wait(A)
(x)           d = b * a
          Section D, Wait(A, B)
(y)           c = a * b + c * f
          End Parallel Sections
(n)       d = d + f
      else
(q)       d = 23
(r)   endif
(r)   e = a + b * c * d
```

Figure 8.2 Example parallel program

example PG, but will often omit the *Exit* node, since for purposes of illustration only a partial representation of the PG is usually needed, and the *Exit* node is seldom a factor concerning the forward flow of information (which we focus on in this chapter) through the PG. The wait-dependence arcs impose a partial order upon the nodes of a PG. If there is no partial order between two sections, they may execute in any order relative to each other – perhaps in parallel. An example precedence graph is shown in Figure 8.1(b), where sections A and C (and B and C) might execute concurrently.

We note that a PG must be acyclic, since any cycles would create a deadlock. A section node of a PG "uses" or "defines" a variable if any of the code within that section uses or defines that variable.

A well-defined interpretation needs to be applied to the case where two sections of

code that can execute in parallel both modify the same variable, or when one section uses a variable modified by another section. We assume copy-in/copy-out semantics in the compiler, where the values of shared variables in a parallel section are defined to be initialized to the values at the beginning of the parallel block. When the parallel block is complete, the global state is updated with any modifications made from the sections. While this gives a well-defined program without volatile variables, and allows independent optimization within each section, we note that this model does not maintain *sequential consistency* [Lam79]. Examine this code:

```
w = 0
x = 0
Parallel Sections
     Section A
        w = 1
        y = x
     Section B
        x = 1
        z = w
End Parallel Sections
output( y,z )
```

A sequentially consistent model permits, for `(y,z)`, the values of (1,1), (1,0), or (0,1) at the `output` statement. However, our copy-in/copy-out model results in the values (0,0).

As an example of how PGs fit into the CFG structure, examine Figure 8.2. To accommodate local CFGs within explicit parallel sections, we add a special type of node to the CFG called a *supernode*. A supernode essentially represents an entire `Parallel Sections` construct (sometimes referred to as a *parallel block*).

For each supernode P, two additional basic block nodes, called the *head* and *tail* nodes for that supernode, are introduced. The *head* node captures all the incoming control flow edges to P and the control flow successors of the *tail* node are those of P in the original CFG. The *tail* node has exactly one control flow predecessor, namely P. Node P is the only control flow successor of the corresponding *head* node. These additional nodes are helpful for both proving correctness and for implementation. We will return to their function in later sections.

The Extended Flow Graph (EFG) is the union of CFGs and PGs representing sequential control flow and parallelism for a single program unit, as originally described

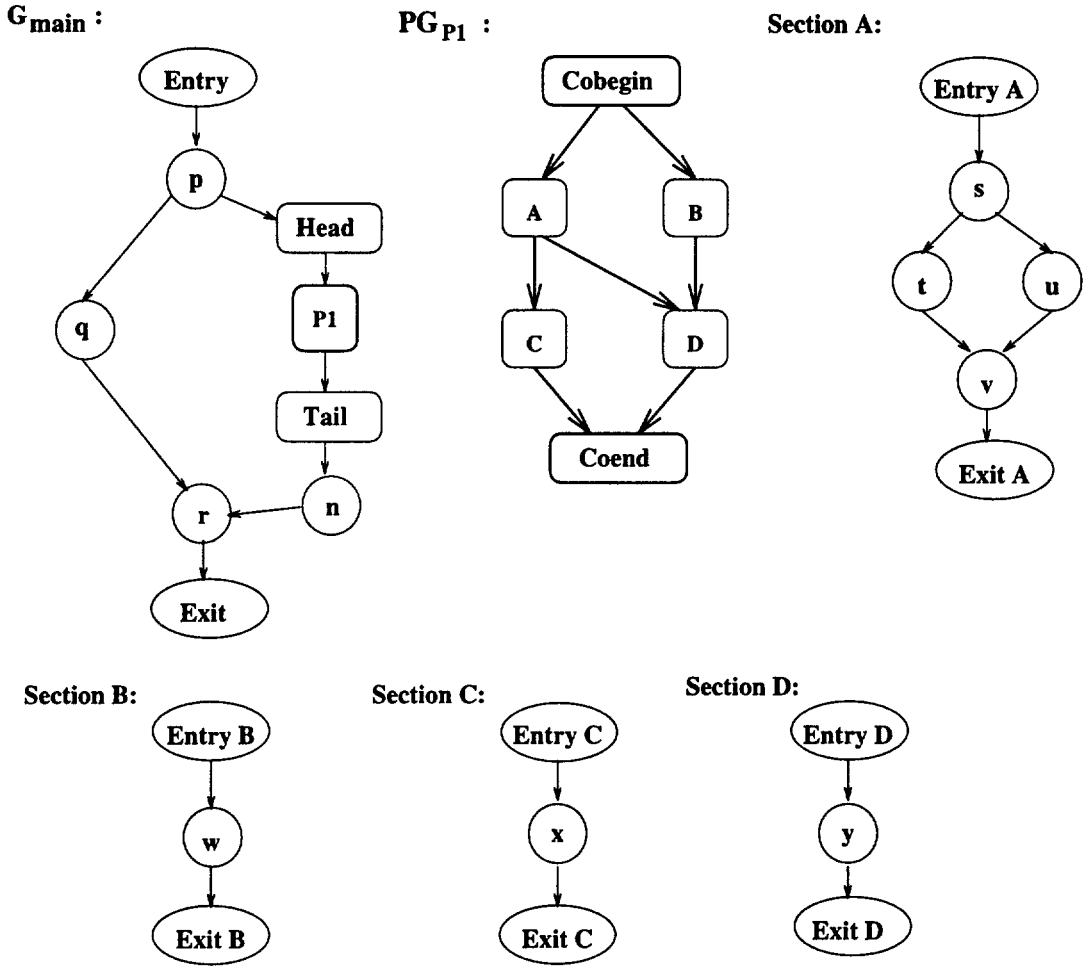


Figure 8.3 EFG for the parallel program of Figure 8.2

by Srinivasan *et al.* The distinguished CFG corresponding to the program unit is called G_{main}. We will talk about the set of nodes in an EFG, which is the union of all the nodes in all the CFGs (main and local) and PGs in the EFG.

The CFG containing any basic block node X is designated G_X, the section node corresponding to G_X is designated S_X, and P_X corresponds to the supernode representing the PG containing S_X.

The EFG for the parallel program in Figure 8.2 is shown in Figure 8.3. The parallel block (supernode) is represented by the node *P1* in G_{main}, which is in turn represented by PG_{P1}. Each section of the parallel construct is represented by a local CFG, as shown in the figure. For example, for basic block node x, G_x is the local CFG for Section C, S_x is the node C in PG_{P1}, and P_x is *P1*.

A confluence node in a PG has quite different semantics than that in a sequential CFG. While precisely one of the predecessors at a confluence node in a CFG will be executed, *all* predecessors of a confluence node in the PG must execute before the confluence node itself executes. Essentially, a confluence node is waiting upon all its predecessors, so they must all execute before the confluence node executes. When paths meet within a PG, information might also merge. It is important to note that it is possible that merging information could be in conflict – since all predecessors are executed, we could have multiple definitions of the same variable, for instance.

We also note an important property of precedence graphs – they are insensitive to transitive edges. This property will become clear when we look at how information flows between section nodes in a precedence graph – with respect to the *reaches* relation for definitions.

8.1.2 The Reaches Relation for Definitions Within a PG

When does node B wait for node A? When there is path in the precedence graph from A to B, *i.e.*, if A can reach B, then B waits upon A. Since any path from A to B is sufficient, it now becomes clear why the addition of transitive edges to a precedence graph adds no new information. In fact, the *transitive reduction*[†] of a precedence graph contains the smallest number of vertices and edges that captures all the information of the original graph.

Definition 8.1 (Reaching Definitions Within a PG.) *Within a PG, a definition of v at section node X* reaches *section node Y if no path from X to Y contains a definition of v, except at X or Y.*

Kill information is computed quite differently within a PG compared to a CFG [GS93]. In a PG, a definition of v in node A is killed before reaching node C if *any* path from A to C passes through node B, where B contains a definition of v. We note that the *shield* relation (introduced in §2.3.2) and *reaches* relation are duals of each other with respect to killing data-flow information:

CFG Information from A can be killed by B before reaching C if *all* paths from A to C pass through B.

[†]The transitive reduction of graph G is any graph G′ with the same vertices as G, but with as few edges as possible, such that the transitive closure of G′ is equal to the transitive closure of G.

PG Information from A can be killed by B before reaching C if *any* path from A to C passes through B.

It is important to contrast two uses of the term "reaches". On the one hand we speak of path reachability; here, A reaches B in a graph if there exists a path from A to B. On the other hand, we will often be referring to reaching definitions, which is concerned with the flow of a particular type of information through a graph. In this case path reachability is not sufficient; definitions (or, more generally, data-flow information) can be killed along paths in a graph due to nonidentity transfer functions. Most of the time the meaning is clear from context, but when not, we will attempt to be explicit concerning usage.

8.2 Merging Reaching Definitions in a Precedence Graph

8.2.1 Interesting Nodes and Merge Nodes Within a PG

At what points in a PG do we need to merge information, specifically reaching definitions? We need to merge definitions at the precedence section nodes in which the definitions first come together. However, as opposed to sequential control flow, a variable may not be defined along every path reaching a confluence point; as long as it is defined along *some* path that reaches a use for that variable, a definition for that variable will be available. This important distinction suggests that identifying merge points as the iterated dominance frontier of a set $\mathcal{S}$ of nodes in the PG may not be correct. To see why, examine Figure 8.4(a) with respect to reaching definitions. If v is only defined at node X, then any use of v at W, Z, or A will have that definition available, since X will always have been executed before any of these other sections execute. But $\mathrm{DF}^+(X) = \{W, Z, A\}$; clearly a merge node is not necessary when only a single definition of a variable reaches any point. Since a precedence graph guarantees execution of all predecessors, we need not be concerned about a definition of v flowing from node Y. Since Y does not define v, it does not contribute to the reaching definitions of v for the other nodes.

To see where merge operators are needed in a PG, first examine Figure 8.4(b), in which v is defined in sections X and Y, and used in section A. Since both definitions reach A without either killing the other, a merge operator is needed at A. However, we need merge only two definitions, even though there are three predecessors. Thus, a merge function for a PG only needs arguments for predecessors with definitions reaching the

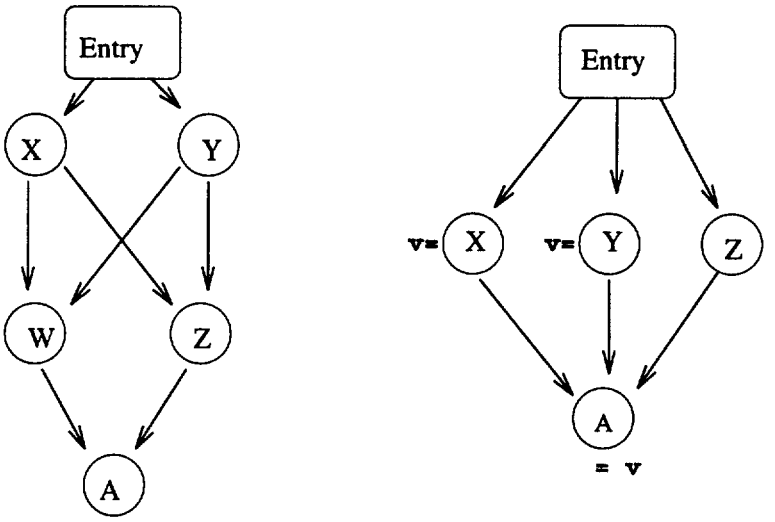


Figure 8.4 Understanding merge operator placement in PGs

confluence node along that path. This observation highlights another major difference between sequential and parallel merges; therefore we will use a new operator, the ψ-function, as the merge operator for reaching definitions within the PG [SHW93]. The ψ-function is similar to the ϕ-function in that it acts as a nonkilling definition in terms of data-flow analysis, but it is also a use for all definitions that reach the ψ-function via its arguments. By collecting multiple reaching definitions the ψ-function linearizes definition chains within a PG in the same manner as the ϕ-function within a CFG.

To identify precisely where to place parallel merge operators in a PG we begin with an important new definition:

Definition 8.2 *In a flow graph, the* meet *of nodes X and Y, M(X,Y) =*

$$\{Z \mid \forall Z_X, Z_Y \text{ with } Z_X \rightarrow Z \text{ and } Z_Y \rightarrow Z, \text{ and } \forall \text{ paths } p_X : X \xrightarrow{*} Z_X, p_Y : Y \xrightarrow{*} Z_Y, p_X \cap p_Y = \emptyset\}$$

We note that within arbitrary flow graphs the *meet* of two nodes is the dual definition to *join*, as it uses a universal quantifier as opposed to the existential quantifier of *join*. For a set of nodes $\mathcal{S}$, $\mathrm{M}(\mathcal{S})$ is defined in the usual pairwise manner: $\mathrm{M}(\mathcal{S}) = \bigcup_{X,Y \in S} \mathrm{M}(X,Y)$. We also define $\mathrm{M}^+(\mathcal{S})$ as the limit of increasing sequences analogous to that used for

join and dominance frontier:

$$\begin{aligned} \mathrm{M}^1(\mathcal{S}) &= \mathrm{M}(\mathcal{S}) \\ \mathrm{M}^2(\mathcal{S}) &= \mathrm{M}(\mathcal{S} \cup \mathrm{M}^1(\mathcal{S})) \\ \mathrm{M}^{i+1}(\mathcal{S}) &= \mathrm{M}(\mathcal{S} \cup \mathrm{M}^i(\mathcal{S})) \end{aligned}$$

The definition of join (Definition 3.1, and the basis of work to place ϕ-functions [CFR+91]) is well-known. Although in a CFG $\mathrm{J}^+(\mathcal{S}) = \mathrm{J}(\mathcal{S})$ [Wol94], the dual definition of join for PGs, meet, does not possess this property. Consider Figure 8.4(a). Let $\mathcal{S}$= $\{X, Y\}$. Then $\mathrm{M}(\mathcal{S}) = \{W, Z\}$; in fact, $A \notin \mathrm{M}(\mathcal{S})$, but $A \in M(\mathcal{S} \cup M(\mathcal{S})) = \mathrm{M}(X, Y, W, Z)$ $= \{W, Z, A\}$.

The meet of two nodes possesses one of the important properties characterizing nodes in a PG: it is unaffected by transitive edges. To prove this claim, we first formalize the concept of a transitive edge as follows:

Definition 8.3 *Edge E: $X \rightarrow Y$ added to graph G is a transitive edge if $\exists Z \in G \ni X \stackrel{*}{\rightarrow} Z \stackrel{+}{\rightarrow} Y$.*

We now show that the central concept of PGs, path reachability, is unaltered in the presence of transitive edges.

Theorem 8.1 *Path reachability in a PG is unaffected by transitive edges.*

Proof:
Consider PG G′, consisting of G plus transitive edge $E : X \rightarrow Y$. Since all edges in G exist in G′, if A reached B in G, A reaches B in G′. Now, let A reach B in G′, but assume that A does not reach B in G. Then path p_1: $A \stackrel{+}{\rightarrow} B$ in G′ must include E, otherwise no distinction is possible between paths in G and G′. Thus, p_1 must be of the form $A \stackrel{*}{\rightarrow} X \rightarrow Y \stackrel{*}{\rightarrow} B$. By Definition 8.3, $X \stackrel{*}{\rightarrow} Z \stackrel{+}{\rightarrow} Y$ in G. Thus, path p_2: $A \stackrel{*}{\rightarrow} X \stackrel{*}{\rightarrow} Z \stackrel{+}{\rightarrow} Y \stackrel{*}{\rightarrow} B$ exists in G. By contradiction, we have demonstrated equivalence of path reachability between G and G′. ▌

We next demonstrate that the meet of a set of nodes is also unaffected in the presence of transitive edges.

Theorem 8.2 *The meet relation is insensitive to transitive edges.*

Proof:
We use G and G′ as defined in the proof of Theorem 8.1, except that E is any transitive

edge added to G. We first show that for nodes X and Y, M(X,Y) in G is equal to M(X,Y) in G$'$ by means of double inclusion.

1. Let $Z \in \mathrm{M}(X,Y)$ in G. We show that $Z \in \mathrm{M}(X,Y)$ in G$'$. By Definition 8.2 for meet, the intersection of all pairs of paths in G from X and Y to predecessors of Z is empty. Now consider G$'$, which includes edge E: $A \rightarrow B$. Assume $Z \in \mathrm{M}(X,Y)$ in G, but $Z \notin \mathrm{M}(X,Y)$ in G$'$. Then, in G$'$ there exists node V such that $V \stackrel{+}{\rightarrow} Z$ with $X \stackrel{*}{\rightarrow} V$ and $Y \stackrel{*}{\rightarrow} V$. If no path from $X \stackrel{*}{\rightarrow} V$ or $Y \stackrel{*}{\rightarrow} V$ passes through A, there is no such V, since the only difference between G and G$'$ is edge E. Thus, without loss of generality, at least X, and perhaps Y, has a path to V that passes through A. But from A, no nodes are reachable in G$'$ that were not reachable in G, as Theorem 8.1 demonstrated. Hence, if there is such a V in G$'$, it exists in G, since its existence is predicated upon reachability. We conclude that since there is no such V in G, there exists no such V in G$'$. By contradiction, $Z \in \mathrm{M}(X,Y)$ in G$'$.

2. Let $Z \in \mathrm{M}(X,Y)$ in G$'$. By Definition 8.2 the intersection of all pairs of paths from X and Y to predecessors of Z is empty in G$'$. Since the edges in G are a subset of the edges in G$'$, any pair of paths from X and Y to predecessors of Z that exists in G exists in G$'$, and has empty intersection in G$'$ by assumption. Thus, that pair of paths is empty in G, and $Z \in \mathrm{M}(X,Y)$ in G.

Now consider M($\mathcal{S}$), where $\mathcal{S}$ is a set of nodes. Since $\mathrm{M}(\mathcal{S}) = \bigcup_{X,Y \in \mathcal{S}} \mathrm{M}(X,Y)$, we apply the property just proved to each pair X,Y to obtain the desired result for $\mathcal{S}$: M($\mathcal{S}$) in G equals M($\mathcal{S}$) in G$'$. ▌

Given a set of section nodes $\mathcal{S}$ defining a variable, merge operators for PGs need to be placed at the iterated meet of $\mathcal{S}$. In terms of reaching definitions, given a variable v and a set $\mathcal{S}$, where $\mathcal{S}$ is the set of section nodes in a PG defining v, ψ-functions need to be placed at $\mathrm{M}^{+}(\mathcal{S})$, where a ψ-function for v at section node A collects all definitions of v that reach A. That is, there is an argument of the ψ-function for each predecessor of A that has a definition of v reaching A. Figure 8.5 shows the case where even though an edge exists from a definition of v (in node A) to the confluence node N, the ψ-function placed at N will only collect the definitions from nodes B and C. That is because the definition at A gets killed by the definition at B in this PG. In this case, $\mathcal{S} = \{A, B, C\}$, and $\mathrm{M}(\mathcal{S}) = \mathrm{M}^{+}(\mathcal{S}) = \{N\}$, but we note that the edge $A \rightarrow N$ is a transitive edge and

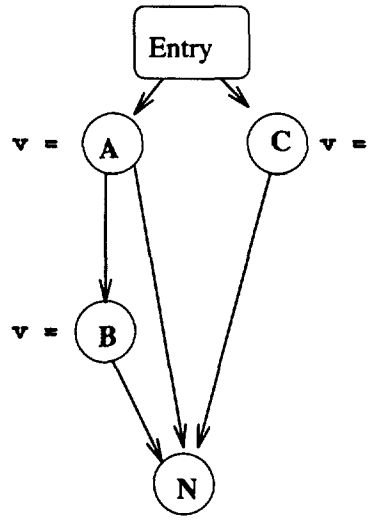


Figure 8.5 Transitive edges do not affect reaching definitions in PGs

$\mathrm{M}(B, C) = \{N\}$.

8.2.2 Proving Correct and Minimal Placement

We will first prove that it is sufficient to place ψ-functions at $\mathrm{M}^+(\mathcal{S})$. The concept of iterated meet is a correction of the ψ-function placement method suggested earlier [SHW93], in that the iterated meet is smaller and, in fact, the minimal set.

How do ψ-functions affect reaching definitions in a PG? If node N is reached by ψ-function s and s is reached by definition d (where s collects d as a ψ-argument), then d reaches N indirectly via a ψ-function. In general, it may be that one or more ψ-functions lie on the path from d to N. In that case, d reaches N indirectly via a *ψ-chain.* Thus, a definition or ψ-function in a PG that reaches node N in the sense of Definition 8.1 is called a *direct* reaching definition, whereas a definition reaching node N via a *ψ-chain* is called an *indirect* reaching definition.

We now prove that placing ψ-functions at the iterated meet of the set of nodes that define a variable maintains the following properties:

1. A unique reaching definition exists for each variable use.
2. A ψ-function will collect all definitions (directly or indirectly) that could reach a node before ψ-function placement.
3. $\mathrm{M}^+(\mathcal{S})$ identifies the minimal set at which to place ψ-functions.

Theorem 8.3 *In a PG, with ψ-functions for v placed at $M^+(\mathcal{S})$, all uses of v within node N will be reached (in the sense of Definition 8.1) by exactly one definition (including ψ-functions) of v.*

Proof:

Let G be a PG before placing ψ-functions, and G_ψ be the same graph after ψ-function placement. Within G, let the set of nodes with definitions of v be $\mathcal{S}$, $\mathcal{S}' \subseteq \mathcal{S}$ be the set of nodes in $\mathcal{S}$ having definitions that reach N, and $\mathcal{T} \subseteq \mathcal{S}$ be the set of nodes in $\mathcal{S}$ with paths that reach N.

EXISTENCE. We first show that any use that had at least one reaching definition in G has at least one reaching definition in G_ψ. Since $\mathcal{S}' \neq \emptyset$, let $A \in \mathcal{S}'$ in G. Then all paths $p_A : A \xrightarrow{*} Z$, with $Z \rightarrow N$, contain no definitions of v (except at A). For all p_A in G_ψ, if the definition of v in A does not directly reach N, then there must be at least one ψ-function along some p_A. In this case, at least one ψ-function reaches N.

UNIQUENESS. We consider cases:

(*i*) Only one $W \in \mathcal{S}'$ reaches N ($|\mathcal{S}'| = 1$). In this case, the definition in W kills any other definitions that may exist in nodes of $\mathcal{T}$. Then, $\forall t1, t2 \in \mathcal{T}$, $\exists p_{t1}$:$t1 \xrightarrow{+} N$, p_{t2}: $t2 \xrightarrow{+} N$, $W \in p_{t1} \cap p_{t2}$. Thus, $N \notin \mathrm{M}(\mathcal{T})$, and more generally, no node on any p_W:$W \xrightarrow{+} N$ (except, perhaps, W) is an element of $\mathrm{M}(\mathcal{T})$. Repeating this argument, no node in any p_W - $\{W\} \in \mathrm{M}^+(\mathcal{T})$. Thus, in G_ψ only the definition of v in W reaches N, since no additional definitions (ψ-functions) created in G_ψ can directly reach N.

(*ii*) Multiple definitions of v reach N from $\mathcal{S}'$, with $N \in \mathrm{M}^+(\mathcal{S})$. Then a ψ-function will be placed at the beginning of node N in G_ψ, and uses of v within N will be reached by that ψ-function.

(*iii*) Multiple definitions of v reach N from $\mathcal{S}'$, with $N \notin \mathrm{M}^+(\mathcal{S})$. Assume N is reached by more than one definition from members of $\{\mathcal{S} \cup \mathrm{M}^+(\mathcal{S})\}$. Call this set R_1. Then either (a) $N \in \mathrm{M}(\mathrm{R}_1)$, which contradicts our assumption, or (b) $\forall A, B \in \mathrm{R}_1$ $\mathrm{M}(A, B)$ is nonempty (since A and B reach N), and let R_2 be the set of all elements of $\mathrm{M}(\mathrm{R}_1)$ having paths that reach N. Repeating this process, we note that R must converge at a limit set R_+, since R_{n+1} is always composed of nodes closer to N, along the paths from nodes in R_1 to N, than the nodes in R_n. If the set R_+ consists of exactly one node (it cannot be zero by the existence proof), we have a contradiction of assumption. If it contains more than one node (which cannot include N by assumption), R has not converged. A contradiction is again reached as long as G contains a finite number of nodes.

Thus, in all cases, we have shown that in G_ψ precisely one reaching definition will

exist for each use that had at least one reaching definition in G. ▌

Theorem 8.4 *Within a PG, with ψ-functions placed at $M^+(S)$, any use of v at node N will be reached directly or indirectly by all definitions of v that reached N before placing ψ-functions.*

Proof:
Let G be a PG before placing ψ-functions, and G_ψ be the same graph after ψ-function placement. We consider two cases:
(*i*) Only one definition of v reaches N in G. This case is handled similarly to case (*i*) in Theorem 8.3, and the single definition that reached N in G will reach N in G_ψ.
(*ii*) Multiple definitions of v reach N in G. All definitions for v from node A that reach N in G reach N indirectly in G_ψ via a ψ-chain. To show this, consider all paths p from A to N in G. By Definition 8.1, no paths from A to N in G contain definitions of v. Since, by assumption, a definition of v in G_ψ does not reach N from A, by Definition 8.1 there must exist a definition along some path from A to N that did not exist in G. That definition can only be a ψ-function. If there is just one ψ-function along the path then it collects all definitions reaching it, and that ψ-function will now reach N, resulting in the definition of v reaching N indirectly. If there is more than one ψ-function along any path from A to N, the argument is repeated. By induction, a definition of v in A will reach N via a ψ-chain. Thus, we have shown that reachability of all definitions is maintained when placing ψ-functions. ▌

We now show that $M^+(S)$ is the minimal set at which to place ψ-functions.

Theorem 8.5 *Within a PG, for a set of nodes S which define v, $M^+(S)$ is the smallest set at which to place ψ-functions in order to insure unique reaching definitions at all nodes.*

Proof:
Given S for variable v, consider any element $N \in M^+(S)$. Let $N \in M^j(S)$, for the minimum $j \geq 1$. Then $\exists X, Y \in M^{j-1}(S)$ (where $M^0(S) = S$) such that all pairs of paths from X and Y to predecessors of N are empty. Since X and Y contain definitions of v (either assignments to v or ψ-functions for v), the definitions at X and Y (or, perhaps, a later definition of v within some node along one of these disjoint paths) both reach N. Thus, by removing the ψ-function for v at N, any use of v within N would be reached by multiple definitions. ▌

8.3 Algorithms and Correctness

In this section, we first discuss several necessary details for implementation. Next, we present the complete algorithm to insert ϕ- and ψ-functions, and correctly create and generate the proper reaching definitions as arguments. We also demonstrate the correctness and safety of these algorithms, which have been successfully implemented in Nascent.

8.3.1 An Introduction to ψ-Function Placement

In our implementation, the compiler finds for each variable the set of nodes in each local CFG or PG where the variable is assigned. A section node in the PG is considered to have an assignment to a variable if the code within the corresponding section assigns the variable. Function (ϕ or ψ) placement is done jointly, with the function type distinguished by the type of node at the confluence point: a supernode or section node tells us that a ψ-function is required.

Our method can result in a ψ-function being initially placed at a PG join point where only one reaching definition within the PG block is defined. This situation can occur in two ways: a variable is defined within a section node that has only one section successor, or a variable is defined within a supernode such that only one definition of that variable reaches outside the parallel block. Although such a ψ-function will have only a single argument, it is a necessary step for the chaining algorithm within parallel sections, as will be detailed in §8.3.3.

8.3.2 Contrasting ψ- and ϕ-Functions

It has already been noted that placing ψ-functions at the DF of nodes in the PG may result in only a single argument to this function. Although every predecessor will have a reaching definition for each variable (we always add an initial definition at program *Entry*), we do not want to include reaching definitions from outside the current parallel block as an argument to a ψ-function – that definition will always be the default reaching definition if none exists within the current parallel block. When chaining arguments to ϕ-functions we *know* how many arguments must be filled in with appropriate definitions – the number of predecessors. But with ψ-functions, we only fill in arguments as needed. There will be at least one ψ-argument, since the presence of the function tells us that a definition exists from another section node in this parallel block. But, there may be *just*

that one argument to a ψ-function. If that is the case, a *singleton-ψ* is created, which serves a special purpose.

There is a crucial distinction to be made between ψ- and ϕ-functions. A ϕ-function is a variable assignment; the choice of assignment is given by the predecessor number of the path taken to reach the merge. A ψ-function, on the other hand, reports anomalous or multiple updates. Hence, ψ-arguments are not necessary for each predecessor of a section node – only those in which an update occurs for a given variable. This property implies that the order of the arguments for ψ-functions is not important, since there is not a one-to-one relationship between arguments and predecessors.

Confluence points in the PG do not represent different possible paths for the sections (as they would in a sequential CFG), since all parallel sections are executed. Rather, they identify those parallel sections that *must* be executed before the merge section (hence altering the copy-in status for all variable in the merge section) and these are sections that *may* redefine variables whose definitions reach beyond the merge section. Thus, even if there is a singleton-ψ, its reaching definition is critical, since it must be propagated to other sections waiting upon the confluence point. This situation is reflected in the complete algorithm, where we propagate the reaching definition of the argument in this case rather than the ψ-function. Once it has served its purpose, we can delete a singleton-ψ, since we have discovered that only one definition reaches this merge point from within the parallel section.

Singleton-ψ's are essentially used as a temporary holding pen for single reaching definitions between explicit parallel sections. Propagating the argument's reaching definition in this case also eliminates redundant links to names, which can otherwise arise. Consider the example program from Figure 8.2, shown in its SSA form in Figure 8.10. Section D uses variables `a`, `b`, `c`, and `f` to redefine `c`. Variable `a`'s reaching definition comes from outside the parallel block and `f`'s reaching definition comes from the ψ-function at *EntryD*, which merges the definitions from Sections A and B. But `b` and `c` have their reaching definitions propagated from single wait–predecessor sections, A and B respectively. To correctly propagate these values to D, a ψ-function (call it $\mathtt{b}' = \psi$) is created in *EntryD* for `b` (and likewise for `c`). Yet if $\mathtt{b}' = \psi(\mathtt{b}_5)$ is treated as a normal definition, $\mathtt{b}'$ would be pushed onto the stack of definitions for `b`. When used in the new generation of `c` a pointer to $\mathtt{b}'$ would be inserted, which only points to $\mathtt{b}_5$. Thus, $\mathtt{b}'$ would be just another link to $\mathtt{b}_5$, which is redundant. The ψ-function *was* necessary to propagate the correct reaching definition of `b` to section D, but after visiting all sections

waiting upon this definition of $\mathtt{b}$, $\mathtt{b}' = \psi$ can be deleted.

Notice, also, that in the ψ-function creation phase, the variable generations of $\mathtt{b}$ at Sections A ($\mathtt{b}_5$) and D ($\mathtt{b}'$) will create another ψ-function at $tail_{P1}$ with arguments $\mathtt{b}_5$ and $\mathtt{b}_5$. By eliminating redundant links this duplication is detected, reducing this ψ-function to a singleton-ψ, hence propagating the correct reaching definition of $\mathtt{b}$ to the rest of the program before being deleted.

8.3.3 Depth-first Renaming

Computing the iterated meet seems somewhat impractical from its definition. After placing ϕ-functions the technique of *chaining* transforms each variable definition into a unique name and each use into the name of its unique reaching definition [CFR+91]. The method employed to perform this renaming is depth-first, in that it recursively traverses the dominator tree in a depth-first order, keeping a stack of current definitions for each variable. The key property that the chaining scheme satisfies is that at each node the correct "current" definition (an original definition or ϕ-function) of each variable is the most recent definition on the depth-first path to this node from *Entry*, i.e., the definition on top of the definition stack [CFR+91, Lemma 10]. In fact, a depth-first traversal of any spanning tree of the CFG will also satisfy this property. Unfortunately, a depth-first traversal of the nodes of a PG will not satisfy this key property with merge operators at $\mathrm{M}^+(\mathcal{S})$. For instance, in Figure 8.6, no ψ-function is needed at node C for either x or y,

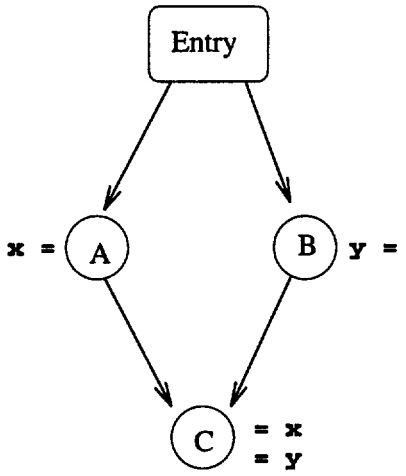


Figure 8.6 All merge points in a PG do not require ψ-functions

since only one definition of each variable reaches node C (in the sense of Definition 8.1). Suppose the depth-first traversal of the PG visits node C after node A; when visiting

node C, the current definition of variable x will be the definition in A, but the current definition of variable y will be wrong.

8.3.4 Efficient Implementation

What method can be used that is relatively efficient and yet correctly propagates information between section nodes of a PG? We need to look more closely at how information flows between nodes in a PG, keeping in mind that a precedence graph has different semantics compared to a CFG.

Since information flowing through the PG is described in terms of reachability, we have found the concept of *reaching frontier* useful. This concept describes reachable nodes in a PG in a way that is analogous to the dominance frontier for nodes within a CFG.

Definition 8.4 *The* reaching frontier *of X, RF(X) =*

$$\{Z \mid X \textit{ reaches a predecessor of } Z\textit{, but } X \textit{ does not reach all predecessors of } Z\}$$

The reaching frontier of a set $\mathcal{S}$, $\mathrm{RF}(\mathcal{S})$, is defined to be the union of the reaching frontiers of all elements of $\mathcal{S}$, i.e., $\mathrm{RF}(\mathcal{S}) = \bigcup_{X \in \mathcal{S}} RF(X)$. The *iterated reaching frontier*, $\mathrm{RF}^+(\mathcal{S})$, is defined similarly to that for join, meet, and dominance frontier. The reaching frontier is used to relate important properties between the meet and dominance frontier. To implement the placement of operators merging information within a PG, we would like to show that $\mathrm{M}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S}) \subseteq \mathrm{DF}^+(\mathcal{S})$.

How are the meet and reaching frontier related? The analogous relations in sequential CFGs, join and dominance frontier, are shown to be equal when iterated, with the provision that $Entry \in \mathcal{S}$. However, *Entry* adds no information to either the meet or the reaching frontier in a PG. $\mathrm{M}(Entry, X) = \emptyset \; \forall X$, since *Entry* reaches all nodes, and thus there is always a path from *Entry* to any node on any path from X. Also, $\mathrm{RF}(Entry) = \emptyset$, since *Entry* reaches all predecessors of all nodes.

We can also show that $\mathrm{RF}^+(\mathcal{S}) \neq \mathrm{M}^+(\mathcal{S})$. Simply choose the set $\mathrm{T} = \{X, Entry\}$. Then $\mathrm{M}(\mathrm{T}) = \emptyset$, so $\mathrm{M}^+(\mathrm{T}) = \emptyset$, while $\mathrm{RF}(\mathrm{T})$ clearly may not be empty. We now show that in general $\mathrm{M}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S})$.

Theorem 8.6 $M^+(\mathcal{S}) \subseteq RF^+(\mathcal{S})$

Proof:

Let $Z \in \mathrm{M}(\mathcal{S})$. Then there is a node $X \in \mathcal{S}$ such that X has a path that reaches a

predecessor of Z, but X cannot reach all predecessors of Z or else there would be no path from any other node that did not intersect some path from X to each predecessor of Z (which would imply that $Z \notin \mathrm{M}(S)$). So, we have $Z \in \mathrm{RF}(X)$ and $Z \in \mathrm{RF}(S)$. Finally, $\mathrm{M}(X) \subseteq \mathrm{RF}(X) \Longrightarrow \mathrm{M}^+(X) \subseteq \mathrm{RF}^+(X)$. ∎

We also show that $\mathrm{DF}(S)$ is not in general a subset or superset of $\mathrm{RF}(S)$. In Figure 8.7, $\mathrm{DF}(X) = \{A,Z\}$, but $\mathrm{RF}(X) = \{Z\}$, since it reaches all predecessors of A. It is also easy to find a graph where X reaches a predecessor of Z but does not dominate any predecessor of Z, so $Z \in \mathrm{RF}(X)$, but $Z \notin \mathrm{DF}(X)$.

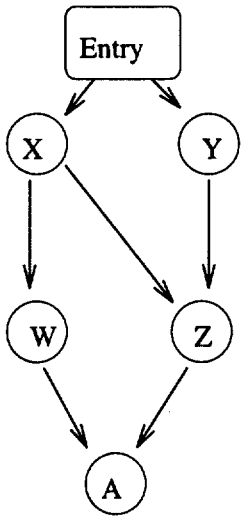


Figure 8.7 $\mathrm{DF}(S)$ and $\mathrm{RF}(S)$ are sometimes unrelated

Next, we show that the iterated dominance frontier is a superset of the iterated reaching frontier on all graphs.

Theorem 8.7 *$DF^+(S) \supseteq RF^+(S)$*

Proof:
It has been shown [CFR+91, Lemma 4] that for any node Z that X reaches, some node $Y \in \{X \cup \mathrm{DF}^+(X)\}$ dominates Z. Now, for any node Z that X reaches, if Z is in $\mathrm{RF}(X)$, then Z is in $\mathrm{DF}^+(X)$; this is because some node in $\mathrm{DF}^+(X)$ must dominate Z. Choose a path p from X to Z. Let Y be the last node on p in $\{X \cup \mathrm{DF}^+(X)\}$; Y must be Z. If Y is not Z, then Y dominates all predecessors of Z, so there is a path from Y to all predecessors of Z; thus there is a path from X to all predecessors of Z, and Z is not in $\mathrm{RF}(X)$.

Thus, $DF^+(X) \supseteq RF(X)$. Hence, $DF^+(\mathcal{S}) \supseteq RF(\mathcal{S})$. $RF^2(\mathcal{S}) = RF(\mathcal{S} \cup RF(\mathcal{S})) \subseteq RF(\mathcal{S} \cup DF^+(\mathcal{S})) \subseteq DF^+(\mathcal{S} \cup DF^+(\mathcal{S})) = DF^+(\mathcal{S})$. By induction, $DF^+(\mathcal{S}) \supseteq RF^+(\mathcal{S})$. ▮

In general $RF^+(X) \neq DF^+(X)$. Although $DF^+(X) \supseteq RF^+(X)$, the converse is not necessarily true. Consider Figure 8.8. $DF(X) = \{B,Z\}$, and $DF^2(X) = DF^+(X)$

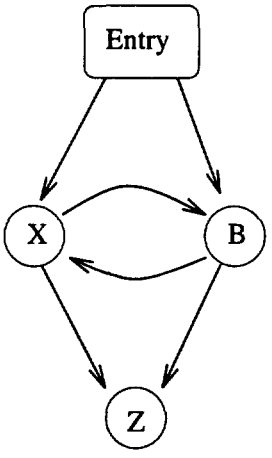


Figure 8.8 $RF^+(X) \not\supseteq DF^+(X)$

$= \{B,Z,X\}$. However, $RF(X) = \{B\}$, and $RF^2(X) = RF^+(X) = \{B,X\}$. Thus, by counterexample, $RF^+(X) \not\supseteq DF^+(X)$.

But, we note that the example above contains a cycle. We are interested in placing ψ-functions in a PG, which we know to be acyclic. We next show that in a DAG $RF^+(\mathcal{S}) = DF^+(\mathcal{S})$.

Theorem 8.8 *In a DAG, $RF^+(\mathcal{S}) = DF^+(\mathcal{S})$*

Proof:
Given a DAG, we demonstrate two preliminary lemmas.

- **Lemma 8.1** *$RF^+(\mathcal{S}) \supseteq DF(\mathcal{S})$.*

  Let $X \in \mathcal{S}$ and let Z be in $DF(X)$. Then X dom A, a predecessor of Z. X $\overline{dom}$ B, some other predecessor of Z, since X $\overline{dom}$ Z. If X does not reach B, then Z is in $RF(X)$. So assume that X reaches B.

  We now show that on some path from X to B, there exists a C such that C is in $RF(X)$. Since X $\overline{dom}$ B, consider a path from entry to B such that X is not on

the path (there must be at least one such path). Let C be the first node on this path that X can reach (C may be B). Then since X can reach C, but not the predecessor of C on this path, C is in $\mathrm{RF}(X)$.

Next, note that C cannot reach A. Otherwise, we would have the path Entry $\rightarrow C \rightarrow A$ (which cannot go through X since the graph is acyclic) that does not pass through X, contradicting the fact that X dom A.

But, this condition means that Z is in $\mathrm{RF}(C)$, since C reaches Z through B, but cannot reach A. We already know that C is in $\mathrm{RF}(X)$, so we have shown that Z is in $\mathrm{RF}^+(X)$. ▮

- **Lemma 8.2** *$RF^+(S) \supseteq DF^+(S)$.*

  Given Lemma 8.1, we know that $\mathrm{RF}^+(X) \supseteq \mathrm{DF}(X)$. So, $\mathrm{RF}^+(\mathcal{S}) \supseteq \mathrm{DF}(\mathcal{S})$. $\mathrm{DF}^2(\mathcal{S}) = \mathrm{DF}(\mathcal{S} \cup \mathrm{DF}(\mathcal{S})) \subseteq \mathrm{RF}^+(\mathcal{S} \cup \mathrm{RF}^+(\mathcal{S})) = \mathrm{RF}^+(\mathcal{S})$. By induction, $\mathrm{DF}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S})$. ▮

Lemma 8.2 together with Theorem 8.7 gives us our result. ▮

Since $\mathrm{M}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S}) \subseteq \mathrm{DF}^+(\mathcal{S})$ (with $\mathrm{RF}^+(\mathcal{S}) = \mathrm{DF}^+(\mathcal{S})$ in a DAG), we have shown that placing ψ-functions within a PG at the $\mathrm{DF}^+(\mathcal{S})$ is a safe approximation for the somewhat smaller set of $\mathrm{M}^+(\mathcal{S})$. However, for the common depth-first implementations using renaming, placing merge operators at $\mathrm{DF}^+(\mathcal{S})$ may well be the method of choice.

How conservative is the use of $\mathrm{DF}^+(\mathcal{S})$ as an approximation for $\mathrm{M}^+(\mathcal{S})$? First, if there is only one member of $\mathcal{S}$, then $\mathrm{M}^+(\mathcal{S})$ will be empty, while $\mathrm{DF}^+(\mathcal{S})$ will usually not be empty. Second, $\mathrm{DF}^+(\mathcal{S})$ assumes a definition lies along all possible paths. Thus, in the case of Figure 8.9 where $\mathcal{S} = \{A, C\}$, $\mathrm{M}(\mathcal{S}) = \mathrm{M}^+(\mathcal{S}) = \{E\}$, while $\mathrm{DF}(\mathcal{S})$ includes D. Third, $\mathrm{M}^+(\mathcal{S})$ is insensitive to transitive edges, while $\mathrm{DF}^+(\mathcal{S})$ is not. Again, examine Figure 8.9, where $\mathrm{DF}^+(\mathcal{S}) = \{D, E, F\}$. A ψ-function is only needed at E, but the insensitivity to transitive edges of $\mathrm{DF}^+(\mathcal{S})$ adds node F to its set.

However, extra ψ-functions are safe, since they only pass along the information collected at those points. Thus, merging information at $\mathrm{DF}^+(\mathcal{S})$ within a PG has been shown to be a safe method, and is relatively efficient since it can be performed with the same complexity as that for ϕ-function placement.

In terms of the space requirements for placing ψ-functions within the PG, we can use the space consumed by ϕ-function placement as an upper bound, since $\mathrm{M}^+(\mathcal{S}) \subseteq$

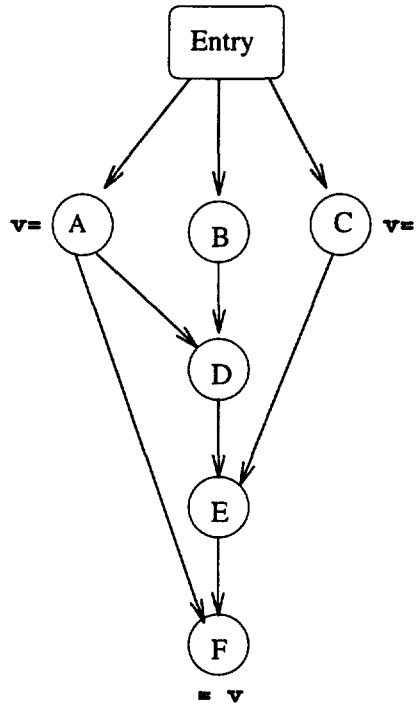


Figure 8.9 Using $\mathrm{DF}^+(S)$ as an approximation for $\mathrm{M}^+(S)$

$\mathrm{DF}^+(S)$. While the worst case scenario could be $O(N^2)$, in practice most programs exhibit linear space requirements when placing ϕ-functions [CFR+91, Hav94].

8.3.5 Complete Algorithms for PGs

The complete transformation of an intermediate representation into parallel SSA form is accomplished in two main phases: function placement and chaining. For these algorithms, *successor* and *predecessor* always refer to nodes in the local CFG, while *children* refers to the dominator tree of the associated local CFG.

We describe here the data structures used for the following algorithms:

- *A(V)* – A list of all nodes with assignments to variable V.
- *symbol(tuple)* – A function that returns the variable symbol (name) associated with this tuple, if it exists. Returns *null* otherwise.
- *V.CurrentDef* – A pointer to the current definition (tuple) of symbol V. Logically points to the top of a definition stack. Initialized to *source*.
- *t.SavedDef* – A pointer to the current definition of *symbol(t)* before processing this

tuple. Used to logically pop definitions off a stack when returning from recursive calls down the dominator tree.

- *T(∗)* – Stack of nodes to hold section nodes of PG for popping in topological order. Initialized to *null*.
- *DF(N)* – Local dominance frontier for node N.
- *WhichPred(N,Q)* – An integer indicating which predecessor of Q in the local CFG is N.
- *Work_List* – An unordered list of CFG nodes. For each variable V, Work_List is initialized to *A(V)*, all assignments to V.
- *HasFunc(∗)* – A reference field to a variable in each CFG node. HasFunc(N) = V means block N already has a ϕ- or ψ-function added for variable V.
- *Work(∗)* – A reference field for each local CFG or PG node. Work(N) = V means that node N has already been added to the Work_List for variable V.
- *set_delete(ψ)* – Marks a singleton ψ-function for later deletion.

The placement of ψ-functions is done concurrently with the placement of ϕ-functions, as is shown in Algorithm 8.1. Functions are placed at the iterated dominance frontier of each assignment per given variable, V. A(V), the list of all initial assignments to V, is found in one pass through the program, storing the definitions of V as a linked list, as was done for Algorithm 3.1. We do not have to reinitialize the fields *HasFunc* and *Work* as each variable is processed, since they are just pointers to each variable under consideration.

At each iterated dominance frontier node[†] we distinguish whether to place a ϕ- or ψ-function by the *type* of node encountered (lines 16 - 20 in Algorithm 8.1) – a basic block node in a local CFG always receives a ϕ-function, while PG nodes indicate that a ψ-function is required. However, note that a ψ-function is not actually placed within the PG node, but rather within the *Entry* node of the corresponding section, unless the PG node is *coend*, in which case it is placed within the *tail* node of the enclosing supernode in the outer local CFG. In this way we correctly propagate definitions reaching the end

[†]Although line 13 in Algorithm 8.1 looks at each Q in the dominance frontier of N, lines 22 - 28 effectively iterate the dominance frontier by placing nodes back into the worklist.

Given: $A(V)$, $\forall\ V$.
Do: compute $DF(\ N\ )$, $\forall\ N \in EFG$.
Initialize with lines 1 - 4
Result: Extended SSA form with ψ-functions for PGs

```
 1:      for all nodes N do
 2:          HasFunc(N) ← ∅
 3:          Work(N) ← ∅
 4:      endfor

 5:      for each variable V do
 6:          Work_List ← ∅
 7:          for each N in A(V)
 8:              Work(N) ← V
 9:              Work_List ← Work_List ∪ { N }
10:          endfor
11:          while Work_List ≠ ∅ do
12:              take N from Work_List
13:              for each Q in DF(N) do
14:                  if HasFunc(Q) ≠ V then
15:                      HasFunc(Q) ← V
16:                      if Q is a basic block of local CFG then
17:                          add-φ(Q,V)
18:                      else if Q is a member of PG then
19:                          add-ψ(Q,V)
20:                      endif
21:                  endif
22:                  if Work(Q) ≠ V then
23:                      Work(Q) ← V
24:                      Work_List ← Work_List ∪ { Q }
25:                      if Q is a section Exit basic block then
26:                          Work_List ← Work_List ∪ {P_Q, S_Q}
27:                      endif
28:                  endif
29:              endfor /* each Q in DF */
30:          endwhile
31:      endfor  /* each variable V */

32:      add-φ(N,V)
33:          i ← number of predecessors of N
34:          place V = φ( V_1, V_2, ..., V_i ) at the beginning of basic block N,
35:              where V_j corresponds to the j^th predecessor of N

36:      add-ψ(N,V)
37:          if N is a section node, then N' ← Entry_N
38:          if N is a coend node, then N' ← tail_{P_N}
39:          place V = ψ at N'
```

Algorithm 8.1 Placement: locations for ϕ- and ψ-functions

Given: Initialized data structures.
Do: Call *ChainEFG($Entry_{main}$)*
Result: Extended SSA form with ψ-functions for PGs

```
1:  ChainEFG(N)
2:      if N is a node of a local CFG then
3:          for all tuples t ∈ N, in forward order do
4:              V ← symbol( t )
5:              if t is an ordinary use of V then
6:                  link( t ) ← V.CurrentDef
7:              endif
8:              if t is an ordinary definition or φ-function of V then
9:                  t.SaveDef ← V.CurrentDef
10:                 V.CurrentDef ← t
11:             else if t is a ψ-function of t then
12:                 t.SaveDef ← V.CurrentDef
13:                 eliminate duplicate arguments to ψ
14:                 num ← number of ψ arguments
15:                 if num = 1 then
16:                     set_delete(ψ)
17:                     V.CurrentDef ← link( t )
18:                 else if num > 1 then
19:                     V.CurrentDef ← t
20:                 endif
21:             endif
22:         endfor  /* all tuples in N */
23:     endif
```

Algorithm 8.2 Chaining an EFG: correctly inserting links

```
24:         if N is a supernode then traversePG( Entry_N ) /* cobegin for N */
25:         if N is a node of a local CFG then
26:             for each Q ∈ Succ(N) do /* Succ(N) in CFG */
27:                 j ← WhichPred(N,Q)
28:                 for each φ-function merge tuple f in Q do
29:                     V ← symbol(f)
30:                     link( j^th argument of f ) ← V.CurrentDef
31:                 endfor
32:             endfor
33:             for each Q ∈ Children(N) do /* children in dom tree */
34:                 ChainEFG(Q)
35:             endfor
36:             for all tuples t ∈ N, in reverse order do
37:                 if t is a definition tuple do
38:                     V ← symbol(t)
39:                     V.CurrentDef ← t.SaveDef
40:                     if t is a ψ-function & set_delete( t ) then
41:                         remove t and its arguments
42:                     endif
43:                 endif
44:             endfor
45:         endif
46: end ChainEFG
```

Algorithm 8.2 (cont.)

Given: Precedence Graph E
Do: Call *traversePG(E)*
Result: Correct node traversal in a PG

```
 1: traversePG(E)
 2:       call dfst(E) /* Algorithm 8.4 */
 3:       while T ≠ ∅ do
 4:           M ← pop(T)
 5:           if M is a section node do
 6:               ChainEFG( Entry_M )
 7:               for all ψ-function merge tuple t ∈ Entry_M do
 8:                   V ← symbol( t )
 9:                   t.SaveDef ← V.CurrentDef
10:                   V.CurrentDef ← t
11:               enddo
12:               for all φ-function merge tuples t ∈ Exit_M do
13:                   V ← symbol( t )
14:                   t.SaveDef ← V.CurrentDef
15:                   V.CurrentDef ← t
16:               enddo
17:               for each Q ∈ Succ(N) do /* Succ(N) in PG graph */
18:                   if Q is a section node, then Q' ← Entry_Q
19:                   if Q is a coend node, then Q' ← tail_PQ
20:                   for all ψ-function merge tuples f ∈ Q' do
21:                       V ← symbol( f )
22:                       if V.CurrentDef is contained within enclosing parallel block then
23:                           add arg = ψ-argument to f with link( arg ) ← V.CurrentDef
24:                       endif
25:                   enddo
26:               enddo
27:           enddo /* of section node */
28:           R ← M
29:           while R ≠ parent(Top(T)) do
30:               for all φ-function merge tuples t ∈ Exit_R do
31:                   V ← symbol( t )
32:                   V.CurrentDef ← t.SaveDef
33:               endfor
34:               for all ψ-function merge tuples t ∈ Entry_R do
35:                   V ← symbol( t )
36:                   V.CurrentDef ← t.SaveDef
37:                   if set_delete( t ) then
38:                       remove t and its arguments
39:                   endif
40:               endfor
41:               R ← parent(R) /* parent set in dfst */
42:           endwhile
43:       endwhile
44: end traversePG
```

Algorithm 8.3 Correct traversal of nodes in the PG

of a parallel section to the sequential flow which follows the supernode in the enclosing local CFG. The distinction made to determine which type of merge node to create also enables a single field, *HasFunc*, to be used for each node; there can never be both a ϕ- and ψ-function placed at the same node.

The other importance difference between ϕ- and ψ-functions in the placement phase can be seen by examining the *add-ϕ* and *add-ψ* routines in Algorithm 8.1. When a ϕ-function is placed at a node, its arity is fixed at i, where i is the number of CFG predecessors of the node. On the other hand, when a ψ-function is placed at a node, we do not know its arity, other than it will be at least one. There is not necessarily a correspondence between ψ-arguments and PG predecessors. Remember, a ψ-argument reflects a definition of that variable within the corresponding parallel section. It may be that no definition of the variable exists within a predecessor section for some ψ-function. It is in the next phase, renaming, that arguments are added to ψ-functions.

Once ϕ- and ψ-function placement is accomplished, the chaining phase is invoked. Algorithm 8.2 fills in the correct argument pointers in the case of ϕ-functions and creates ψ-arguments when needed, filled in with the current reaching definition for each variable. This algorithm also links each ordinary use to its unique reaching definition.

The chaining algorithm we present in this chapter works the same as Algorithm 3.2 when traversing a local CFG except that ψ-functions as well as ϕ-functions are treated as variable generations and pushed logically onto a definition stack. However, when looking for ϕ-functions at CFG successors, we will never examine a node which could contain both a ϕ-function and a ψ-function. A ψ-function can only be placed at two types of nodes: Section entry nodes and the tail node for a supernode. In the former case, we have a node with no nodes in its dominance frontier, and in the latter we have a node with exactly one predecessor.

Algorithm 3.2 performs a depth-first traversal of the dominator tree of the CFG. We modify this algorithm for parallel constructs as follows:

1. Begin the traversal with the *Entry* node for G_{main}.
2. When visiting a basic block node or *Entry* node, the algorithm works the same as originally presented.
3. When visiting a supernode, the procedure recurses to perform a traversal of the nodes in the corresponding PG. The order of traversal of these nodes is important: the traversal of section nodes must preserve topological order — that is, every

```
(p)     a1 = 2
(p)     b1 = 3
(p)     c1 = 4
(p)     if (Q) then
            Parallel Sections
            Section A
(s)             if (P) then
(t)                 b2 = a1 * 5
                else
(u)                 b3 = a1 + 7
(u)                 f1 = b3 * a1
(v)             endif
(v)             b4 = φ(b2, b3)
(v)             f2 = φ(f0, f1)
(ExitA)         b5 = φ(b4, b1)
(ExitA)         f3 = φ(f0, f2)
            Section B
(w)             c2 = c1 + 15
(w)             f4 = c2 * 16
(ExitB)         c3 = φ(c1, c2)
(ExitB)         f5 = φ(f0, f4)
            Section C, Wait(A)
(x)             d1 = b5 * a1
(ExitC)         d2 = φ(d0, d1)
            Section D, Wait(A, B)
(EntryD)        f6 = ψ(f3, f5)
(y)             c4 = a1 * b5 + c3 * f5
(ExitD)         c5 = φ(c3, c4)
            End Parallel Sections
(tail_P1)       f7 = ψ(f3, f6)
(n)             d3 = d2 + f7
(p)     else
(q)         d4 = 23
(r)     endif
(r)     b6 = φ(b5, b1)
(r)     c6 = φ(c5, c1)
(r)     d5 = φ(d3, d4)
(r)     f8 = φ(f0, f7)
(r)     e1 = a1 + b6 * c6 * d5
```

Figure 8.10 SSA form of parallel program

predecessor of a section node must be visited before visiting the section node itself. Since the PG is acyclic, it is fairly easy to discover a correct order. We call the routine *dfst*() to build a correct traversal order (see Algorithm 8.4 and Theorems 8.12 - 8.14).

4. When visiting a section node, singleton ψ-functions can be identified for deletion (a separate pass is not needed to actually delete these functions, since the deletion occurs at lines 37 - 39 of Algorithm 8.3, the same time definitions are popped off the stack). First, for each ψ-function at this section node, remove any duplicate ψ-arguments. If there is only one remaining ψ-argument, then that argument can be marked for future deletion. If there is more than one remaining ψ-argument, the ψ-function is necessary. The procedure then recurses to visit the dominator tree of the corresponding local CFG. Insertion of ψ-arguments is done to the wait-dependence successors in a fashion similar to the chaining process for arguments of ϕ-functions.
5. When visiting an *Exit* node for a section, the SSA name for every variable modified in that section must be propagated back to the section node, as though there were an assignment in that section node. Due to slice edges, all variables defined within the section will have corresponding ϕ-functions at the section *Exit*.
6. Similarly, when visiting a *coend* node, each SSA name modified in the parallel block must be propagated to the corresponding supernode. We accomplish this task by placing ψ-functions for *coend* nodes at the parallel block *tail* node. If only a single reaching definition of a variable reaches *coend*, a singleton-ψ will be created.

The SSA form of the parallel program (where the EFG includes *slice* edges) in Figure 8.2 is shown in Figure 8.10.

A major revision to Algorithm 3.2 occurs when a supernode is encountered. At this point we must traverse the section nodes of the corresponding PG by calling *traversePG*(), and recursively calling *ChainEFG*() on each local local CFG (see Algorithm 8.3). If that local local CFG has a supernode, then *traversePG*() will again be called. Thus, *ChainEFG*() and *traversePG*() seesaw back and forth as needed.

8.3.6 Safety and Correctness of the Algorithms

We show in this section that the algorithms presented perform as intended. We first demonstrate that ϕ-functions are correctly placed, and that ψ-functions are placed at all

points identified by Definition 8.2. Our algorithm may place ψ-functions at *more* points than required, but these functions are useful for implementation, notably as singleton-ψs, and are deleted later. Next, we show that the correct reaching definitions are propagated and inserted as arguments to ϕ- and ψ-functions. Finally, we prove that the *traversePG*() routine visits nodes within the PG graph in the correct order, and we also provide complexity analysis for the algorithms.

Within an EFG, ϕ-functions for variable v need to be placed at all points in each local CFG that correspond to the $\mathrm{DF}^+(\mathcal{S})$, where $\mathcal{S}$ is the set of nodes in a local CFG that define v. Furthermore, if the local *Exit* node is in $\mathrm{DF}^+(\mathcal{S})$, ϕ-functions need to be placed in $\mathrm{DF}^+(P_M)$, for M $\in \mathcal{S}$. This concept was formalized earlier [SHW93], and called the *Parallel Precedence Frontier* (PPF) of a node. It was also proved in the work by Srinivasan *et al.* that ϕ-functions are needed at $\mathrm{PPF}^+(\mathcal{S})$, the limit of increasing sequences of PPF sets. Here, we provide mechanisms to implement these concepts efficiently, and prove their safety and correctness. Formally, the PPF of a node is recursively defined as follows:

Definition 8.5 *Given the following descriptions:*

- $\mathrm{PPF}_{local}(X)$ *is the sequential dominance frontier of X, defined within* G_X*;* $\mathrm{PPF}_{local}(X)$ *is defined between nodes and supernodes in* G_X *and does not consider nodes within supernodes, and*
- P_X *is the supernode containing X, as defined earlier. (If* $X \in G_{main}$ *then* $P_X = \emptyset$*.)*

We define the parallel precedence frontier *of a basic block node or supernode X, denoted* $\mathrm{PPF}(X)$*, as follows:*

If $Exit_{G_X} \notin \mathrm{PPF}_{local}(X)$ *then* $\mathrm{PPF}(X) = \mathrm{PPF}_{local}(X)$.

If $Exit_{G_X} \in \mathrm{PPF}_{local}(X)$ *then* $\mathrm{PPF}(X) = \mathrm{PPF}_{local}(X) \cup \mathrm{PPF}(P_X)$.

Theorem 8.9 *The placement algorithm inserts a ϕ-function at all points in* $\mathrm{PPF}^+(\mathcal{S})$ *for any variable, and a ψ-function at all points identified by Definition 8.2.*

Proof:
We first consider the proper placement of ϕ-functions. As described above, for variable v, ϕ-functions belong at $\mathrm{PPF}^+(A(v))$. Thus, we need to show that Algorithm 8.1 places ϕ-functions at precisely those points. For each element in $A(v)$, lines **16** and **24** operate

the same as the original sequential algorithm. This procedure satisfies the first half of Definition 8.5, while lines 25 - 27 satisfies the second half of the definition (adding S_Q on this line to the worklist generates a ψ-function, as seen by lines 18 - 19). Finally, lines 22 - 24 insure that the PPF is iterated.

Next, we show that the points identified by Definition 8.2 are a subset of those identified in Algorithm 8.1. Due to the slice edge in each section local CFG, every local *Exit* node is always in the iterated dominance frontier of all nodes (except local *Entry* and *Exit*) within the section (Theorem 3.1). Thus, if variable v is defined within a section, the local *Exit* node will always have a ϕ-function created for v. Lines 25 - 27 from Algorithm 8.1 guarantees that the section in the PG graph containing the variable definition is added to the worklist. Similarly, the slice edge in the PG graph insures that the coend node is added to the worklist via the DF^+. Any section node from Definition 8.2 is contained in the DF^+ of a variable definition by Theorems 8.6 and 8.7, and we have shown that our algorithm identifies *all* nodes in the DF^+ for ψ-function placement. ∎

Theorem 8.10 *The correct reaching definitions for ψ-functions are propagated by Algorithms 8.2 and 8.3.*

Proof: By exhaustive cases. Let g be any ψ-function for variable v. From Algorithm 8.1 we know g is either (i) at a Section *Entry* node or (ii) at the tail node of a supernode.

case (i). Let $g \in$ Section B for arbitrary Sections A and B, such that B waits upon A. We must show that all reaching definitions of v from Section A are correctly propagated to g. By Theorem 3.1, any downward-exposed definition of v in A results in a ϕ-function, f, being created for v in $Exit_A$. Lines 12-16 of Algorithm 8.3 logically push a pointer to f onto a definition stack, which at this point is $v.CurrentDef$. We have two subcases. In subcase $(i.a)$, B waits directly upon A. Lines 17 - 18 of Algorithm 8.3 will examine B (which we know contains $v = \psi$ by Theorem 8.9), and create a ψ-argument in line 23 with $link(\ \psi\text{-}arg\ ) = f$. In subcase $(i.b)$, B waits transitively upon A with no intervening definition of v. Here, since there are no intervening definitions of v, $v.CurrentDef$ remains unchanged until reaching B as long as it is not logically popped off the definition stack, which could only occur if $v.CurrentDef$ is reset. The only issue concerns whether the section nodes are visited in the correct order. This issue is dealt with in Theorems 8.12 – 8.14 later in this section.

case (ii). Let $g \in tail_P$. This case is actually a special case of (i), where coend $\in$

DF^+. Line 19 of Algorithm 8.3 insures that the reaching definition is propagated to g in this case. ▮

Theorem 8.11 *The correct reaching definitions for ϕ-functions are propagated by Algorithms 8.2 and 8.3.*

Proof:
Consider any ϕ-function f for variable v. If f is within a local CFG, then Algorithm 8.2 works the same as Algorithm 3.2. We need only consider the case where the local CFG contains a supernode, P, and (*i*) f is within P, or (*ii*) f is at a point reached by P.

case (*i*). Either $f \in DF^+$ of $A(v)$, or not. If not, then $v.CurrentDef$ is a pointer to the current reaching definition of v, propagated from its local *Exit* node ϕ-function, and logically pushed onto a stack of definitions by lines 12 - 16 of Algorithm 8.3 . If so, then Algorithm 8.1 guarantees that a ψ-function was created at $Entry_{G_X}$, and Theorem 8.10 assures us that it possesses the proper reaching definition.

case (*ii*). The last definition of v from one branch of a local CFG reaching f comes from inside P. But here $v.CurrentDef$ will be the ψ-function at $tail_P$ when filling in the correct ϕ-argument in lines 28 - 31 of Algorithm 8.2. ▮

Algorithm 8.3 provides the details for traversing the PG section nodes in the right order: they must be visited in topological order, but must also be visited in a depth-first fashion of *some* spanning tree of the PG graph. Algorithm 3.2 visits nodes for chaining in a depth-first order of its dominator tree. We visit the section nodes of a supernode in topological order. Note that a depth-first order of a graph will not, in general, visit the nodes in topological order, and all topological orders do not visit a directed graph in a depth-first manner of some spanning tree of that graph. The key idea of Algorithm 3.2 is that when a depth-first search of the dominator tree visits a node all reaching definitions of previous nodes are logically on a stack. This task is accomplished by the depth-first search, as it will visit all a node's dominator tree children before completing its call, and only then pop off definitions within the node. For sequential code, visiting nodes in a depth-first order of the dominator tree effectively produces a 'must-precedes' ordering; for a supernode, we visit section nodes in the 'must-precede' order by examining them topologically, while we insure that the correct reaching definitions between section nodes exists by visiting these nodes in a depth-first order of *some* spanning tree of the PG.

Thus, we would like to find a spanning tree of the PG such that there exists a depth-first search of that tree that maintains topological order. We prove that Algorithm 8.4

Given: Precedence Graph with root R
Do: Initialize with lines 1 - 2
call *dfst*(R)
Result: Topological order of some spanning tree of PG

```
1:      set of edges E ← ∅
2:      stack of nodes T ← ∅

3:      dfst(V)
4:          mark visited(V)
5:          for each child ( successor in a PG ) of V do
6:              if unvisited(child) then
7:                  add edge V → child to E
8:                  parent(child) ← V
9:                  dfst(child)
10:             endif
11:         enddo
12:         push V onto T
13:     end dfst
```

Algorithm 8.4 Ordering PG nodes for processing

accomplishes the desired task, which is called by the routine *traversePG*().

Theorem 8.12 *Popping T will visit the nodes of G in topological order.*

This result is well-known [Sed88]. ▮

Theorem 8.13 *E is a spanning tree of G.*

Proof:
Choose any node N of G. We know N is visited (Theorem 8.12), and visited only once, since it is marked when visited the first time, and will not be revisited once so marked. Since each node N has at most one edge in E with head N, we need only show that N can be reached from the root, R. Simply follow the parent links repeatedly from N. The unique parent P of N corresponds to an edge $P \rightarrow N$ in E. Since G is finite (a necessary assumption), this chain will terminate at the only node without a parent, R. ▮

Theorem 8.14 *Popping T will visit the nodes of E in a depth-first order.*

Proof:
In the context of visiting tree E, a depth-first order of E means that we want to visit all descendants of node N before any unvisited siblings of N. Let N and M be siblings in E, with D a descendant of N. We must show that, given N, M, and D unvisited, if N is visited first, D will be visited before M (by Theorem 8.12 we know N will be visited before D). Assume, to the contrary, M is visited before D. This assumption implies that M is between N and D in stack T. Since D is reachable from N, and *dfst(N)* reached M before completing, D must be a descendant of M in E. But this fact implies two paths from R to D in E (one through N and one through M), since M and N are siblings. However, this conclusion contradicts the fact that E is a tree. Thus, D will be visited before M. ▌

In order to assess the asymptotic complexity of the algorithms given in this chapter, let $\hat{P}$ be the number of sequential sections of code in the parallel program, and $\hat{N}$ and $\hat{E}$ be the maximum of the total number of nodes and edges respectively in the local CFGs corresponding to each of these sections. If $\hat{V}$ is the number of variables in the program, we calculate the running time of our algorithms as follows: the first phase, ϕ- and ψ-function placement, takes *worst case* $O(\hat{N}^2 + \hat{E})$ per section [CFR+91], thus over all sequential sections it will take $O(\ \hat{P} \times (\hat{N}^2 + \hat{E})\ )$ time. Then, for the second phase, the *ChainEFG*() routine take maximum time of $O(\hat{N} \times \hat{V})$ (per sequential section), while *traversePG*() will traverse all sections ($O(\hat{P})$), calling *dfst*() (constant time with respect to the *traversePG*() call), *ChainEFG*(), and processing ϕ- and ψ-functions on each section ($O(\hat{V})$). Thus, the running time of the second phase, over all variables, is $O(\hat{P} \times (\ \hat{N} \times \hat{V} + \hat{V}\ ))$.

8.4 Notes on Implementation

Here, we will examine some of the salient features observed while implementing the algorithms presented in this chapter:

- The slice edges proved to be an invaluable tool for propagating reaching definitions. All variables defined within a section have a ϕ-function at the local *Exit* node, but ψ-functions (inserted at the section *Entry* nodes) are never propagated within that section, since *Entry* nodes have an empty local dominance frontier. Thus, by looking first for ψ-functions at *Entry*, followed by ϕ-functions at *Exit*, the proper reaching definition will always be on the top of the stack when proceeding to a new

section.

- Removing duplicate ψ-arguments. We have seen how duplicate arguments can occur. At first glance, it may appear that in order to remove duplicates, the arguments would need to be sorted, taking $N \log N$ time. Although we expect N to usually be fairly small, we can, in fact, perform the duplicate elimination in linear time, by using a variant of a bucket sort. For each ψ-function s, examine its arguments, marking a reference flag (pointer to a symbol) at the end of each argument's *link* with s. Since each ψ-function is unique, we can immediately identify duplicate entries and remove them. Note that this technique is possible, since we can follow the *use* of a variable (from the ψ-argument in this case) to its *definition* via our reference chain implementation.

Chapter 9

Conclusions

9.1 Future Applications of Reference Chaining

Reference chaining can be applied in many instances. Once a problem is identified and cast into a framework where reaching and reachable references are needed, the GRC algorithm can be invoked to insert merge operators and capture the desired information. We saw just such an instance with load-range analysis in §5.2.

We mentioned early in this thesis that FUD chains to date have been the most useful application of reference chaining. That is because they extend SSA form (which already has many applications developed around its structure) to include def-def links, which have allowed more thorough analysis of many problems. We anticipate using FUD chains to analyze another important compiler problem: alias sets. In general, this problem is undecidable, and some languages, notably C, allow arbitrary pointer declaration and pointer arithmetic, making the problem all but impossible to solve. Using a restricted set of pointers, such as provided in Fortran 90, will enable more precision of pointer analysis. The Fortran 90 standard [ABM+92] permits pointers in a much more restricted sense than C: no pointer arithmetic is allowed and objects need to be identified as being the potential target of a pointer.

We briefly examine here two techniques for following FUD chains to analyze alias sets. The problem is somewhat similar to analysis of arrays (in which each element could be considered an alias of all other elements), where each definition of an array is treated as a nonkilling definition. Thus, one may follow chains until a particular array element is reached, or until the element in question cannot be determined. We used this method to perform array constant propagation in §4.6.

Aliases can occur in several ways. Formal arguments can become aliases to each other, as we see with this example:

```
S1:     call sub( X,Y,Y )
S2:     call sub( X,X,Y )

S3:        ...
S4:     sub( P,Q,R )
S5:        ...
```

When sub is called at S_1, Q and R become aliases of each other, and likewise for P and Q at S_2. These are *must aliases*, since the compiler can statically establish their relationship. With this C code fragment

```
S1:     int C
S2:     int *A, *B
S3:     A = &C
```

*A and C are must aliases, but if the compiler cannot determine any information about B, it may also point to the address of C. Thus, B and A, and *B and C are *may aliases*, a less precise relationship than must alias, but a necessary conservative approximation.

We currently envision two methods for performing alias analysis with FUD chains:

1. Create alias *equivalence classes*. When analyzing a variable, the compiler must follow the chains of all elements in the equivalence class. There are several questions to answer:

   - Are the data structures kept small, as we expect?
   - Does this construction lead to an explosion in the number of links to traverse?
   - How can we take advantage of incremental information?
   - How much of a problem is *false* aliases? For the first example above, P and R will be put into the same equivalence class (since both at some point are classified as aliases of Q), but in fact are never aliases of each other.

2. Treat definitions as killing definitions of themselves, but nonkilling definitions of the alias set. When analyzing a variable, the compiler will traverse a chain that weaves through all references of variables in an alias set. This technique is similar to work previously done [WZ91, CG93]. In this case we may have larger data structures, but taking advantage of incremental information may be much easier.

9.2 Assessment and Conclusion

We have described a general method of chaining arbitrary references in both forward and backward data-flow problems. At this point, how do we evaluate the strengths and weaknesses of this approach? First, we assess how GRC stands in comparison to our thesis as described in §1.2:

1. **Efficient implementation.** We have shown empirically that building reference chain graphs is linear in the number of variables and program statements. Compared to the actual analysis (performing constant propagation, building live variable sets, etc.) the construction cost is fairly minimal.

2. **Alternative solutions.** We have shown how to use GRCs as an alternative method for performing constant propagation and solving live variables and anticipatable expressions.

3. **Problems previously neglected.** GRC has been shown to be applicable to the problem of scalar data dependence. We present the first sparse solution to the scalar dependence problem, important since it avoids the additional overhead associated with general dependence analysis.

4. **Extension into parallel languages.** We have provided a detailed extension for reaching definitions into explicit parallel constructs. We envision extensions for other constructs, such as the HPF `FORALL` statement.

We now offer the following observations, based upon using GRC on a large assortment of problems and scenarios:

- GRC has great appeal due to its use of standard basic block and CFG representations. By augmenting the CFG, we obviate the need for separate structures (such as that used by sparse evaluation graphs [CCF91]), yet provide the maximum degree of sparsity allowed by each problem.

- Merge operators provided by GRC capture information that is stored until needed and which can be extracted on demand; an example is the λ-function when used for computing live variable information.

- By developing the general reference chaining algorithm (Algorithms 5.1 and 5.2) we have provided a mechanism for collecting information on many problems, not

just those applications we have focused on in this thesis. The GRC framework can result in efficient and fast solutions to problems that before were limited by the cost of expensive operations in the traditional iterative style. Solving the reaching uses problem with FRDU chains, which allowed an efficient method of detecting anti- and input dependence, is a prime example of the usefulness of the GRC technique.

- Empirical testing of solutions to problems using a demand-driven method has shown that, in the absence of cycles, sparse *links* combined with the demand for classification is quite an efficient technique. Separate solvers for cycles have also been shown to be very effective.

- The cost of constructing reference chains (all structures involve some overhead, albeit some more than others) can often be amortized when multiple applications use the same chains for analysis.

- Array analysis remains as critical as ever. FUD chains have improved the ability to process data structures such as arrays and records, but further research is necessary to continue the development of techniques to analyze data structures with multiple components.

Instead of providing *links* between references to achieve sparse data-flow representations, another approach is to use standard methods, as described in §2.3.1, and identify those equations in the system that carry redundant information [DGS94]. By eliminating redundancies, fixed point computation may be faster. This approach maps data-flow equations into congruences (those equations with identical maximum fixed point solutions), which is an equivalence relation that partitions all the equations into congruence classes. Then, a reduced set of equations can be solved, with one equation from each class. Although the authors note that SSA form is an alternate technique, Duesterwald *et al.* point out that "the benefits of using SSA for data flow analysis are limited to problems that are based on definition-use connections, such as constant propagation. A problem like available expressions does not benefit from SSA." In this dissertation we have seen that extending the concept of SSA to GRC allows solutions to many data-flow problems (including available expressions) that require connecting arbitrary reference pairs, not just definitions and uses.

In conclusion, we have presented a framework that aids in the overall analysis of the intermediate compiler form, and have demonstrated that reference chaining is an option for many problems where sparse representation of data-flow information is profitable.

By implementing the techniques described, we have shown that GRC is a viable and efficient approach for analyzing real programs.

Bibliography

[ABC+87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. In Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos, editors, *Supercomputing: 1st International Conference*, number 297 in Lecture Notes in Computer Science, pages 194–211. Springer-Verlag, Berlin, 1987.

[ABC+88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988. (update of [ABC+87])

[ABM+92] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook.* McGraw-Hill Book Company, New York, NY, 1992.

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 1(3):137–147, 1976.

[ADNP88] Arvind, M. L. Dertouzos, R.S. Nikhil, and G.M. Papadopoulos. Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Massachusetts Institute of Technology Laboratory for Computer Science, March 1988.

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

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Department of Mathematical Sciences, Rice University, 1983. (available from University Microfilms Inc., document 83-14916)

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

[Aut94] Tito Autrey. Interprocedural constant propagation: Implementation and evaluation. *Student research proficiency paper*, Department of Computer

Science and Engineering, Oregon Graduate Institute of Science & Technology, May 1994.

[BB89] William Baxter and Henry R. Bauer III. The Program Dependence Graph and vectorization. In *Conference Record Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, Austin, TX, January 1989.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, Orlando, FL, June 1994.

[BCD+92] David S. Blickman, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Jr., Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, and William B. Noyce. The GEM optimizing compiler system. *Digital Technical Journal*, 4(4):121–135, 1992.

[BCFH89] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. Automatic generation of nested, fork–join parallelism. *The Journal of Supercomputing*, 3(2):71–88, July 1989.

[BH73] Per Brinch Hansen. *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, ON, June 1991.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, FL, January 1991.

[CCF94] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

[CCKT86] D. Callahan, K.D. Cooper, K. Kennedy, and L. M. Torczon. Interprocedural constant propagation. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, pages 152–161, Palo Alto, CA, June 1986.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, Portland, OR, June 1989.

[CF93] Ron K. Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 461–476. Springer-Verlag, 1993.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing Static Single Assignment form. In *Conference Record Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing Static Single Assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In *ACM SIGPLAN '90 Conference on Progamming Language Design and Implementation*, pages 337–351, White Plains, NY, June 1990.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, June 1993.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *Proceedings ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 54–68, Portland, OR, June 1989.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. In *4th ACM International Conference on Supercomputing*, pages 254–266, Amsterdam, The Netherlands, June 1990.

[DGS94] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *5th International Conference, Compiler Construction*, number 786 in Lecture Notes in Computer Science, pages 357–373. Springer-Verlag, 1994.

[EHLP92] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua,

editors, *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 65–83. Springer-Verlag, 1992.

[F+93] Geoffrey C. Fox *et al.* Common runtime support for high-performance parallel languages. In *Proceedings of Supercomputing 93*, pages 752–757, Portland, OR, November 1993.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–54, 1991.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, Wokingham, England, 1988.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler.* Benjamin-Cummings, Menlo Park, CA, 1988.

[FO83] J. Ferrante and K.J. Ottenstein. A program form based on data dependency in predicate regions. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, pages 217–231, Austin,TX, January 1983.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GS93] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, San Diego, CA, May 1993.

[GSW] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. To appear in *ACM Transactions on Programming Languages and Systems*, 1995.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, Albuquerque, NM, June 1993.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM*, 23:172–202, 1976.

[GWS94] Michael P. Gerlek, Michael Wolfe, and Eric Stoltz. A reference chain approach for live variables. Technical Report 94-029, Oregon Graduate Institute of Science & Technology, 1994.

[Hal91] Mary Hall. *Managing Interprocedural Optimization.* PhD thesis, Department of Computer Science, Rice University, 1991.

[Har85] Don Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[Hav93] Paul Havlak. Construction of Thinned Gated Single-Assignment form. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 477–499. Springer-Verlag, 1993.

[Hav94] Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Department of Computer Science, Rice University, 1994.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.

[HP92] Mohammed R. Haghighat and Constantine D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science, pages 65–83. Springer-Verlag, 1992.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of Program Dependence Graphs for representing programs. In *Conference Record Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.

[HS78] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, Potomac, MD, 1978.

[HU72] M.S. Hecht and J.D. Ullman. Flow graph reducibility. *SIAM Journal of Computing.*, 1(2):188–202, June 1972.

[HWW93] William C. Hsieh, Paul Wang, and William E. Weihl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 239–248, San Diego, CA, May 1993.

[Joh94] Richard Craig Johnson. *Efficient Program Analysis using Dependence Flow Graphs.* PhD thesis, Department of Computer Science, Cornell University, 1994.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, NM, June 1993.

[JPP93] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report 93-1365, Department of Computer Science, Cornell University, July 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, Orlando, FL, June 1994.

[Kas75] V.N. Kas'janov. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady*, 16(5):448–450, 1975.

[Ken81] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conference Record 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, TX, January 1989.

[KH93] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *Proceedings ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 268–277, Albuquerque, NM, June 1993.

[Kil73] Gary A. Kildall. A unified approach to program optimization. In *Conference Record First ACM Symposium on the Principles of Programming Languages*, pages 194–206, October 1973.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffan. Partial dead code elimination. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158, Orlando, FL, June 1994.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LFK+93] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1/2):51–142, 1993.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[Mar89] Thomas J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Department of Computer Science, Rutgers University, 1989.

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In *Conference Record 21st Annual ACM Symposium on Principles of Programming Languages*, pages 311–325, Portland, OR, January 1994.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[MJ92] Carl McConnell and Ralph E. Johnson. Using Static Single Assignment form in a code optimizer. *ACM Letters on Programming Languages and Systems*, 1(2):152–160, June 1992.

[MS93] R. Metzger and S. Stroud. Interprocedural constant propagation: An empirical study. *ACM Letters on Programming Languages and Systems*, March–December 1993.

[Muc88] Steve S. Muchnick. Optimizing compilers for SPARC. *Sun Technology*, pages 161–173, Summer 1988.

[NP94] Cindy Norris and Lori L. Pollock. Register allocation over the Program Dependence Graph. In *Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 266–277, Orlando, FL, June 1994.

[OBM89] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. Gated Single-Assignment form: Dataflow interpretation for imperative languages. Technical Report LA-UR-89-3654, Los Alamos National Laboratory, 1989.

[Par91] Parallel Computing Forum. PCF Parallel Fortran extensions. *Fortran Forum*, 10(3), September 1991.

[RL77] J.H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record Fourth ACM Symposium on the Principles of Programming Languages*, Los Angeles, CA, January 1977.

[RWZ88] B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *Conference Record Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, CA, January 1988.

[Sed88] Robert Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1988.

[Sel89] Rebecca Parsons Selke. A rewriting semantics for Program Dependence Graphs. In *Conference Record Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–24, Austin, TX, January 1989.

[SG93] Vugranam C. Sreedhar and Guang R. Gao. Computing ϕ-nodes in linear time using DJ-graphs. Technical Report 757, McGill University School of Computer Science, ACAPS Laboratory, Montreal, PQ, January 1993.

[SGW94] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of 27th Annual Hawaii International Conference on System Sciences*, pages 43–52, January 1994.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. Static Single Assignment for explicitly parallel programs. In *Conference Record Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 16–28, Charleston, SC, January 1993.

[SS93] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 633–655. Springer-Verlag, 1993.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[TS85] Jean-Paul Tremblay and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, New York, NY, 1985.

[Tse93] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Deptartment of Computer Science, Rice University, January 1993. (available as Technical Report TR93-199, Rice University)

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: representation without taxation. In *Conference Record 21st Annual ACM Symposium on Principles of Programming Languages*, pages 297–310, Portland, OR, January 1994.

[WGS93] Michael Wolfe, Michael P. Gerlek, and Eric Stoltz. Nascent: A next-generation, high performance compiler. Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology. *unpublished manuscript*, 1993.

[WGS94] Michael Wolfe, Michael P. Gerlek, and Eric Stoltz. Demand-driven data flow analysis. Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology. *unpublished manuscript*, 1994.

[Wol78] Michael Wolfe. Techniques for improving the inherent parallelism in programs. M.S. thesis UIUCDCS-R-78-929, University of Illinois, Dept. Computer Science, July 1978.

[Wol82] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Department of Computer Science, University of Illinois, October 1982. (available from University Microfilms Inc., document 83-03027)

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing, London, 1989. (also available from MIT Press)

[Wol92a] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Department of Computer Science, Stanford University, August 1992. (available as Technical Report CSL-TR-92-538, Stanford University)

[Wol92b] Michael Wolfe. Beyond induction variables. In *Proceedings ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, CA, June 1992.

[Wol94] Michael Wolfe. $J^+ = J$. *ACM Sigplan Notices*, 29(7):51–53, July 1994.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

Biographical Note

Eric James Stoltz was born on September 19, 1954, in Modesto, California, the son of Jack and Cathy Stoltz. After living in California and Washington, he attended Willamette University in Salem, Oregon, earning a Bachelor of Science degree in Mathematics in 1976. There followed 14 years of teaching math in public school, where each day entertained the possibility of joy and tears. In 1982 he received a Masters of Science and Teaching degree in Mathematics from Portland State University.

Eric decided to pursue a second career in 1990. Thinking a further advanced degree in mathematics would not offer a wide range of career opportunities, he entered the computer science Ph.D. program at Oregon Graduate Institute of Science & Technology with no formal computer science background of any sort. This decision led to an interesting first year, but involvement in a compiler analysis research group led by Professor Michael Wolfe offered work that seemed suited to his nature. Eric received his Ph.D. in Computer Science in January, 1995.

During his tenure at OGI, Eric had considerable opportunity to improve his writing skills, via technical reports, conference papers, journal articles, and of course this dissertation.

Living on the West Coast his entire life, Eric continued his eventful life in February, 1995, by accepting a job in Texas, a state that he visited for the first time in December, 1994.