2012

Future value based single assignment program representations and optimizations

Shuhan Ding
Michigan Technological University

FUTURE VALUE BASED SINGLE ASSIGNMENT PROGRAM REPRESENTATIONS AND OPTIMIZATIONS

By

Shuhan Ding

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2012

This dissertation, "Future Value Based Single Assignment Program Representations and Optimizations," is hereby approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE.

Department of Computer Science

Signatures:

Dissertation Advisor _____

Dr.Soner Önder

Department Chair _____

Dr. Steven M Carr

Date _____

To my parents and my husband

Contents

List of Figures **11**

List of Tables **15**

Acknowledgments **19**

Abstract **21**

1 Introduction **23**

1.1 Motivation 23

1.2 Research Goals 26

1.3 Contribution 27

1.4 Organization of the Dissertation 28

2 Background **29**

2.1 Program Representations 29

2.1.1 Single Assignment Semantics 30

2.1.2 Static Single Assignment Form (SSA) 30

2.1.3 Gated Single Assignment Form (GSA) 32

2.1.4 SSA and GSA Comparison 33

2.2 Computing Single Assignment Forms 33

2.2.1 Construction algorithms and Problems 33

2.2.2 Inverse Transformation Algorithms and Problems 36

2.3 Other Important Representations 39

2.4 Code Motion and Compiler Optimizations . . . 40
2.4.1 PRE and strength reduction . . . 42
2.4.2 Scheduling and register allocation . . . 44
2.5 Future Values . . . 46

3 Future Gated Single Assignment (FGSA) . . . 48
3.1 Motivation and Comparison . . . 49
3.2 Congruence Classes and Path Separability . . . 51
3.3 Efficiently Computing FGSA . . . 55
3.3.1 Identification of Congruence Classes . . . 55
3.3.2 Gating Function Construction and Insertion . . . 57
3.4 Interval Analysis and T1/T2 Transformations . . . 59
3.4.1 Acyclic Regions: T2 Transformation . . . 59
3.4.1.1 Computing Path Predicate Expressions . . . 61
3.4.1.2 Merging Edges . . . 62
3.4.2 Cyclic Regions: T1 Transformation, Exit Function . . . 62
3.5 Irreducible Graphs and T_R Transformation . . . 65
3.6 Experimental Analysis . . . 69
3.7 Complexity of FGSA Construction . . . 72
3.8 Executable FGSA . . . 72
3.9 Conclusion . . . 74

4 Live Variable Analysis on FGSA . . . 75
4.1 Extended Liveness . . . 76
4.2 Associating Liveness with Congruence Classes . . . 80
4.2.1 Computing The Anticipated Window and The Gating Region . . . 81
4.2.2 Interference Under Extended Liveness . . . 83
4.3 Computing and Associating Liveness with Congruence Classes . . . 85
4.4 Conclusion . . . 88

5 Inverse Transformation From FGSA 90
5.1 Simple Inverse Translation from FGSA 91
5.2 Path Separability, C-FGSA, T-FGSA and Isolation 93
5.2.1 Checking for Path Separability 96
5.3 Minimizing Copies 98
5.3.1 Handling Interferences for an Isolated CC 100
5.3.2 Handling Non-Gated to Gated CC Interferences 102
5.3.3 Handling Gated to Gated CC Interferences 103
5.3.4 Representation of the Problem 103
5.4 Common Use Form and Global Optimal Solution 108
5.4.1 Global Solution Through Complete Isolation 110
5.4.2 Approximation of Global Optimal Solution 111
5.4.3 Validity of Proposed Approach 112
5.5 Conclusion 113

6 Optimizations on FGSA 114
6.1 Constant Propagation on FGSA 114
6.2 Global Value Numbering (GVN) on FGSA 116

7 Recursive Future Predicated Form 119
7.1 Code Motion in Acyclic Code 120
7.1.1 Future Predicated Form 122
7.1.2 Elimination of ϕ-nodes 122
7.1.3 Merging of Instructions 124
7.2 Instruction-Level Recursion 128
7.3 Code Motion in Cyclic Code and Recursive Future Predicated Form 129
7.3.1 ϕ-nodes in Loop Header 130
7.3.2 Conversion of Loops into Instruction-Level Recursion 130
7.4 Code Motion Involving Memory Dependencies and Function Calls 134

7.5 Directly Computing RFPF 138
7.6 Conclusion 140

8 Optimizations on RFPF 141

9 Conclusion 145
9.1 Summary of Work 145
9.2 Future Research 146

References 147

List of Figures

2.1 SSA form 31
2.2 Single assignment semantics and special functions 31
2.3 Insert copy assignments to eliminate a ϕ-function 37
2.4 Split critical edges 37
2.5 The semantics of simultaneous evaluation of ϕ-functions and break circular definition 38
2.6 The concept of future data and control dependencies 47

3.1 An FGSA Example 49
3.2 A non-path-separable CC: use belongs to two CCs 53
3.3 Local CCs computation 55
3.4 T2 example 57
3.5 CC cases 59
3.6 Algorithm 1: T2-CC incorprating 61
3.7 Algorithm 2: T2-CC merging 63
3.8 A self-referencing gating function 63
3.9 T_R transformation on an irreducible graph 66
3.10 Predicated instructions 73

4.1 Traditional liveness 76
4.2 Partial liveness 77
4.3 Two liveness approaches in general single assignment form 78
4.4 Running FGSA Example 80

4.5 Extended liveness on FGSA 82
4.6 Gating region 84
4.7 Algorithm 1 with live analysis 87
4.8 Algorithm 2 with live analysis 89

5.1 Translation from FGSA 92
5.2 Path separability example 93
5.3 Isolation and path-separability 95
5.4 Basic algorithm for checking path separability 97
5.5 A non-path-separable CC and its NPSG 101
5.6 Interferences of uses in different regions 103
5.7 Weighted Interference Graph 105
5.8 Dynamic programming and search tree 106
5.9 Common use form 109

6.1 Evaluation rules for (a) ψ_P and (b) ψ_R functions 115
6.2 Constant propagation 116
6.3 A modified example from Gargi's work. Predicates are contained in <>.
Line f1, f2, and f3 contain FGSA gating functions for corresponding ϕs. . . 117

7.1 Splitting code motion 121
7.2 Merging code motion 121
7.3 Code motion across control dependent regions 123
7.4 ϕ-node elimination 124
7.5 Instruction propagation 126
7.6 Instruction merging 127
7.7 Algorithm 3: Compute RecursivePredicate and ExitPredicate 131
7.8 Theorem 7.3.1 132
7.9 Program 1:Conversion of a cyclic program into RFPF 133
7.10 Predicated memory and reordered memory 135

7.11 ϕ-node of predicates before a store 135
7.12 ϕ-node of predicates before a load 136
7.13 Rewriting memory operations: placement of ϕ-functions 136
7.14 Rewriting memory operations: rewriting memory instructions 137
7.15 Algorithm 4: Directly computing RFPF 139

8.1 Partial redundancy elimination during the code motion 142
8.2 Merging and converting back to CFG 143
8.3 Constant propagation on RFPF 144

List of Tables

3.1 CCs vs pruned ϕ-functions over REAL 70
3.2 Number of definitions in CCs 71
3.3 Length of CC predicate expressions 71

Preface

This dissertation contains the material (Chapter 7 and Chapter 8) that has been published in CC'10/ETAPS'10 Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction Springer-Verlag Berlin, Heidelberg 2010, which is co-authored by me and Soner Önder. Dr.Önder proposed the key concepts of *future values* and *instruction level recursion*. I designed the algorithms under the supervision of Dr. Önder.

Acknowledgments

Though only my name appears on the cover of this dissertation, a great many people have contributed to the success of this work [1], each in their own unique ways.

First, my sincere thanks to my advisor, Dr. Soner Önder. I have been amazingly fortunate to have an advisor whose patience and support helped me overcome many crises to finish this dissertation. Your advice and insight were invaluable to the success of this work. Thank you for your guidance and help in reaching this important milestone.

A heartfelt thanks to Dr. Steven Carr, who introduced me into the wonderful compiler world. Many thanks to Dr. Zhenlin Wang, Dr. Nilufer Önder, Dr. Steven Seidel, Dr. Ching-Kuang Shene and Dr. Jean Mayo. Thank you for teaching me in various computer science courses and helping me to build a solid foundation for this dissertation and for my career.

Many wholehearted friends have helped me to keep my sanity through these years. Sincere thanks to Wei Wang, Yunhua Li, Zheng Zhang, Xinxin Jin. Thank you for accompanying me through laughters, happiness and even hardships. Your support and care helped me overcome setbacks and to stay focused on my graduate study. I greatly value your friendship and I deeply appreciate your belief in me. You are not just friends to me, but you also give me a family feeling in the darkest of times.

Many thanks to my colleagues: Changpeng Fang, Peng Zhou, Lihui Hu, Alicia Thorsen, Roland Scott and Weiming Zhao. Your wise counsel and willingness to share your experience during your PhD study, assisted me greatly to finish my dissertation. Besides, many thanks to my friends around and outside the Michigan Tech: Ming Xie, Zhiyao An, Xiaofei Qu and Fengqiong Huang.

Finally and most importantly, none of this would have been possible without the love and patience of my family. My parents, raising and supporting me, also aided and encouraged me throughout this endeavor. I truly and deeply appreciate their generosity

[1] This work is supported by an NSF CAREER award (CCR-0347592) to Soner Önder.

and understanding. To my soul mate Mingsong Bi, thanks for supporting me constantly all these years.

Abstract

An optimizing compiler internal representation fundamentally affects the clarity, efficiency and feasibility of optimization algorithms employed by the compiler. Static Single Assignment (SSA) as a state-of-the-art program representation has great advantages though still can be improved. This dissertation explores the domain of single assignment beyond SSA, and presents two novel program representations: *Future Gated Single Assignment (FGSA)* and *Recursive Future Predicated Form (RFPF)*. Both FGSA and RFPF embed control flow and data flow information, enabling efficient traversal program information and thus leading to better and simpler optimizations. We introduce *future value* concept, the designing base of both FGSA and RFPF, which permits a consumer instruction to be encountered before the producer of its source operand(s) in a control flow setting. We show that FGSA is efficiently computable by using a series T1/T2/T_R transformation, yielding an expected linear time algorithm for combining together the construction of the pruned single assignment form and live analysis for both reducible and irreducible graphs. As a result, the approach results in an average reduction of 7.7%, with a maximum of 67% in the number of gating functions compared to the pruned SSA form on the SPEC2000 benchmark suite. We present a solid and near optimal framework to perform inverse transformation from single assignment programs. We demonstrate the importance of unrestricted code motion and present RFPF. We develop algorithms which enable instruction movement in acyclic, as well as cyclic regions, and show the ease to perform optimizations such as *Partial Redundancy Elimination* on RFPF.

Chapter 1

Introduction

1.1 Motivation

An optimizing compiler internal representation fundamentally affects the clarity, efficiency and feasibility of optimization algorithms employed by the compiler. A strong program representation is built upon sound principles so that it facilitates correctness. A good representation is never eclectic, yet, it represents information that is needed by a large number of optimizations. Most state-of-the-art optimizing compilers in this regard rely on Static Single Assignment (SSA) form initially developed by Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck in the 1980s. The strength of SSA comes from its ability to represent programs in *s*ingle-assignment form. In *single assignment form*, every assignment is unique and each definition dominates all its uses. These two properties together enable SSA for the development of strong, provably correct sparse optimization algorithms for a variety of problems. SSA achieves single assignment semantics by inserting a gating function, called a ϕ-function at the merge nodes of the control-flow graph which returns one of the definitions reaching the merge points. The code size of SSA is linear to the original program. The state of art SSA transformation and inverse transformation algorithms are very efficient although the inverse transformation is problematic. Overall SSA is a good IR because it brings in significant

improvement on efficiency and simplicity and only small overhead. However, there is still significant room for improvement in the domain of single assignment representations. At a ϕ-function, which definition to return is not explicitly represented in SSA. Therefore programs in SSA form are not directly executable and an inverse transformation is inevitable before machine code can be generated. A fundamentally similar representation Program Dependence Web/Gated Single Assignment (Ottenstein et al. 1990) employs executable gating functions by encoding the control flow information in the form of predicates in the gating functions. Thus the return value of the gating function is decided by the predicates controlling it. However with the initial design for data, control, demand -driven interpretation, PDW suffers complicated transformation algorithms. Both SSA and GSA insert gating functions at control flow convergence points. In some control flow cases, gating functions are inserted in such a way they refer to each other and no new value is indeed generated, which causes unnecessary data flow traversal. And because the gating functions are location specific, optimizations that rely on code movement cannot be easily performed.

In this dissertation, we explore the domain of single assignment beyond SSA and GSA. We present two novel program representations: Future Gated Single Assignment (FGSA) and Recursive Future Predicated Form (RFPF). They both retain the single assignment property, i.e., each use has a single definition. Both representations are significantly different from SSA in the way of handling how to converge multi-definitions and how to deliver a value of a definition to a use. In the case of FGSA, instead of inserting gating functions at the confluence points, we identify a group of uses which receive the same set of definitions, forming a *congruence class*. We also identify the control flow under which each definition flows into the uses and encode this information in the form of a set of path expressions. Assigning a different name to each definition and a common, unique name to the uses, the single-assignment semantics can easily be achieved by using a single gating function per congruence class. This gating function is controlled by the set of path expressions computed from control flow.

Both FGSA and RFPF embed either essential or extensive control flow information. In FGSA, when the control flow involves determining which value should flow to a use, the control information is explicitly represented. In RFPF, control dependencies are completely converted into data dependencies, resulting a linear non-graphic representation. Just as SSA makes data flow traversal easier, our new IRs make both data and control flow traversal more efficient and thus lead to better and simpler optimizations.

In FGSA and RFPF, we introduce *future value* concept, which permits a consumer instruction to be encountered before the producer of its source operand(s) in a control-flow setting. Future values concept allows FGSA to place the gating function above the definitions of its uses, enabling definitions to be encountered down the control flow. It also gives RFPF the capability to move the consumer instruction above its value producers and permits unrestricted code motion.

In summary, these representations extend the state-of-art in program representations on several points:

1. Executable semantics which permits direct execution by a appropriate architecture, yet serving the dual role as a compiler internal representation;

2. A formal framework for inverse transformation;

3. Uniform treatment of control and data dependencies resulting simplification of compiler optimization algorithms such as PRE;

4. In case of RFPF, possession of capabilities for code motion which are only possible with code restructuring in the existing representation without code restructuring during the optimization phase.

It is important to mention that there are other representations which aim at combine data flow and control flow setting as our new representations do, such as Dependence Flow Graphs (Ferrante et al. 1987), ThinnedGSA (TGSA) (Havlak 1993), Static Single Information Form (SSI) (Ananian and Rinard 1999; Singer 2006), and Extended Static

Single Assignment (e-SSA) (Bodík et al. 2000). In terms of program analysis capability, these representations have somewhat equal power but each have significantly different levels of implementation complexity as well as the overhead imposed by the representation. However these representations do not address fundamental problems that FGSA and RFPF address, namely a framework to address inverse transformation, a provision of efficient executable semantics and a framework in which unrestricted code motion is feasible. Finally, FGSA and RFPF are efficiently computable and as such, they are very promising to be effective and practical representation.

1.2 Research Goals

This dissertation aims to design two new IRs. FGSA is more close to SSA and can be used as a replacement of SSA. RFPF further extends FGSA which maintains all the good properties of FGSA and is designed for arbitrary code motion. For details, in this work, our goals are:

1. To design efficient and provable correct transformation algorithms for FGSA, which identifies congruence classes, computes the control predicates and inserts the gating function at the proper position;

2. To examine the overhead of FGSA in terms of transformation complexity, code size, and complexity of predicate expressions;

3. To perform program analysis and adapt optimizations on FGSA;

4. To investigate inverse transformation on FGSA based on congruence classes and control predicates;

5. Based on FGSA, to design RFPF transformation algorithms;

6. To adapt optimization algorithms to RFPF.

1.3 Contribution

This dissertation contributes two single assignment representations, algorithms to compute them as well as algorithms to transform them back to multi-assignment form. This dissertation also introduce the concept of unrestricted code motion to the degree that any instruction can be moved to any program point while maintaining correct program semantics. As well, it illustrates that the concept of traditional liveness can be effectively used in the presented single assignment program representations. These contributions enable a number of additional contributions listed below:

1. We present an algorithm that computes FGSA using a series of T1/T2 transformations. To the best of our knowledge, utilization of T1/T2 transformations for single assignment computation has not been explored before.

2. We develop a novel transformation T_R which permits T1/T2 based algorithm to handle irreducible loops without *node splitting*. As a result, computation of single-assignment form and irreducible loop elimination can be efficiently combined.

3. We demonstrate that live variable analysis can be combined with algorithms for translation into single assignment.

4. We present a near optimal inverse transformation of FGSA in terms of number of copies.

5. We illustrate that FGSA is convenient to use as an IR by presenting two cases studies of optimization algorithms on FGSA.

6. We develop the concepts of future values, future dependencies, future predicates and *instruction level recursion*. These concepts together enable any instruction including loops to be moved beyond data dependencies and control dependencies.

7. We present an algorithm to convert conventional programs into the RFPF. These algorithms operate by propagating instructions and predicates and use only the local information available at the vicinity of moved instructions.

8. We illustrate that unrestricted code motion itself can be used to analyze programs for optimization opportunities.

9. FGSA approach results in an average reduction of 7.7% in the number of gating functions compared to the pruned SSA form.

1.4 Organization of the Dissertation

In the rest of the dissertation, in Chapter 2, various important program representations that this research is built upon and several optimizations that are utilized in the following chapters are reviewed. Next, Chapter 3 through Chapter 5 present the construction, live analysis and inverse transformation of FGSA. In Chapter 6, several optimizations are demonstrated on FGSA and compared to existing techniques. Next Chapter 7 presents the construction and inverse transformation of RFPF and Chapter 8 presents optimizations on FGSA. Finally, a summary and conclusion is given in Chapter 9.

Chapter 2

Background

2.1 Program Representations

An IR is a data structure used in most modern compilers which is transformed from source program and from which the target program is generated. A typical source program can be one of various high level language programs or one type of IRs and the target program can be anther type of IR or machine code. IRs are important tools for representing a program either for direct execution or for better employing compiler optimizations. The IRs presented in this work are based on a number of ideas introduced in existing representations, such as static single assignment form(SSA) and its extension, gated single assignment(GSA). This section reviews some of these representations.

We begin by introducing Control flow graph(CFG) which is a primary means of representing programs in optimizing compilers. A CFG $G =< N, E, s, e >$ is a directed graph, where N is the set of nodes, E is the set of the edges and s and e represent two special nodes *start* and *end*. The nodes in a CFG are *basic blocks*. A basic block is a group of instructions that have one entry point, one exit point and no branch instructions are contained within the group. The edges represent the transfer of control between basic blocks. There's an edge from *start* to any entrance basic block and there's an edge from any exiting basic block to *end*. For an edge X $\rightarrow$ Y in the graph, Y is a *successor* of X and

X is a *predecessor* of Y. A basic block with multiple predecessors is a *join* node. Similarly, a basic block with multiple successors is a *branch* node.

As control dependencies between instructions are represented in a CFG, data dependencies were traditionally represented by use-def chains(UD Chain) and def-use chains(DU Chain) (Aho et al. 1986). A UD chain consists of a use of a variable and all the definitions of the variable that can reach the use without being intervened by other definitions. A DU chain consists of a definition of a variable and all the uses of the variable that can be reached by the definition. Before the SSA form was invented, both UD and DU chains were a prerequisite for many compiler optimizations.

2.1.1 Single Assignment Semantics

UD and DU chains are extra structures which represent the data flow in a program. Single assignment semantics on the other hand explicitly embeds UD and DU information into the representation. Two popular representations that implement single assignment semantics are Static Single Assignment(SSA) form and Gated Single Assignment(GSA) form. In both representations every variable is assigned only once, that is each use is explicitly related to a single definition site. At control confluence points special functions are inserted to merge and select values. SSA employs non-executable ϕ-functions while GSA employs several forms of executable gating functions. Many optimizations algorithms become simpler with single assignment semantics. Detailed properties of these two representations are summarized in the following sub-sections.

2.1.2 Static Single Assignment Form (SSA)

Static single assignment(SSA) was developed by Ron Cytron and coworkers, researchers from IBM in the 1980s (Cytron et al. 1991). In SSA, existing variables in the original representation are split into versions and new variables are indicated by the original name with a subscript such that every definition gets its own version. The usefulness

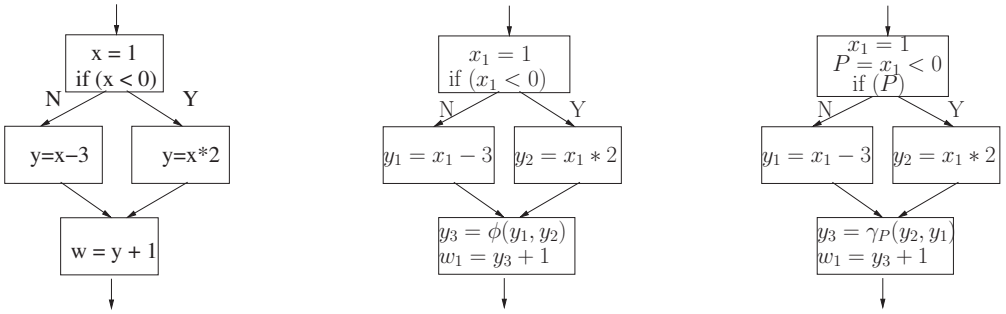
of SSA comes from how it simplifies the properties of variables, i.e., use and definition relationship of each variable is explicit. For example, consider the following piece of code (Figure 2.1(a)):

$y \leftarrow 1$	$y_1 \leftarrow 1$
$y \leftarrow 2$	$y_2 \leftarrow 2$
$x \leftarrow y$	$x_1 \leftarrow y_2$
(a) example code 1	(b) SSA form

Figure 2.1: SSA form

In this very simple example, we need reaching definition analysis to determine that it is the second definition of y that reaches x and hence the first definition is not necessary. But if the program is in SSA form (Figure 2.1(b)), the result is straight-forward.

Unlike the above straight-line code example, most programs have branch nodes and join nodes. At a join node, a use of a variable can be reached by multiple definitions due to control flow. SSA introduces a special function, called ϕ to choose the correct value. Consider Figure 2.2(a). In constructing the SSA, it is clear which definition version reaches each use except the use of y in $w = y + 1$. In Figure 2.2(b), a ϕ-function is inserted at the beginning of the block. A new definition of y, namely y_3 is generated, where the ϕ-function chooses either y_1 or y_2 according to which way control arrived from.



(a) example code 2 (b) SSA form (c) GSA form

Figure 2.2: Single assignment semantics and special functions

The SSA form of a program enables or enhances many analysis and transformations.

Wegman and Zadeck presented two approaches to constant propagation with conditional branches in (Wegman and Zadeck 1991), one is in SSA and the other is not. Clearly the approach which uses SSA has significant advantages. Rosen et al., proposed the approach to redundancy elimination, applying global variable numbers on SSA in (Rosen et al. 1988). They found out that using SSA enables easy removal of trivial assignments and exposes identical expressions which may not be visible in the normal form. Recent applications of SSA to eliminate redundancies can be found in (Kennedy et al. 1999; VanDrunen and Hosking 2004b,a).

2.1.3 Gated Single Assignment Form (GSA)

ϕ-functions in SSA are not directly executable. Ballance et al., proposed a new program representation, called Program Dependence Web(PDW) (Ottenstein et al. 1990) based on SSA. In PDW, ϕ-functions are replaced with a family of *gating functions* which are executable. The three 'gating' functions used in PDW are:

† $\gamma(P, v_1, v_2)$: A γ-function contains a predicate and two values. It returns v_1 if predicate P is true or v_2 if P is false.

† $\mu(P, v_1, v_2)$: A μ-function also contains a predicate and two values and represents a loop entry. Predicate P determines whether control will pass into the loop body. Once P becomes true, the μ-function returns v_1 for the first iteration of the loop and v_2 for the subsequent iterations.

† two η-functions, η^T and η^F: A $\eta^T(P, v)$ returns the value v when the loop predicate P is true. A $\eta^F(P, v)$ behaves similarly when P is false

The resulting graph is called Gated Single Assignment(GSA) form. Figure 2.2(c) shows the GSA form of the code.

Due to several reasons, GSA is not as popular as SSA. However, there are still some applications. For example, Arenza et. al., (Arenaz et al. 2003) proposed a GSA-based

compiler infrastructure to detect parallelism in loops that contain complex computations. These computational kernels form the strongly connected components (SCC) in the graph of use-def chains for the corresponding GSA form. They identified different scenarios of SCC graphs and use the information to guide the generation of parallel code for the loops.

2.1.4 SSA and GSA Comparison

Despite of using different forms of gating functions, both SSA and GSA can represent programs single assignment form. ϕ-functions in SSA at a control convergence point select values according to which edge of the graph was taken by control flow. Gating functions in GSA embed control dependence information in the form of predicates, and they select values based on the value of the predicate. Basically, GSA provides the same information SSA does as well as extra control information that helps to select values at gating functions. However construction of GSA is more complicated, which restricts its usefulness. Next construction and destruction algorithms for both representations are discussed.

2.2 Computing Single Assignment Forms

2.2.1 Construction algorithms and Problems

ϕ-function placement is a central issue in the SSA construction. Two concepts, namely, dominance relation and dominance frontiers are crucial to the understanding where to insert ϕ-functions.

Let X and Y be nodes in the CFG of a program. If X appears on every path from *start* to Y, then X *dominates* Y. If X dominates Y and $X \neq Y$, then X *strictly dominates* Y. The *immediate dominator* of Y (denoted $idom(Y)$) is the closest strict dominator of Y on any path from *start* to Y. The dominance relation can be summarized as a tree structure, called the *dominator tree*, in which any node Y other than *start* has idom(Y) as its parent in the tree. The problem of finding the dominators of a flowgraph has

been studied extensively, Lengauer-Tarjan (Lengauer and Tarjan 1979) being the most well known algorithm. This algorithm relies on the observation that a node's dominator must be above it in the depth-first-spanning tree, which provides an initial guess at the dominator. Then in a second pass, the real dominator is found by correcting the initial guess. Lengauer-Tarjan has a good asymptotic time complexity as O($E \log N$) or even better O(E$\alpha(E,N)$) using a sophisticated implementation. Keith Cooper et al., (D.Cooper et al. 2001) proposed a straight-forward algorithm to find dominators, which formulates the dominance as a global data-flow problem. The approach yields an efficient iterative solver. With a carefully designed data structure, the implementation of the algorithm can run as fast as Lengauer-Tarjan in practice.

The *dominance frontier* of a node X (denoted DF(X)) is the set of nodes Y such that X dominates a *predecessor* of Y but does not strictly dominate Y. Cytron et al. (Cytron et al. 1991) proposed finding the dominance frontier set for each node in a two step manner. This algorithm walks over the dominator tree in a bottom-up traversal order. At each node X, X's successor nodes that are not dominated by X are added into DF(X). Next the dominance frontier sets of X's children in the dominator tree are traversed and any node that is not dominated by X is added into DF(X).

Dominance frontiers are exact places where ϕ-functions may be needed. Consider a definition that resides at node A. For the nodes that are in DF(A), the definition can reach them together with other definitions of the same variables brought in through other control flow. Further, once a ϕ-function of some variable is inserted at node B, B becomes one of the definition sites regarding to the variable, more ϕ-functions may need to be inserted at nodes in DF(B), which demonstrates the idea of iterative dominance frontier.

An efficient approach to inserting a minimum number of ϕ-functions using iterative dominance frontiers is proposed by Cytron et al. (Cytron et al. 1991). The outline of Cytron's general SSA construction algorithms is as below:

1. Compute the dominance frontier for each node in the control flow graph.

2. Using the iterative dominance frontier information, determine the locations of the

ϕ-functions for each variable.

3. Rename each variable by replacing the name of an original variable V by a proper version name V_i.

Cytron's algorithm is minimal in terms of ϕ-functions in the sense that it captures the exact number of joining cites of multiple definitions. However, it can insert the ϕ-functions that are never used after joining. Choi et al. (Choi et al. 1991) propose the *pruned* SSA form, which contains no dead ϕ-functions. Based on live analysis information, the pruned SSA form only inserts ϕ-functions where the variable is live. It contains fewer ϕ-functions and uses fewer SSA names at the cost of live analysis and more condition evaluations during the ϕ-function insertion. These two variations of SSA form favor different applications. Applications like register allocation always benefit from accurate live information provided by pruned SSA while surprisingly global value numbering, an optimization that combines constant propagation and redundancy elimination can benefit from extra information provided by dead ϕs inserted by the minimal SSA algorithm. Briggs et al. (Briggs et al. 1998) propose the third variation, called semi-pruned SSA form which balances the number of ϕ-functions and the cost of construction between the previous two. Bilardi and Pingali (Bilardi and Pingali 2003) investigated the existing ϕ-placement algorithms and put them into a single framework based on a new relation called *merge*. Using the *merge* relation, they describe several new algorithms for ϕ-functions insertion which are optimal for a single variable, as well as an optimal algorithm for multiple variable ϕ-placement in structured programs.

Building the dominator tree and the dominance frontier has non-trivial costs. Targeting this optimization opportunity, several new techniques have been proposed to generate the SSA form without precomputing the dominator and dominance frontier information. Aycock and Horspool (Aycock and Horspool 2000) discover where to insert ϕ-functions by inserting ϕ-functions for every variable at every node in the flow graph and iteratively deleting the extraneous ones. This approach achieves the minimal ϕ-functions in reducible graphs. Brandis and Mössenböck (Brandis and Mössenböck 1994) generate the SSA form

in one pass for structured CFGs. Their algorithm inserts ϕ-functions at the joining nodes by employing different rules for different structures, such as *if, while and repeat*. The algorithm performs variable renaming in the same pass.

The first work in which GSA is proposed (Ottenstein et al. 1990) employs Control Dependence Graph (CDG) (Ferrante et al. 1987; Cytron et al. 1991) to replace ϕ-functions into gating functions. CDG is a graph representation that summarizes the control dependence information of a CFG. It can be computed by computing dominance frontiers on a reverse graph of CFG. During the replacement, for each ϕ-function, two conditions are computed: conditions under which the arguments flow to the ϕ-function and conditions under which the ϕ-function should be executed. For example, given the condition information, gating function $\gamma(P, v_1, \gamma(Q, v_2, \perp)$ is built. Here special symbol $\perp$ signifies that control flow cannot reach the corresponding ϕ under the corresponding predicate value.

Havlak (Havlak 1993) proposed a simplified version of GSA, called Thinned GSA(TGSA) which has a simpler algorithm. Invented for symbolic analysis only, TGSA contains less control information than original GSA. For example, TGSA omits the loop predicate in μ-functions and the predicates in the γ-functions where control cannot flow to the functions under the conditions represented by the predicates. γ-function replacement is the main part of the construction. The approach constructs a DAG which has the same dataflow predecessors and successors as the original ϕ. It also takes the result of a branch as input. By visiting the DAG, each reaching ϕ-argument and the branch conditions under which the argument flows is computed. Tu and Padua (Tu and Padua 1995) improved the computation of gating functions by converting the problem into a path compressing problem. Unlike the above two algorithms, the algorithm does not require the SSA form to generate the GSA form. Gating functions and their positions in the graph are computed at the same time.

2.2.2 Inverse Transformation Algorithms and Problems

The SSA form of a program is an intermediate representation that enables efficient implementations of many compiler optimizations. However, the program needs to be

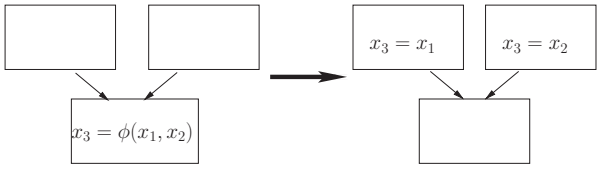


Figure 2.3: Insert copy assignments to eliminate a ϕ-function

transformed back to an executable form since ϕ-functions are not directly executable. SSA form is translated back to the traditional form by replacing each ϕ-function with some ordinary assignment. This algorithm involves insertion of copy operations in the predecessor nodes of a ϕ-function, as it is shown in Figure 2.3. This naive copy insertion approach will break in some subtle cases, namely, the existence of critical edges in the flow graph(the bold edge in Figure 2.4 left) and the simultaneous evaluation of ϕ-functions of the same block. A critical edge is an edge, of which the source node has multiple successors and the target node has multiple predecessors. The critical edges are handled by splitting them (Figure 2.4 right). For the case involving simultaneous evaluation of ϕ-nodes in the same block, consider Figure 2.5 (a). The naive copy insertion results in Figure 2.5(b), which destroys a_1's value before b_1 gets it and thus breaks the original semantics. The solution is to insert a temporary to keep a_1's value as in Figure 2.5 (c).

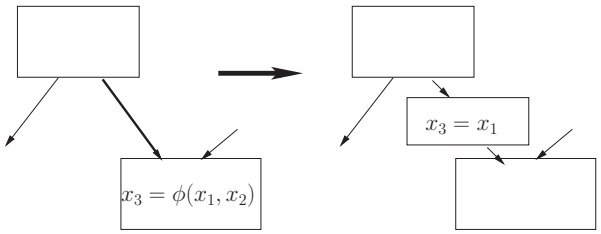


Figure 2.4: Split critical edges

According to Sreedhar et al (Sreedhar et al. 1999), SSA form can be classified into Conventional SSA(C-SSA) and Transformed SSA(T-SSA) based on whether copy insertion is needed during the inverse transformation. C-SSA is the form when the SSA form is just built and arguments in the same ϕ-function do not interfere. To transform a C-SSA program, a representative name is used to replace all the arguments and the destination

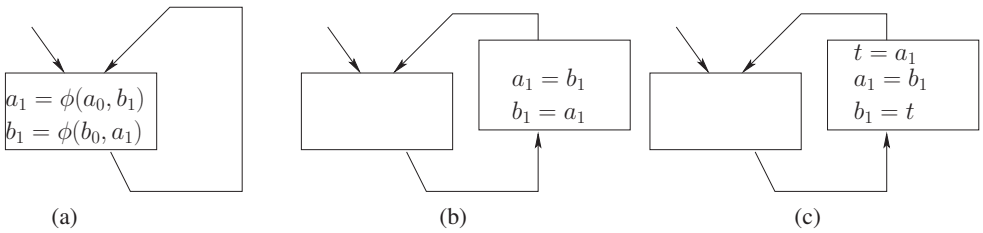


Figure 2.5: The semantics of simultaneous evaluation of ϕ-functions and break circular definition

of the ϕ-function and later the trivial assignment statement is deleted. T-SSA form results from the SSA form undergoes optimization passes during which live range of the ϕ-arguments become overlapped. Due to the overlapping, using a single representative variable to replace the ϕ-arguments does not work since the representative variable cannot represent multiple values in the overlapped regions. Sreedhar et al (Sreedhar et al. 1999) proposed the approach to transform T-SSA into C-SSA first and then back into normal CFG based on the concept of phi-congruence classes. At the beginning, all the arguments and the destination of a ϕ-function are in the same congruence classes. During the transformation, when two names in the same class conflict (i.e., live ranges overlap), copy operations are introduced for one of the names to divide the class into two. Eventually, when there are no conflicts between the names in the same class, T-SSA is transformed into C-SSA and each class is given a distinctive representative name. Several copy insertion algorithms are proposed. The best one employs both control flow information and live range dependence graph and results in a significant reduction in the number of copy instructions. However minimizing copy insertion during inverse transformation of SSA is still an open research question.

A different approach to reducing copy instructions follows the idea of introducing as many copies as necessary and eliminating the redundant copies by applying coalescing algorithms developed specially for this purpose. Two names can be coalesced if they are not both live at any point of the program. Boissinot et al. (Boissinot et al. 2009) proposed a coalescing algorithm by taking values of variables into account. By their definition, two names do not conflict even when they are both live at some point of the program as long as

their values are the same.

2.3 Other Important Representations

In addition to above mentioned representations, there are some other program representations which are significant for various reasons. Static Single Information (SSI) form (Ananian and Rinard 1999; Singer 2006), an extension to SSA places σ-functions at control branches. A σ-function is a multi-destinations function that each destination has the same value and a unique name. Through the embedded information of branch conditions, SSI supports predicate analysis and backward dataflow analysis. A distinguished property of SSI is its aggressive splitting of live ranges which gives it predicate analysis power without explicitly using predicates. Extended-Static Single Assignment (e-SSA) (Bodík et al. 2000) has the same property and similar program analysis capability. Particular in case of SSI, aggressive splitting of live ranges results in interference graphs which are interval graphs, providing significant advantage in coalescing and register allocation tasks if done on single assignment form. Of course, split live ranges must later be combined, and the approach can make sense only with an effective coalescer. However, this approach also results in an enormous increase in the number of gating functions used. For example, Singer (Singer 2003) reports an average of six fold increase in the number of gating functions in SSI compared to SSA. Program dependence graph(PDG) (Ferrante et al. 1987) is a representation which makes both the data and the control dependencies for each instruction in a program explicit through graph links. Ball and Horwitz (T.Ball and S.Horwitz 1992) give algorithms to reconstruct a control flow graph from a control dependence graph such as PDG. Pingali (Pingali et al. 1990) analyzes CFG, data dependence graph and a combination representation with control flow and data dependence information and summarizes the crucial properties for any good program representation such that a data structure can be easily traversed for dependence information. Then a program representation called dependence flow graph (DFG) which is based on dependence driven execution model is given. DFG naturally incorporates the best aspects of many

other representations and leads to a better algorithm for solving the constant propagation problem.

2.4 Code Motion and Compiler Optimizations

Code motion is an essential tool for many compiler optimizations. In this section, we first briefly discuss Partial Redundancy Elimination(PRE), one of the powerful optimizations that is carried out by code motion.

PRE combines and extends two other techniques:

1. Common subexpression elimination which eliminates the redundant computations. An expression is redundant at a program point p if it is computed along every path leading to p and none of its subexpressions has been redefined. If an expression is redundant at p, its computation at p can be replaced by a reference to a variable holding the computed value.

2. Loop-invariant code motion which moves loop-invariant expression out of the loop. An expression is loop-invariant if its value remains the same at each loop iteration. By moving the expression out of the loop, we reduce the computation times of the expression to once and still obtain the correct value.

A more common optimizable case is that the expression is redundant along some, but not all paths leading to p, which is defined as *partial redundancy*. PRE are the algorithms to remove partial redundant expressions to achieve execution speedup. The basic idea of PRE is to convert partially redundancy to (full) redundancy, that is, to copy and move a target expression to a proper point such that the expression at the original location becomes fully redundant. Once the original computation becomes fully redundant it can be replaced with a reference to the computed value. Various PRE algorithms exist aiming to address the question of where to move the copy instructions with different approaches.

Strength reduction is an optimization that replaces a costly operation with a set of equivalent, but less expensive operations. Strength reduction is very powerful especially

when the target expressions are in the loops. Code motion is used to move the code to a different point in the program where specialized circumstances allow the code to be replaced by less expensive sequence of operations.

Instruction scheduling is a compiler optimization that reorders the operations to improve instruction-level parallelism (ILP). Multiple operations can be executed in parallel on the processors that are equipped with pipelined functional units or multiple parallel functional units. The former processors are represented by pipelined machines and the later are represented by superscalar and VLIW architectures. We refer to machines with multiple functional units, namely superscalar and VLIW as ILP architectures. The basic idea of pipelining is to split the processing of an instruction into a series of independent steps so that CPU is allowed to issue instructions at the rate of the slowest step, which is much faster than the time needed to process all the steps at once. As a result, multiple instructions are executed simultaneously at any time on a pipelined machine. Hazards can happen when data dependencies between these instructions occur. When hazards happen, pipeline stalls or *No-Operations* must be inserted to ensure correctness. ILP architectures have the ability to issue more than one instruction per cycle by dispatching the instructions to multiple functional units. Scheduling algorithms are suggested for both superscalar and pipelined machines. While for a pipelined machine the goal is to issue a new instruction every cycle by eliminating pipeline stalls, for an ILP architecture with n functional units, the basic idea is to execute as many as n instructions each cycle. For both machines, the compiler is required to rearrange the code properly to better utilize the machine sources.

Memory scheduling is an important issue since memory instructions usually have longer latency than other instructions. In order to hide the latency of accessing memory, some techniques allow lifting of load instructions to an early position in the program so that when an instruction using the load value is executed, the value will be ready.

PRE and strength reduction are machine independent optimizations while scheduling is machine dependent. The state-of-art algorithms involving these optimizations will be summarized in the next two subsections.

2.4.1 PRE and strength reduction

PRE algorithms consist of initializing an auxiliary variable with a candidate computation and replacing original computations by reloading the variable. In 1979, Morel and Renvoise proposed an algorithm to suppress partial redundancies (Morel and Renvoise 1979). Their work became the first to combine redundancy elimination and loop-invariant code motion together. They developed the bi-directional iterative approach to global-flow analysis for code placement. Their idea is to put the computations as early as possible. An alternative code placement strategy called Lazy Code Motion(LCM) (Knoop et al. 1992) was proposed by Knoop, Rüthing and Steffen. Their algorithm decomposes the bi-directional structure of Morel and Renvoise's work and thus is more efficient. LCM achieves computational optimality in the sense that computations on each path can not be reduced further by means of safe code motion and the lifetime optimality in the sense that the lifetimes of the introduced variables are minimized. The basic idea is to move target expressions to the early program points to expose the maximum number of redundancies and then to push them to the latest points where the redundancies still remain to minimize register pressure. A practical implementation algorithm for LCM was proposed later by the same authors in (Knoop et al. 1994).

An algorithm called SSAPRE is presented in (Chow et al. 1997; Kennedy et al. 1999) for performing PRE based on SSA form. This work is one of the earliest that look into the relationship between use-def information for variables represented in SSA and the redundancy property for expressions. The algorithm is based on a sparse representation of expressions, the factored redundancy graph(FRG). In FRG, the real expression computations and Φ-nodes which are inserted at points where the value of the expressions may change represent the nodes and the control flow edges in original CFG and use-def edges for expressions represent edges. Analysis performed in FRG is similar to LCM, which first moves expressions up the graph and then pushes them down to determine the code motion region. SSAPRE achieves the same computation optimality and life-time

optimality as LCM.

Cliff Click proposed a different code motion strategy called Global Code Motion(GCM) (Click 1995). GCM algorithm first hoists instructions out of the original blocks (i.e., move code out of loops) and then schedule them according to data dependencies between instructions. An instruction can be moved as far up as it is dominated by its input and can be moved as down as it dominates its uses. Between the two points is the code motion region for the instruction. Cliff also presents the algorithm for global value numbering (GVN), which aims to replace a set of instructions that compute the same value with one instruction. Combined with GVN, GCM can achieve a net effect of performing constant propagation and PRE. Click's algorithm separates code motion from optimization issues. It is simple and fast, however, it may introduce extra computations along some paths.

Code motion alone can not eliminate all the partial redundancies. According to Bodik, Gupta and Soffa (Bodík et al. 1998), 73% of loop-invariant statements in SPEC benchmarks can not be eliminated by code motion alone. They proposed an algorithm based on the integration of code motion and CFG restructuring to achieve the complete removal of partial redundancies. Their work resorts to restructuring merely to remove the obstacles to code motion, which reduces the code growth resulting from code duplication. Additionally, using a profile to guide the optimization further reduces the code growth by selecting those computations that have sufficient run time gains over the cost of duplications.

Strength reduction has a close connection with PRE. Strength reduction methods can be classified into two families: One family treats it as a loop optimization issue that requires explicit detection of loop induction variables. The other unifies code motion in PRE with strength reduction. Compared to PRE, besides initializing an auxiliary variable with the computation and replacing original computations by reloading the variable, strength reduction techniques additionally update the variable between its initialization and uses.

Several other PRE algorithms extend their reach by incorporating strength reduction. An example of such an algorithm is lazy strength reduction (Knoop et al. 1993), proposed by

Knoop, Rüthing and Steffen. In lazy strength reduction, the candidate expressions are in the form of $v * c$ where c is a constant. The algorithm performs a similar technique of pushing up first to expose optimization opportunities and then pushing down to minimize register pressure as in LCM, with several refinements. Kennedy, Chow and their co-workers also proposed their strength reduction algorithm based on SSAPRE framework (Kennedy et al. 1998). This algorithm covers a broader class of candidates than lazy strength reduction. Since the algorithm is performed on expressions one by one, it can find and optimize new candidates that are formed by optimizing previous candidates.

Max Hailperin proposed a general framework called Thrift Code Motion (TCM) (Hailperin 1998). The technique is based on cost which can be instantiated to perform strength reduction. A computation has different costs when it is placed at different points in the program, e.g., a computation can be hoisted up to points where it is constant foldable, which means smaller cost. The general goal is to place computations so as to minimize their cost, rather than their number. So TCM consists of first moving computations as early as possible, then delaying them as long as the cost does not increase. The cost function can be computed via forward dataflow analysis. Hailperin's algorithm covers more candidates than any other previous works.

2.4.2 Scheduling and register allocation

Scheduling algorithms consist of scheduling for ILP and scheduling for Memory Level Parallelism (MLP). Early work for scheduling for ILP is mainly to find data independent instructions within basic blocks. List scheduling (Fisher 1979) using the *highest-level-first* priority scheme is the classic one. In this algorithm a directed acyclic graph representing the dependencies between instructions is constructed. The nodes represent the instructions and the edges represent the dependence relation between instructions, with latencies labeled on them. Any topological sort is a valid schedule. In order to eliminate stalls during the execution some heuristics are commonly used, e.g., if a candidate instruction is on the critical path, its priority is increased.

Trace scheduling (Fisher 1982) is a global acyclic scheduling algorithm which allows code motion across the basic block boundaries. A trace is an acyclic sequence of basic blocks in the CFG, forming a path through the program. Traces are selected and scheduled according to their execution frequency. Instructions in a trace are scheduled as if they were in a basic block. Rules of inter-block code motion are specified in (Fisher 1982). Trace scheduling puts its total focus on the current trace and neglects the rest of program. A different approach to global acyclic scheduling is based on the concept of *superblock*. Allen et al., (Allen et al. 1983) proposed the $if-conversion$ which converts all the branches into predicates and thus eliminates the branches in the flowgraph. The resulting code with one entrance and multiple exits is called a *superblock*. Superblocks can be scheduled using local scheduling techniques. After scheduling, reverse IF-conversion (Warter et al. 1993) is performed to regenerate CFG.

Instruction scheduling at the basic level is inadequate for superscalar processors. Bernstein and Rodeh (Bernstein and Rodeh 1991) proposed a scheme for intra-loop scheduling, which uses the control and data dependence information summarized in the program dependence graph (PDG) (Ferrante et al. 1987) to move instructions beyond the basic block boundaries. An instruction can be moved to a block which has the same control dependence as the block originally holding the instruction. Further, an instruction can be moved to its predecessor block speculatively (speculative instruction). A set of heuristics are used to pick the instruction to be scheduled next. For example, non-speculative instructions have high priority over speculative instructions.

For cyclic scheduling, the most common way is to unroll loops some number of iterations so that global acyclic scheduling algorithms can find enough basic blocks to perform the code motion. This approach still has the scheduling barrier at the back-edge and the cost of increased code size. A better approach is software pipelining (Rau and Glaeser 1981; Rau 1994).

Scheduling for MLP typically involves movement of multiple load instructions, in most cases speculatively. A speculative load is a load instruction that does not incur any

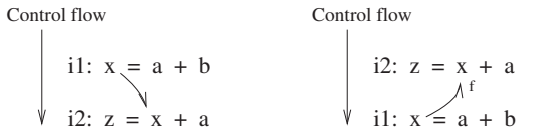
exception until another instruction uses the value loaded. Rogers and Li (Rogers and Li 1992) described a hardware mechanism to support speculative loading. Then they described how to lift a load within a basic block region, over a branch and across the loop region respectively. The highest point a load can be lifted within a basic block is after the instruction which the load is data dependent on. Considering the register pressure, a load should be lifted away from the first use of the loaded value merely to cover memory accessing latency. A load can be lifted over a branch if the load is from the block that is predicted to be executed most often. For loops, when there are not enough instructions to hide the latency of some speculative loads, these loads can be lifted across iterations.

2.5 Future Values

As it is known, code motion is prohibited due to data dependencies, as well as the control dependencies. Future values concept allows instruction movements beyond control and data dependencies. In order to see how this is possible, consider the statements shown in Figure 2.6(a). In this example, the control first encounters instruction `I1` that computes the value `x`, and then encounters the instruction `I2` which consumes the value. In Figure 2.6(b), the instruction `I2` has been hoisted above `I1`, and its source operand `x` has been marked to be a *future value* using the subscript `f`. If the machine buffers any instructions whose operands are future values alongside with any operand values which are not future until the producer instruction is encountered, the instructions can be executed with proper data flow between them even though the order at which the control has discovered them is reversed. In other words, when an instruction is hoisted beyond an instruction that defines the hoisted instruction's source operand, a *future dependency* results:

Definition 1. *When instructions I and J are* true dependent *on each other and the instruction order is reversed, the true dependency becomes a* future dependency *and is marked on the source operand with the subscript* f.

This execution semantics requires that when an instruction is hoisted in an arbitrary manner,



(a) Traditional data−flow (b) Future (reversed) data−flow

i_1: if($a < b$)
i_2: x = x +1

(c) Traditional control-flow

i_1: P=($a < b$)
i_2: $[P]x = x+1$

(d) If-conversion

i_2: $[P_f]x = x+1$
i_1: P=($a < b$)

(e) Future control-dependence

Figure 2.6: The concept of future data and control dependencies

the compiler has to make sure that a definition of the value is encountered across all paths. In the same manner, it's possible to represent control dependencies in future form too. Consider Figure 2.6(c). In this example, i_2 is control dependent on i_1. In Figure 2.6(d) predicate P is used to guard i_2, which represents the same control dependence. When the order of i_1 and i_2 is reversed (Figure 2.6(e)), predicate P becomes a future value and thus the original control dependence becomes future control dependence.

Future data and control dependencies may enable unrestricted code motion with proper program representations. They together form the foundation upon which this work will be built.

Chapter 3

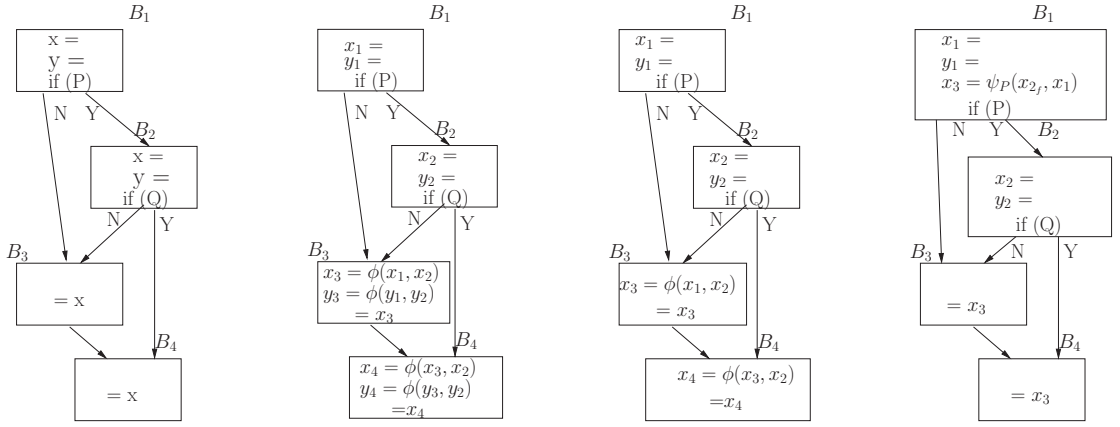
Future Gated Single Assignment (FGSA)

We have reviewed several important existing program representations, including SSA and GSA. Both SSA and GSA achieve single assignment semantics by inserting gating functions at the confluence nodes of the control-flow graph. In this chapter, we develop an alternative representation called Future Gated Single Assignment (FGSA) form which can potentially lead to better program analysis and optimization algorithms. This representation is built on the two fundamental concepts, namely, future values and congruence classes. Together, they provide the foundation for a sparse representation that can associate data and control-dependence information with the variables to the extent possible while permitting well-established program optimization algorithms work with ease with the new representation.

FGSA approach to sparse representation is unique. FGSA disassociates program facts from the control flow graph as much as possible by identifying a group of uses which receive the same set of definitions, forming a *congruence class*. It also identifies the control flow under which each definition flows into s uses and encodes this information int the form of a set of path expressions. Assigning a distinct name to each definition and a common unique name to the uses, single assignment semantics is easily achieved by using a *single gating function* per congruence class. This function is controlled by the set of path expressions, computed from control-flow and the function is placed at the lowest common dominator of the uses, instead of the confluence points of the program. Because this point maybe above some

of the definitions, *future dependencies* may result, giving the name *Future Gated Single Assignment* (FGSA) form to the representation.

3.1 Motivation and Comparison



(a) original CFG (b) minimal SSA (c) pruned SSA (d) FGSA

Figure 3.1: An FGSA Example

We present our motivation through an example shown in Figure 3.1(a). In this example, the same set of values flow into uses of x at node B_3 and B_4. y is defined at two places but not used. After applying Cytron et al.s' algorithm (Cytron et al. 1991) the graph shown in Figure 3.1(b) is obtained. Dead ϕ-functions can be eliminated through pruning (Choi et al. 1991) yielding the graph in Figure 3.1(c). Clearly, there is a single useful congruence class consisting of two uses of x, one at node B_3 and the other at node B_4 and the two definitions x_1 and x_2. Congruence classes involving y are not useful, since they do not reference a use. Observe that a query such as *which definitions flow into the use x in block* B_4 cannot be readily answered even in the pruned version without visiting the ϕ function in node B_3 although this node does not alter dataflow. This is a key observation for program analysis and optimization: since SSA places gating functions at the confluence points, control-flow information that is irrelevant to program analysis has to be dealt with, at times visiting multiple ϕ functions.

The FGSA version shown in Figure 3.1(d) includes an executable function, ψ which returns the value of the first argument if its predicate expression P is true and the second argument if it is false. This function is placed at the closest common dominator node of the uses so that its result dominates all its uses. Since some definitions may not be available at the node ψ is inserted, unavailable arguments become a *future value* and are marked with a subscript f (Ding and Önder 2010; Önder 2010). In the above example, when the predicate P is true, the function returns the value of x_2 *to be defined along the taken path* of the branch P. A separate pruning step is not necessary, since definitions of y do not have any uses.

FGSA is a single assignment representation which builds on the strengths of SSA and GSA by preserving single definition and dominance properties of these representations. As a result, it provides equivalent functionality to that of SSA and GSA and existing optimization algorithms can directly use it with minimal changes. It however improves significantly upon these representations by aiming for a clear separation of data-flow and control-flow aspects of the program. In FGSA, congruence classes represent participants in data-flow, predicate expressions represent the effect of control-flow and the gating function placement is data driven. Furthermore, predicate expressions are minimal, in the sense that if a given predicate expression is true, corresponding definition is guaranteed to reach the congruence class uses, although the program may traverse many more predicates before it reaches to one of the uses. These properties together enable FGSA to represent the same program by using fewer gating functions than either SSA or GSA. Our primary goal however is not to reduce the number of gating functions, but to represent essential and precise information about the program.

The separation of control flow structure from the gating and the grouping of definitions and uses into congruence classes also make it easy to compute the representation. The information that is necessary to generate FGSA is quite different than that of SSA or GSA, therefore the representation lends itself well to interval analysis. As a result, it is possible to use a series of T1/T2 (Hecht and Ullman 1974; Aho et al. 1986) transformations to compute the representation. We further simplify the construction by introducing a novel

transformation T_R which enables the interval analysis based algorithm to handle irreducible graphs without node replication, yielding a clean, reducible graph in single assignment form.

We now formally introduce the concept of congruence classes and the concept of path separability, the key for computing gating predicates.

3.2 Congruence Classes and Path Separability

We first define the *congruence classes* and *Gated congruence classes*:

Definition 2. *Let $U = \{u_1, u_2, \cdots, u_m\}$ be a set of uses which have the same reaching definition set $D = \{d_1, \cdots, d_n\}$. Such sets form a congruence class $CC = \{D, U\}$.*

In the following discussion, we use the notation $CC.U$ and $CC.D$ to refer to the use set and the definition set of the congruence class CC respectively.

Definition 3. *A Gated Congruence Class is a triple, given by $CC_G =< CC, G, \psi_G >$, where $G = \{g_1, \cdots, g_n\}$ is the gating predicate expressions separating definitions, ψ_G is the gating function which returns one of the definitions in CC.D such that $\forall i \in [1, n]$, if g_i is true, d_i is returned.*

A gating predicate expression g_i is a boolean expression that consists of branch conditions(predicates) in the control flow as variables and operators AND($\wedge$), OR($\vee$) and NOT($\neg$). For acyclic code, the predicate expression set G can be computed directly from the control flow. For cyclic code there are no predicate expressions one can observe which could control the gating at the loop header between the values flowing from the outside of the loop and loop carried values in a single-assignment form. We therefore separate the congruence classes into two types. Those congruence classes for which we can compute the gating predicates from control flow are called *path-separable congruence classes* and congruence classes involving cyclic regions are called *non-path-separable congruence*

classes. [1]

Definition 4. *A CC=$\{D,U\}$ is path-separable* if and only if *there $\exists$ a function f, such that $G = f(P)$ where $G = \{g_1, \cdots, g_n\}$ is the set of gating predicate expressions and $P = \{p_1, \cdots, p_n\}$ is the set of path expressions for CC.D.*

Generally, a *path expression* is defined as a regular expression over edges which represents all the paths from a given source node to the target node in the CFG (Tarjan 1981). In this work, we define a path expression over predicates by labeling each edge with a predicate, such that :

1. The taken edge of a conditional jump with condition P is labeled P;
2. The not taken edge of a conditional jump with condition P is labeled $\neg P$;
3. An unconditional edge is labeled T, representing the predicate value $true$.

Definition 5. *The path expression for a given path in the CFG is defined as the conjunction of the edge labels which form of the path.*

Therefore, if there are multiple paths from node u to v, the path expression from u to v is given by the union of individual path expressions, each of which representing a distinct path from u to v. Note that to compute gating functions, we compute path expressions for all the definitions in the same CC. Those path expressions must represent the paths starting from the same node, from which each definition in the CC has a chance to execute. The closest node in the control-flow graph where all definitions in a given CC have a chance to execute is the lowest common ancestor of the nodes where the definitions reside in the dominator tree, referred to as LCDOM. To compute the minimal path expressions for gating functions, we define:

Definition 6. *Given a definition $d_i \in CC.D$ defined in node z and node v=LCDOM$(CC.D)$, the (minimal) path expression for definition d_i is the path expression from v to z.*

[1] A simple congruence class CC={{d}, U} is trivially path-separable irrespective of the region involved and the congruence class value is d.

This definition leads us to the theorem that for path-separable congruence classes, the gating path expressions can be computed from the predicate expressions controlling the definitions:

Theorem 3.2.1. *Given $CC = \{\{d_1, d_2\}, U\}$ and path expressions p_1 for d_1, p_2 for d_2, the gating predicate expression for d_1 is given by $g_1 = \neg p_2 \wedge p_1$ if there exists a path on which d_2 kills d_1, and $g_1 = p_1$ otherwise.* [2]

Proof. Let u be the node where d_1 resides and v be the node where d_2 resides. Let w represent the node for any use in CC.

(1). If d_1 and d_2 do not kill each other, then the path $u \rightarrow w$ is clear of the other definition, i.e., when d_1 is executed, CC 's value is d_1. Since d_1 is executed when p_1 is true, $g_1 = p_1$.

(2). If d_2 kills d_1 along some path, then the path from $u \rightarrow w$ is not clear. d_1 can reach to uses in CC only when d_2 is not executed and d_1 is executed. $g_1 = \neg p_2 \wedge p_1$. □

More generally, given $CC = \{\{d_1, \cdots, d_m\}, U\}$ and path expression $P = \{p_1, \cdots, p_m\}$, gating predicate expression $G = \{g_1, \cdots, g_m\}$ is computed as:

$\forall j \in [1, m]$, $g_j = \neg p_{k_1} \wedge \neg p_{k_2} \wedge \cdots \wedge \neg p_{k_l} \wedge p_j$ when d_j is killed by $d_{k_1}, \cdots, d_{k_l}$ along some path where $k_1, \cdots k_l \in [1, m]$.

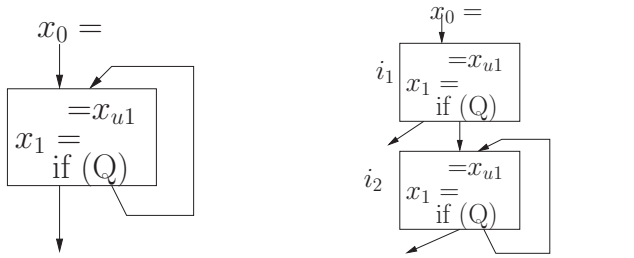


Figure 3.2: A non-path-separable CC: use belongs to two CCs

Computing the gating predicates for non-path-separable congruence classes requires splitting the congruence classes, computing gating predicates for each sub-congruence

[2] In the case of two definitions, expression $\neg p_2$ implies p_1, so the gating path expression can be simplified to $g_1 = \neg p_2$. However, for the general case, this does not hold and the conjunction is required.

class and recombining back using a special predicate. This is because the definitions outside the loop flow into the CC only *once*, but once this happens, the CC receives only the definitions within the loop. Consider the example in Figure 3.2(a). Peeling the loop once (Figure 3.2(b)), we observe that x_{u1} in the first iteration only receives x_0 while x_{u1} in successive iterations only receives x_1. Clearly, this equivalent program contains two sub-congruence classes, one of which represents the flow of values from the outside and another represents loop carried data flow. Both of these classes are trivially path-separable. The key issue is, splitting a given CC in this manner results in sharing of uses between the two sub-CCs. We therefore introduce *read-once predicates* and use them to combine the two CCs:

Definition 7. *The read-once predicate is a special predicate which becomes false once it is read.*

Note that, a read-once predicate is set to true before entering the loop and becomes false once it is read. Now we can construct a new CC by introducing a read-once predicate R: $CC_{x_{u1}} = \{v = \psi_R(CC_1.D, CC_2.D), \{x_{u1}\}\}$, which in this case can be simplified to $CC_{x_{u1}} = \{v = \psi_R(x_0, x_1), \{x_{u1}\}\}$. This new CC essentially represents the semantics of regular loops allowing us to construct the gating predicates using path predicates and use the same gating function uniformly across the representation without overloading its semantics.

So far, we have introduced the concept of congruence classes and showed how the concept can be used to construct single-assignment form by computing the gating predicates. Once the congruence classes and their gating predicates are computed, we can identify the points where the gating functions should be inserted and finalize the construction of FGSA. In the following sections we first give an overview of the algorithm and discuss the handling of reducible programs using T1/T2 transformations. In Section3.5, we show how to handle irreducible programs using our novel transformation T_R which permits the T1/T2 to proceed normally upon encountering an irreducible loop.

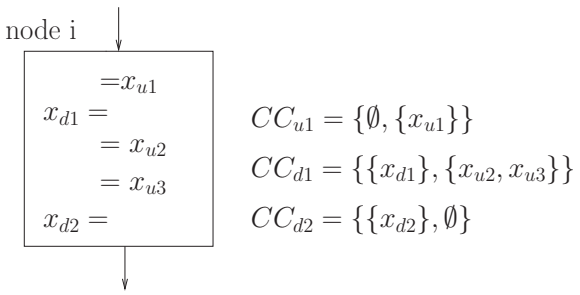


Figure 3.3: Local CCs computation

3.3 Efficiently Computing FGSA

FGSA construction algorithm consists of two steps. In the first step, the congruence classes are identified alongside with their gating predicates by using a bidirectional global flow analysis algorithm with edge placement (Graham and Wegman 1976; Dhamdhere and Patil 1993). This algorithm employs a *local computation* and a *global propagation* such that during local computation, definitions and uses in each basic block are grouped to construct local CCs and during global propagation local CCs that can communicate with other blocks (i.e., CCs for which definitions come from the predecessor blocks or values which flow into the successor blocks) are propagated globally using T1/T2 transformations alongside edges. Once the congruence classes are computed, the second step of the algorithm uses the dominator information to place the gating functions at the appropriate points on the CFG and the representation is finalized.

3.3.1 Identification of Congruence Classes

Consider Figure 3.3. x_{u1} is *upward exposed* since it receives definition(s) outside of node i. x_{d2} is *downward exposed*, whose value can flow into the successors of node i. x_{d1}'s value flows into both x_{u2} and x_{u3} locally. By scanning each statement in the block, uses and definitions are grouped by Definition 2 to form three congruence classes within node i as shown. Note that a single use can also form a CC which has an unknown value.

For global dataflow analysis, only CCs that are either upward or downward exposed in a

node can communicate with others, such as CC_{u1} and CC_{d2} in Figure 3.3. We refer to the two types of CCs as CC_{up} and CC_{down} of the block. We use the combination of the two types of CCs to categorize nodes and edges in the following description of T2 and T1 transformations:

[T1]*: Remove any edge that points from a node to itself. This is the key to the construction of the gating predicates for non-path-separable CCs. During this transformation, the loop node information is pushed onto the edges between the node and its predecessors and its successors.*

[T2]*: If a node v has a single predecessor u, T2 transformation consumes node v by node u. All the successor edges from node v become successor edges of node u. This step propagates partially computed CCs globally and computes the path predicates.* T2 transformation for node v therefore pushes v's information (i.e., locally computed CCs of node v) onto the edge (u,v) and collects the path expression from u to v. When v is consumed by u, the CCs are combined with existing CCs on the edges to form the propagating data in the next step.

When the whole program has only the *start* node left after a series of T1/T2 transformations, the construction of all the CCs is complete.

Now let us see through an example how the global propagation of *CC* information is handled. Consider the program shown in Figure 3.4(a). The only applicable transformation is a T2, therefore, first, node B2 is selected. The local CCs for B2 are $CC_{up}(B2) = \emptyset$ and $CC_{down}(B2) = \{\{x_{d1}\},\{x_{u1}\}\}$. After incorporating B2's local information with information on edge (B1,B2) and (B2,B3) which by default are all $\emptyset$s, we consume B2 and push the resulting CCs onto the edge. Since B2 was on the not taken path of its predecessor, predicate $\neg P$ is propagating down along the edge (Figure 3.4(b)). On the taken path of the branch (B1,B3), there is no computation, so both CCs on that edge is empty, with predicate P. Now there are two edges, both from B1 to B3 and they are to be merged. Both edges contain $\emptyset$ as CC_{up}, the resulting CC_{up} is $\emptyset$. Since one edge contains CC_{down} with a definition and the other doesn't, there is an unknown value that can reach the uses

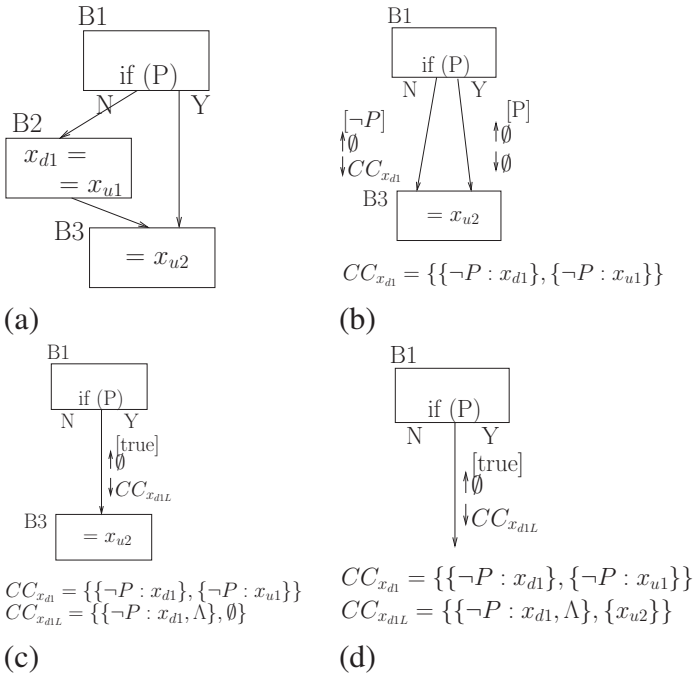


Figure 3.4: T2 example

below. A new CC, namely $CC_{x_{d1L}}$ is created, taking the union of definition sets of $CC_{x_{d1}}$ and a special symbol Λ which represents this unknown definition which reaches to this point. Predicates guarding the two edges are also merged, resulting in a *true* predicate to guard the merged edge (Figure 3.4(c)). Eventually, $CC_{x_{d1L}}$'s value will flow into x_{u2}, as shown in Figure 3.4(d).

Note in this example, during congruence class construction, predicates are associated with both the definitions and the uses. Use predicates are used during the gating function construction and insertion (Section 3.3.2) to avoid a partially dead gating function. In the following sections, we give a detailed algorithm for global propagation and *CC* construction.

3.3.2 Gating Function Construction and Insertion

Gating functions for non-path-separable CCs and exit functions are constructed and inserted during T1. For path-separable CCs, construction of gating path predicates requires

the use of *reduced reachable* set (Boissinot et al. 2008) for each node:

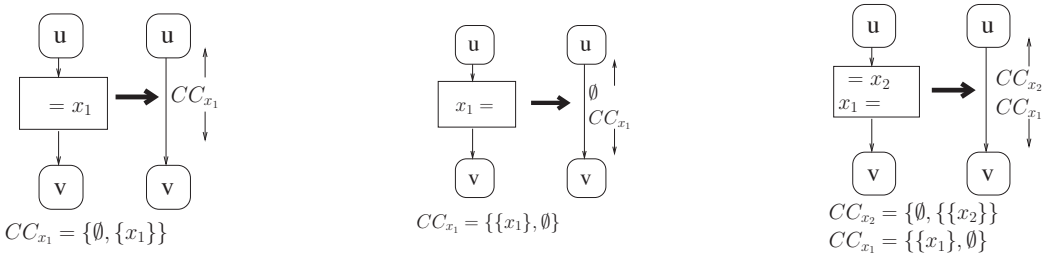
Definition 8. *The reduced reachable set of node v is a set of nodes that are reachable from v after all the back-edges in the CFG are eliminated, given by R(v)={x | there is a path from node v to node x containing no back-edges}*

The reduced reachability relation of nodes is used in gating function construction since it can answer the query whether the definitions in the same CC may kill each other. Together with the reduced reachable set information, Theorem 3.2.1 given in Section 3.2 enables us to compute the gating predicates from path predicates. Note that both the reduced reachable set information and path predicates are computed during the T1/T2 transformations. According to Theorem 3.2.1, the key question must be answered is whether definitions in the same CC may kill each other. Assuming definitions d_i and d_j belong to the same CC and they reside in node v_i and v_j, we observe that d_j may kill d_i if $v_j \in R(v_i)$ holds. Once the gating predicates are computed gating functions can be inserted at the flow-graph for each CC that contains more than one definition to divert those definitions. On one hand, in order to ensure that each use in a CC gets the value of the gating function, the gating function must dominate each use in the CC. Also considering minimizing the live range, the gating function is inserted at the lowest common dominator (LCDOM) node of all the uses in the CC. On the other hand, the LCDOM node may not be post dominated by the uses, in which case the gating function inserted at that point may be partially dead. To avoid that, we compute the disjunction of the predicates controlling the uses and use it to guard the gating functions. Any definition that appears below the gating function is marked as a future value (Ding and Önder 2010; Önder 2010).

3.4 Interval Analysis and T1/T2 Transformations

3.4.1 Acyclic Regions: T2 Transformation

Consider node v as the next T2 transformation candidate, which has a single preceding edge (u,v) and possibly multiple successor edges. (v,w_i) represents the i^{th} successor edge of v. For each of these edges, two congruence classes representing the upward exposed value and the downward exposed value are associated with the edge, annotated as CC_{up} and CC_{down}. During the computation and propagation, four cases of the combination of CC_{up} and CC_{down} may occur. We illustrate here using edge (u,v). The combinations happening within node v and on the edge (v,w_i) are the same. Predicates USE and DEF are used to identify each of the cases:



(a) Case 2 (b) Case 3 (c) Case 4

Figure 3.5: CC cases

Case1 : $CC_{up} = CC_{down} = \emptyset$. We set $USE(u,v) = false$ and $DEF(u,v) = false$. This is the case for an unprocessed edge or where the nodes processed between u and v are transparent to the given variable.

Case2 (Figure 3.5(a)): $CC_{up} = CC_{down} \neq \emptyset$. We set $USE(u,v) = true$. Between u and v and before node v is processed, a node with a use of x was eliminated through T2 transformation. A CC is created for the use, namely CC_{x_1} in the example. Since such a CC is upward exposed to any definition and downward exposed to any use, it is associated with edge (u,v) as both CC_{up} and CC_{down}.

Case3 Figure 3.5(b): $CC_{up} = \emptyset$ and $CC_{down} \neq \emptyset$. We set $DEF(u,v) = true$. Figure shows a case where the intermediate node contains a downward exposed definition. A CC of a single definition is created and uses further down will later join. To the upward direction, the CC is *closed* since definitions further up cannot join (because they are killed by this definition and cannot be in the reaching definition set of the uses of the CC), neither can the uses above (because their reaching definition set won't contain the definition). In such a case, CC_{up} is $\emptyset$.

Case4 Figure 3.5(c): $CC_{up} \neq CC_{down}$ and neither is $\emptyset$. We set $USE(u,v) = true$ and $DEF(u,v) = true$. This case represents a combination of the above two cases.

To perform T2 transformation on node v, assuming local CCs for node v, $CC(v)$ (i.e., $CC_{up}(v)$ and $CC_{down}(v)$) is computed, Algorithm 1 (Figure 3.6) is applied to compute $CC(u,w_i)$ by incorporating $CC(v)$ with $CC(u,v)$ in the first phase and the intermediate results (annotated as $CC(u,v]$) with CC(v,w_i) in the second phase. During two CCs incorporation, if the preceding CC contains no definition, two $CC_{up}.U$ are merged to form the new CC_{up}; otherwise, the succeeding $CC_{up}.U$ meets their definitions in the preceding CC_{down} and thus joins the preceding CC_{down}. Definitions in the preceding $CC_{down}.D$ are killed by definitions in the succeeding $CC_{down}.D$. A global list is maintained, called *globalCC*, which records the valid global congruence classes. Initially, $CC(u,v), CC(v), CC(v,w_i)$ are all listed in *globalCC*.

Note in Algorithm 1 line (11), a special symbol Λ may appear in a CC_{down}. Λ is created when two CCs merge, resulting from two edges' merging (see Merging Edges). When it appears in a CC_{down}, it represents upward exposed definitions, which are bound to the value of the corresponding CC_{up}. The Λ is replaced by the same definitions once the CC_{up} finds them. In order to compute the *reduced reachable set* information, we associate a node set S with each edge within Algorithm 1 and 2 by lines annotated by (R).

Algorithm 1
We define *Function f* such that :
$CC(u,w_i) = f(f(CC(u,v),CC(v)),CC(v,w_i))$

Function f:
INPUT: $CC(X)$, $CC(Y)$
OUTPUT: $CC_{up}(Z), CC_{down}(Z)$

begin

//case 1 and 2
(1)if ($\neg DEF(X)$) then
(2) $CC_{up}(Z).U = CC_{up}(X).U \cup CC_{up}(Y).U$
(3) remove $CC_{up}(X), CC_{up}(Y)$ from globalCC
(4) add $CC_{up}(Z)$ into globalCC
(5) $CC_{down}(Z) = DEF(Y)?CC_{down}(Y) : CC_{up}(Z)$

// case 3 and 4
(6)else
(7) $CC_{down}(X).U = CC_{down}(X).U \cup CC_{up}(Y).U$
(8) remove $CC_{up}(Y)$ from globalCC
(9) $CC_{up}(Z) = CC_{up}(X)$
(10) $CC_{down}(Z) = DEF(Y)?$ $CC_{down}(Y) : CC_{down}(X)$
(11) if (DEF(Y) and $\Lambda \in CC_{down}(Z)$) then
(12) $CC_{down}(Z).D = CC_{down}(Z).D \cup CC_{down}(X).D$
// compute R sets
//(R1) add $S(Y)$ to reduced reachable
//(R1) set of each node in $S(X)$
//(R2) $S(Z) = S(X) \cup S(Y)$
end

Figure 3.6: Algorithm 1: T2-CC incorprating

3.4.1.1 Computing Path Predicate Expressions

Path predicate expressions are also computed during the T2/T1 transformation and participate in congruence class propagation. T2 transformation involves two types of path merging:

1. When concatenating path (u,v) and (v,s) to form (u,s), given $\alpha(u,v)$ for path predicate expression of (u,v) and $\alpha(v,s)$ for (v,s), $\alpha(u,s) = \alpha(u,v) \wedge \alpha(v,s)$.

2. When combining two paths q_1, q_2 that have the same predecessor s and the successor t, given $\alpha(q_1), \alpha(q_2)$ as the path predicate expressions for q_1 and q_2, $\alpha(s,t) = \alpha(q_1) \vee \alpha(q_2)$.

In a local CC computation, such as the precomputed $CC(v)$, definitions and uses are guarded with the predicate T, representing the value *true*. Given path predicate expression $p(u,v)$ for path (u,v), $p(u,v)$ becomes the path expression for each definition and use in $CC(v)$ from u to v. The predicated $CC(v)$ then is combined with $CC(u,v)$. The same procedure applies when $CC(u,v]$ incorporates with $CC(v,w_i)$.

Path predicate expressions involving loops are computed by the following and used in Section 3.4.2. Suppose path p consists of a loop on path p_1, let $\alpha(p_1)$ be the path predicate expression for p_1, $\alpha(p) = \alpha(p_1)^*$ (infinite conjunction of the predicate expression of $\alpha(p_1)$).

3.4.1.2 Merging Edges

After eliminating node v, there may be multiple edges from u to w_i. Such edges are merged into a single edge. The edge merging results in merging of the CCs associated with the edges as well as the path predicate expression. Given two edges e_i and e_j, with the same predecessor and successor node , we compute the resulting CC_{up} and CC_{down} using Algorithm 2 (Figure 3.7). All the use sets that are upward exposed are unioned to form a new CC since all these uses must receive the same set of definitions. Definitions flowing out are merged into set D. If any existing CC contains such a definition set, it represents the CC_{down}, otherwise a new CC is created with the definition set D.

3.4.2 Cyclic Regions: T1 Transformation, Exit Function

Each candidate for T1 transformation is a self pointing node v and a back-edge e_{back}. Note that before the T1 step, CC(v) for local CC of node v and the summary of dataflow information within the loop excluding the node v CC(e_{back}) have already been computed. The path predicate expression from v to e_{back} is also available. Merging CC(v) and CC(e_{back}) using Algorithm 1 forms the base CC for T1 transformation, referred to as CC(lp). T1 transformation computes $CC(lp^*)$, which represents the effect of iteration on CC(lp). Once the back-edge is processed and deleted, the rest part of propagation is handled by additional T2 transformations.

When there is no definition in the loop (i.e., DEF(lp)=false), the uses in the loop, as well as the uses following the loop, get the same value that initially comes to the loop, therefore, $CC_{up}(lp^*) = CC_{down}(lp^*) = CC_{up}(lp)$.

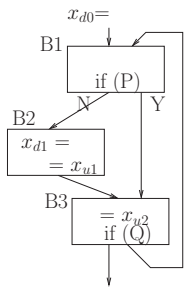
When there are definitions in the loop (i.e., DEF(lp)=true), the algorithm needs to deal

Algorithm 2
INPUT: $CC_{up}(e_i), CC_{down}(e_i), CC_{up}(e_j), CC_{down}(e_j)$
OUTPUT: CC_{up}, CC_{down}
begin

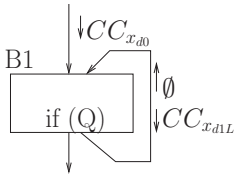
$CC_{up}.U = CC_{up}(e_i).U \cup CC_{up}(e_j).U$
add CC_{up} in the globalCC
remove $CC_{up}(e_i)$ and $CC_{up}(e_j)$ from globalCC
if ($DEF(e_i)$ or $DEF(e_j)$) then
  $D = CC_{down}(e_i).D \cup CC_{down}(e_j).D$
  // $CC(e_i)$ or $CC(e_j)$ does not contain definitions
  if($\neg DEF(e_i)$ or $\neg DEF(e_j)$) then
    $D = D \cup \{\Lambda\}$
  if (CC_D not exist in globalCC)
    $CC_{new} = \{D, \emptyset\}$
    add CC_{new} to globalCC
    $CC_{down} = CC_{new}$
  else
    $CC_{down} = CC_D$
else
  $CC_{down} = CC_{up}$
// compute R sets
//(R) $S(u,v) = S(e_i) \cup S(e_j)$
end

Figure 3.7: Algorithm 2: T2-CC merging

with two issues. First, the loop defined definitions may flow through the back-edge to uses in $CC_{up}(lp)$, which also receive initial values. Second, when a loop defined value flows outside of the loop, it must be separated from iterative values. Consider the two cases:



(a)



$CC_{x_{d1}} = \{\{\neg P : x_{d1}\}, \{\neg P : x_{u1}\}\}$
$CC_{x_{d1L}} = \{\{\neg P : x_{d1}, \Lambda\}, \{x_{u2}\}\}$
$CC_{x_{d0}} = \{\{x_{d0}\}, U\}$

(b)

$CC_{x_{u2.1}} = \{\{\neg P : x_{d1}, x_{d0}\}, \{x_{u2}\}\}$
$CC_{x_{u2.2}} = \{\{Q^+ \wedge \neg P : x_{d1}, Q^+ : x_{u2}\}, \{Q^+ : x_{u2}\}\}$
$CC_{x_{u2}} = \{v = \psi_R(CC_{x_{u2.1}}, CC_{x_{u2.2}}), \{x_{u2}\}\}$

(c)

Figure 3.8: A self-referencing gating function

(1) $CC_{down}(lp).D$ does not contain Λ. In this case loop carried definitions are all defined within the loop. The uses in $CC_{up}(lp)$ can receive values initially flowing into the loop, as well as the values defined in the previous loop iteration. Figure 3.2 as we discussed before shows such an example. We set $CC_{up}(lp^*) = CC_{up}(lp) = \{\emptyset, \{x_{u1}\}\}$ during T1, which later merges with some definition(s) to form CC_1. We also incorporate $CC_{up}(lp).U$ and $CC_{down}(lp).D$ to form CC_2 and combine them using the *read-once* predicate.

Read-once predicate needs to be initialized before entering the loop. A read-once predicate is introduced for each T1 transformation at the immediate dominator of the T1 candidate node. If a read-once predicate is needed by multiple CCs in the loop during the same iteration, a temporary (normal) predicate must be introduced at the loop header and assigned to the read-once predicate, ensuring read-once semantics.

(2) $CC_{down}(lp).D$ contains Λ, which represents an upward exposed definition set. Consider the example shown in Figure 3.8 in which loop carried values consists of both the initial value and the loop defined value. Within the loop (Figure 3.8(a)), the CC(lp)s are computed, as shown in Figure 3.8(b). x_{u2} receives a loop carried value x_{d1} and an initial value, which turns out to be x_{d0}. Note that, along the taken path of P, x_{d0} flows to x_{u2} for the first iteration , while in the subsequent iterations the previous iteration value of x_{u2} flows, which can be either x_{d0} or x_{d1}. Hence x_{u2} must belong to two CCs ($CC_{x_{u2.1}}$ and $CC_{x_{u2.2}}$) (Figure 3.8(c)). It should be noted that, within $CC_{x_{u2.1}}$, definitions are path-separable and the gating function can be constructed as $\psi_P(x_{d0}, x_{d1})$. The same reasoning applies to $CC_{x_{u2.2}}$. Therefore, the gating function for $CC_{x_{u2}}$ is simplified to $x_{u2} = \psi_P(\psi_R(x_{d0}, x_{u2}), x_{d1})$.

To handle the value flow outside the loop, we introduce a new function:

Definition 9. *The exit function $\eta(d_i)$ returns the last value of an iteratively executed definition d_i.*

When a loop defined value flows to uses both inside and outside the loop region (i.e., statically those uses have the same reaching definition set), only the value of the last iteration flows to the uses outside of the loop. For a $CC_i = \{D, U\}$ such that definitions are all defined in the loop and some uses are within the loop while others are not, we divide

CC_i into two CCs, namely CC_{i1} and CC_{i2} such that CC_{i1}={D, U_1} where U_1 is the set of uses in the loop and CC_{i2}=$\{\eta(D), U_2\}$ where U_2 is set of uses outside of the loop. Insertion of *exit* functions therefore requires classifying uses in the CC based on whether the use is in the loop region. Assuming the path predicate expression on the back-edge is p_1 and the path predicate of a use outside the loop is p_2, the property $p_1 \wedge p_2 = false$ holds. Applying this property divides the uses into two parts and the CC for each part is computed.

3.5 Irreducible Graphs and T_R Transformation

The interval analysis algorithm presented so far can only handle reducible graphs. In order to handle irreducible graphs, the graph can be converted into a reducible graph using node splitting (Janssen and Corporaal 1997; Unger and Mueller 2002). Unfortunately, node splitting is a technique which may result in significant code growth and the complexity of analysis that is necessary to minimize the growth is not trivial. We therefore introduce a novel transformation called T_R which eliminates irreducibility at the cost of single basic block per irreducible loop by inserting a unique loop header n_{head} each time it is applied. We first give a few key definitions adopted from the previous works (Janssen and Corporaal 1997; Unger and Mueller 2002) before we present an intuitive view of the transformation.

Definition 10. *Given a loop L, an* entryedge *is an edge such that the source node is not in L and the target node is in L. The target node of an* entryedge *is an* entrance *of L.*

For loop L to be irreducible, there must be more than one *entrance*. It is easy to show that all the entrances of L share an external node *e* as their immediate dominator, referred to as the shared external dominator *SED*.

Definition 11. *Given a loop L, node e as SED of all the entrances, we define* external path *from e to entrance n_i as the union of all the paths from e to n_i only consisting of nodes that are not in L; the corresponding* external path predicate *represents the* external path *as a predicate expression, referred to as* EPP.

Note that an entrance gets executed either when the corresponding *EPP* is true or the control flow is already in the loop. This simple reasoning leads to a transformation based on read-once predicates, since we can use a read-once predicate to test whether the control flow is in the loop or not.

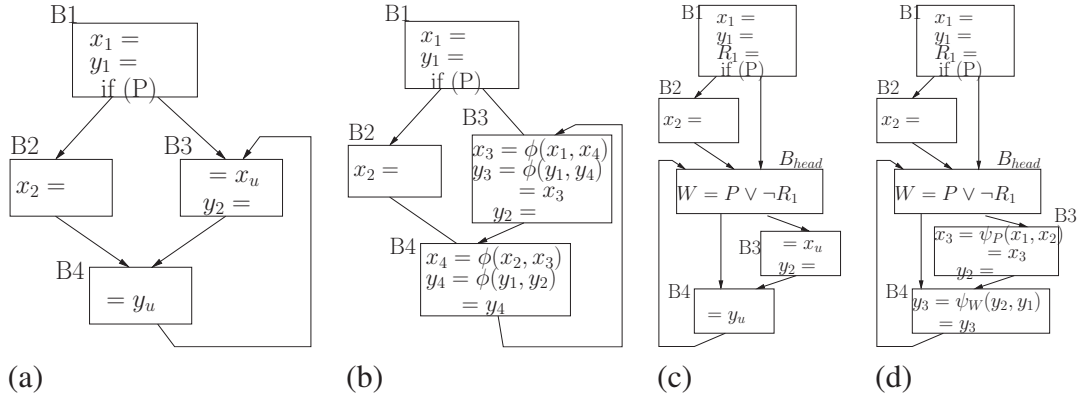


Figure 3.9: T_R transformation on an irreducible graph

Now consider Figure 3.9. In this graph, the loop is $\{B3, B4\}$ and $B1$ is the SED. We select node $B3$ in Figure 3.9(a) as the target entrance, and introduce a new loop header B_{head}. Edge $(B2, B4)$ and $(B1, B3)$ are redirected to this header and the edge $(B4, B3)$ now is the back-edge yielding the graph Figure 3.9(b). Since the selected entrance $B3$ is on the P path from SED, the branch condition in B_{head} is given by $W = P \wedge \neg R_1$.

Upon completing the T_R transformation, the interval analysis algorithm can proceed to compute the CCs using the algorithms presented in Section 3.4.1 through Section 3.4.2. The resulting CCs are given by $CC_{x_u} = \{\{P : x_1, \neg P : x_2\}, \{x_u\}\}$ and $CC_{y_u} = \{\{W : y_2, \neg W : y_1\}, \{y_u\}\}$ and FGSA form for the loop is shown on Figure 3.9(c). The SSA version of this irreducible loop is given in Figure 3.9(d) for comparison. Note that even though the variable x_3 is loop invariant, it is harder to move out of the loop region in case of SSA version, whereas in case of FGSA the move can easily be done by some PRE algorithm as the loop is now reducible. We now give a formal definition of T_R transformation using read-once predicates as follows:

Definition 12. *T_R transformation: Let $G =< N, E >$ be an arbitrary (irreducible)*

control-flow graph, L an irreducible loop in G, n_0 a selected entrance of L, then the transformation $G' =< N', E' >= T_R(G, L, n_0)$, is defined as follows:

† $N' = N \cup \{n_{head}\}$, *(3.4.0)*

† $E' \subset N' \times N'$ *such that the following restriction holds:*

$(x,y) \in E \wedge x \notin L \wedge y \in L \Leftrightarrow (x, n_{head}) \in E', (n_{head}, y) \in E'$, *(3.4.1)*

$(x,n_0) \in E \wedge x \in L \Leftrightarrow (x, n_{head}) \in E'$, *(3.4.2)*

otherwise, $(x,y) \in E \Leftrightarrow (x,y) \in E'$.

† *Let EPP of n_0 be P. Introduce a read-once predicate R_1. The branch condition in node n_{head} is set to $P \vee \neg R_1$. (3.4.3)*

The above transformation algorithm inserts a unique loop header n_{head} for the loop (rule 3.4.0). n_{head} dominates nodes in *L.* Any edge from a node outside the loop to a node inside the loop is directed through n_{head} (rule 3.4.1). Any edge heading to the selected entrance is redirected to the header as a back-edge (rule 3.4.2). After applying the T_R transformation, the selected entrance becomes a single predecessor node (n_{head} is its single predecessor) so that it can be consumed by a T2 transformation. This property guarantees that the size of irreducible loop is decreasing. Therefore, an irreducible graph will become reducible after a finite number of T_R transformations together with T1/T2 transformations.

In general, elimination of irreducible loops is exponential (Carter et al. 2003) in the sense that for an irreducible graph equivalent to a complete graph to be converted, the resulting reducible graph contains at least 2^{n-1} nodes. However, this is only true for traditional node splitting conversion. As indicated by the same reference, and exploited by Erosa and Hendren (Erosa and Hendren 1994) this is not applicable to guard based irreducibility elimination. Contrary to node-splitting, T_R transformation is a linear transformation and the irreducible graphs do not cause exponential code growth or exponential processing time either during the conversion to FGSA or during the inverse transformation. This is because, each T_R transformation adds a single node (basic block) containing a single instruction for

each irreducible core that is reduced. Next we prove that T_R eliminates the irreducible loops with linear code growth.

Definition 13. *Given a loop L, SED-set(L)=*$\{n_i \in L | idom(n_i) = e \notin L\}$. *MSED-set is the maximal SED-set of L.*

Definition 14. *A loop L is SED-maximal if there is no other loop L' such that* $L \subset L'$ *and MSED-set(L)*$\subset$ *MSED-set(L').*

With these definitions, we know that if we can reduce the size of MSED-set of each irreducible loop down to one, the graph becomes reducible.

Theorem 3.5.1. *Let L be an SED-maximal loop, K be its MSED-set,* n_i *be one of the entrances of L, then* $n_i \in K$.

Proof. Let e be SED of K and let e' be the immediate dominator of n_i. We claim that e dominates e'. This can be proved by contradiction. Let p_1 be the path from start to n_i which does not contain e and let k be an arbitrary node in K. There must be a path p_2 from n_i to k because they are both in L. By concatenating p_1 with p_2, we obtain a path to k which does not contain e, which contradicts the assumption that e immediate dominates k. Since e dominates n_i and e' immediate dominates n_i, e dominates e'. Similarly, we can prove that e' dominates e. Then e is the immediate dominator of n_i. Because K is the maximal set that contains all the nodes in L that share e as immediate dominator, $n_i \in K$. □

Theorem 3.5.2. *Let L be an SED-maximal loop, K be its MSED-set. Each* $T_R(L)$ *reduces the size of K by at least one.*

Proof. Let n_0 be the selected entrance. According to Theorem3.5.1, n_0 is an element in K. After the transformation, n_0 has a single predecessor which is n_{head}, therefore it no longer belongs to K. n_{head} immediate dominates nodes in K, therefore, n_{head} does not belong to K. For nodes in $L \backslash K$, their dominance relationship does not change. No node is added into K after T_R. Therefore, the size of K is reduced by at least one. □

Theorem 3.5.3. *T_R eliminates the irreducible loops with linear code growth.*

Proof. Let L be an SED-maximal loop in G, K be its MSED-set, and e be SED of K. $T_R(L)$ does not change dominance information of G except that n_{head} becomes the immediate dominator of nodes in K and e immediate dominates n_{head}. Therefore, $T_R(L)$ cannot affect other SED-maximal loops in G. According to Theorem 3.5.2, the T_R transformation needs to be performed at most $|K|-1$ times on L to make size of K to become one. Given G has t number of SED-maximal loops and each MSED-set is $K_i, i \in (1,t)$, at most $\Delta = \sum_{i\in(1,t)}(|K_i|-1)$ T_R transformations are required. Since each T_R inserts a single node with a single branch instruction, the code size is increased by Δ. □

3.6 Experimental Analysis

We compute the number of gated CCs and compare it with the number of ϕ-functions on SSA. For this purpose, we modified GCC 4.2.4 and compiled the SPEC2000 program suite with optimization flags -O3. This computation permits a direct comparison between FGSA and SSA since the number of FGSA gating functions is equal to the number of gated CCs. In order to compute the number of gated CCs, given the SSA form, we check each argument for each ϕ-function. The ϕ-function argument can either be defined by a real instruction or another ϕ. In the latter case, the argument is replaced by the arguments in its definition and the procedure is recursively applied.

In this way, uses can be classified into groups according to different definition sets, which are then organized into CCs. GCC builds SSA based on Cytron's algorithm but includes a pruning procedure which detects and avoids inserting dead ϕ-functions. Our experiment is performed both with the procedure on and off.

Without pruning, the number of CCs is 73.7% less than number of ϕ-functions on an average. Table 3.1 shows the results over real variables after pruning. Data is collected based on each function in a benchmark program. Therefore, item *Max* refers to the maximum reduction in the functions while *Average* refers to the average reduction over

Table 3.1
CCs vs pruned ϕ-functions over REAL

	vars	phis	ccs	% Reduction Max	% Reduction Average
164.gzip	3715	624	514	42.86	8.85
175.vpr	16648	1309	1092	61.11	7.39
176.gcc	125212	15810	14206	66.67	4.8
181.mcf	899	161	117	60	12.17
186.crafty	14341	1485	1226	67.47	10.55
197.parser	18720	2887	2653	50	6.08
253.perlbmk	20330	1789	1656	50	2.83
255.vortex	36585	1913	1747	50	1.9
256.bzip2	3598	342	286	50	12
300.twolf	21676	2653	1991	64.91	10.22
177.mesa	4446	3511	2779	53.33	21.03
179.art	1383	173	155	19.23	10.4
183.equake	1670	131	125	10	4.58
188.ammp	13735	1433	1297	51.72	9.49

functions within the same program.The pruning procedure reduces the number of ϕs significantly. However, comparing CCs with pruned ϕs, we still observe a maximum reduction of 67.47% from a function in 186.crafty and an average reduction of 7.7%. Note that FGSA doesn't require a separate pruning procedure yet produces fewer number of gating functions. Benchmark 171, 172, 173, 200 and 301 are not reported. Number of CCs and ϕ-functions are almost the same in these Fortran programs due to simple control flow structures they have. When all the variables including virtual ones are taken into account, number of CCs are reduced more over ϕs, resulting an average reduction of 10% with ϕ pruning and 72.3% without pruning.

Table 3.2 shows the distribution of CCs based on the number of definitions for SPEC2000 INTEGER suite. From the table we can observe, CCs consisting of two definitions are dominant, making up 62% or more of all the CCs across the suite. for all the benchmarks. CCs consisting of more than two definitions require nested gating functions when single predicate controlled gating functions are used. However, the number of CCs of more than four definitions takes 13.38% in the worst case.

Table 3.3 shows the distribution of the length of predicate expressions found in congruence classes. The data has been obtained by adapting Tu and Padua's (Tu and Padua 1995)

Table 3.2
Number of definitions in CCs

	ccs	2defs%	3defs%	4defs%	4^{+}defs%
164.gzip	514	78.79	11.87	4.28	5.06
175.vpr	1092	81.32	7.97	7.97	2.75
176.gcc	14206	76.95	10.14	4.65	8.26
181.mcf	117	68.38	27.35	1.71	2.56
186.crafy	1226	62.07	14.52	10.03	13.38
197.parser	2653	79.8	16.66	2.41	1.13
253.perlbmk	1656	79.71	8.33	7.13	4.83
255.vortex	1747	87.58	5.15	3.15	4.12
256.bzip2	286	80.42	12.24	5.59	1.75
300.twolf	1991	76.49	10.9	9.94	2.66

Table 3.3
Length of CC predicate expressions

Benchmark	median	average	$\% > 4$	$\% > 8$	max
164.gzip	1	1.98	12.5	0.4	13
175.vpr	1	2.06	7.1	1.4	31
176.gcc	2	3.79	20.3	9.2	132
181.mcf	1	1.97	6.0	1.7	9
186.crafty	2	3.15	16.7	6.1	95
197.parser	2	2.27	12.9	1.3	83
253.perlbmk	1	2.5	12.6	5.3	31
255.vortex	1	2.01	11.2	3.4	17
256.bzip2	1	1.71	4.6	1.4	15
300.twolf	1	2.23	8.1	3.5	32

GSA path predicate computation algorithm to FGSA. We observe that some benchmarks, such as 176.gcc exhibit complicated control flow which manifests itself in the length of predicate expressions. On the other hand, the median across the whole suite is no more than two, and predicate expressions which are longer than eight elements make up less than 10% of the congruence classes indicating that in these benchmarks, there are some CCs which span large regions of complicated control flow with large live ranges of definitions. However, even for 176.gcc, congruence classes which has more than 16 predicates is 3.7%. It is important to note that the length of the predicate expressions is a property of the program at hand. As a result, optimization algorithms have to traverse and evaluate these conditions either through predicate expressions built onto the CCs, or by traversing a chain of ϕ functions.

3.7 Complexity of FGSA Construction

In the following discussing, we follow the practice for most recent SSA work (Bilardi and Pingali 2003; Das and Ramakrishna 2005) and report the complexity per variable. Given a program, let the number of nodes, edges, user defined variables and instructions be N, E, V and I respectively. The FGSA construction is done in three steps:

1. The local CC computation scans each instruction in each node. This is done once for all the variables, hence the time complexity per variable is O(I)/V.

2. During CC propagation, Algorithm 1 runs for each node that contains a single predecessor, therefore its running time is bounded by O(N). Algorithm 2 runs over edges which is bounded by O(E). Running time for T1 is bounded to O(N).

3. Let the total number of global CCs that require gating functions be C_{tot}, bounded by the number of join nodes in the program, which is bounded by O(N). For each definition in one of the CCs, computing its gating path predicate requires a query of reduced reachable sets. The total number of queries is the total number of definitions in the CCs, represented as $\sum_{CC_i} |CC_i.D|$. A very loose bound for the number of definitions in a CC would be O(N), which will yield a worst time complexity of $O(I)/V + O(N+E) + O(N^2)$. However, the experiments in Section 3.6 shows that for most of CCs, the size of $CC_i.D$ is a small number. The term $\sum_{CC_i} |CC_i.D|$ can be folded into C_{tot}, yielding a bound of O(N).

Putting it all together, the expected time complexity for FGSA construction per variable is $O(I)/V + O(N+E)$.

3.8 Executable FGSA

FGSA can be executed in a predicated architecture or on an architecture that is fully equipped with the ability to execute the gating functions in FGSA.

(1)$x_1 = \cdot$ if true
(2)$x_2 = .$ if P
(3)$x_3 = x_1$ if $\neg P$
(4)$x_3 = x_2$ if P

Figure 3.10: Predicated instructions

Generally, in a predicated architecture such as IA-64, the execution of an instruction is guarded by a predicate(one bit predicate register), for example: Figure 3.10 shows the sequence of code is the Program in Figure 4.1(b) after it is converted into predicated instructions. A compiler employs *if-conversion* (Allen et al. 1983) to eliminate branches and to convert a control-flow region of program into a linear sequence of predicated code. Note that gating functions in FGSA can be easily transformed into predicated instructions. Based on the definition of gated congruence classes, a gating function $v_1 = \psi_{(p_1,p_2,,p_n)}(d_1,d_2,,d_n)$ can be transformed into: $v_1 = d_1$ if p_1, $v_2 = d_2$ if p_2,$\cdot$, $v_n = d_n$ if p_n. Note that gating functions with a read-once predicate can be converted in the same approach as long as the read-once predicate is initialized and reset properly after being read. Because gating predicates are always disjoint, the predicates of predicated copy instructions resulting from a particular gating function are disjoint as well, which is a unique property of FGSA.

Assuming the architecture is a traditional predicated architecture which does not support future value execution, the future values must be eliminated after we convert the gating functions into predicated copy instructions . For each gating function argument, there is a particular point of the program where the value of the argument is produced and available. Therefore, in order to eliminate future values, we can move the affected copy instruction right below the definition of the future argument.

Predicates present a challenge to program analysis and optimizations such as register allocation. This is because the interference between variables depends on not only the live ranges but also the relationship of predicates of live ranges. Consider the code in Figure 3.10, x_1 is live between instruction (1) and (3) while x_2 is live between (2) and (4).

Without considering predicates, x_1 and x_2 interfere and cannot be put into the same register. However, because x_2 is defined and used on P while x_1 is used on $\neg P$ and P and $\neg P$ are complementary, x_1 and x_2 can share the same register. Several papers (Gillies et al. 1996; Hoflehner 2010) address the issue. These techniques analyze predicate relationships and allocate registers for predicated code. Predicated code resulting from FGSA can equally be analyzed using these techniques although the disjointness of gating predicates can further simplify the predicate analysis.

FGSA can directly be executed on an architecture which supports future values. This support is no more complicated than allocating a register upon encountering an instruction with a future value through the renamer. When the producer instruction is encountered, it is simply renamed to the allocated register. It implies that the live range of a variable which has a future value use starts from the future use and ends at the last use or a definition depending on which one comes last. Gating functions can be directly executed similar to conditional moves, encountered in many architectures.

We don't investigate further execution of FGSA based code on predicated and future valued architectures in this dissertation. Instead we concentrate on compiling for transitional architectures.

3.9 Conclusion

We have presented a new program representation, its computation using an interval analysis based approach, a novel transformation that allows conversion of irreducible loops to reducible loops without node replication. FGSA representation facilitates expected linear time conversion of programs from a control-flow graph, yields the same semantics as SSA and GSA by using fewer gating functions, eliminates irreducibility and provides additional information in the form of path expressions which can further simplify analysis and optimization algorithms. Our algorithms do not require the computation of *iterative dominance frontiers* and do not need a separate live-analysis to generate pruned forms of the graph.

Chapter 4

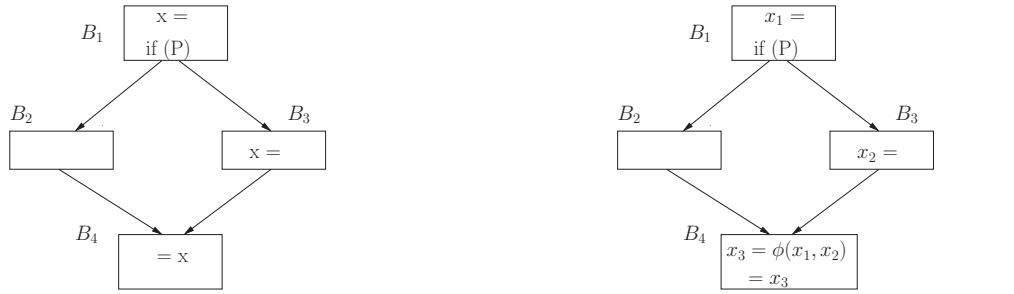
Live Variable Analysis on FGSA

Live variable analysis is important for coalescing and register allocation. Traditional liveness is conservative and imprecise. In a multi-assignment form, as shown in Figure 4.1(a), x is reported to be live at the exit of B_1. The fact is, B_1 reaches a use of x by following path $B_1B_2B_4$. However along path B_1B_3, x is dead. To be more accurate, x is partially live at B_1, which cannot be represented by traditional backward data flow analysis that is used to solve the liveness problem. Therefore, the conservative result that x is (definitely) live at B_1 is returned. In other words, although the traditional definition of liveness is complete, it is not sound (Hoflehner 2010): "Completeness means at any point of program where a variable is actually live, the liveness computation report it as live. Soundness means that at any point where the liveness computation reports a variable as live, it is actually live."

The problem partly originates from the fact that conventional live information is computed and reported based on blocks. Because the blocks contain computations that may change the liveness, we have to employ two sets for each block, LIVEIN and LIVEOUT to represent the liveness at the entry of the block and the exit of the block respectively. Consider data flow equation used to compute liveness. LIVEOUT of block n_i is taking the union of LIVEIN sets of all the succeeding blocks of n_i. Therefore when a variable is dead at LIVEIN of some succeeding block of n_i, it is regarded as live in n_i, although we know it is not definitely live. If liveness is computed and reported on edges, conventional

LIVEIN and LIVEOUT for blocks can be simplified to LIVE. Furthermore, given LIVE information for edges, the live information for blocks is easily computable. Unfortunately, adopting an edge based approach addresses only part of the problem.

In an SSA program, if we assume that ϕ nodes are true functions, their arguments must be treated as uses. This does not work in SSA and the ϕ-function arguments are considered to be live at the exit of the preceding nodes , but not at the entrance of the node containing the ϕ. For example, in Figure 4.1(b), the SSA form of the program in Figure 4.1(a), x_1 is live at the exit of B_2 and x_2 is live at the exit of B_3, but neither is live at the entry of B_4 while x_3 is considered to be live at the entry of B_4. This is not consistent with how liveness is interpreted in the rest of program but it is unavoidable with an imprecise definition of liveness.



(a) Multi-assignment liveness (b) SSA liveness

Figure 4.1: Traditional liveness

Given the impreciseness of the traditional concept of liveness and the new perspective brought in by FGSA, this chapter extends the concept of liveness to cover single assignment in general and SSA and FGSA in particular. As we demonstrate later, this extension makes the availability and consumption of values and the conditions under which this happens precise, yielding a computation of liveness that is both complete and sound.

4.1 Extended Liveness

Contrary to the simple *live* and *dead* attributes which define live ranges, we define a variable to be *definitely live*, *(definitely) dead*, *partially live* or *exclusively live*. We consider a

variable to be definitely live at a program point p if the value is available at p and used along all paths from p to the program exit; a variable is dead if the value is not used along any path from p to the program exit; a variable is partially live if the value is available at p and used along some (but not all) paths from p to the program exit. We will delay the definition of exclusive liveness for now as it is closely related to single assignment form and congruence classes.

In general, the partial liveness of a variable indicates that the computation of the variable is partially dead. There are two cases which make a variable partially live. Both Figure 4.2(a) and (b) show that x is partially live at the exit of B_1. In Figure 4.2(a) there are no uses of the variable along some path to the exit. In Figure 4.2(b) the value is killed before it reaches the use along some path to the exit.

(a) x is not used along some path (b) x is killed along some path

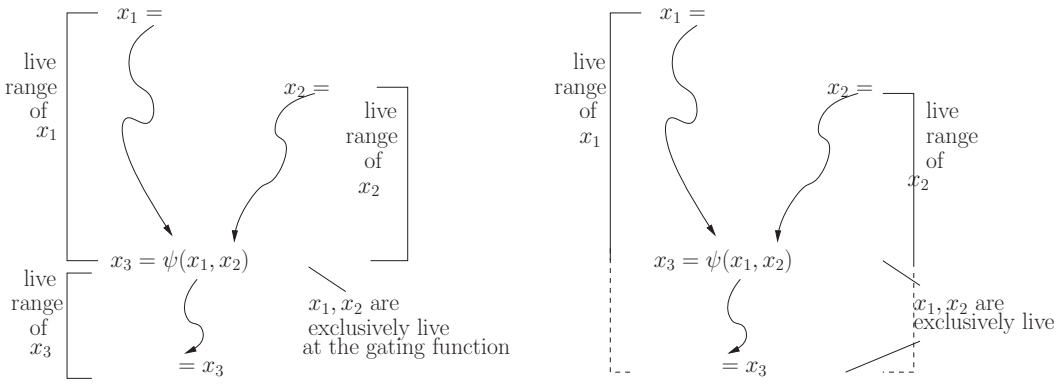
Figure 4.2: Partial liveness

Traditional liveness unifies live ranges formed by multiple definitions as long as these definitions are under the same variable name. Consider Figure 4.1(a) again. The liveness of x is computed by unifying live ranges formed by the definition in B_1 and the use in B_4 and the definition in B_2 and the use in B_4 into a single live range. In single assignment form, these two live ranges represent separate live ranges. When such live ranges share a set of uses, we consider each definition to be *exclusively live* at the uses.

Let's review SSA liveness under this extension. We pointed out at the beginning that given Figure 4.1(b), variable x_1 is live at exit of B_2 but not at the entry of B_4 while x_3 is live at the entry of B_4. With the extended liveness, we can interpret arguments of ϕ functions to be

actual uses which are conditionally used. As a result, the variables become exclusively live at the block where the ϕ node is placed. This is because only one of the argument values can be used at any time. The concept of exclusive liveness therefore addresses a significant problem regarding gating function arguments in single assignment form.

Because of their conditional nature, gating functions in single assignment form offer two alternative interpretations. One approach is to view that gating functions split the live ranges, which is the approach the traditional SSA liveness analysis adopts. In this approach, live ranges of gating function arguments end at the gating function and the live range of the gating function result starts. Another approach which hasn't been explored before is to view gating functions to be transparent. In this approach, uses of gating function result can be treated as the uses of the variables that reach the gating function, in essence extending the live ranges of gating functions arguments. Figure 4.3 compares the two approaches where ψ represents a general gating function.



(a) live range splitting gating function (b) live range transparent gating function

Figure 4.3: Two liveness approaches in general single assignment form

Compared to live range splitting gating functions, live range transparent gating function treatment has several important advantages. First of all, reaching definition information that is obscured by gating functions gains its traditional meaning. In Figure 4.3 x_1 and x_2 are (exclusively) live at the use of x_3 as they both reach this point. Note that extracting this information in an SSA graph would require traversal and analysis (i.e., closure) of several ϕ nodes. Second, live range of x_3 is still computable if desired. Third, if one considers the

register allocation problem, the live range reported by the transparent approach is much closer to the reality (precise if no copy operations are needed to eliminate the gating functions) whereas the live range splitting approach optimistically reports shorter live ranges. Furthermore, as we show later, this form is easily computable as part of single assignment form computation. It appears that the splitting approach is most appropriate if one desires to execute the gating functions. In this case, we can envision gating functions as instructions, and starting of a new live range is the appropriate semantics for this case. Similarly transparent gating functions are most appropriate for program analysis and optimizations, but they must provide a direct relationship between the definitions and uses if awkward traversals are to be avoided. Interestingly, SSA adopts live range splitting approach although the gating functions are not executable, and it can't readily adopt the transparent approach because of chaining of ϕ nodes. As we shortly discuss, FGSA can do both.

Let's apply the two approaches on FGSA respectively. Since FGSA gating functions are executable and the conditions of the gating are precisely specified through the predicate expressions, it is possible to view the gating functions as executable, conditional instructions. This view splits the live ranges at the gating functions. Live range of variables which participate in a CC end at the gating function when dataflow is traditional or start at the gating function when dataflow is future.

By applying the transparent gating function approach in FGSA, a particular definition can reach the uses represented by the CC. As previously stated, this view is most appropriate for code optimization and generation on traditional architectures, and it allows us to compute liveness before the gating functions are placed during FGSA construction.

Now consider again our running example from Chapter 3 (Figure 4.4). With the extended liveness, x_2 is definitely live at the exit of B_2 because it is used (through x_3 in B_3 and B_4) along all paths. x_1 is partially live at the exit of B_1 because it is not anticipated along B_1B_2 due to x_2 and is anticipated along B_1B_3. The congruence class of x_3 (CC_{x_3}) is formed and used at B_3 and B_4. Therefore, at the point where the CC is constructed, the definitions in

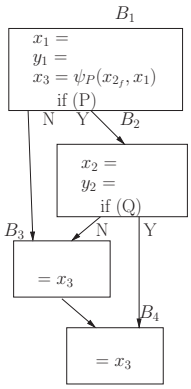


Figure 4.4: Running FGSA Example

the CC namely, x_1 and x_2 are exclusively live. This view permits us to associate liveness with congruence classes, a significant advantage of the representation.

4.2 Associating Liveness with Congruence Classes

The extended definition of liveness when combined with congruence classes permits simplification of many program analysis and optimization algorithms. Particularly inverse transformation from single assignment form may benefit from reporting the live ranges of each variable with respect to each CC the variable is a member of. In other words, we can compute a variable's total live range by unifying its live ranges with respect to each CC it participates in. Given a gated CC, any definition in the CC arrives at the uses of the CC only when its gating predicate expression is evaluated to be true. At any time, only a single gating predicate expression can become true, which guarantees that a single definition can be live and flow into the uses while other definitions either don't flow or are killed. As a result, we define an important concept, *anticipated region* of a definition:

Definition 15. *Given $CC_G =< CC, G, \psi_G >$, the Anticipated Region (AR) of $d_i \in CC.D$ is a set of program points P such that $\forall p \in P$, d_i is the sole anticipated value at p among the definitions in CC.D.*

The anticipated region depends on where the definition is placed. When the definition site moves or the definition is replaced by another definition which resides in a different

block, the anticipated region changes. However, the gating path predicate expression for the value shall not change and it defines the *maximal anticipated region* of the definition. For that purpose, we compute the *anticipated window* for each definition. Intuitively, the anticipated window of a variable represents the set of program points such that a variable may become anticipated at that point due a program transformation, such as copy folding even if the variable is not currently anticipated.

4.2.1 Computing The Anticipated Window and The Gating Region

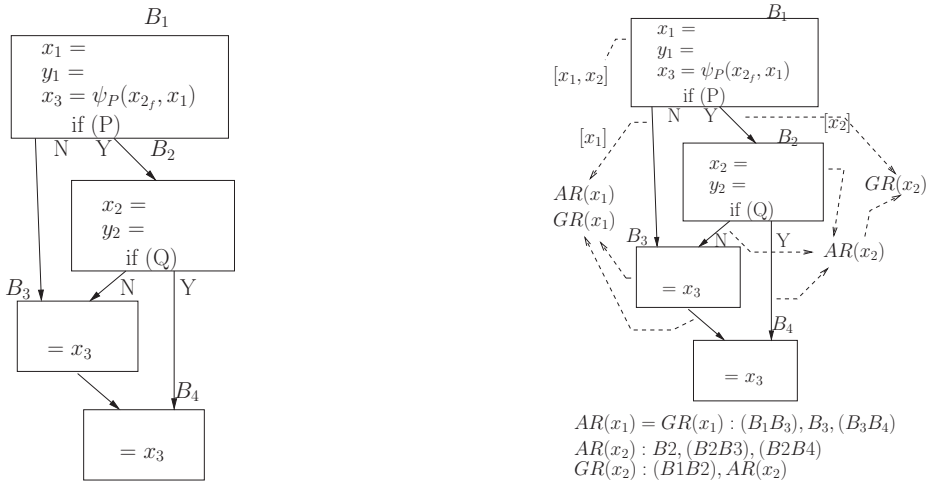
Anticipation window of a given variable can be computed by assuming that all the definitions are available at the LCDOM of all the definitions in a CC and propagating this information using path predicates associated with each variable. Therefore, we start at LCDOM to traverse the CFG. Each CFG edge or block will contain a set of definitions that are possibly anticipated at it. Clearly LCDOM contains all the definitions. The definition set then is propagated and classified at each branch. The taken edge of branch P contains the set of definitions where P appears as a term in their gating path predicate. Similarly, the not-taken edge contains the rest of the definitions. If P is irrelevant to the gating path predicates of the definitions coming to the branch, the whole set is passed to the first control independent node of P. The process ends when the edge or the block contains a single anticipated definition. If the CC is not fully anticipated at LCDOM, we need to mark the *dead ends* before the processing. When a program point contains a single definition, it starts the maximal anticipated region of the definition:

Definition 16. *Given $CC_G =< CC, G, \psi_G >$, the maximal anticipated region of $d_i \in CC.D$, or referred to as the gating region (GR) of d_i is a set of program points P such that $\forall p \in P$, either $p \in AR(d_i)$ w.r.t CC or p is in the anticipated window of d_i such that it contains only d_i.*

As we demonstrate shortly, anticipated region and gating region are crucial concepts in analyzing the interaction between various live ranges. In Figure 4.5 (a), x_1 and x_2 are

members of CC_{x_3} only. The result of computing the anticipated window is shown in Figure 4.5 (b). B_1 is the LCDOM which contains x_1 and x_2. Next we encounter branch P. Since x_1's gating path predicate contains $\neg P$, the not-taken edge (B_1B_3) gets x_1. Similarly, because x_2's gating path predicate contains P, the taken edge B_1B_2 gets x_2. Because both edges contain a single definition, the processing ends and (B_1B_3) marks the beginning of $GR(x_1)$ while (B_1B_2) marks the beginning of $GR(x_2)$. Based on this, we compute the anticipated region and gating region of x_1 which are the same while the gating region of x_2 includes every point in $AR(x_2)$ plus edge B_1B_2.

The anticipated region depends on where the definition is placed while the gating region depends on the corresponding gating predicate expression. Therefore, when optimization algorithms such as copy propagation are applied on the CC, the anticipated region of each definition may change but the gating region will not. The gating region is therefore a central concept as it is strongly tied to the semantics of the program. While it is possible to envision program changes that can modify the gating region through code motion and rewriting, in general, safety constraints in optimizations respect the semantics indicated by the gating region. As such, gating region concept precisely defines the range of program points a definition can be safely placed. For example, a definition that is inside the gating



(a) Running FGSA Example (b) Anticipated Region and Gating Region

Figure 4.5: Extended liveness on FGSA

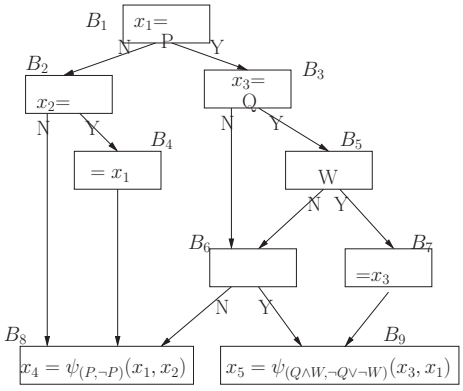
region of a CC will never be partially dead with respect to this CC. In other words, if the CC is executed, the value is used. Furthermore, the gating region of a variable with respect to a given CC classifies interferences into two main groups, namely those which occur outside the gating region and those which occur inside the gating region.

Consider Figure 4.6. There are two gated CCs: CC_{x_4} and CC_{x_5} and two non-gated CCs: CC_{x_1} and CC_{x_3}. It is easy to see that the program has undergone some optimizations and some definitions are not in their original places. However the gating predicate expressions which represent the semantics of the program remain. According to the gating predicate, the gating regions w.r.t the two gated CCs are listed in Figure 4.6(b). The gating regions for non-gated CCs are trivial.

4.2.2 Interference Under Extended Liveness

When two variables are *simultaneously live* at a given program point, we consider these two variables interfere. Under the extended liveness, this description still holds except that when two variables are exclusively live, they don't interfere. Understanding and analyzing interferences between variables in single assignment forms is key in many optimizations such as coalescing, inverse transformation and register allocation. With extended liveness and CCs, we can now classify the interferences into different types and look for a solver for each type.

1. D-to-D interference: Definition-to-definition interferences result when definitions that belong to the same CC are placed above their gating regions such that two or more definitions simultaneously flow through a program region towards their gating region. In Figure 4.6(a), the interference between x_1 and x_3 is a D-to-D interference. In order to solve it, we can place a copy of one definition or move one definition into its gating region depending on the approaches we employ. The points of placement must dominate its (exclusive) uses in the CC. For that purpose, we define *exits* of a gating region:



(a)

$GR(x_1)$ w.r.t CC_{x_4}:
$(B_1B_3), B_3, (B_3B_6), B_6, (B_6B_8)$
$GR(x_2)$ w.r.t CC_{x_4}:
$(B_1B_2), B_2, (B_2B_4), B_4, (B_2B_8), (B_4B_8)$
$GR(x_3)$ w.r.t CC_{x_5}:
$(B_5B_7), B_7, (B_7B_9)$
$GR(x_1)$ w.r.t CC_{x_5}:
$(B_3B_6), (B_5B_6), B_6, (B_6B_9)$

(b)

Figure 4.6: Gating region

Definition 17. *An exit of a gating region of $d_i \in CC_i.D$ is an edge B_iB_j such that d_i is partially or definitely live at block B_i and exclusively live at block B_j.*

There exists either a single point or multiple points which (collectively) dominate all the exits of the gating region. These points are the potential placement points for the new definition. If a single point exists, a single copy is enough to eliminate the interference. Otherwise, a copy is needed at each point. Placing copies at such points guarantees that the value of the original definition can flow to the CC (uses in the CC) along the original path predicate expression. In other words, the placement keeps the program semantics. Assuming we place the new definition beyond its gating region, the definition becomes no longer exclusive live at some B_j. Similarly, if we place the new definition at a point which doesn't dominate all the exits, the value flows into the CC along a different path predicate. In either case, the program semantics will change.

2. U-to-D interference type A: Given $d_1, d_2 \in CC_i.D$, d_1 has a use either in d_2's anticipated region if d_2's definition is placed in its gating region or d_1 has a use in d_2's gating region. In such a case, d_1 is live at d_2's definition point, which implies that d_1 and d_2 interfere. For example, in Figure 4.6(a), the use of x_1 at B_4 is in x_2's anticipated region. Because d_1 has an actual use, definition of d_1 and its uses form

a non-gated CC: CC_{d_1}. CC_{d_1} and CC_i both contain d_1 as one of their definitions. In order to solve this type of interference, we have two options. One option is to insert a single copy of d_1 given by $d_1' = d_1$ to isolate the two CCs (i.e., replace d_1 in CC_i by d_1'). The other option is to shrink d_2's anticipated region such that the region is below the use of d_1. In Figure 4.6(a) this would involve insertion of two copies of x_2 at the exits of its gating regions.

3. U-to-D interference type B: Given $d_1 \in CC_i.D$, d_1 has a use at the point where d_1 is exclusively live with respect to CC_i. Type B is similar to type A in the sense that CC_{d_1} and CC_i share d_1. Because the use is at a multiple value anticipated point, the only option to solve the interference is to isolate two CCs by inserting a single copy of d_1.

These types of interferences will manifest themselves in inverse transformation and register allocation. The different options we select to solve the interferences have different costs in terms of number of copy instructions inserted. In the next Chapter, we focus on inverse transformation and demonstrate the developed algorithms to reduce the number of copy insertions.

4.3 Computing and Associating Liveness with Congruence Classes

Given the definitions provided by extended liveness, we can now illustrate how the extended liveness can be further enhanced and computed to make it useful for optimization algorithms.

Definition 18. *When v_1 and v_2 are both live at a program point p such that v_1 is live at predicate P_1 and v_2 is live at predicate P_2 and P_1 and P_2 are disjoint (i.e., P_1 and P_2 cannot be simultaneously be true), then v_1 and v_2 are exclusively live w.r.t each other.*

By convention, variables interfere when they are simultaneously live at some program

point. This is not true for variables that are exclusively live. For a CC in FGSA, all the definitions that participate the CC are exclusively live at the gating function point because gating path predicates are always disjoint.
For FGSA, we define liveness based on CCs:

Definition 19. *$CC = \{D, U\}$ is live at a program point p if and only if:*
1) there exists a use $u_i \in CC.U$ and a path from p to u_i; AND
2) for each definition $d_i \in CC.D$, there exists a path from d_i to p such that d_i is not killed along the path.

No computations are performed on the edges, therefore conventional LIVEIN and LIVEOUT for blocks can be simplified to simply LIVE for edges. To further improve the accuracy, the live information can be predicated, i.e., we say *CC* is live on edge *e* on *P* when *CC* is partially live and only along the path presented by *P*. In FGSA construction, information is propagated along the edges together with the path predicates, and as a result live information can also be computed at the same time. In FGSA, a given use can only be a member of a particular CC eventually, although it may form a CC initially without definitions. In the following discussion, we use the phrase "use *u* is live at edge *e*" and the phrase "CC_u is live on *e*" which *u* participates in eventually interchangeably.
Consider local CC computation. If block B_i contains a non-empty upward exposed CC, it implies that the CC is live at e_i assuming e_i is a preceding edge of B_i. Then during the propagation, the CC (or definitions participate in the CC) is live along the edges that are consumed by T1/T2 right before the CC meets its definitions.
Globally T1/T2 propagates data flow bidirectionally. Therefore we can compute liveness on edges using availability and anticipation of variables. In the forward direction, we can collect definitions which are available along the edges. In the backward direction, we can collect CCs (represented by the uses) that are anticipated along the edges. Given this information, we consider a definition to be live on a given edge if the definition is available and at least one CC is anticipated along the same edge. When multiple definitions are live at on edge, they are exclusively live. Based on anticipation information, we can further tell

Algorithm 1
We define *Function f* such that :
$CC(u,w_i) = f(f(CC(u,v),CC(v)),CC(v,w_i))$

Function f:
INPUT: $CC(X)$, $CC(Y)$
OUTPUT: $CC_{up}(Z)$,$CC_{down}(Z)$

begin

(L1)if (DEF(X) or USE(X)) then
(L2) for each e in $Edge_{up}(Y)$
(L3) insert $CC_{down}(X)$ into AVAIL[e]
(L4)if (USE(Y)) then
(L5) for each e in $Edge_{down}(X)$
(L6) insert $CC_{up}(Y)$ into ANTIC[e]
(L7)if ($\Lambda \in CC_{down}(Y).D$
and $CC_{down}(Y).U \neq \emptyset$)
(L8) for each e in
$Edge_{down}(X) \cup Edge_{up}(Y)$
(L9) insert $CC_{down}(Y)$ into ANTIC[e]
(L10)if ($\neg DEF(X)$) then
(L11) $Edge_{up}(Z) = Edge_{down}(X)$
$\cup Edge_{up}(Y)$
(L12)else
(L13) $Edge_{up}(Z) = Edge_{up}(X)$
(L14)if (DEF(Y) and $\Lambda \notin CC_{down}(Y).D$) then
(L15) $Edge_{down}(Z) = Edge_{down}(Y)$
(L16)else
(L17) $Edge_{down}(Z) = Edge_{down}(X)$
$\cup Edge_{down}(Y)$

//X is case 1 or 2
(1)if ($\neg DEF(X)$) then
(2) $CC_{up}(Z).U = CC_{up}(X).U \cup CC_{up}(Y).U$
(3) add $CC_{up}(Z)$ into globalCC
(4) $CC_{down}(Z) =$
$DEF(Y)?CC_{down}(Y) : CC_{up}(Z)$
(5) remove $CC_{up}(X), CC_{up}(Y)$
from globalCC
//X is case 3 or 4
(6)else
(7) $CC_{down}(X).U = CC_{down}(X).U$
$\cup CC_{up}(Y).U$
(8) $CC_{up}(Z) = CC_{up}(X)$
(9) $CC_{down}(Z) = DEF(Y)?$
$CC_{down}(Y) : CC_{down}(X)$
(10) if (DEF(Y) and $\Lambda \in CC_{down}(Z).D$) then
(11) $CC_{down}(Z).D = CC_{down}(Z).D$
$\cup CC_{down}(X).D$
(12) remove $CC_{up}(Y)$ from globalCC

// compute R sets
//(R1) add $S(Y)$ to reduced reachable
//(R1) set of each node in $S(X)$
//(R2) $S(Z) = S(X) \cup S(Y)$
end

Figure 4.7: Algorithm 1 with live analysis

respect to which CC a definition is live.

Similar to associating CC_{up} and CC_{down} with each edge (node), we additionally associate $Edge_{up}$ and $Edge_{down}$ with each edge (node) during the propagation. $Edge_{up}$ represents the list of edges that are transparent to the uses in the corresponding CC_{up}. $Edge_{down}$ represents the list of edges that are reachable from the definitions in CC_{down}. We also keep two global vectors ANTIC[ ] and AVAIL[ ], which are indexed by edges and updated during the propagation. $ANTIC[e_i]$ contains CCs that are anticipated along edge e_i while $AVAIL[e_i]$ contains CCs (definitions in the CCs) that are available along edge e_i.

We now modify Algorithm 1 and 2 to propagate $Edge_{up}$ and $Edge_{down}$ and to update CC

live information during FGSA construction. In Algorithm 4.7, line L1-L17 computes live information for CCs. In the modified algorithm, when CC(X) contains downward exposed definitions or uses, edges in $Edge_{up}(Y)$ will get the same values as CC(X). For those edges, we add $CC_{down}(X)$ into their AVAIL[] set (line L1-L3). When CC(Y) contains upward exposed uses, edges in $Edge_{down}(X)$ all reach the uses. For those edges, we add $CC_{up}(Y)$ into their ANTIC[] set (line L4-L6). When $CC_{down}(Y).D$ contains Λ, it implies values are partial transparent through the region. Therefore when $CC_{down}(Y).U$ contains uses, these uses are anticipated along edges in $Edge_{up}(X) \cup Edge_{up}(Y)$ (line L7-L9). The resulting $Edge(Z)$ is also updated during the process (line L10-L17). When edge merging happens (Algorithm 4.8), $Edge_{up}$ and $Edge_{down}$ are merging respectively.

During T1, when loop lp contains definitions which reach the back-edge (i.e., $CC_{down}(lp).D$ is not empty) and lp contains upward exposed uses(i.e., $CC_{up}(lp).U$ is not empty), edges in $Edge_{up}(lp) \cup Edge_{down}(lp)$ represent the edges to which the uses are either reduced reachable or reachable through the back-edge. Therefore, we add $CC_{down}(lp)$ to AVAIL[] and ANTIC[] entries that are corresponding to these edges.

4.4 Conclusion

We have extended the concept of liveness on FGSA with respect to predicates and congruence classes. By doing so, we are able to classify interferences among variables into categories and look for solutions for each category, which can be utilized by inverse transformation and register allocation algorithms. Finally in this chapter, we have presented algorithms to combine FGSA construction with liveness computation.

Algorithm 2
INPUT: $CC_{up}(e_i), CC_{down}(e_i), CC_{up}(e_j), CC_{down}(e_j)$
OUTPUT: CC_{up}, CC_{down}
begin

(1)$CC_{up}.U = CC_{up}(e_i).U \cup CC_{up}(e_j).U$
(2)add CC_{up} in the globalCC
(3)if ($DEF(e_i)$ or $DEF(e_j)$) then
(4) $D = CC_{down}(e_i).D \cup CC_{down}(e_j).D$
// $CC(e_i)$ or $CC(e_j)$ does not contain definitions
(5) if($\neg DEF(e_i)$ or $\neg DEF(e_j)$) then
(6) $D = D \cup \{\Lambda\}$
(7) if (CC_D not exist in globalCC)
(8) $CC_{new} = \{D, \emptyset\}$
(9) add CC_{new} to globalCC
(10) $CC_{down} = CC_{new}$
(11) else
(12) $CC_{down} = CC_D$
(13)else
(14) $CC_{down} = CC_{up}$
(15) remove $CC_{up}(e_i)$ and $CC_{up}(e_j)$ from globalCC

(L1) $Edge_{up} = Edge_{up}(e_i) \cup Edge_{up}(e_j)$
(L2) $Edge_{down} = Edge_{down}(e_i) \cup Edge_{down}(e_j)$

// compute R sets
//(R) $S(u,v) = S(e_i) \cup S(e_j)$
end

Figure 4.8: Algorithm 2 with live analysis

Chapter 5

Inverse Transformation From FGSA

We have introduced FGSA, algorithms to transform a given program into this representation and we have presented an extended definition of liveness for single-assignment forms in general and congruence classes in particular. We now illustrate that a congruence class based single-assignment form coupled with the extended definition of liveness also leads to an efficient inverse translation algorithm. This algorithm presents a fresh perspective and a promise of an optimal solution to the long-standing problem of inverse translation from single-assignment forms.

Key to our approach is the classification of interferences under a well-formed taxonomy and developing solutions for each element of the taxonomy. This taxonomy is enabled by the extended liveness definition as well as the use of congruence class concept in specifying the interferences.

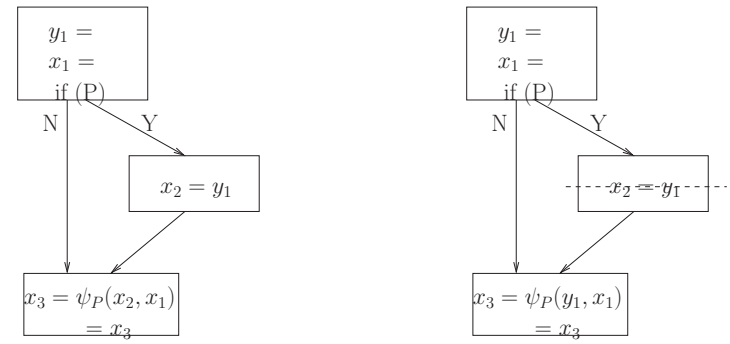
In the rest of this chapter, first in Section 5.1 we give a simple algorithm to translate a given FGSA program back into multi-assignment form. This algorithm is analogous to an algorithm developed for SSA by Boissinot et al. (Boissinot et al. 2009). Our purpose in providing the algorithm is to show that using a single gating function per congruence class and placing this gating function at a point in the program at times involving future dependencies will not affect the applicability of existing inverse transformation techniques on FGSA. We follow this simple algorithm with a taxonomy of interferences in Section 5.2

and present key notions of *path-separability* and *isolation*. Also In this section, we demonstrate that inverse translation is the process of ensuring that all congruence classes in the representation are path-separable and isolated. In Section 5.3, we discuss the problem of minimizing copy instructions inserted for this purpose and develop solutions for each case arising from our taxonomy. In Section 5.4, we introduce an important concept, common use form which further extends the concept of congruence class to combine two or more congruence class into a single congruence class. Finally, in Section 5.5 we summarize our approach and conclude the chapter.

5.1 Simple Inverse Translation from FGSA

Inverse transformation from FGSA requires elimination of gating functions on an architecture that is not designed to execute gating functions with future values. Similar to CSSA and TSSA defined in Sreedhar et al.'s (Sreedhar et al. 1999), we define C-FGSA (conventional FGSA) as the form computed using the set of algorithms in Chapter 3. Inverse transformation from C-FGSA is straight-forward. All the variable occurrences in a gating function are replaced by a representative variable, and later the gating function can be removed. Given a program in C-FGSA form, optimizations may transform it into a state in which this simple procedure will not result in a semantically correct outcome because of interferences. Therefore, given a gating function $v = \psi_G(D)$, the algorithm for translating out of FGSA consists of three steps: (1) For each definition d_i, insert $g_i : v = d_i$ at the gating function, where g_i is d_i's gating path predicate; (2) Eliminate the gating function; (3) Move copy instruction $g_i : v = d_i$ along the control flow such that the predication is eliminated and d_i is not in future form. Critical edges are split in order to put the copy instructions in proper control flow positions. This algorithm follows a similar approach to Boissinot et al. (Boissinot et al. 2009). The first step inserts all the copies necessary and the redundant copies are assumed to be eliminated by applying coalescing algorithms developed specifically for this purpose.

Consider the lost copy problem of SSA inverse transformation in Figure 5.1(a) within the

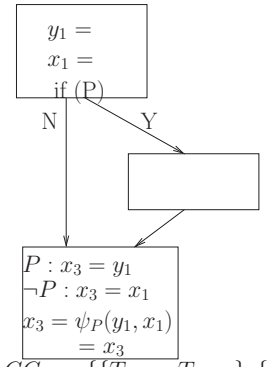


$CC_{x_3} = \{\{T : x_1, P : x_2\}, \{x_3\}\}$

(a) Lost copy problem

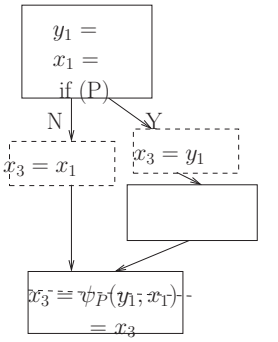
$CC_{x_3} = \{\{T : x_1, T : y_1\}, \{x_3\}\}$

(b) Copy folding applied



$CC_{x_3} = \{\{T : x_1, T : y_1\}, \{x_3\}\}$

(c)Inverse transformation: step 1



$CC_{x_3} = \{\{T : x_1, T : y_1\}, \{x_3\}\}$

(d)Inverser transformation: step 2&3

Figure 5.1: Translation from FGSA

FGSA framework. After the copy propagation, CC_{x_3} and the gating function are updated as shown in Figure 5.1(b). With the updated graph, both x_1 and y_1 are above their gating regions which results in a D-to-D interference and the D-to-D interference is the reason why the graph is in T-FGSA form. Note that the detection of the interferences is not necessary for the correctness of the above algorithm.

When we apply the steps of the algorithm, two copy instructions are inserted at the gating function (Figure 5.1(c)), and are later moved to a proper point of the program to eliminate the predication (Figure 5.1(d)). Note that the final points of insertion for these variables are precisely the beginning of their gating regions. Coalescing algorithms will remove $x_3 = x_1$, however the other one will not be removed.

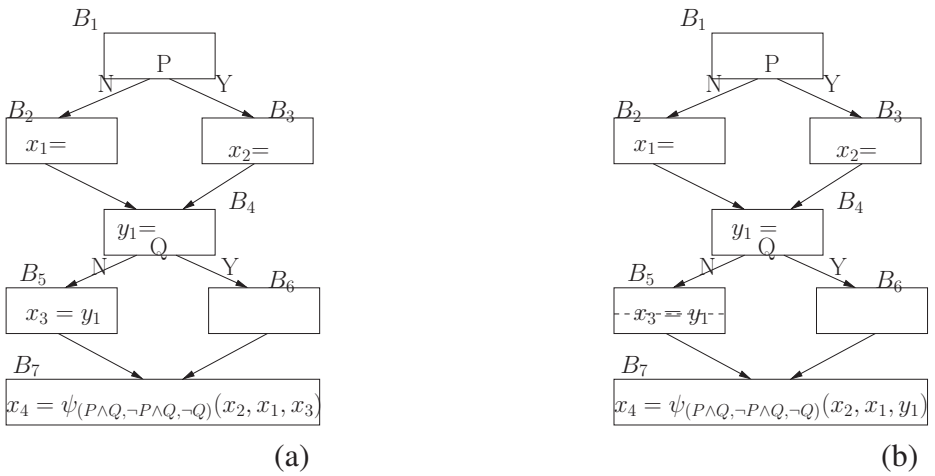


Figure 5.2: Path separability example

5.2 Path Separability, C-FGSA, T-FGSA and Isolation

Before we can develop efficient inverse transformation algorithms which are provably optimal, we need to understand the implication of various optimizations on a given single assignment form. Although it is intuitively clear that one cannot simply drop gating functions and rename variables to a single representative variable to go back to the multi-assignment form after most optimizations, existing approaches are limited to handling *interferences* created due to optimizations.

In order to gain further insight, consider the program shown in Figure 5.2(a). Before any optimizations, this program clearly is in a form where one an simply drop the gating function and rename all instances x_1, x_2, x_3 to x to go into the multi-assignment form. It is important to observe that the gating predicates of the gating function were easily computed using the presented algorithms and they are still computable after the variables have been renamed into the unique definition form using subscripts. All that's necessary is to assume that a new definition x_i *kills* any x_j reaching to its definition point. In other words, the graph is still path separable based on the definition of path separability (Definition 4).

Consider now the problem of recalculating the gating path predicates on Figure 5.2(b). Assuming y_1 kills x_2 and x_1 by the same definition, it is clear that predicates computed as

such would be wrong and hence would not represent the semantics of the program correctly. Since the gating path predicates are invariant, we can test for path separability using a new definition given below:

Definition 20. *Given a* $CC = \{D, U\}$ *and its corresponding gating function* $\psi_G(D)$*, definitions in CC.D are path separable if and only if* $\forall d_i \in CC.D$*,* d_i *is the last definition from LCDOM(CC.D) to uses in CC.U along path represented by* g_i*, where* g_i *is the gating path predicate for* d_i*.*

In fact, optimizations such as code motion and copy propagation destroy either the path separability of definitions or destroy the *isolation* property of congruence classes, or, both. While path separability deals with interferences within definitions, isolation deals with interferences between uses of one definition and other definitions. We define isolation as such:

Definition 21. *Given* $CC = \{d_v = \psi_G\{d_1, d_2, \cdots, d_n\}, U\}$ *where* d_v *is the destination of the gating function, the* related use set *is defined as:*

$\hat{U} = CC.U \cup \bigcup_{d_i \in CC.D} CC_{d_i}.U$*, where* CC_{d_i} *is the congruence class consisting of the single definition* d_i *and its uses.*

Correspondingly, the related definition set *is defined as:*

$\hat{D} = CC.D \cup \{d_v\}$.

CC is isolated if and only if $\forall u \in \hat{U}$ *and its definition* x^u*, condition (I) and one of the conditions (II)(a) and (II)(b) must hold:*

(I) when x^u *is not* d_v*,* x^u *reaches u without being intervened by definitions from* $\hat{D} / \{x^u\}$.

(II)(a) when x^u *is* d_v *and the gating function contains no future values,* x^u *reaches u without being intervened by definitions from* $\hat{D} / \{x^u\}$.

(II)(b) when x^u *is* d_v *and the gating function contains future values, let the set of future values be* D_f*,* $\forall d_{i_f} \in D_f$*,* d_{i_f} *reaches u.*

Theorem 5.2.1. *If a gated CC is path separable and isolated, the* C-FGSA Property *holds for the CC.*

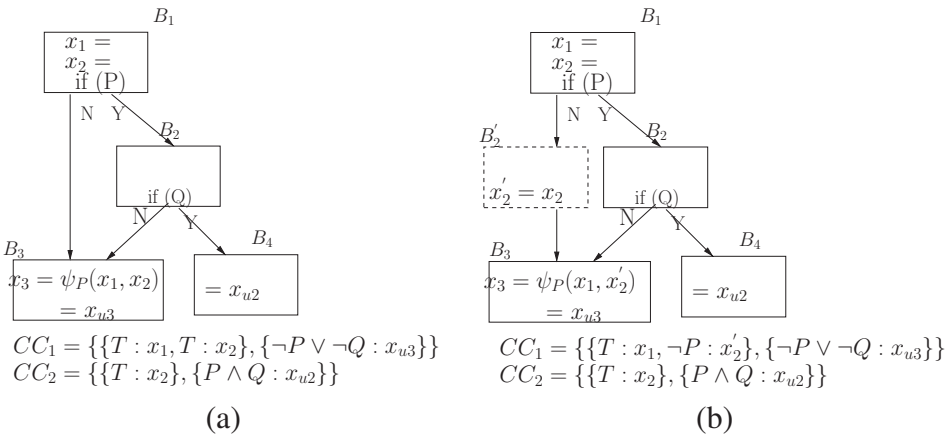


Figure 5.3: Isolation and path-separability

Proof. Let $CC_i = \{d_v = \psi_G(D), U\}$ be a path separable and isolated gated CC. Because CC_i is path separable, $\forall d_i \in CC_i.D$ it is not killed by any definition along g_i. Further, d_v is not killed by any definition when its value flows into uses. Therefore renaming d_i or d_v into a representative name will not affect its value flowing into any use of $CC_i.U$. d_v cannot appear in any other CCs so renaming d_v cannot affect other CCs. We can classify CC_i into two cases:

† If CC_i shares no definition with other CCs, renaming of CC_i will not affect the semantics of other CCs. Therefore CC_i has C-FGSA property.

† CC_i shares definition(s) with other CCs. Let $d_i \in CC_i.D \cap CC_j.D$. Because of condition (I) in Definition 21, the value that flows between d_i and any use in $CC_j.U$ is not intervened. Along the renaming d_i during translating CC_i, uses in $CC_j.U$ can be renamed using the same representative. CC_j must be a CC consisting of single definition d_i, so renaming of CC_j cannot affect other CCs.

□

To perform inverse transformation from FGSA, we can check each CC with respect to path separability and isolation criteria. If both are satisfied, eliminating the corresponding gating function is trivial. Otherwise, copy insertion and code motion are applied. Consider Figure 5.3. This program is in a T-FGSA form. Since path predicate expressions for both

x_1 and x_2 are T, CC_1 is not path-separable. Furthermore, because of CC_2, gated CC_1 is not isolated with respect to x_2. In order to make CC_1 path-separable, we can insert either a copy of x_1 on path P or a copy of x_2 on path $\neg P$ according to the gating function $\psi_{(P,\neg P)}(x_1,x_2)$. Considering that CC_1 may need to be isolated from CC_2 w.r.t x_2, the copy of x_2 has priority over x_1. Once $x_2^{'} = x_2$ is inserted as shown in Figure 5.3(b), CC_1 obtains C-FGSA property. Next we present the algorithm to check path separability for CCs in T-FGSA. We'll discuss how to minimize copy insertion in Section 5.3.

5.2.1 Checking for Path Separability

Checking for path separability involves the concept of the gating region for a congruence class. Intuitively, Definition 20 can be understood as a variable being the sole anticipated value in its gating region. However, when the congruence class in question is of the *nested* form, we need to analyze each sub-congruence class and locate the gating region appropriately.

In order to understand the problem better, consider Figure 5.2 again. Although there is a single congruence class represented by the gating function $\psi_{(P\wedge Q,P\wedge\neg Q,\neg P)}(x_2,x_1,x_3)$, this congruence class consists of a nested congruence class $\psi_{(P,\neg P)}(x_2,x_1)$ as an embedded component. In other words, x_1 and x_2 are exclusively live in block B_4, and the gating function $\psi_{(Q,\neg Q)}(\psi_{(P,\neg P)}(x_2,x_1),x_3)$ selects either this value or x_3 depending on the predicate Q. From this perspective, the beginning of the gating region for x_1 is edge (B_1B_2), for x_2 it is (B_1B_3), for $\{x_1,x_2\}$ (B_4B_6) and finally for x_3, it is edge (B_4B_5). Our algorithm which detects path separability involves insertion of a *holder block* at the beginning of the gating region for each and recursively checking for path separability, in essence decomposing the gating path predicate expressions. Given $CC_i = \{v = \psi_{(p_1,p_2,\cdots)}(d_1,d_2,\cdots),U\}$, the algorithm consists of several steps:

1. For each definition, locate or place (if no such block exists) a block B_i, referred to as *holder block* based on its gating path predicate, marking the beginning of a gating region. Note that the gating path for d_i is the path from LCDOM(CC_i.D) through B_i

```
(1)for d_i ∈ CC.D do
(2)     let B_i be d_i's holder block
(3)     if def(d_i) reaches B_i then
(4)       if AVAIL(B_i) ≠ {d_i} then
(5)         d_i is not path separable
(6)     for d_j ∈ CC.D and i ≠ j do
(7)       if B_i reaches def(d_j) then
(8)         d_i is not path separable w.r.t d_j
```

Figure 5.4: Basic algorithm for checking path separability

to any use of CC_i.

2. Divide CC_i.D into subsets $S_1, \cdots, S_m$ based on distinctive holder blocks such that $S_i = \{d_{i1}, d_{i2}, \cdots\}$ contains all the definitions that have the same holder B_i. This results from the fact that all the definitions in S_i share the common suffix of their gating path predicates, resulting in some variables being exclusively live.

3. Apply basic algorithm on $S_1, \cdots, S_m$ to check path separability among subsets.

4. For any S_i that contains more than one definition, cut the common suffix of the gating path predicates, repeat the algorithm to check path separability within the subset.

Figure 5.4 presents the basic algorithm that checks path separability for a set of definitions in a CC with distinctive holder blocks. For efficiency, this algorithm precomputes reduced reachability (Boissinot et al. 2008). Given a directed flow graph $G =< N, E >$ with node set N and edge set E, the reduced graph $G' =< N, E' >$ is the subgraph of G such that it contains no back-edges. The reachability on G' is referred to as reduced reachability. Also reaching definition is computed on the subgraph involving the CC. $def(d_i)$ represents the node where d_i is defined. In the algorithm, line (3)-(5) checks whether d_i is the only definition in the reaching definition set of B_i if it is defined before B_i. Line (6)-(8) checks whether d_i is the only definition on all paths from B_i to any use in CC.

Theorem 5.2.2. *Algorithm in Figure 5.4 correctly computes path separability.*

Proof. On path from LCDOM(CC.D) to B_i, line (3)-(5) guarantees d_i is the only definition that reaches B_i without being killed. Suppose there exists $d_j \in AVAIL(B_i)$ where $d_j \in CC.D$ and $i \neq j$, then upon arriving at B_i, d_i and d_j are not distinguishable using path predicate of d_i.

Line (6)-(8) guarantees that d_i is not killed by other definitions, and it does not kill other definitions either. There are two cases:

1. $d_i \in AVAIL(B_i)$. According to line (7), d_j must kill d_i along path $B_i \rightarrow def(d_j)$. So the result that d_i and d_j are not path separable holds.

2. $d_i \notin AVAIL(B_i)$. Assume d_i is path separable. Then along all the path from B_i to any use of CC, d_i must flow, which means d_i must be defined along all the path from B_i on. In other words, $def(d_i)$ post-dominates B_i. Since B_i reaches $def(d_j)$, the fact that either d_j kills d_i along the path $def(d_i) \rightarrow def(d_j)$ or $def(d_i)$ post-dominates $def(d_j)$ holds. In either case, d_i and d_j are not path separable.

□

Given $CC_i = \{v = \psi_{(p_1,p_2,\cdots)}(d_1,d_2,\cdots),U\}$,

† case1: d_1 and d_2 are not path separable, CC_i is isolated with respect to d_1 and d_2, we can insert either a copy of d_1 on p_1 or d_2 on p_2;

† case2: d_1 and d_2 are not path separable, CC_i is not isolated with respect to d_1 only, we can insert a copy of d_1 on p_1. If CC_i is not isolated with respect to d_2 only, a copy of d_2 can be inserted instead.

† case3: d_1 and d_2 are path separable, CC_i is not isolated with respect to either d_1 or d_2 or both. In Section 5.3, we discuss the solutions to handle such cases.

5.3 Minimizing Copies

As we have discussed in Chapter 4, interferences among variables can broadly be classified into three categories, namely, D-to-D, U-to-D type A and type B. When definitions are

shared among congruence classes, whether these classes are gated or not, the data-flow aspects of the involved congruence classes must be compatible, since the inverse translation will force all the CCs to use the same name. While some of the data-flow incompatibilities can be eliminated by inserting copies to divert the data flow in gating regions, there are incompatibilities (such as U-to-D type B interferences) which require total isolation of the involved congruence classes. As a result, the optimal solution to a given set of interferences has to take into account how each type of interference manifests itself and whether the interference is localized to a particular CC or multiple CCs are involved in the solution of the problem.

Although programs which are in C-FGSA form are isolated by definition, this definition of isolation is difficult to test and use. Furthermore, this definition mixes two distinct cases of isolation together; programs in which the target CC is *strictly isolated* (i.e., share no definitions with other CCs) and those which share definition(s) but shared variables do not interfere. When we consider the interference testing based on the extended liveness, it becomes clear that the separation of the two cases enables a divide-and-conquer solution in which we first develop algorithms specifically for strictly isolated CCs and then extend these solutions to the *global problem* of solving interferences optimally when CCs share their definitions. It should be clear that a CC that is isolated but not strictly isolated should lead to a solution in which no copy instructions are placed by the global algorithm. We now define strict isolation formally:

Definition 22. *A $CC_i = \{\{d_1, d_2, \cdots, d_n\}, U\}$ is strictly isolated w.r.t d_i if and only if there does not exist $CC_j, i \neq j$ such that $d_i \in CC_j.D$. CC_i is strictly isolated if and only if $\forall d_i \in CC_i.D, CC_i$ is isolated w.r.t d_i.*

Based on the concept of strict isolation, we divide the problem of inverse transformation into three subproblems:

1. Eliminating interferences when a CC is strictly isolated;

2. Eliminating interferences when a gated CC shares definitions with one or more

non-gated CCs;

3. Eliminating interferences when a gated CC shares definitions with one or more gated CCs.

Note that insertion of a copy instruction targeting a particular variable can isolate a given CC with respect to others. As a result, we can rely on isolation to convert case (2) and (3) to a problem of case (1), although the process may insert more copies than necessary. Therefore, elimination of interferences when a CC is strictly isolated is a key algorithm which is used whenever a particular problem is reduced to case (1). Similarly, most CCs share variables with non-gated CCs and the solution of this case (2) is another key component of a global optimization solution.

In the following subsections, we present a provably optimal copy placement algorithm in terms of the number of copy instructions placed for case (1) and case (2) listed above. We further provide an upper bound for a near-optimal algorithms for case (3).

5.3.1 Handling Interferences for an Isolated CC

A strictly isolated CC is C-FGSA if it is path-separable. Otherwise, inserting copies of definitions to their gating regions will make it path-separable. Any non-path-separability between two definitions can be solved by inserting copies for either one of the definitions at its gating region. Note that the copy placement must dominate all the exits of the gating region and when there are multiple possible placements, the one with the minimum number of copies is selected. We refer to this placement as *D-placement* in the gating region. As a result, we can map copy minimization to achieve path-separability into a weighted vertex cover problem (selecting a subset of vertexes to cover all the edges with minimal cost). For the CC in question, we construct Non-Path-Separability Graph (NPSG) which is a special interference graph. Vertexes in an NPSG are definitions in a CC and there is an edge between two vertexes if the definitions are non-path-separable w.r.t each other. There is a cost associated with each vertex, which represents the number of copies that are

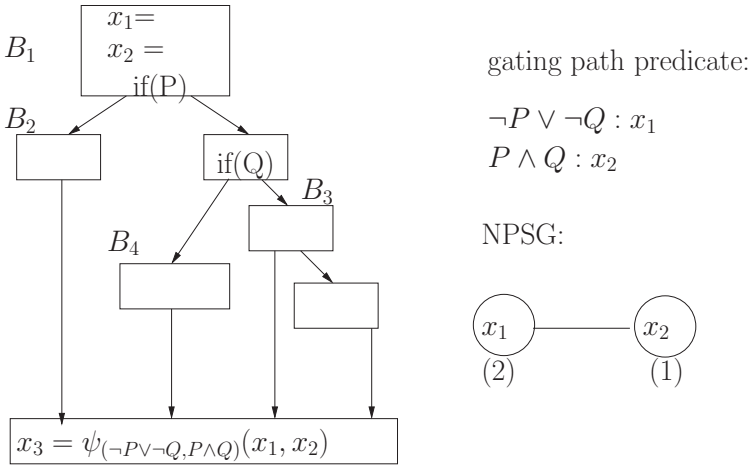


Figure 5.5: A non-path-separable CC and its NPSG

necessary to be inserted to cover the corresponding gating path. To find the solution with the minimal number of copies to achieve path separability for a CC is to find the solution to the optimal weighted vertex cover on the NPSG. Figure 5.5 shows a strictly isolated and non-path-separable CC and its NPSG. x_1 and x_2 are not path separable and their gating path predicates are shown at the up right corner of the figure. For x_1, no single basic block in the CFG can represent the starting point of its gating region but B_2 and B_4 together can. Therefore we need two copy instructions to be inserted at x_1's gating region. Similarly, we compute the cost for x_2. Now we can construct the NPSG as it is shown at the down right corner of the figure. It is easy to see, in this example, picking x_2 is the optimal solution. By solving the vertex cover on the NPSG and inserting one copy for x_2, we make the CC path separable.

Although the general minimal vertex cover problem is NP-complete, it has polynomial time optimal solution on *chordal graphs* (Gavril 1972). Sebastian (Hack 2005, 2007) proves that the interference graphs on SSA is chordal, based on the fact that when two variables x_1 and x_2 interfere, they are both live a program point l, definition sites of x_1 and x_2 both dominate l and thus either x_1 dominates x_2 or x_2 dominates x_1. The same dominance relationship can be found in FGSA when two variables are non path separable. Following the same proof mechanism used by Sebastian, the proof is straight-forward that the NPSG is also chordal.

Therefore, minimizing the copies for inverse transformation of CC_i is achievable using a polynomial time algorithm. Since such a CC has no communication with other CCs, the local optimal solution is also the globally optimal solution.

5.3.2 Handling Non-Gated to Gated CC Interferences

A non-gated CC has a single definition and it does not require specific handling. Given d_1 and d_2 in the same CC_i, d_1 and d_2 have U-to-D interfere if d_1 has a use in d_2's region. A U-to-D interference implies that d_1 is placed above its gating region and the use of d_1 is placed in d_2's gating region or exclusive live region. There are two cases:

† U-to-D type A: The use of d_1 is in d_2's gating region. For this type, we either place a copy of d_1 to isolate the two CCs or shrink d_2's anticipated region. Note that these operations are with different costs. The isolation operation cost is always one. The shrinking operation places copies to avoid the use and dominate all the exits. Its cost depends on where the uses are. Shrinking operation may have advantage over isolation operation in terms of minimizing number of copies when multiple definitions have U-to-D interference with d_2.

† U-to-D type B: The use of d_1 is in d_2's exclusive live region. For this type, we have no choice, but isolate d_1.

Figure 5.6 demonstrates uses of x_1 in different regions of x_2. The use at block B_4 is U-to-D type B interference because x_1 and x_2 are exclusively live here. The uses at block B_1 and B_3 cause U-to-D type A interference. However, the costs of operation to solve them are different. For the use at block B_1, a placement of a single copy at block B_2 is sufficient. For the use at block B_3, two copies of x_2 are necessary for the placement to be placed below the use and dominate all the exits.

When interferences caused by non-path-separability are combined with those caused by uses in other definitions' regions, optimal copy insertion must take both causes into account. Our method to solve the combined interferences is based on the observation that

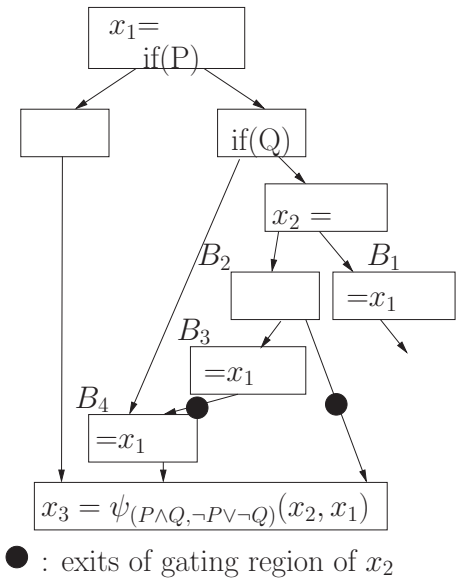


Figure 5.6: Interferences of uses in different regions

when k variables have D-to-D interferences with each other, there exist $k-1$ number of ways to solve them. Therefore, we enumerate all possible D-to-D solvers and optimally solve copy insertion problem on each one of them. If copy insertion problem without D-to-D interference is polynomial solvable, the general problem is too. We'll show that copy insertion problem without D-to-D has polynomial time solution in Section 5.3.4.

5.3.3 Handling Gated to Gated CC Interferences

In this category, the local optimal solution may not be the global one. When two CCs share definitions, they can be separated by inserting copies for the shared definitions and then be treated respectively as one of the above two cases. An alternative way is to merge the two CCs into one if possible. We discuss this case in Section 5.4.

5.3.4 Representation of the Problem

We now concentrate on the case where CC_i shares definitions with non-gated CCs . We know from the analysis, inferences caused by type B must be solved with copies of involved

variables while cases of type A have choices. We focus on minimizing copy insertion for solving U-to-D Type A interferences within a gated CC. We introduce a directed weighted interference graph (WIG) where each node represents a definition involved and each edge represents a interference which is caused by a use of d_1 taking place in d_2's region. Because d_1 has a use in d_2's region and definition site of d_1 dominates its use sites, definition site of d_1 dominates definition site of d_2, annotated as $d_1 \prec d_2$. Therefore, the direction of the edge represents the partial order of dominance of the two nodes involved. On the WIG, we have two types of operations, isolation operation g and shrinking operation eg. Given any node i, $g(i)$ represents inserting a copy of i , which covers all the outgoing edges associated with node i because the two CCs are isolated. Cost of g(i) is always one. The shrinking operation $eg(i)_j$ which is related to edge $\widehat{ji}$ represents inserting copies of i as a placement in its gating region such that it handles uses of j in i's region. Because such a placement may also solve uses other than j, $eg(i)_j$ covers some incoming edges including edge $\widehat{ji}$. Meanwhile, the placement in the gating region of i serves as an isolation and thus all the outgoing edges associating with node i are covered. The cost of $eg(i)_j$ is the number of copies that should be inserted for that placement to work. Minimizing copy insertion is equivalent to covering all the edges in WIG with operations of minimal cost.

Lemma 5.3.1. *WIG is acylic.*

Proof. Assume there exists a cycle $i_1 i_2 ... i_n i_1$. Because the edge direction represents dominance relationship, $i_1 \prec i_2 ... \prec i_n \prec i_1$ which implies $i_1 \prec i_n$ and $i_n \prec i_1$ both hold. Consequently the assumption cannot hold. □

Lemma 5.3.2. *Given $i_1 \preceq i_2 \preceq i_3$ and $\widehat{i_1, i_3}$ is an edge in WIG, then i_1 also interferes with i_2.*

Proof. The edge $\widehat{i_1, i_3}$ implies i_1 has a use that is live at i_3's definition site. Because $i_2 \preceq i_3$, that use of i_1 is also live at i_2's definition site. Therefore, i_1 and i_2 interfere. □

The interference between i_1 and i_2 is a U-to-D type A interference. The use of i_1 is exposed to i_2's gating region once we shrink i_3's region. We can incorporate such interferences into

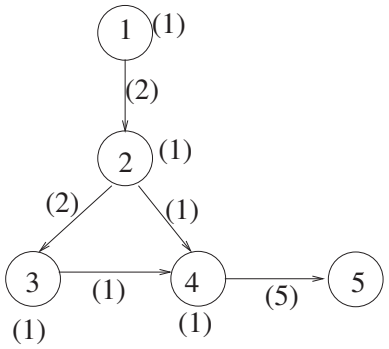


Figure 5.7: Weighted Interference Graph

WIG by introducing a directed edge from i_1 to i_2.

Figure 5.7 shows an example of WIG where numbers in the parenthesis are costs of gating and extended gating operations.

Because WIG is a DAG, there exists a topological order to traverse the graph, which is the order of dominance. We can use dynamic programming to solve the problem because the problem has overlapped sub-problems and optimal substructure. Consider WIG in Figure 5.7. Each node is processed based on the dominance relationship starting from node 1. Node 1 can take g(1) or blank operation b(1) which takes no operation on the node so that each of its outgoing edges should be covered by the eg operation of the node that the edge points to. Corresponding to the two operations, two induced sub-graphs namely $G1$ and $G1'$ results. The optimal solution to the root WIG is between the optimal solution to $G1$ plus g(1) and optimal solution to $G1'$. A *s*olution tree can be constructed to represent each level of operations and the resulting sub-graphs. On the left branch (g(1) operation) of the level, sub-graph $G1$ is a fresh start, with node 2 as next processing node. Again we can have g(2) and blank operation on it and get two sub-graphs $G2$ and $G2'$. On the right branch(b(1)) of the first level, node 2 has a dangling edge, which forces eg(2) to be the next operation resulting exact same sub-graph as $G2$. Further expansion on this branch is unnecessary. The whole search tree is shown in Figure 5.8.

This property of subproblem overlapping is not a coincidence. It is determined by the property of WIG. At any level of search tree, the sub-graphs which result from operation g(i) and b(i), namely G_i and G_i' have the same number of nodes. G_i' contains the entire G_i

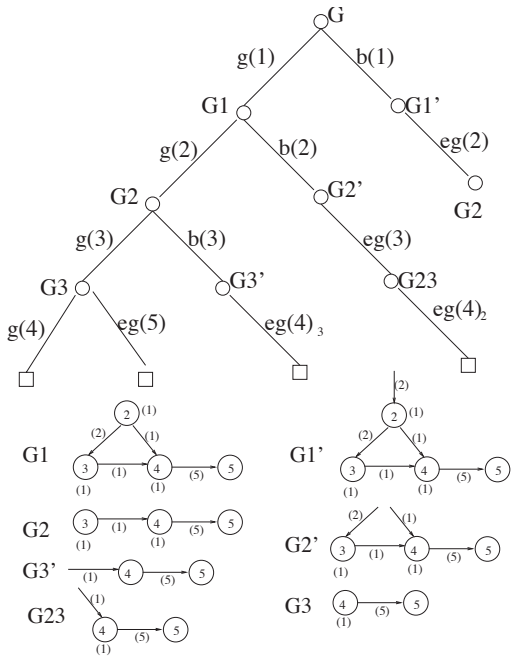


Figure 5.8: Dynamic programming and search tree

plus some dangling edges from node i. Based on Lemma 5.3.2, nodes that have dangling edges are consecutive in the processing order. Assume that the dangling edges point to node j_1 through j_k which are the next k nodes about to be processed. j_1 has no incoming edges besides the dangling edge from i. $eg(j_1)_i$ is the only choice. Based on the same reasoning, $eg(j_2)_i$ through $eg_(j_k)_i$ must be taken in the next steps, which results in G_x. On the other hand, G_i will become G_x after taking $g(j_1)$ through $g(j_k)$ consecutively. So the optimal solution on $G(x)$ is used repeatedly to construct optimal solutions on G_i and G_i'.
If any sub-graph is no longer connected, there is no affect among each connected component. Solving each connected component independently, the summation of the cost of individual optimal solutions is the optimal solution to the combined graph.

Theorem 5.3.3. *The solution tree contains the optimal solution in term of operation cost for covering all the edges in the WIG.*

Proof. We construct the solution tree by enumerating all possible operations of nodes when traversing nodes in a topological order (i.e., given edge $\widehat{i,j}$, enumerate i's operations before j's). Therefore, when we enumerate operations of nodes in a reverse order at some step

(i.e., given edge $\widehat{i,j}$, enumerate j's operations before i's), we may generate solutions that do not belong to the solution tree. We'll show that those solutions cannot be optimal. Assume we enumerate operations according to the topological order of all nodes except for nodes i and j where edge $\widehat{i,j}$ exists. We process node j before node i. If operation g(j) or b(j) is selected (depending on whether j has outgoing edges or not), enumeration of node i in next step is not affected. In another word, selecting g(j)/b(j) first and then processing i is the same as processing i first and then selecting g(j)/b(j), which is covered by the solution tree. Let us consider cases when operation eg(j) is selected.

1. when node j has the only incoming edge $\widehat{i,j}$, there are several subcases:

   a) node i has no incoming edges and the only outgoing edge $\widehat{i,j}$, we have no choice but to select operation b(i). $b(i)+eg(j)$ is the segment that the solution tree would generate;

   b) node i has no incoming edges and multiple outgoing edges including $\widehat{i,j}$, we can select operation g(i) or b(i). If we select b(i), this is the segment of solution that solution tree would generate. If we select g(i), segment $g(i)+eg(j)$ costs more than $g(i)+g(j)/b(j)$, which covers the exact same edges. Note that $g(i)+g(j)/b(j)$ is the segment of the solution tree;

   c) node i has an incoming edge, we have to select operation eg(i). Similar to the above subcase, $eg(i)+eg(j)$ covers the exact the same edges as $eg(i)+g(j)/b(j)$ while costs more. $eg(i)+g(j)/b(j)$ is covered by the solution tree;

2. when node j has incoming edge $\widehat{h,j}$ besides $\widehat{i,j}$, we know incoming edge $\widehat{h,i}$ must exist. During processing of node i, we have to select $eg(i)$:

   a) if $eg(j)_h$ covers both edge $\widehat{i,j}$ and $\widehat{k,j}$ (i.e., cost of $eg(j)_h$ is greater than cost of $eg(j)_i$), $eg(i)+eg(j)_h$ is the solution segment that the solution tree would generate;

   b) if $eg(j)_h$ does not cover edge $\widehat{i,j}$ (i.e., cost of $eg(j)_h$ is smaller than cost of $eg(j)_i$), to enumerate $eg(j)$, we have the choice of $eg(j)_h$ and $eg(j)_i$. If $eg(j)_h$ is selected, the rest is the same as the above subcase, which we will obtain a solution

segment that the solution tree would generate. If $eg(j)_i$ is selected, segment $eg(i)+eg(j)_i$ covers the same edges as $eg(i)+eg(j)_h$ covers while costs more. $eg(i)+eg(j)_h$ is covered by the solution tree.

After analyzing each case, we know solutions that do not belong to the solution tree all costs more than their counterpart in the solution tree. Therefore, those solutions cannot be optimal. This proves the solution tree contains optimal solutions. □

To analyze the complexity of the algorithm, consider a WIG with N nodes. The enumeration of solutions is bounded by O(N) and each solution requires O(N) operations. Therefore time complexity of our algorithm is bounded by $O(N^2)$.

5.4 Common Use Form and Global Optimal Solution

Given the algorithms presented so far, we have developed an optimal solution to the inverse translation problem when interferences are restricted to gated-non-gated CC interferences. Optimal inverse translation when multiple gated CCs share variables is much involved.

At the end of inverse transformation of a gated CC, all variables in the definition set are to be named by a single representative name. Consider two gated CCs that share a definition. The shared definition must be named by representatives chosen by the two CCs respectively. However, if the representatives chosen by two CCs are different, it is impossible for a variable to be named with two different names. Therefore either the two CCs must be isolated by inserting copies of the shared definitions or the two CCs must use the same representative. Renaming two CCs with the same name implies a union operation on the use set and definition set of the two CCs, which results in what we call Common Use Form(CUF). Isolating each shared variable or putting CCs into CUF are two alternatives. While isolation is always feasible, constructing CUF is restrictive.

When multiple gated CCs share definitions and they can be put into common use form, WIG and NSPG graphs based solutions developed earlier in this chapter can be used to solve the resulting CUF to obtain the optimal solution to the combined CC. Consider

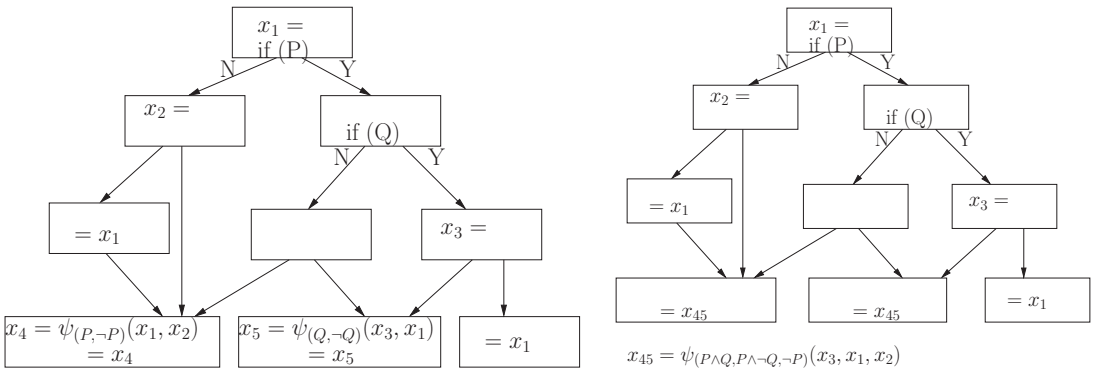


Figure 5.9: Common use form

Figure 5.9(a), in which two CCs CC_{x_4} and CC_{x_5} share definition x_1 is shown. Corresponding CUF for these CCs is given in Figure 5.9(b). Note that a single gating function represented by x_{45} routes definitions to the appropriate set of uses, although this combined CC does not have the maximal sharing of definitions property for its use set. It is easy to observe that two gated CCs cannot be put into the CUF when the gating path predicates of definitions between the two CCs are not compatible.

Unfortunately, not only that CUF is not always possible, but also the optimal solution to the CUF may not be the globally optimal solutions as there may be a global solution with a better cost based on isolation of shared variables. As a result, for a set of CCs that are tied to each other due to definitions sharing during inverse transformation, approaches vary among two extreme approaches and those in between. On one end of approaches, all CCs are put into a single combined CC so that the optimal solution for the CC is searched. For the other end, optimal solutions for all CCs are searched independently under the restriction that each shared variable must be isolated. Between the two extremes, the whole set can be divided into several subsets such that the CUF approach is performed intra-subsets and the isolation is performed inter-subsets. Any valid configuration of subsets forms an approach and therefore leads to a solution.

5.4.1 Global Solution Through Complete Isolation

Consider complete isolation approach. Given m CCs, namely $CC_1, CC_2, ..., CC_m$, sharing a single definition v_1. To solve each CC independently, we can apply algorithms discussed in this chapter and get m local optimal solutions, namely $sol_1, sol_2, ..., sol_m$. For CC_i and its corresponding local solution sol_i, if sol_i contains an operation on v_1, CC_i is isolated from other CCs. This is because an operation on v_1 (i.e., copies of v_1 are inserted) not only serves as an interference solver within the CC, but also serves as an isolation instance. v_1 is covered once when a local solution contains an operation on it or an instance of isolation is performed on it. To isolate m CCs completely, v_1 must be covered at least $m-1$ times. Let n be the number of coverage for v_1 by the union of all the local optimal solutions. If $n = m-1$, the union of the local optimal solutions is the global optimal solution. Otherwise, $m-1-n$ instances of isolation must be performed. Each instance of isolation costs a copy of v_1 inserted at the definition site of v_1. The local optimal solutions plus these instances of isolation is the global optimal solution to the set of CCs. Proof sketchy:
Assume we use an alternative local solution sol_i' for CC_i to replace sol_i which contains no operation on v_1 and keep all other local optimal solutions the same. Even if sol_i' contains an operation on v_1, the cost difference between sol_i and sol_i' cannot be less than one which is the isolation cost for sol_i. Therefore the union of the local optimal solutions is the global optimal solution when a single variable is shared under the complete isolation paradigm.

For the general case, consider variables $v_1, v_2, ..., v_n$ are shared by $CC_1, CC_2, ..., CC_m$. We know each variable requires at least $m-1$ instances of coverage. In this case, the union of local optimal solutions may not be the global optimal one. Alternative local solutions may contain operations that cover more variables with less cost in terms of global cost than the corresponding local optimal ones. Next, we present an approximate solution to this problem.

5.4.2 Approximation of Global Optimal Solution

Let's look at a related problem: Given n variables, namely $v_1, v_2, ..., v_n$ which are shared by $CC_1, CC_2, ..., CC_m$, we look for the global optimal solution GS' with cost $\delta_{GS'}$ such that the solution solves each CC's local interferences and covers each variable m times. Consider the difference between GS' and our original goal GS which covers each variable at least $m-1$ times with cost δ_{GS}. We can find the lower bound of cost of GS given GS'. GS' covers each variable at most once more than GS. For n variables, the total cost difference $\Delta_1 = \delta_{GS'} - \delta_{GS}$ is less than n. In other words, if we find GS', cost of GS cannot be less than $\delta_{GS'} - n$. In order to find GS', let's annotate local solutions for CC_i as $sol_{i,*}$. We order local solutions based on their costs so that the local optimal solution for CC_i is $sol_{i,0}$. Given local solution $sol_{i,j}$, it has cost $\delta_{i,j}$ and covers $\chi_{i,j}$ number of variables. Given two local solutions for CC_i, namely $sol_{i,j}$ and $sol_{i,k}$, the two solutions are globally equal if $sol_{i,k}$ covers t more variables and costs t more than $sol_{i,j}$. This is because in order to cover the same number of variables, $sol_{i,j}$ can always match by using isolation. The total isolation cost for $sol_{i,j}$ is t which is the same as the cost difference between $sol_{i,j}$ and $sol_{i,k}$. Therefore among local solutions the global optimal solution tends to pick the one which relatively covers more and costs less. For that purpose, we compute for each local solution a unique value Adv. $Adv_{i,j} = (\chi_{i,j} - \chi_{i,0}) - (\delta_{i,j} - \delta_{i,0})$. We order the local solution based on Adv value and the solution for CC_i with the largest Adv is $sol^*_{i,0}$. We can prove that the approximate global optimal solution consists of the union of $sol^*_{i,0}$ for each CC_i.

Theorem 5.4.1. *GS' consists of* $\bigcup_i sol^*_{i,0}$ *where* $i \in (1,m)$.

Proof. We replace $sol^*_{i,0}$ by an arbitrary solution $sol^*_{i,t}$ and fix other local CC solutions to form a global solution $GS'_{i,t}$. $sol^*_{i,0}$ requires $n - \chi^*_{i,0}$ instances of isolation. Similarly $sol^*_{i,t}$ requires $n - \chi^*_{i,t}$ instances of isolation. The cost difference between $GS'_{i,t}$ and GS' is given by $\Delta_2 = (\delta_{i,t} * + n - \chi^*_{i,t}) - (\delta_{i,0} * + n - \chi^*_{i,0})$, which is $Adv^*_{i,0} - Adv^*_{i,t}$. Because $sol^*_{i,0}$ has the largest Adv among the local solutions, Δ_2 is always greater than 0. It implies that GS' always costs less than $GS'_{i,t}$. Therefore GS' is the approximate global optimal solution. □

Based on the proof, we compute the cost of GS' as $\delta_{GS'}, given by \sum_i \delta^*_{i,0} + n \times m - \sum_i \chi^*_{i,0}$. Therefore we obtain the lower bound and upper bound of cost of the global optimal solution which is $\delta_{GS'} - n$ and $\delta_{GS'}$ respectively.

5.4.3 Validity of Proposed Approach

As it can be seen, an optimal solution to the inverse translation is involved. While a simple interference graph based solutions all appear to be NP-Complete solutions developed so far are all polynomial time algorithms when we seek the global solution through appreciate application of these algorithms. It then becomes a legitimate question, as to whether the combined solution would be globally optimal solution in terms of minimum number of copies placed. We claim that Theorem 5.3.3 is also applicable to the global case, that is a global optimal solution consists of only solutions from each CC's solution tree. In a global solution there may be isolations and the number of isolations is determined by the operations on nodes in WIGs. Also the global cost is computed from the local costs and costs of isolations. Now the question is that can there be a solution S which is not generated by a solution tree and contributes less to the global cost? This is not possible since there is always a solution generated by a solution tree that it has operations on the exact same nodes as the solution S and costs less. In other words there is always a solution (generated by a solution tree) with exact same number of isolations as S. It just suffices to consider some sub-cases in the proof of Theorem 5.3.3. For instance:

"node i has no incoming edges and multiple outgoing edges including $\widehat{i, j}$. If we select g(i), segment g(i)+eg(j) costs more than g(i)+g(j)/b(j), which covers the exact same edges."

Can a solution with "g(i)+eg(j)" be part of global solution? No. Because we can instead use g(i)+g(j) with less local cost. Both of "g(i)+eg(j)" and "g(i)+g(j)" have operations on i and j, so both will result in the same contribution to the number of isolations globally.

5.5 Conclusion

We have presented an approach to the inverse translation problem in the single assignment program representation domain. We have demonstrated that the problem is polynomial time solvable to generate minims number of copy instructions when definition sharing is restricted to gated-non-gated CC combinations. we also presented an algorithms which is at most $n-1$ away from an optimal solution when n variables are shared among multiple gated CCs. In the next chapter, we illustrate the power of the representation on two important optimizations.

Chapter 6

Optimizations on FGSA

The executable semantics of FGSA combined with its direct representation of congruence classes make the representation quite powerful in its ability to implement various optimizations. Although the adaptation and implementation of optimization algorithms are beyond the scope of this work, we illustrate the clarity the representation provides through two case studies, namely, constant propagation and global value numbering respectively. In the two case studies, we show that by unifying control flow and data flow traversal into one, FGSA achieves the same results as SSA does and enables simpler and more efficient optimization algorithms.

6.1 Constant Propagation on FGSA

Wegman and Zadeck (Wegman and Zadeck 1991) proposed two constant propagation algorithms, namely *Simple Constant* (SC) propagation and *Conditional Constant* (CC) propagation algorithms and the SSA versions of these algorithms, referred to as SSC and SCC respectively. The CC and SCC algorithms find more constants than the SC and SSC algorithms by using an extra data structure to keep track of the edges that are executable based on the evaluation result of the conditional expressions.

These algorithms can be implemented on an FGSA representation in a straight-forward manner. Interestingly however, the executable semantics of FGSA representation coupled

with the direct congruence class representation gives similar powers to the *simple constant* algorithm to that of *conditional constant* on FGSA. When a conditional expression becomes constant, any gating function which uses its result can be simplified and evaluated. As a result, we just need to redefine the *meet* operation for FGSA gating functions. Figure 6.1(a) shows the evaluation rules for gating functions. For simplicity, assume the gating function is in the form of $\psi_P(d_1, d_2)$. The rules are represented in the three-level lattice (Wegman and Zadeck 1991) where $\top$ at the highest level represents undetermined values, constant values such as c_1 and c_2 are in the middle and $\perp$ at the lowest level represents non-constant values. Evaluation of read-once predicates requires a different set of rules, as unlike other predicates, we assume their value is always $\perp$. For the gating function $\psi_R(d_1, d_2)$, the evaluation rules are given in Figure 6.1(b). We refer to the modified simple constant algorithm as the FGSA-SSC algorithm.

In order to see the application of the simple constant algorithm on both SSA and FGSA, consider Figure 6.2. When SSC is applied on Figure 6.2(a), evaluation of the *SSA edges* results in the discovery that x_1, P, x_2 are all constants, whereas Q is not a constant. Flow of two different constant values (namely, five and six) onto X_3 makes it $\perp$, which in turn makes x_4 also $\perp$. When the FGSA-SSC algorithm is used on Figure 6.2 (b), depending on the evaluation order, rules given in Figure 6.1 are applied, for the gating function, initially giving it a $\top$ value and the value constant 6 when x_2 is evaluated. Since x_3 represents the

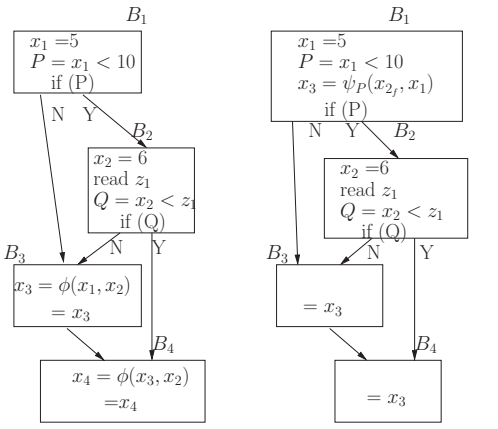
Rule	P	d_1	d_2	$\psi_P(d_1, d_2)$
1	$true$	X_1	X_2	X_1
2	$false$	X_1	X_2	X_2
3	$\perp$	c_1	c_2	$(c_1 == c_2)?c_1 : \perp$
4	$\perp$	$\perp$	X_2	$\perp$
5	$\perp$	X_1	$\perp$	$\perp$
6	$\perp$	$\top$	$\top$	$\top$
7	$\perp$	$\top$	c_2	$\top$
8	$\perp$	c_1	$\top$	$\top$
9	$\top$	$\top/c_1$	$\perp$	$\top$
10	$\top$	$\perp$	$\top/c_2$	$\top$
11	$\top$	$\perp$	$\perp$	$\perp$

(a)

Rule	R	d_1	d_2	$\psi_R(d_1, d_2)$
1	$\perp$	c_1	$\top$	c_1
2	$\perp$	$\top$	c_2	c_2
3	$\perp$	c_1	c_2	$(c_1 == c_2)?c_1 : \perp$
4	$\perp$	$\perp$	X	$\perp$
5	$\perp$	X	$\perp$	$\perp$

(b)

Figure 6.1: Evaluation rules for (a) ψ_P and (b) ψ_R functions



(a) SSC on SSA (b) SSC on FGSA

Figure 6.2: Constant propagation

congruence class, all uses of x_3 become a constant. Similar results can be achieved by applying the conditional constant algorithm on Figure 6.2(a).

Note that the future values have no effect on this algorithm as it chases def-use chains implemented through single assignment. Although this simple example shows the power of FGSA, this is by no means a proof that SSC and SCC are equivalent when used on FGSA.

6.2 Global Value Numbering (GVN) on FGSA

GVN(Rosen et al. 1988)(Click 1995)(Simpson 1996) is an optimization based on SSA form. It maps *value-congruent*[1] variables/expressions into the same class and thus achieves constant propagation, redundant computation and unreachable code elimination. In classic GVN, any computation at a confluence point that uses a ϕ result as an operand is split by renaming the operand with ϕ-arguments and moving up along corresponding incoming edges. The goal is to locate some local redundancy. On FGSA, this step is easy to perform because a ψ-function is executable and forward propagation is directly applicable.

[1]Term *congruent* is used by GVN to indicate two expressions are value-equivalent. To distinguish from congruence classes as we define in this dissertation, we use the term *value-congruent* in the following discussion.

A more recent work on GVN by (Gargi 2002) which builds on Simpson's work(Simpson 1996) exploits predicates of branch conditions to find more value-congruences. There are two key ideas in this work. One is to infer values from predicates (*value inference* and *predicate inference*). For example, given a branch $(if(x_0 == y_0))$, x_0 and y_0 are assumed to be in the same class in the region dominated by the taken edge of the branch. The other is to associate the arguments of acyclic ϕ-functions with the predicates that control their arrival (ϕ-*predication*). Two ϕ-functions are said to be value-congruent if their arguments are value-congruent and predicates of corresponding edges are value-congruent. We adopt Gargi's motivating example with some simplifications (Figure 6.3). Within the whole example region, y_0 can be replaced by x_0 due to predicate $R1$. After computing ϕ-*predication* for p_3 and q_2, they are put into the same class according to ϕ-function value-congruence definition, which eventually causes i_3 to become constant 1.

```
01 i0 = 1
02 if (x0 == y0) < R1 >
03    p0 = 0
04    if (x0 ≥ 1) < R2 >
05       if (i0 ≠ 1) < R3 >
06          p1 = 2
07       elseif(x0 ≤ 9) < R4 >
08          p2 = i0
09    p3 = φ(p0, p2, p1)
f1    (p3 = ψ(R2∧R3,R2∧¬R3∧R4,¬R2∨¬R3∧¬R4)
f1    (p1, p2, p0))
10    q0 = 0
11    if(i0 ≤ y0) < R2' >
12       if(9 ≥ y0) < R4' >
13          q1 = 1
14    q2 = φ(q0, q1)
f2    (q2 = ψ(R2'∧R4',¬R2'∨¬R4')(q1, q0))
15    i2 = p3 − q2 + 1
16 i3 = φ(i0, i2)
f3 i3 = ψ(R1,¬R1)(i2, i0)
```

Figure 6.3: A modified example from Gargi's work. Predicates are contained in <>. Line f1, f2, and f3 contain FGSA gating functions for corresponding ϕs.

The algorithm can be applied on FGSA with no changes. In fact, ϕ-*predication* analysis (time complexity is $O(E^2)$(Gargi 2002)) can be saved because gating predicates of ψ-functions have the same information. On line f1, because R3 is false (line 05) and hence $R2 \wedge R3$ is false, p_1 is irrelevant to p_3. The ψ-function is simplified to $\psi_{(R2 \wedge R4, \neg R2 \vee \neg R4)}(1,0)$. On line f2, because R2' and R2, R4' and R4 are value-congruent, the ψ for q_2 becomes $\psi_{(R2 \wedge R4, \neg R2 \vee \neg R4)}(1,0)$, which clearly shows p_3 and q_2 are value-congruent. FGSA simplifies the task of inferring values from predicates as well resulting in further simplification of the algorithm. Observe that at any gating function, any argument (definition) flows to the CC when its gating predicate is true. Based on Theorem 3.2.1 and its generalization, a gating expression consists of the path predicate (P1) controlling the execution of the definition, and another set of predicates (P2) which is true when the value is not killed. In other words, a definition can be computed only when its path predicate P1 is true and therefore, its computation can exploit its path predicate to infer values. For example, on line 3, i_2's gating predicate, namely $R1$, which in this case also is i_2's path predicate. Value inferring from $R1$ can be applied to all operands of i_2 and applied recursively, which results in i_2 being updated to constant 1 and in turn i_3 becoming constant 1. More aggressively, value inferring from the P2 part can also be applied. However the result using P2 inferring can be used only to update the gating function (since P2 is true if the gating function returns that definition) and may not be used to update the definition (since P2 may not be true at the definition site). As it can be seen, unlike value inferring on SSA, value inferring on FGSA is more targeted and may bring more sparseness to the algorithm.

Chapter 7

Recursive Future Predicated Form

[1] In this chapter, we revisit the concept of future values and demonstrate that the concept permits unrestricted code motion to the degree that entire procedures can be collapsed into a singe basic block under a new program representation call *Recursive Future Predicated Form (RFPF)*. Similar to FGSA, RFPF also is a representation built on the principle of *single-assignment* (Bilardi and Pingali 2003; Cytron et al. 1991) and it subsumes general *if-conversion* (Allen et al. 1983). RPFP can co-exist with any single-assignment representation, and can provide the framework in which existing represent can perform code motion with ease, or use the representation itself as the primary internal representation. In this respect, RFPF properly extends the single assignment forms and covers the domain of legal transformations resulting from instruction movements. Furthermore, we illustrate that under this representation, both the construction of the representation and program analysis itself can be performed using the code motion as the only mechanism. In this respect, possible transformations range from the starting SSA form where all data-flow is traditional, to a final reduction where the *entire procedure becomes a single block* through upward code motion, possibly with mixed (i.e., traditional and future) data-flow. We refer to a procedure which is reduced to a single block through code motion

[1]The material contained in this chapter was previously published in CC'10/ETAPS'10 Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction.

to be in *complete RFPF*. *Complete RFPF* expresses the program semantics without using control-flow edges except sequencing. During the upward motion of instructions, valuable information is collected and can be used to perform several sophisticated optimizations such as *Partial Redundancy Elimination* (PRE). Such optimizations typically require program analysis followed by code motion and/or code restructuring (Morel and Renvoise 1979; Knoop et al. 1992; Bodík et al. 1998).

Complete RFPF can be constructed starting from any single assignment form, including FGSA. Because of the property of FGSA gating functions, FGSA is a better starting point than SSA for generating complete RFPF transformation. In the following sections, Section 7.1 through Section 7.4 present transformation algorithms from SSA using code motion as the primary means. During the entire process, the single-assignment property is maintained. In Section 7.5, we further discuss how to use T1/T2/T_R transformations to directly build RFPF.

7.1 Code Motion in Acyclic Code

We first discuss code motion using future values in acyclic regions involving control dependencies. For an acyclic control-flow graph $G =< s, N, E >$ such that, s is the start node, N is the set of nodes and E is the set of edges, instruction hoisting involves one of three possible cases. These are: (1) movement that does not involve control dependencies (i.e., straight-line code), (2) *splitting* (i.e., parallel move to predecessor basic blocks), and (3) *merging* (i.e., parallel move to a predecessor block that dominates the source blocks). Note that movement of a ϕ-node is a special case and normally would destroy the single-assignment property. We examine each of these cases below:

Case 1 (Basic block code motion)**.** *Consider instructions I and J. Instruction J follows instruction I in program order. If I and J are true dependent, hoisting J above I converts the true dependency to a future dependency. Alternatively, if the instructions are future dependent on each other, hoisting J above I converts the future dependency to a true*

dependency (Figure 2.6(a) and (b)).

When code motion involves control dependencies, the instruction propagation is carried out using instruction predication, instruction cloning and instruction merging. An instruction is cloned when the instruction is moved from a control independent block to a control dependent block. Cloned copies then propagate along the code motion direction into different control dependent blocks. When cloned copies of instructions arrive at the same basic block they can be merged.

Case 2 (Splitting code motion)**.** *Consider instruction I that is to be hoisted above the block that contains the instruction. For each incoming edge e_i a new block is inserted, a copy of the instruction is placed in these blocks and a ϕ-node is left in the position of the moved instruction (Figure 7.1).*

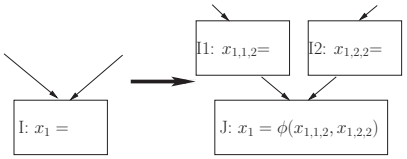


Figure 7.1: Splitting code motion

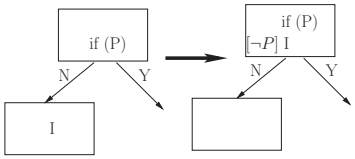


Figure 7.2: Merging code motion

Note that in Figure 7.1, when generated copies I1 and I2 are merged back into a single instruction, the inserted ϕ-node can safely be deleted and the new instruction can be renamed back to x_1. The two new names created during the process, namely, $x_{1,1,2}$ and $x_{1,2,2}$ are eliminated as part of the merging process. In order to facilitate easy merging of clones, we adopt the naming convention $v_{i,j,k}$ where v_i is an SSA name, j is the copy version number and k is the total number of copies. Generated copies can be merged when they arrive at the immediate dominator of the origin block, and in case of reduction to a single block, all copies can be merged. We discuss these aspects of merging later in Section 7.1.3.

Case 3 (Merging code motion)**.** *Consider instruction I that is to be hoisted into a block where the source block is control dependent on the destination block. The instruction I is converted to a predicated instruction labeled with the controlling predicate of the edge (Figure 7.2).*

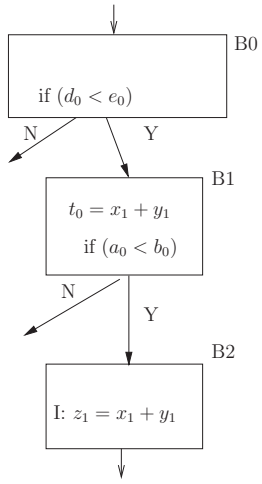
7.1.1 Future Predicated Form

When a predicated instruction is hoisted above the instruction which defines its predicate, the predicate guarding the instruction becomes *future* as the predicate is also a value and the data dependence must be updated properly. Figure 7.3 shows a control dependent case. Instruction I is control dependent on condition $a_0 < b_0$. When the instruction I is moved from $B2$ to $B1$, it becomes predicated and is guarded by Q (Figure 7.3(b)). In the next step, the instruction is hoisted above the definition of Q and its predicate Q becomes future (i.e., Q_f) (Figure 7.3(c)).

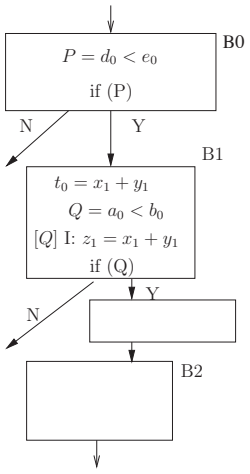
When a predicated instruction is hoisted further, it may cross additional control dependent regions and will acquire additional predicates. Consider Figure 7.3(c). Since the target instruction is already guarded by the predicate Q_f, when it moves across the branch defined by P, it becomes guarded by a nested predicate (Figure 7.3(d)). In terms of control flow, it means that predicate P must appear, and it will appear before Q. Similarly, if P is true, then Q must also appear since if the flow takes the true path of P the predicate Q will eventually be encountered. In other words, the conjunction operator has the short-circuit property and it is evaluated from left to right. Semantically, a nested predicate which involves future predicates is quite interesting as it defines *possible* control flow.

7.1.2 Elimination of ϕ-nodes

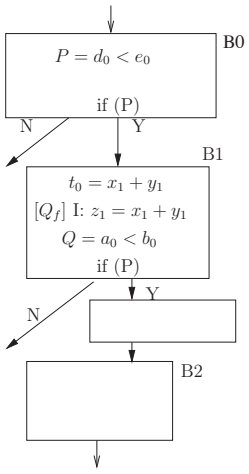
RFPF transformations aim to generate a single block representing a given procedure. The algorithms developed for this purpose hoist instructions until all the blocks, except the start node are empty. Proper maintenance of the program semantics during this process requires the graph to be in single-assignment form. On the other hand, movement of ϕ-nodes as regular instructions is not possible and the elimination of ϕ-nodes result in the destruction of the single-assignment property. For example, elimination of the ϕ-node $x_3 = \phi(x_1, x_2)$ involves insertion of copy operations $x_3 = x_1$ and $x_3 = x_2$ across each incoming edge in that order. Such elimination creates two definitions of x_3 and the resulting graph is no longer in



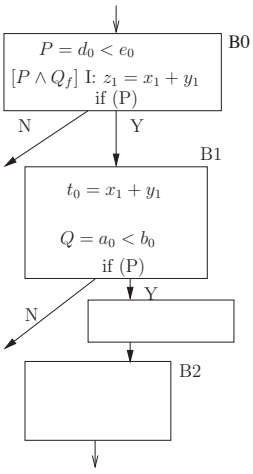
(a) before code motion



(b) after code motion



(c) future predicate



(d) nested predicate

Figure 7.3: Code motion across control dependent regions

single-assignment form. Our solution is to delay the elimination of ϕ-nodes until the two definitions can be merged, at which time a *gating function* can be used if necessary.

Since the gating functions created in this manner would be binary, instead of the gating path predicates introduced with FGSA representation, we redefine ψ to have a single controlling predicate P:

Definition 23. *We define the gating function* $\psi_p(a1, a2)$ *as an executable function which returns the input* $a1$ *if the predicate* p *is true and* $a2$ *otherwise.*

Note that during merging, cloned copies already bring in the necessary information for

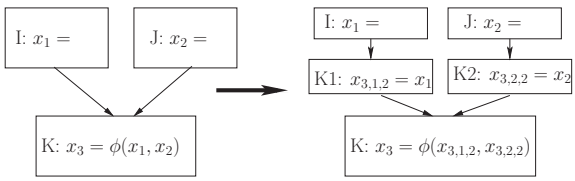


Figure 7.4: ϕ-node elimination

computing the controlling predicate for the gating function. The merging process is enabled by transforming the ϕ-node in a manner similar to the *splitting* case described above:

Case 4 (ϕ-node elimination)**.** *Consider the elimination of the ϕ-node $x_3 = \phi(x_1, x_2)$ (Figure 7.4). ϕ-node elimination can be carried out by placing copy operations $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ across each incoming edge in that order and updating the ϕ-node with the new definitions to become $x_3 = \phi(x_{3,1,2}, x_{3,2,2})$.*

Merging of the instructions $x_{3,1,2} = x_1$ and $x_{3,2,2} = x_2$ requires the insertion of a *gating function* since the right-hand sides are different. Once the instructions are merged, the ϕ-node can be eliminated. It is important to observe that until the merging takes place and the deletion of the ϕ-node, instructions which use the ϕ-node destination x_3 can be freely hoisted by converting their dependencies to *future dependencies*.

7.1.3 Merging of Instructions

In general, upward instruction movement will expose all paths resulting in many copies of the same instruction guarded by different predicates. This is a desired property for optimizations that examine alternative paths such as PRE and related optimizations since partial redundancy needs to be exposed before it can be optimized. We illustrate an example of PRE optimization in next chapter. On the other hand, the code explosion that results from the movement must be controlled. RFPF representation allows copies of instructions with different predicates to be merged. Merging can be carried out between copies of instructions which result from a splitting move, as well as those created by ϕ-node elimination. As previously indicated, merging of two instructions with the same derivative destination (i.e., such as those which result from ϕ-node elimination) requires the insertion

of the *gating function* ψ into the appropriate point int program, whereas merging of the two copies of the same instruction can be conducted without the use of a gating function. When the merged instructions are the only copies, the resulting instruction can be renamed back to the ϕ destination. Otherwise, a new name is created for the resulting instruction, which will be merged with other copies later during the instruction propagation.

Definition 24. *Two instructions* $\gamma : x_{i,m,k} \leftarrow e1$ *and* $\delta : x_{i,n,k} \leftarrow e2$*, where* γ *and* δ *are predicate expressions, represent the single instruction* $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow e1$ *if e1 and e2 are identical.*

Definition 25. *Two instructions* $\gamma : x_{i,m,k} \leftarrow e1$ *and* $\delta : x_{i,n,k} \leftarrow e2$*, where* γ *and* δ *are predicate expressions represent the single instruction* $\gamma \vee \delta : x_{i,(m,n),k} \leftarrow \psi_P(e1, e2)$ *if e1 and e2 are not identical. The predicate expression P is the first predicate expression in* γ *and* δ *such that P controls* γ *and* $\neg P$ *controls* δ*.*

Definition 26. *Instruction* $\gamma : x_{i,(p,\ldots,q),k} \leftarrow e$ *can be renamed back to* $\gamma : x_i \leftarrow e$ *if* $(p, \ldots, q)$ *contains a total of k version numbers.*

Theorem 7.1.1. *Copy instructions generated from a given instruction I during upward propagation are merged at the immediate dominator of the source node of I, since all generated copies will eventually arrive at the immediate dominator of the source block.*

Proof. Let node A be the immediate dominator of the source node I has originated from in the forward CFG. Assume there's one copy instruction I' which does not pass through A during the whole propagation. For this to happen, there must be a path p, which from the start node reaches I' and then reaches the source node of I. The fact that p does not pass through node A conflicts the assumption that A is the immediate dominator node of I. □

Let us now see through an example how the instruction merging effectively eliminates unnecessary code duplication. Consider the CFG fragment shown in Figure 7.5(a). Suppose that instruction I needs to be moved to block $B1$. Further note that instruction I is control independent of the block B1. We first insert the branch condition $P = a < b$ in block

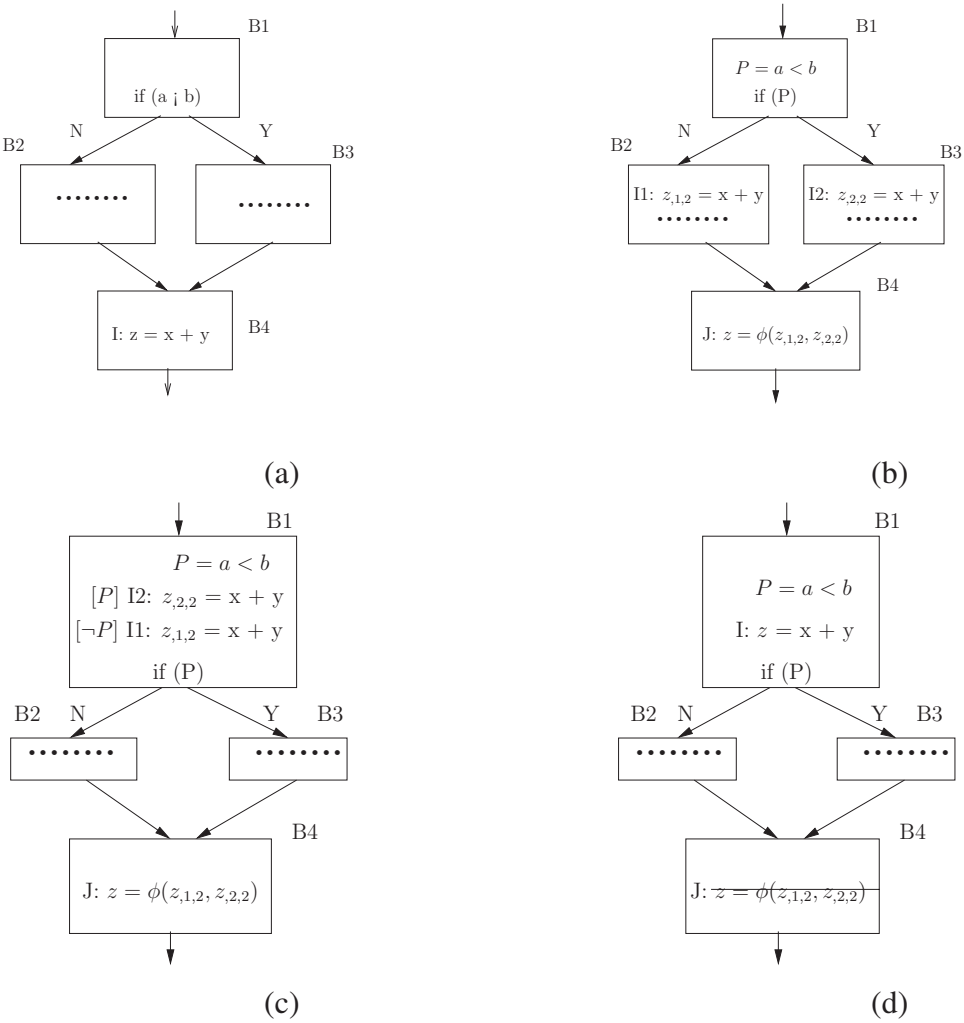


Figure 7.5: Instruction propagation

$B1$. Moving of I is accomplished by applying the *splitting* transformation, followed by progression of $I1$ and $I2$ into blocks B2 and B3 respectively and the deletion of temporary nodes inserted during the movement (Figure 7.5(b)). Next, the instructions $I1$ and $I2$ are propagated using a *merge* move which predicates them with $\neg P$ and P respectively and places them in block $B1$ (Figure 7.5(c)). At this point, using Definition 24, the two instructions can be reduced to a single instruction I without a predicate (Figure 7.5(d)) and the ϕ-node can be deleted. Note that the merging of the instructions and the deletion of ϕ node must be carried out in the same step to maintain single-assignment property.

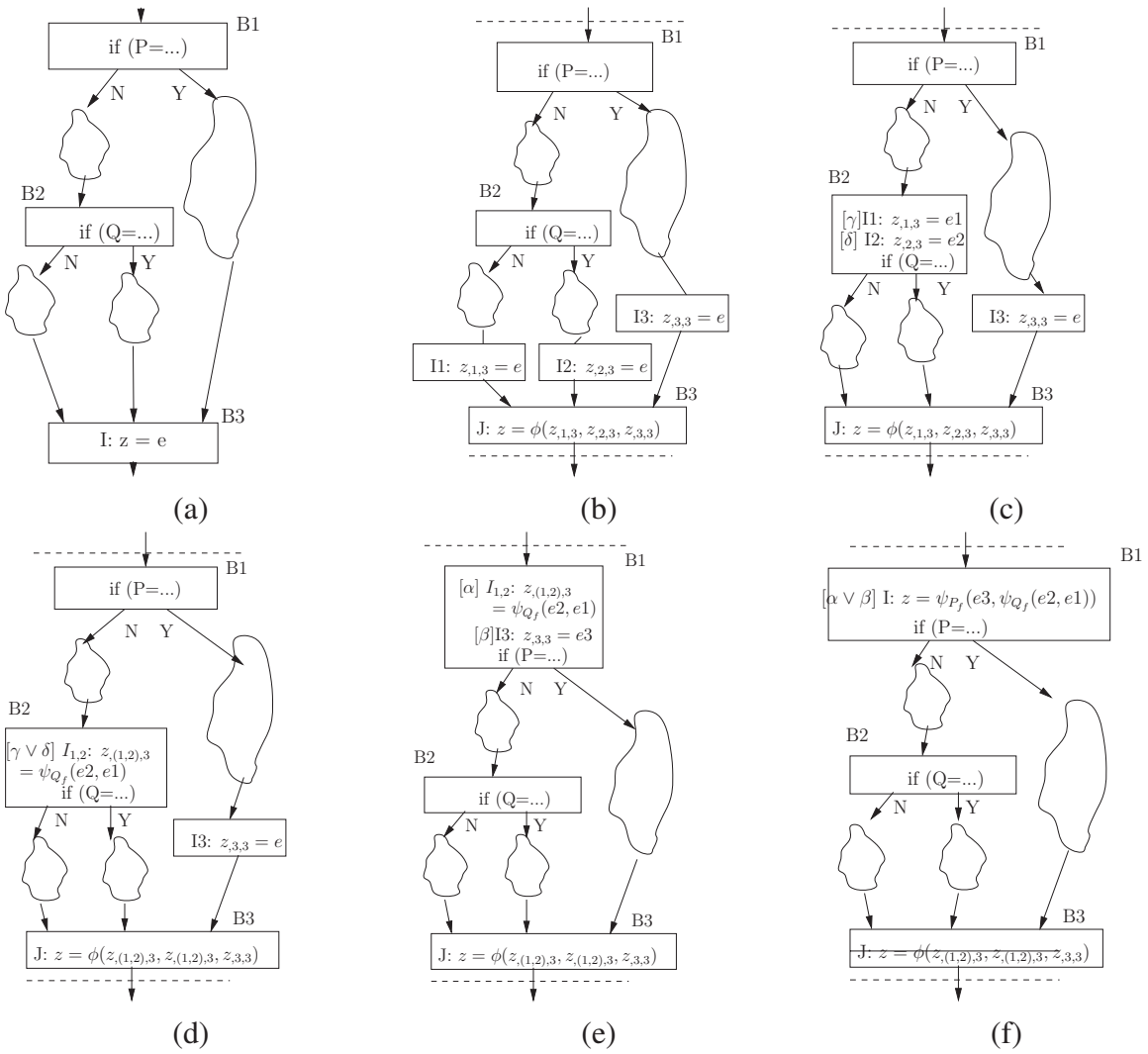


Figure 7.6: Instruction merging

A detailed example which shows how the adopted naming convention facilitates instruction merging is illustrated in Figure 7.6(a). In this example regions represented as *clouds* are arbitrary control and dataflow regions an instruction has to pass through and cloud regions have no incoming or outgoing edges except for the explicitly indicated ones. Instruction I, which computes e is moved across block $B3$ by applying a *splitting* transformation (Figure 7.6(b)). Next, two of the total three copy instructions, namely, $I1$ and $I2$ converge in block $B2$ and during propagation may acquire different expressions, namely, $e1$ and $e2$. These two instructions are merged into $I_{1,2}$ using Definition 25 (Figure 7.6(c)(d)).

Note that future predicate Q_f is used in the gating function for choosing between $e1$ and $e2$. At this point, checking the name of destination $z_{,(1,2),3}$, indicates that there are copies that are not merged yet. Further instruction propagation results in the merging of $I_{1,2}$ and $I3$ in block $B1$. Applying Definition 25 and 26, all the copy instructions are reduced into a single instruction I, which is represented through a nested gating function [2]. At this point the ϕ-node can be deleted. The final result is shown in Figure 7.6(f).

7.2 Instruction-Level Recursion

In a reducible control-flow graph, a loop region is a strongly connected region where the loop header forms the upward propagation boundary. Therefore moving instructions across the loop header requires a new approach. This approach is to convert every instruction within the loop region to an equivalent instruction that can iterate in parallel with the loop execution independently. We define an instruction that *schedules* its next iteration, a *recursive instruction*.

Conceptually, a recursive instruction appears as a function call that is spawned at the point the control visits the instruction. The instruction executes within this envelope and checks a predicate to see if it should execute in the next iteration. If the predicate is true, a recursive call is performed. Otherwise the function returns the last value it had computed. In this way, as long as the predicate which controls the loop iteration is known, any loop instruction can iterate itself and hence it can be separated from the loop structure (or pushed out of the loop region). In other words, an instruction that is hoisted above the loop header becomes a recursive instruction controlled by a special predicate called the *Recursive Predicate*:

Definition 27. *Recursive Predicate: In a loop L that has a single loop header H and a single backedge e, the predicate expression which allows control flow to reach e from H without going through e is Recursive Predicate for L.*

For loops with multiple edges we can use the disjunction of the recursive predicates

[2] These nested gating functions can be properly represented using FGSA ψ functions

computed for each edge. This follows from the observation that we can insert an empty block such that all the backedges are connected to this block and removed from the loop header and a single exit from this block becomes the single backedge for the graph. Since the controlling predicate of the newly inserted block's outgoing edge is the disjunction of the controlling predicates of all the incoming edges, such graphs can be reduced into a single backedge case described above.

Since the instruction returns only its last value, we can establish proper data dependencies with instructions outside the loop region. Note that, a recursive instruction should also include a predicate to implement the control flow within the loop body:

Definition 28. Recursive Predicated Instruction: $x_i = (R)[P]\{I : x_{i_j} = ...\}$, *where I is the instruction,* x_i *is the single-assignment name of the instruction's destination, j is the loop nest level, P is the predicate guarding I obtained through acyclic instruction propagation into the loop header and R is the recursive predicate the instruction iterates on.*

Note that the recursive instruction renames the destination of the original instruction by appending the loop nest level, and the function returns the original name. In Section 7.3.2 we revisit this renaming. From an executable semantics perspective, a recursive predicate must need to know the number of readers it is being waited by and should generate a new value after all the readers have read it.

7.3 Code Motion in Cyclic Code and Recursive Future Predicated Form

We follow a hierarchical approach to perform code motion in cyclic code. For this purpose, starting with the inner-most loops, we convert the loops into groups of recursive instructions, propagate them to the loop header of the immediately enclosing loop and apply the procedure repeatedly until all cyclic code is converted into recursive instruction form, eventually leading to a single block for the procedure.

7.3.1 ϕ-nodes in Loop Header

Although any code motion within a given loop can be carried out using the acyclic code motion techniques, the ϕ-nodes in the loop header cannot be eliminated using the techniques developed for acyclic regions. This is because a ϕ-node placed in a loop header controls data flow coming from outside the loop and loop carried values. As discussed in section 3.2, this data flow is not path-separable using predicates computed from control-flow. Instead, we can use the appreciate gating function developing on the single assignment from over which RFPF is being built. This function is the μ-function (Ottenstein et al. 1990) if RFPF is built on SSA or GSA and the ψ function with a read-once predicate, if it's built on FGSA. For ease of reference, we define the μ function as given in (Ottenstein et al. 1990):

Definition 29. *"We define the gating function $\mu(a^{init}, a^{iter})$ as an executable function. a^{init} represents external definitions that can reach the loop header prior to the first iteration. a^{iter} represents internal definitions that can reach loop header from within the loop following an iteration. a^{init} is returned when control reaches loop header from outside of the loop. a^{iter} is returned in all subsequent iterations."*

7.3.2 Conversion of Loops into Instruction-Level Recursion

The conversion is achieved by following the following steps:

1. Identify a single-entry, multi-exit region where the entire region is dominated by an inner-most loop header.

2. Propagate all instructions except branches to the loop header using acyclic code motion discussed before.

3. Calculate the controlling predicates for the exit edges and calculate the *Recursive Predicate* using *Algorithm 3* shown in Figure 7.7.

Algorithm 1
for each back-edge and exit edge e do

begin
  let b be the block which e originates from
  let I be any instruction originally in block b
  let $\alpha(I)$ be the predicate expression guarding I
  if block b has a branch on predicate P then
    if e is on the true path of the branch then
      $p(e) \leftarrow \alpha(I) \wedge P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge P$ if e is an exit edge
    else
      $p(e) \leftarrow \alpha(I) \wedge \neg P$ if e is a back-edge
      $q(e) \leftarrow \alpha(I) \wedge \neg P$ if e is an exit edge
    end
  else /* e is fall through backedge */
    $p(e) \leftarrow \alpha(I)$ if e is a back-edge
  end
  if e is a back edge then
    $RP \leftarrow p(e)$
end

Figure 7.7: Algorithm 3: Compute RecursivePredicate and ExitPredicate

4. Pick an unused single assignment name for the *RecursivePredicate*.

5. Convert ϕ-nodes to gating function μ or ψ with a read-once predicate..

6. Insert $(RP)[T]RP = ...$ at the very beginning of loop header where *RP* is the single assignment name picked in the previous step and it is assigned to the computed *RecursivePredicate* by converting all the predicate variables in the computed predicate to future form.

7. Convert every instruction in the header to recursive form using *RP* and delete the back edges and branches. The conversion involves renaming all instructions which are in the loop body such that each single assignment name that is defined in the block is appended the loop nest level, starting with zero at the inner-loop and incrementing. This renaming will update any uses which are loop carried to the new name while keeping names which are defined outside the loop unchanged.

Once the above process is completed, an inner-most loop has been converted to sequential code. We apply the above process until the entire procedure is converted into a single block.

Theorem 7.3.1. *The predicate expression controlling the backedge e can be computed correctly using Algorithm 1.*

Proof. Figure 7.8 that contains an arbitrary innermost loop is used to demonstrate the proof. $B1$ is the loop header, $e1$ is a backedge originating from block $B3$ which contains instruction J. Assume a trivial instruction K is inserted in $e1$ as shown in Figure 7.8(b). The predicate expression controlling K, namely γ is the same as the one controlling $e1$. γ is computed by propagating instruction K to $B1$. For that purpose, K is first moved into block $B3$. K becomes $\beta : K$ in $B3$ where three cases may happen:
case1: $\beta = P$ if $e1$ is the taken edge of $B3$,
case2: $\beta = \neg P$ if $e1$ is the fall through edge of $B3$,
case3: $\beta = true$ which means K is not guarded by any predicate if $B3$ is ended with an unconditional jump.
Propagate $\beta : K$ and instruction J to $B1$. Since $\beta : K$ and J propagate from the same block, the predicates guarding these two instructions are the same when they reach $B1$. Assume J becomes $\alpha : J$ in $B1$, then K becomes $\alpha : \{\beta : K\}$. Combining nested predication yields $\gamma = \alpha \wedge \beta$. □

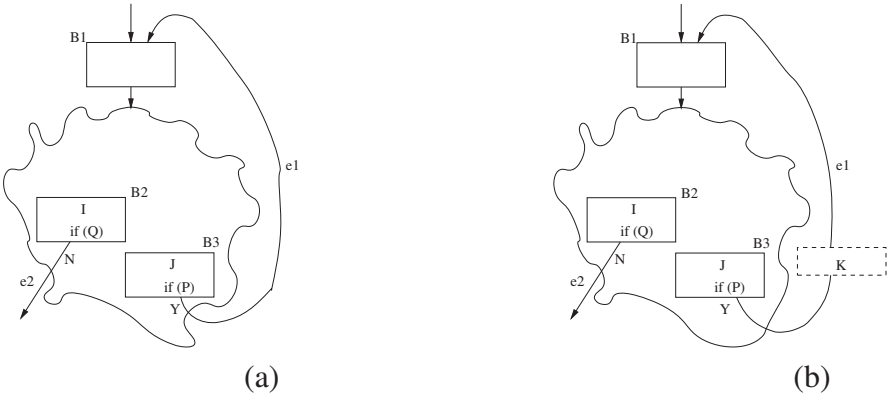
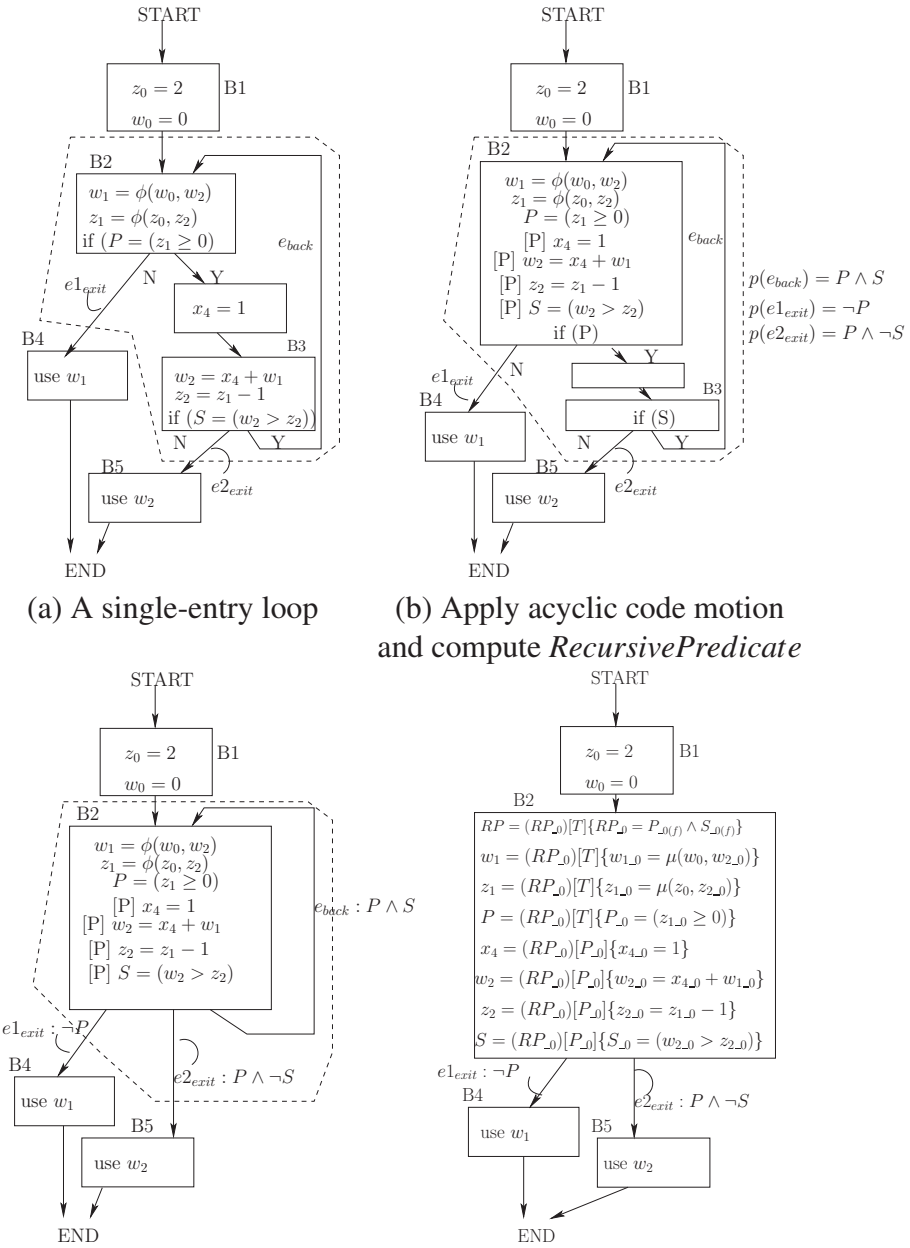


Figure 7.8: Theorem 7.3.1

Note that although K may be split into multiple copies during the propagation, the last copy instruction is merged and hence the resulting instruction is renamed back to K in the loop header $B1$ if it is not merged before reaching $B1$.



(a) A single-entry loop

(b) Apply acyclic code motion and compute *RecursivePredicate*

(c) Eliminate loop except loop header

(d) Convert to recursive form

Figure 7.9: Program 1:Conversion of a cyclic program into RFPF

Figure 7.9(a) is an example that shows the steps of transforming cyclic code. The region cut out is a loop region with a single loop header $B2$. Following the algorithm, we first propagate every instruction inside the loop into the loop header(Figure 7.9(b)). During the instruction propagation, the necessary predicate information to compute the *RecursivePredicate* and controlling predicates for the exit edges are collected naturally, shown on the right side of Figure 7.9(b). Next, everything in the loop region except the loop header and the back edge is deleted.(Figure 7.9(c)). The result of the conversion is shown in Figure 7.9(d).

7.4 Code Motion Involving Memory Dependencies and Function Calls

Memory dependencies pose significant challenges in code motion. There are many cases a compile time analysis of memory references would not yield precise answers. Our solution is to assume dependence and enforce the original memory ordering in the program through predication. Since a series of consecutive load operations without intervening stores have no dependence on each other, RFPF allows these loads to be executed in any order once the dependence of the first load in the series is satisfied. We define the memory operations as: `MEM, @P` where MEM represents a Load/Store operation and P is a predicate whose value is set to 1 when the memory operation MEM gets executed. Any memory operation that has a dependence with MEM will be guarded by P as a predicated operation. In this way, the dependence among memory operations are converted into data dependencies explicitly. Once the memory operations are converted in this manner, they can be moved like any other instruction. Because of the predication, if a memory operation is hoisted above another which defines its controlling predicate, the controlling predicate becomes a future value (Figure 7.10).

Taking control flow into account, one memory operation may have multiple dependencies due to multiple paths through which it can be reached. In this case, a gating-function is

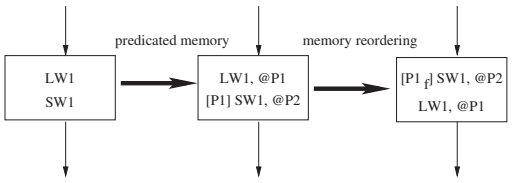


Figure 7.10: Predicated memory and reordered memory

used to choose which dependence finally comes to the memory operation. Figure 7.11 shows a case where a store, namely ST2 is in a converging node of LD1 and ST1. Note that the store ST2 cannot execute before LD1 or ST1 completes, therefore the ϕ-function selects the destination predicates of previous memory operations. Hence $P_4 = \phi(P_1, P_3)$ is inserted to select the dependencies with LD1 and ST1 [3].

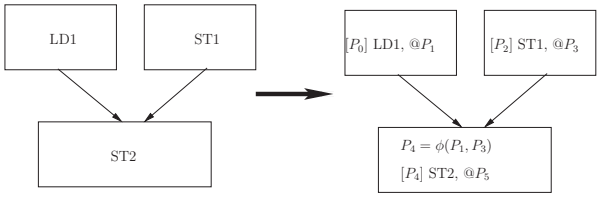


Figure 7.11: ϕ-node of predicates before a store

Figure 7.12 shows a slightly different case where a load, namely LD2 is in a converging node. A load operation normally is independent of another load whereas a store after a series of load operations is dependent on all of them. For this reason, a new predicate representing the dependence from a series of loads so far needs to be computed after every load instruction since the last store operation in the path. Note that, in Figure 7.12, if the control flow takes the left edge, LD2 is controlled by the same predicate that LD1 is controlled by, which is produced by the last store in the path. Further notice, a new predicate needs to be computed after the load as representing the dependence from all the loads so far. Therefore two ϕ-nodes are inserted before LD2, where P_4 is for guarding LD2, and P_6 is for computing the new predicate after LD2, namely P_7. Again, note that, since our goal is to impose a minimal ordering of memory operations, store and load at the converge node need different number of ϕ-functions and each ϕ is computed differently as they are

[3] In case of FGSA, the corresponding gating function would be $\psi_{(P_0,P_2)}(P_1, P_3)$

shown in Figure 7.11 and 7.12. In a general rewriting algorithm, without knowing the next memory operation is a load or a store, the combination of all three ϕs are needed at each convention node.

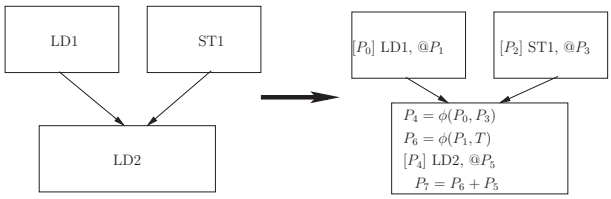


Figure 7.12: ϕ-node of predicates before a load

For building RFPF on SSA, our algorithm to rewrite memory operations is based on Cytron et al's SSA construction algorithm (Cytron et al. 1991). Since all the load/store operations can be treated as assignments to the same variable, Cytron et al's algorithm can be modified to accomplish the rewriting. Cytron et al's algorithm has two phases: placement of ϕ-functions and renaming. The original ϕ-function placement algorithm uses the iterative *dominance frontier* information to place ϕs. Figure 7.13 shows the modifications on this phase.

> Treat each load/store instruction as an assignment to a special variable M.
> Modify placement of ϕ-functions algorithm in (Cytron et al. 1991):
> instead of placing one ϕ-function for each variable at each proper node,
> place three ϕ-functions for the special variable M, namely
> $P1 = \phi(T,\ldots,T)$, $P2 = \phi(T,\ldots,T)$, $P3 = \phi(T,\ldots,T)$,
> where T represents boolean value *true*

Figure 7.13: Rewriting memory operations: placement of ϕ-functions

We put three ϕ-functions at each converge node, where P1 computes the predicate guarding the next load operation, P2 computes the predicate guarding next store operation and P3

computes the predicate representing the consecutive loads so far. The purpose of P3 is the same as P_7 in Figure 7.12. Extra ϕ-nodes that are never used later can be eliminated each by pruning.

Modify renaming algorithm in (Cytron et al. 1991):
Elements stored in stack S for the special variable M are in the form of a tuple $< P_j, P_k, P_l >$

(1) for each memory operation MEM do
(2) $< P_j, P_k, P_l >$ = pop(S)
(3) if MEM is a *load*
(4) rewrite MEM with $[P_j]MEM, @P_i$
(5) insert $P_{i+1} = P_l \wedge P_i$ behind the current memory operation
(6) push(P_j, P_{i+1}, P_{i+1})
(7) i=i+2
(8) if MEM is a *store*
(9) rewrite MEM with $[P_k]MEM, @P_i$
(10) push(P_i, P_i, T)
(11) i=i+1

Figure 7.14: Rewriting memory operations: rewriting memory instructions

The original renaming algorithm performs a top-down traversal of the dominator tree. The visit to a node processes the statements associated with the node in a sequential order, starting with any ϕ-function that may have been inserted. Each variable is associated with a stack, keeping the current version number on the top. The right hand side variable of each statement is renamed by the top of the corresponding stack. The left hand side variable is given a new version number which then is pushed into the corresponding stack. Figure 7.14 shows the modification on the renaming phase. Corresponding to the three ϕ-functions at the converge nodes, each memory operation is associated with a tuple, namely $< P_j, P_k, P_l >$. P_j and P_l are for the next load and P_k is for the next store. Note that i keeps the current predicate version number and gets updated for each memory operation. Rewriting the load operations is performed by line(4) and the new predicate after a load is

computed by line (5). Rewriting store operations is performed by line (9) of the algorithm. We employ a similar algorithm for handling function calls. Because of their side effects such as input/output, function calls may not be reordered without a proper analysis of the functions referenced. Therefore, we introduce a single predicate for each call instruction which is set when the call is executed. A single ϕ node is needed at merge steps to enforce the function call order on any path.

7.5 Directly Computing RFPF

As discussed at the beginning of the chapter, RFPF can be constructed on any single assignment form. Instead of starting with an SSA program, we can start with FGSA. Unlike ϕ-functions in SSA, gating functions in FGSA are not tied to the program points and thus they can be split, moved and merged back the same way as any executable instruction. FGSA contains fewer gating functions which will results in a smaller RFPF. More importantly, T_R transformation eliminates irreducible loops, which enables the RFPF computation algorithms to handle general graphs. Intuitively, FGSA already contains the part of information required for computing RFPF, which is collected through T1/T2/T_R. Next we are going to demonstrate how to directly compute RFPF from a multi-assignment form through the concept of congruences classes and T1/T2/T_R.

Given any instruction, computing its RFPF format requires computing the path predicate expressions from *start* to this particular instruction, constructing gating functions for its operands if necessary and renaming its operands. Constructing gating functions and renaming operands are the exact same tasks as they are in FGSA, which are achieved by the set of algorithms in Chapter 3. In FGSA, we compute partial path predicate expressions representing the path from LCDOM of a CC's definition set to a definition. In RFPF, a full path predicate expression which represents the path from the *start* node to any definition (instruction) must be computed, which will be used to guard the instruction in the complete RFPF. The complete path predicate expression can be obtained through T2 transformation with an extra computation. Consider an instruction in node v. When T2 is performed on

```
Pred(v) = ∅
Link(v) = ∅
RP(v) = ∅
Loop(v) = ∅
while(there is a node except start left) do
if (T1 is performed on v)
  let path predicate expression at back-edge be p
  RP(v) = p
  Loop(v) = {v}
  for(w is dominated by v)
     if(Pred(w) is not empty AND
       Pred(w) ∧ RP(v) is not false)
       add w into Loop(v)
  for (w in Loop(v))
     convert each instruction in w into recursive form with RP(v)
if (T2 is performed on v with predecessor u)
  let path predicate expression at edge (uv) be q
  Pred(v) = q
  Link(v) = u
endo
Traverse each node v in the dominator tree in pre-order
  Pred(v) = Pred(v) ∧ Pred(Link(v))
```

Figure 7.15: Algorithm 4: Directly computing RFPF

v, path predicate expression which represents the path from u to v was already computed and pushed onto edge (uv). It is easy to see that u must dominate v. We can create a link between u and v. Later when u is consumed by T2, the path predicate expression from u's dominator to u will be computed and u and the dominator is also linked. Finally by concatenating the path predicate expressions following the links in the dominator tree from v all the way back to *start*, the complete path predicate expression is computed.

For an instruction in a cyclic region, computing its RFPF format requires computing the recursive predicate. In fact, when T1 is performed, the path predicate expression at the back-edge is the recursive predicate for the loop. Then it is easy to transform a loop instruction into its recursive form. As discussed before, a gating function with a read-once predicate can be used at the loop headers. However if an instruction is below and outside the loop, it shouldn't be transformed into the recursive form. In Section 3.4.2, we demonstrated

that we can identify whether a node is inside or outside a loop by testing the result of its path predicate (to the loop header) AND the recursive predicate.

Based on T1/T2/T_R presented in Chapter 3, we present complete path predicate expression and recursive predicate computation in Figure 7.15. Note that T1/T2/T_R also computes congruence classes and constructs gating functions, therefore direct computing RFPF from a multi-assignment form is complete.

7.6 Conclusion

We have presented a new approach, Recursive Future Predicated Form to program representation and optimization. The most significant difference of our approach is to move instructions to collect the necessary data and control flow information, and in the process yield a representation in which compiler optimizations can be carried out. RFPF representation is a complete framework which exposes optimizations which are only possible through code restructuring. In the next Chapter, we visit PRE under this representation.

Chapter 8

Optimizations on RFPF

[1] Many optimizations can be carried out on the *complete RFPF* and as well as during the transformation process. One of the advantages of RFPF is its ability to perform traditional optimizations while keeping the graph in single-assignment form with minimal book keeping. We show two examples of optimizations, one which can be employed during the transformation and another after the graph is converted into full RPFP.

Case Study 1. *PRE during the transformation:*

Consider Figure 8.1(a). There's a redundant computation of $x_0 + y_0$ along the path (B2 B4 B5). Most PRE algorithms cannot capture this redundancy because node B4 destroys the available information for $x_0 + y_0$. On the other hand, instruction propagation and RFPF cover the case. Observe that during the instruction propagation, one of the clones, namely, ($I1$) reaches node B2(Figure 8.1(b)). By applying *Value numbering* (Aho et al. 1986) in the basic block, $x_0 + y_0$ in I1 is subsumed by z_1(Figure 8.1(c)).

By further propagating and merging, instruction I1 and I2 are merged in B1 with the addition of the gating function ψ (Figure 8.2(a)) yielding the complete RFPF (Figure 8.2(d))).

Figure 8.2(b) gives the result of transforming RFPF back into SSA. This graph is

[1]The material contained in this chapter was previously published in CC'10/ETAPS'10 Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction.

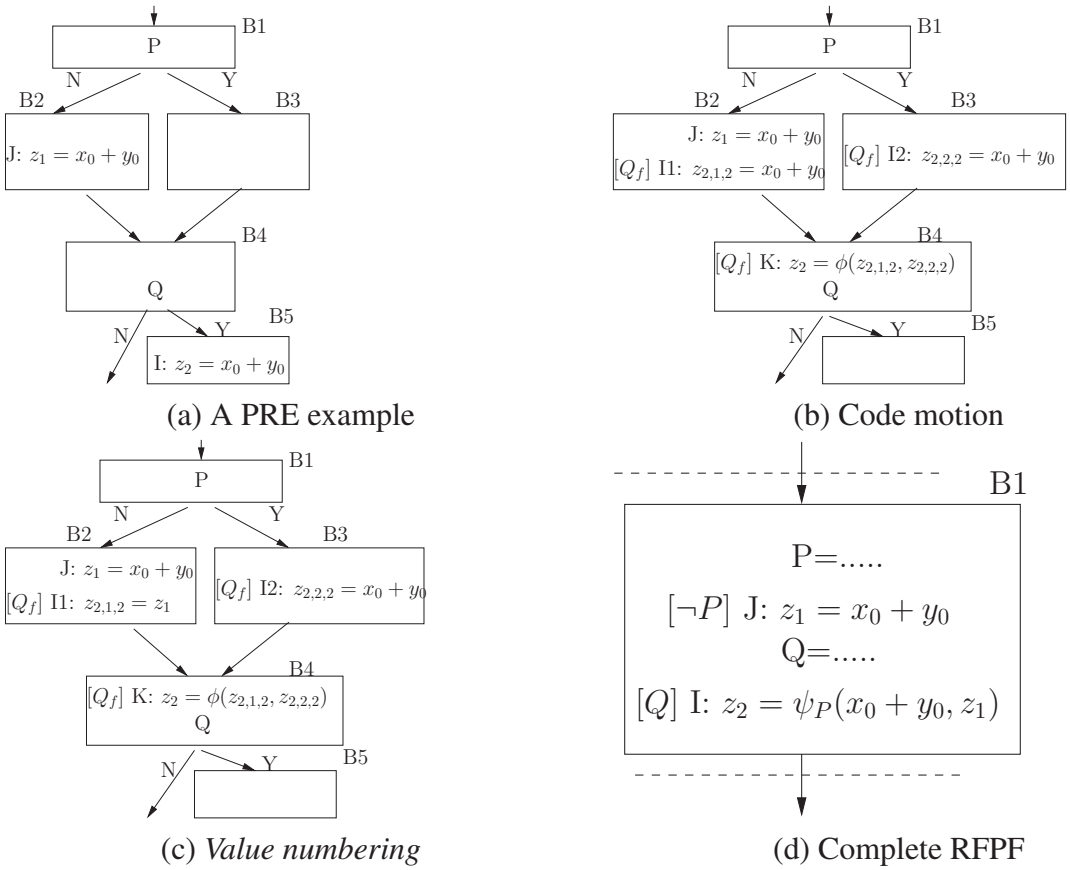


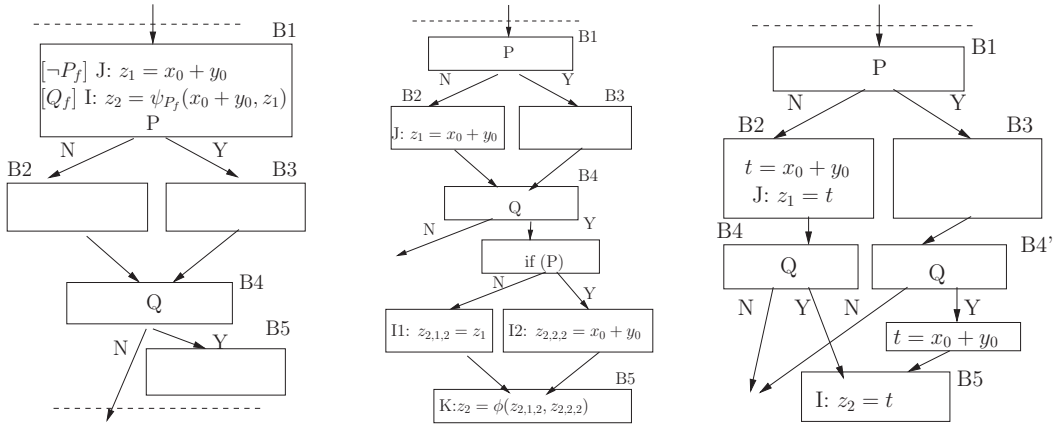
Figure 8.1: Partial redundancy elimination during the code motion

functionally equivalent to Figure 8.2(c), which shows the result by using the PRE algorithm of Bodik et al. (Bodík et al. 1998). This algorithm separates the expression available path from the unavailable path by node cloning which eliminates all redundancies. As it can be seen, RFPF can perform PRE and keep the resulting representation in the SSA form.

The dependency elimination in our example is not a coincidence. By splitting instructions into copies, we naturally split the dataflow information available path from unavailable path. From the perspective of the total number of the computations, RPFP yields essentially the same result. The optimality of RPFP and code motion based PRE in RFPF is yet to be studied, but its ability to catch difficult PRE cases is quite promising.

Case Study 2. *Constant propagation in* complete RFPF*:*

We use another example(Figure 8.3(a)) to show how to do constant propagation(CP) in

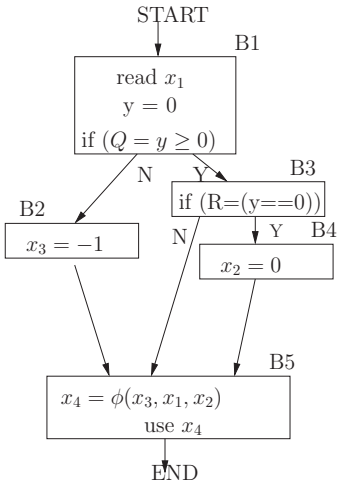


(a) Instruction merging (b) RFPF to CFG (c) Bodik et al. (Bodík et al. 1998)

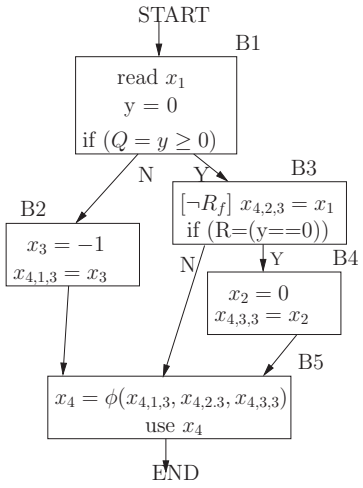
Figure 8.2: Merging and converting back to CFG

complete RFPF. As in the PRE example, constant propagation chances are caught in node B2 and B4(Figure 8.3(b)). Figure 8.3(c) and (d) shows complete RFPF of the program and the result after optimization. We use the conditional constant propagation(CCP) approach described in (Wegman and Zadeck 1991). Note that x_4 becomes a constant in our representation because gating function ψ can be evaluated given the constant information of the predicate and the variable values.

The choice of applying various optimizations during or after the transformation has to be decided based on foreseen benefits. This is an open research problem and is left as future work.



(a) A CP Example



(b) Transform to RFPF

read x_1
$y = 0$
$Q = (y \geq 0)$
$Q : R = (y == 0)$
$Q \wedge R : x_2 = 0$
$\neg Q : x_3 = -1$
$x_4 = \psi_Q(\psi_R(x_2, x_1), x_3)$
use x_4

(c) Complete RFPF

read x_1
$y = 0$
$Q = true$
$true : R = true$
$true : x_2 = 0$
$false : x_3 = -1$
$x_4 = \psi_{true}(\psi_{true}(0, x_1), x_3)$
use $0(x_4)$

(d) Apply CCP

Figure 8.3: Constant propagation on RFPF

Chapter 9

Conclusion

This dissertation explores the field of single assignment forms beyond SSA and GSA and improves the state-of-the-art on several points, including use of interval analysis to compute single assignment form, elimination of irreducibility without node replication as well as the provision of a framework which can potentially subsume optimizations which require program restructuring.

9.1 Summary of Work

Chapter 3 presents FGSA, a congruence class based single assignment form. We employ interval analysis T1/T2 to collect the information necessary for FGSA construction. We introduce T_R transformation to eliminate irreducible loops without node replication, which enables interval analysis to handle both reducible and irreducible programs. FGSA representation facilitates expected linear time conversion of programs from a control-flow graph, yields the same semantics as SSA and GSA by using fewer gating functions and provides executable semantics with extra information in the form of path expression. The information embedded in FGSA can be used to simplify analysis and optimizations, which is demonstrated in Chapter 4 and Chapter 6 respectively. In Chapter 4, we've extended the concept of liveness on FGSA with respect to predicates and congruence classes. By doing so, we are able to classify interferences among variables into categories and look

for solutions for each category. In Chapter 6 we've adapted two optimizations algorithms on FGSA to show that existing optimization algorithms are easy to be adapted to FGSA and have good chance to be simplified on FGSA. Based on the idea, in Chapter 5, we've designed a framework of optimal translation from FGSA. We first present the taxonomy of the interferences resulting from optimizations and then presented solutions for each category. To our knowledge this is an approach that has never been investigated to solve single assignment inverse transformation problem.

Chapter 7 presents RFPF, an novel approach to program representation and optimization. We've presented complete sets of algorithms to move code in acyclic regions as well as cyclic regions, including memory instructions and function calls. The property of unrestricted code motion of RFPF is very powerful. In Chapter 8 we've demonstrated the ability of RFPF through two optimization problems.

9.2 Future Research

This dissertation work can be extended to the following areas:

1. A generalized framework to adapt existing analysis and optimizations onto FGSA and RFPF;

2. Development of provably optimal inverse transformation algorithms for gated-gated interferences in FGSA;

3. Development of optimized inverse transformation algorithms for RFPF and investigation of whether RFPF can subsume program structuring based optimization;

4. A new design of architecture to directly execute FGSA/RFPF.

References

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers: principles, techniques, and tools*. 2nd Ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Allen, J. R., K. Kennedy, C. Porterfield, and J. Warren. 1983. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, pp. 177–189. ACM.

Ananian, C. S. and M. Rinard. 1999. *Static single information form.* Master's thesis. Massachussets Institute of Technology.

Arenaz, M., J. Touriño, and R. Doallo. 2003. A gsa-based compiler infrastructure to extract parallelism from complex loops. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, New York, NY, USA, pp. 193–204. ACM.

Aycock, J. and N. Horspool. 2000. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, Volume 1781 of *Lecture Notes in Computer Science*, pp. 110–125. Springer.

Bernstein, D. and M. Rodeh. 1991. Global instruction scheduling for superscalar machines. *SIGPLAN Not. 26*(6): 241–255.

Bilardi, G. and K. Pingali. 2003. Algorithms for computing the static single assignment form. *J. ACM 50*(3): 375–425.

Bodík, R., R. Gupta, and V. Sarkar. 2000. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, New York, NY, USA, pp. 321–333. ACM.

Bodík, R., R. Gupta, and M. L. Soffa. 1998. Complete removal of redundant expressions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, New York, NY, USA, pp. 1–14. ACM.

Boissinot, B., A. Darte, F. Rastello, B. D. de Dinechin, and C. Guillon. 2009. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, Washington, DC, USA, pp. 114–125. IEEE Computer Society.

Boissinot, B., S. Hack, D. Grund, B. Dupont de Dine hin, and F. e. Rastello . 2008. Fast liveness checking for ssa-form programs. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, New York, NY, USA, pp. 35–44. ACM.

Brandis, M. M. and H. Mössenböck. 1994, Nov. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems. 16*(6): 1684–1698.

Briggs, P., K. D. Cooper, T. J. Harvey, and L. T. Simpson. 1998, Jul. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience. 28*(8): 859–881.

Carter, L., J. Ferrante, and C. Thomborson. 2003. Folklore confirmed: reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, New York, NY, USA, pp. 106–114. ACM.

Choi, J.-D., R. Cytron, and J. Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, pp. 55–66. ACM.

Chow, F., S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. 1997. A new algorithm for partial redundancy elimination based on ssa form. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, New York, NY, USA, pp. 273–286. ACM.

Click, C.. 1995. Global code motion/global value numbering. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, New York, NY, USA, pp. 246–257. ACM.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*(4): 451–490.

Das, D. and U. Ramakrishna. 2005, May. A practical and fast iterative algorithm for ϕ-function computation using dj graphs. *ACM Trans. Program. Lang. Syst. 27*: 426–440.

D.Cooper, K., T. J.Harvey, and K. Kennedy. 2001. A simple, fast dominance algorithm. *Softw. Pract. Exper.*.

Dhamdhere, D. M. and H. Patil. 1993, April. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Trans. Program. Lang. Syst. 15*: 312–336.

Ding, S. and S. Önder. 2010. Unrestricted code motion: A program representation and transformation algorithms based on future values. In R. Gupta (Ed.), *Compiler Construction*, Volume 6011 of *Lecture Notes in Computer Science*, pp. 26–45. Springer Berlin / Heidelberg. 10.1007/978-3-642-11970-5_3.

Erosa, A. and L. J. Hendren. 1994. Taming control flow: A structured approach to eliminating goto statements. In *In Proceedings of 1994 IEEE International Conference on Computer Languages*, pp. 229–240. IEEE Computer Society Press.

Ferrante, J., K. J. Ottenstein, and J. D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*(3): 319–349.

Fisher, J. A.. 1979. *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources.* Ph. D. thesis, New York, NY, USA.

Fisher, J. A.. 1982, July. Trace scheduling: A technique for global microcode compaction. *IEEE Transcations on Computers c-30*: 4778–490.

Gargi, K.. 2002. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, New York, NY, USA, pp. 45–56. ACM.

Gavril, F.. 1972. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *Siam Journal on Computing 1*: 180–187.

Gillies, D. M., D.-c. R. Ju, R. Johnson, and M. Schlansker. 1996. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, Washington, DC, USA, pp. 114–125. IEEE Computer Society.

Graham, S. L. and M. Wegman. 1976, January. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*: 172–202.

Hack, S.. 2005, June. Interference Graphs of Programs in SSA Form. Technical Report 2005-15, Universität Karlsruhe.

Hack, S.. 2007, October. *Register Allocation for Programs in SSA Form*. Ph. D. thesis, Universität Karlsruhe.

Hailperin, M.. 1998. Cost-optimal code motion. *ACM Trans. Program. Lang. Syst. 20*(6): 1297–1322.

Havlak, P.. 1993. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Volume 768 of *Lecture Notes in Computer Science*, pp. 477–499. Springer.

Hecht, M. S. and J. D. Ullman. 1974, July. Characterizations of reducible flow graphs. *J. ACM 21*: 367–375.

Hoflehner, G. F.. 2010. *Strategies for Predicate-Aware Register Allocation*, Volume 6011, pp. 185–204. Springer.

Janssen, J. and H. Corporaal. 1997. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst. 19*(6): 1031–1052.

Kennedy, R., S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. 1999, May. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems 21*(3): 627–676.

Kennedy, R., F. C. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. 1998. Strength reduction via ssapre. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, London, UK, pp. 144–158. Springer-Verlag.

Knoop, J., O. Rüthing, and B. Steffen. 1992. Lazy code motion. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, New York, NY, USA, pp. 224–234. ACM.

Knoop, J., O. Ruthing, and B. Steffen. 1993. Lazy strength reduction. *Journal of Programming Languages 1*: 71–91.

Knoop, J., O. Rüthing, and B. Steffen. 1994. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst. 16*(4): 1117–1155.

Lengauer, T. and R. E. Tarjan. 1979, Jul. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems 1*(1): 121–141.

Morel, E. and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM 22*(2): 96–103.

Önder, S.. 2010. Methods and systems for ordering instructions using future values. *US. Patent* 7,747,993, Filed Dec. 2004, Issued Jun. 2010.

Ottenstein, K. J., R. A. Ballance, and A. B. MacCabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not. 25*(6): 257–271.

Pingali, K., M. Beck, R. C. Johnson, M. Moudgill, and P. Stodghill. 1990. Dependence flow graphs: An algebraic approach to program dependencies. Technical report, Cornell University, Ithaca, NY, USA.

Rau, B. R.. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, New York, NY, USA, pp. 63–74. ACM.

Rau, B. R. and C. D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl. 12*(4): 183–198.

Rogers, A. and K. Li. 1992. Software support for speculative loads. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, pp. 38–50. ACM.

Rosen, B. K., M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, pp. 12–27. ACM.

Simpson, L. T.. 1996. *Value-driven redundancy elimination*. Ph. D. thesis, Houston, TX, USA. AAI9631092.

Singer, J.. 2003. Ssi extends ssa. In *Work in Progress Session Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques*.

Singer, J.. 2006. *Static program analysis based on virtual register renaming.* PhD's thesis, University of Cambridge. UCAM-CL-TR-660.

Sreedhar, V. C., R. D.-C. Ju, D. M. Gillies, and V. Santhanam. 1999. Translating out of static single assignment form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, London, UK, pp. 194–210. Springer-Verlag.

Tarjan, R. E.. 1981. Fast algorithms for solving path problems. *J. ACM 28*(3): 594–614.

T.Ball and S.Horwitz. 1992. Constructing control flow from control dependence. Technical report, University of Wisconsin-Madison.

Tu, P. and D. Padua. 1995. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pp. 47–55.

Unger, S. and F. Mueller. 2002, July. Handling irreducible loops: optimized node splitting versus dj-graphs. *ACM Trans. Program. Lang. Syst. 24*: 299–333.

VanDrunen, T. and A. L. Hosking. 2004. Anticipation-based partial redundancy elimination for static single assignment form. *Softw. Pract. Exper. 34*(15): 1413–1439.

VanDrunen, T. and A. L. Hosking. 2004. Value-based partial redundancy elimination. In *13th International Conference on Compiler Construction*, Volume 2985 of *Lecture Notes in Computer Science*, pp. 167–184. Springer.

Warter, N. J., S. A. Mahlke, W.-M. W. Hwu, and B. R. Rau. 1993. Reverse if-conversion. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, New York, NY, USA, pp. 290–299. ACM.

Wegman, M. N. and F. K. Zadeck. 1991, Apr. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*(2): 181–210.