INTEGRATING SCALAR ANALYSES AND OPTIMIZATIONS IN A PARALLELIZING AND OPTIMIZING COMPILER

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Bixia Zheng

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Professor Pen-Chung Yew, Advisor

February 2000

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral dissertation by

Bixia Zheng

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Professor Pen-Chung Yew

Name of Faculty Adviser

Signature of Faculty Adviser

Date

GRADUATE SCHOOL

ABSTRACT

Scalar analyses and optimizations are indispensable and challenging in parallelizing and optimizing compilers. Scalar analyses statically collect information on how programs manipulate variables. Scalar optimizations, on the other hand, improve the efficiency of the compiled codes and the precision of the analyzed results. Performing scalar analyses and optimizations is challenging, especially when the programming languages being considered support general pointer usage. This is because many existing pointer analysis algorithms are either imprecise or inefficient, and the interactions between analyses and optimizations raise the issue of ordering the analyses and the optimizations.

In this dissertation, we propose a new approach to perform scalar analyses and optimizations. We first analyze both pointer and non-pointer variables interprocedurally. The analyzed results are represented by an extended version of the traditional *static single assignment* (SSA) form. We then perform the optimizations in tandem using the SSA form. Each optimization incrementally updates the SSA form to reflect its refinement to the dataflow information. This incremental update can improve the precision of the analyzed result and expose more optimization opportunity. The final SSA form after the optimizations can be used by later optimization and parallelization.

Our approach improves the efficiency and effectiveness of scalar analyses and optimizations by two means. First, we partition the variables based on their *pointer levels* and analyze the variables one *pointer level* at a time. The *pointer level* of a variable is the maximum level of indirection to the variable. Analyzing one pointer level at a time can avoid the complexity in modeling multiple levels of dependency due to multiple levels of pointers, and thus allows simpler and more efficient algorithms without sacrificing the precision of the analyzed results. Second, in contrast to many existing approaches which interleave optimizations at the procedure level, we interleave optimizations at the

statement level to allow faster response to the interactions among optimizations and to reduce the number of program traversals needed to perform the optimizations.

We implement our approach in the Agassiz Compiler. Using the SPEC benchmark suite and several other pointer intensive C programs, we demonstrate the effectiveness and efficiency of our approach.

ACKNOWLEDGMENTS

I would never have been able to complete this work without the support and contributions of many other people.

I especially thank my academic advisor, Professor Pen-Chung Yew, for funding and directing my research. He not only contributed many valuable ideas to the work in this dissertation but also taught me methods to do research, write papers and give presentations. I feel privileged to have been a student of his.

I would like to thank the other members of the committee for my degree program, Professors David J. Lilja, Mats P.E. Heimdahl, Matthew T. O'Keefe and Wei-tek Tsai, for their time in reviewing my work and for their helpful advice. I would also like to thank Professor Zhiyuan Li, Dr. Yong-Feng Lee, and Dr. Youfeng Wu for reviewing papers I wrote and providing me with valuable advices.

I owe many thanks to the people who designed and support the Agassiz Compiler infrastructure on which this work is built. Jenn-Yuan Tsai, Binyu Zang and Tong Chen participated in the earlier design of the compiler infrastructure and exposed me to a wide range of issues in constructing high-performance compilers. Our cooperating research group in Fudan University spent the majority of effort in designing and implementing the compiler intermediate representation, the parsers for C and FORTRAN, and the cleanup algorithm which establish the base for this work. Jenn-Yuan, Tong Chen and Jun Liang spent lots of effort in setting up our software environment, such as check-in/check-out tools and common makefiles, which makes team development effort possible. Tong Chen, Jun Liang, and Yan Zhen who utilize the results of this work in later phases of the compiler provided me with valuable feedback. Xin Wang, Sangyeun Choi, Qing Zhao, Yonghong Song and Yan Zhen cooperated with me in developing the HLI, which is an interface between the front-end and the back-end compilers.

I thank my family in China for their unfailing support and constant encouragement. Lastly, I thank my husband for his support and encouragement, and my son for lightening my days.

TABLE OF CONTENTS

INTRODUCTION **1**

1.1 The Purpose of Scalar Analyses and Optimizations 2

1.2 The *Phase Ordering Problem* 4

1.3 Existing Approaches 6

1.4 Our Approach 9

1.5 Contributions 10

1.6 Organization 11

THE PROGRAM REPRESENTATION **12**

2.1 Abstract Syntax Trees 12

2.2 Expressions 12

2.3 Statements 14

2.4 Intraprocedural Control-flow 14

2.5 Interprocedural Control-flow: Call Graph 17

2.6 Program Traversal 18

AN EXTENDED STATIC SINGLE ASSIGNMENT FORM **20**

3.1 Background 20

3.1.1 Properties of SSA Forms *20*

3.1.2 Representing Control-flow Joints *22*

3.1.3 The Advantage of SSA Forms *23*

3.2 Motivations for an Extended SSA From 25

3.3 An Extended SSA Form 27

3.3.1 Representing Implicit Variable Accesses *27*

3.3.2 Representing Weak-Updates *28*

3.3.3 Definition Graph *31*

3.3.4 Predicates and β*-functions* *31*

3.4 Other Related Work 35

3.4.1 Location Factored SSA Forms Versus Assignment Factored SSA Forms *35*

3.4.2 A Comparison with Hasti and Horwitz's SSA Form *39*
3.4.3 Partial SSA Forms Versus Full SSA Forms *40*
3.4.4 SSA Forms with Predicates *41*

PARTITIONING THE PROGRAM ANALYSIS USING POINTER LEVELS AND AN SSA FORM **43**

4.1 Background 43
4.1.1 Representing Pointer-Related Information *44*
4.1.2 Flow-Insensitive Versus Flow-Sensitive *45*
4.1.3 Context-Insensitive Versus Context-Sensitive *47*
4.1.4 Partitioned Analysis *48*
4.2 The Idea of Our Partitioned Approach 49
4.3 Basic Concepts 53
4.3.1 Pointer Level *53*
4.3.2 Representing Memory Locations *54*
4.3.3 Naming the Intermediate SSA Forms *57*
4.4 Analyzing One Procedure 57
4.4.1 Analyzing Assignments *58*
4.4.2 Handling CFG Joint Nodes *61*
4.4.3 Pointer Analysis Using an SSA Form *65*
4.4.4 Passing Pointer Values to the Succeeding Pointer Levels *68*
4.4.5 An Example *68*
4.5 Analyzing Multiple Procedures 71
4.5.1 Algorithm Description *71*
4.5.2 Context-Sensitivity *79*
4.5.3 Complexity *86*
4.6 Handle Some Language Specific Features 87
4.7 Other Related Work 88
4.7.1 Comparing with Existing Partitioned Pointer Analysis Approaches *89*
4.7.2 Comparing Points-to Representation with Alias Pair Representation *91*
4.7.3 Using SSA Form in Pointer Analysis *94*

4.7.4 Analyzing Recursive Data Structures 96

INTEGRATING OPTIMIZATIONS USING AN SSA FORM 101

5.1 The Considered Optimizations 101

5.2 Calculating Values for Definitions 102

5.2.1 An Iterative Approach 104

5.2.2 A Demand-Driven Approach 105

5.2.3 An Improved Demand-Driven Approach 108

5.3 Integrating Optimizations 109

5.4 Interactions Affect Ordering 110

5.5 The Optimization Algorithm 114

5.6 An Example 115

5.7 Related Work 118

5.7.1 Integrated Optimizations Using SSA Forms 119

5.7.2 Updating SSA Forms During Optimizations 121

5.7.3 Integrate Optimizations Using Other Intermediate Representations 121

5.7.4 A Framework to Explore the Interactions among Optimizations 122

EXPERIMENTAL EVALUATION 124

6.1 Benchmark Program Description 124

6.2 Evaluating the Partitioned Analysis Algorithm 125

6.2.1 Analysis Times 125

6.2.2 Analysis Results 127

6.2.3 Evaluating the Two Methods to Achieve Context-Sensitivity 129

6.2.4 Calibrate Our Pointer Analysis with Yong et. al.'s Analysis Results 133

6.3 Evaluating the Integrated Scalar Optimizations 134

6.3.1 Optimization Times 134

6.3.2 Counting the Number of Optimizations 135

6.3.3 Comparing with a Non-integrated Approach 136

6.3.4 The Effect of the Optimizations 140

CONCLUSIONS AND FUTURE WORK 143

7.1 Applications of the Analysis Result 144
7.2 Future Work 145

BIBLIOGRAPHY 148

LIST OF TABLES

Table 1: The benchmark programs and some important characteristics of the programs (rec. DS = recursive data structure, max. PL = maximum pointer level) 125
Table 2: The analysis time (in seconds). 126
Table 3: Statistics about the precision of the analyzed result and the SSA form with explicit alias and *Ref/Mod* information (ANT = average number of targets).......... 128
Table 4: Parameterize procedures' modification effect (*Mod*) to pointers...................... 129
Table 5: The effect of the parameter *m* (*: for programs *m88ksim* and *vortex*, we are not able to complete the analysis and collect the statistics in one day with an unrestricted *m*.)... 131
Table 6: The effect of the parameter *m* on the analysis time (in seconds). 132
Table 7: Calibrate our pointer analysis algorithm with Yong et. al.'s analysis results [YHR99] for three large programs. 133
Table 8: Optimization time (in seconds)... 135
Table 9: Optimizations performed on each programs (CE=useless code elimination, IR=indirection removal, CP=constant propagation, CF=constant folding, BE=branch elimination, NZL=non-zero iteration loop recognition)... 136
Table 10: The integrated approach is faster (speed up = $T_{\text{non-integrated}}$: $T_{\text{integrated}}$). 137
Table 11: The integrated approach finds more optimizations. 138
Table 12: The integrated approach is less sensitive to the optimization orders. 139
Table 13: The optimizations can reduce the number of *use-def* links represented by the SSA forms. 140
Table 14: The optimizations can reduce the number of assembler instructions generated for the benchmark programs... 141

LIST OF FIGURES

Figure 1: An overview of the Agassiz Compiler 3

Figure 2: An example to illustrate the *phase ordering* problem 5

Figure 3: Comparing three approaches to implement a group of optimizations 7

Figure 4: An example program and its abstract syntax tree representation. 13

Figure 5: The control-flow structures for *if* statements, loops and *unstructured regions*. 15

Figure 6: An example program and its SSA form. 21

Figure 7: Comparing an SSA form with a *use-def* representation using an example program. 23

Figure 8: Missing pointer alias information may cause incorrect optimizations. 25

Figure 9: Missing interprocedural *Ref/Mod* information may cause incorrect optimizations. 26

Figure 10: Examples for incrementally updating SSA forms during optimizations. 26

Figure 11: An example to illustrate definition graphs. 32

Figure 12: The extended SSA form (with predicates) for the program in Figure 11(a) and the definition graph for the definitions. 33

Figure 13: The truth table for predicate R_{may}. 34

Figure 14: The truth table for predicate R_{must}. 34

Figure 15: Comparing a location factored SSA form with an assignment factored SSA form using the example program in Figure 8(a). 37

Figure 16: Comparing an n-way branch approach with a 2-way branch approach using indirect assignment "*p = …" (assumption: pointer *p* may point to *a*, *b*, and *c*; definitions a_0, b_0, and c_0 reach the indirect assignment). 39

Figure 17: Comparing a full SSA form with a partial SSA form using an example program. 41

Figure 18: Two example programs for comparing flow-insensitive pointer analysis with flow-sensitive pointer analysis. 46

Figure 19: An example to illustrate that pointers complicate a context-sensitive analysis.48

Figure 20: An example program and the dependence among the analysis of its variables.50
Figure 21: Examples for representing memory locations for scalar, array and structure variables.......... 55
Figure 22: An example for representing the memory locations for a union variable........ 56
Figure 23: The algorithm to analyze a program with one procedure. 57
Figure 24: Calculating the *l*-level MemObjs accessed on the left-hand side of an assignment.. 60
Figure 25: The algorithm to insert β-functions in CFG joint nodes (procedure *Handle_Procedure_Joint*). 61
Figure 26: The algorithm to insert β-functions in CFG joint nodes (procedure *Handle_Joint*). 62
Figure 27: An algorithm to calculate pointer values using the SSA form.......... 66
Figure 28: Partitioning the SSA form construction for the program in Figure 20(a) (Note: (1) We assume a_0 and b_0 represent the initial values of a and b respectively; (2) We use " $pp_4 = pp_3$" to represent that pp_4 has the same β-function on its RHS as pp_3 does. Similar notation is also used for p_5 and q_5.). 69
Figure 29: Calculating pointer values for variables at pointer level 1 using the SSA form in Figure 28(b) (S=START, E=END, G=GOING, M=MergeWith). 70
Figure 30: Overview of the interprocedural algorithm (assume source programs without some language specific features). 72
Figure 31: An example program and its intermediate SSA forms. (Note: we do not show the *exit-node* of *main*, because it is not necessary to calculate the *Ref/Mod* for main. We avoid putting an unused initial definition to the *entry-node* of a procedure)...... 75
Figure 32: The SSA form for the program in Figure 31(a) (after analyzing *p* and *q*). (Note: we will show the predicates of the β-functions later.).......... 77
Figure 33: The final SSA form for the program in Figure 31(a). (Note: we will show the predicates of the β-functions later.).......... 78
Figure 34: An improved SSA form of the SSA form in Figure 33. 83
Figure 35: An example program for type casts. 88

Figure 36: Comparing the points-to representation with the alias pair representation (from [EGH94]). 92
Figure 37: An example program with accesses to recursive data *structures*.... 93
Figure 38: The points-to representation may infer spurious aliases. 93
Figure 39: An example program and its SSA forms created by Hasti and Horwitz's pointer analysis algorithm. 95
Figure 40: A recursive data structure type and an aggregate of its data structures. 97
Figure 41: A standard 3-level lattice and its meet operator. 103
Figure 42: A working list based iterative algorithm to calculate values for definitions. 104
Figure 43: An example program to illustrate the working-list based algorithm. 104
Figure 44: A demand-driven algorithm to calculate values for definitions. 106
Figure 45: An example program and a *use-def* link cycle in the program. 107
Figure 46. Optimizing statement s_i exposes new opportunities for optimizing its succeeding statement s_{i+} and its preceding statement s_{i-} 112
Figure 47: The algorithm to optimize a procedure. 113
Figure 48: The utility functions used by the algorithm in Figure 47. 115
Figure 49: An example program to illustrate the integrated optimization algorithm. 116
Figure 50. The un-optimized and optimized SSA forms for the program in Figure 49. . 117

CHAPTER 1

INTRODUCTION

This dissertation discusses techniques to improve the efficiency and effectiveness of scalar analyses and optimizations in an integrated parallelizing and optimizing compiler. By *scalar analysis*, we refer to a dataflow analysis that does not distinguish the elements in an array. This is in contrast to *array dataflow analysis*. Scalar optimizations refer to the traditional optimizations, such as constant propagation, that are not parallelism- or cache-oriented [Muc97].

Scalar analyses and optimizations are indispensable in a parallelizing and optimizing compiler. The goal of scalar analyses is to statically collect information on how a program manipulates its data. This information is extremely valuable for other analyses and optimizations. Its accuracy can potentially affect the effectiveness of almost all compiler optimizations and programming tools. Scalar optimizations, such as unreachable code elimination, on the other hand, can improve the efficiency of the compiled codes as well as the precision of the analyzed result.

Performing scalar analyses and optimizations in a parallelizing and optimizing compiler is very challenging, especially when the programming language being considered supports arbitrary pointer usage. First, the presence of pointers complicates scalar analyses because a pointer dereference may potentially access any memory location. Many existing pointer analysis algorithms, which are used to determine the memory locations that may be accessed by a pointer dereference, are either imprecise or inefficient. The effectiveness of many scalar optimizations is often inhibited by such impreciseness. Second, the analyses and the optimizations interact with each other in the sense that an analysis provides information for an optimization, and an optimization, on the other hand, can also refine the analyzed result. Furthermore, performing an optimization can also expose opportunity for further optimization. These interactions make it difficult to determine an optimal ordering among the analyses and the optimizations, so called the *phase ordering problem* [Bin94, CC95, Cli95, WS97]. To achieve good results, we often have to repeat the analyses and the optimizations multiple times. Finding a way to exploit the interaction and to avoid the expensive repetition is important and challenging.

In this dissertation, we present a new algorithm to improve the efficiency and effectiveness of scalar analyses and optimizations. This algorithm is implemented in the Agassiz Compiler [ZTZ99+], which is an integrated parallelizing and optimizing compiler that can compile both FORTRAN77 and C programs for concurrent multithreaded architectures [TY96].

1.1 The Purpose of Scalar Analyses and Optimizations

We first highlight the purpose of performing scalar analyses and optimizations in the Agassiz Compiler. Figure 1 shows the compiler infrastructure. It consists of a parallelizing front-end and an optimizing back-end. The front-end compiler can take a C or FORTRAN source program. After a parsing stage, the program is transformed into an abstract syntax-tree like intermediate representation (IR). The IR created in the parsing stage is then normalized by a cleanup phase. More specifically, the cleanup phase eliminates some language specific features, such as side-effect expressions and multiple entries of a procedure. The cleanup phase also eliminates irregular control-flow and transforms the program into an equivalent structured program. After the cleanup phase, we perform scalar analyses and optimizations. The resulting improved IR with annotated dataflow information is used by the parallelization phase for parallelism-oriented optimizations and parallel code generation. Some dataflow information, including pointer-induced alias information, loop-carried data dependence information and interprocedural reference and modification (*Ref/Mod*) information are also exported to the back-end compiler via a platform independent format call High-Level Information (HLI) [CTS+98]. Thus, the front-end compiler generates two outputs: an annotated parallel code for concurrent multithreaded architectures [TY96] and a HLI file.

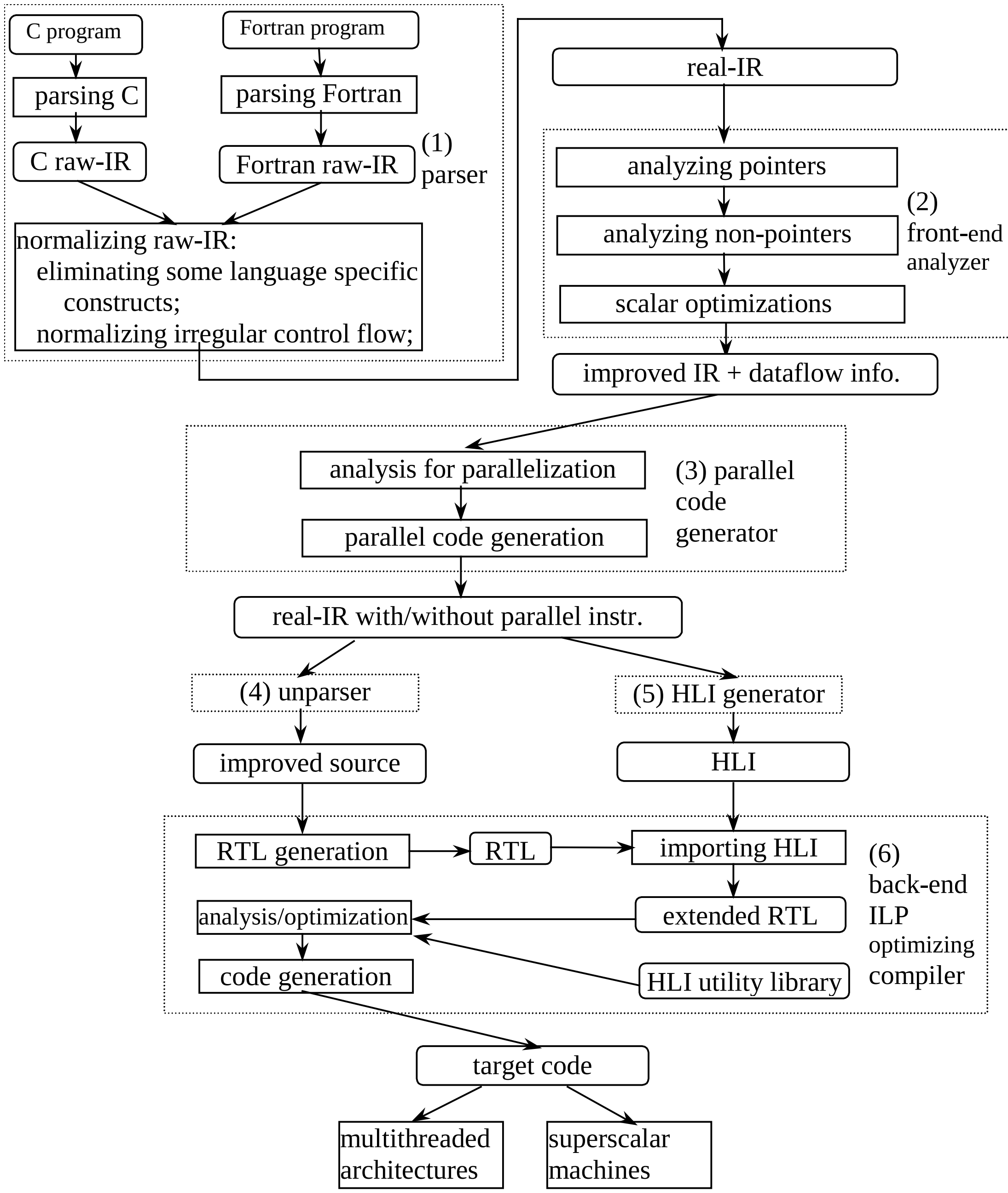


Figure 1: An overview of the Agassiz Compiler

The back-end compiler, which is currently a modified version of GCC, reads in the annotated parallel codes and creates a RTL representation for the program. It also imports the HLI information into the RTL and uses the information to perform instruction level optimizations. Utility functions are provided to maintain the correctness of the HLI information during the optimization. The final target code can be either a parallel code or a sequential code. The parallel version of the target code can be fed into a simulator for concurrent multithreaded architectures while a sequential version of the target code can be executed on a real superscalar machine.

In the Agassiz Compiler, scalar analyses and optimizations are performed right before the parallelization phase. There are two main tasks in this scalar phase: to perform interprocedural analysis on both pointer variables and non-pointer variables, and to perform some traditional code improvement transformations. The analyses collect important information including pointer-induced alias information, variable *use-def* information and interprocedural *Ref/Mod* information. The collected information is used by the compiler in two ways: supporting parallelization in the front-end and exporting to the back-end via HLI to improve the effectiveness of the back-end optimizer. The optimizations, such as constant propagation and unreachable branch elimination, not only can improve the efficiency of the compiled code but also improve the preciseness of the analyzed results. Thus, the purpose of the scalar analysis and optimization phase in the Agassiz Compiler is to obtain high quality dataflow information for later phases in the compiler and to improve the efficiency of the compiled code.

1.2 The *Phase Ordering Problem*

The *phase ordering problem* refers to the difficulty in determining an optimal order for a group of analyses and optimizations [Bin94, CC95, Cli95, WS97]. This is because the analyses and optimizations interact with each other: one analysis/optimization may provide more information for another analysis/optimization. To achieve a good result thus often requires repetitive analyses and optimizations.

We illustrate this phase ordering problem in Figure 2. We give an example program in Figure 2(a). An optimized version of the example program is shown in Figure 2(b). In order to obtain the optimized code, we analyze the program and perform a group of optimizations based on the analyzed result. The involved optimizations, as shown in Figure 2(c), include constant propagation, constant folding, unreachable branch elimination, indirection removal and useless code elimination. Of these optimizations, indirection removal is specific to C programs and all the other optimizations are common to both C and FORTRAN programs. Indirection removal refers to an optimization to replace a pointer dereference, such as **p* in statement *s8*, with the definitely points-to target *a* of the pointer *p*. Chow et. al. also perform indirection removal in their compiler [CCL+96].

(a) An example program

```
s1:  a = 1;
s2:  b = 2;
s3:  c = a + b;
s4:  if ( c > 2 )
s5:      p = &a;
s6:  else
s7:      p = &b;
s8:  *p = 3;
s9:  c = a + b;
//use a, b, c, p.
```

(b) An optimized version of the program

```
s2:  b = 2;


s5:     p = &a;


s8:  a = 3;
s9:  c = 5;
….
```

(c) The interactions among the analyses and the optimizations.

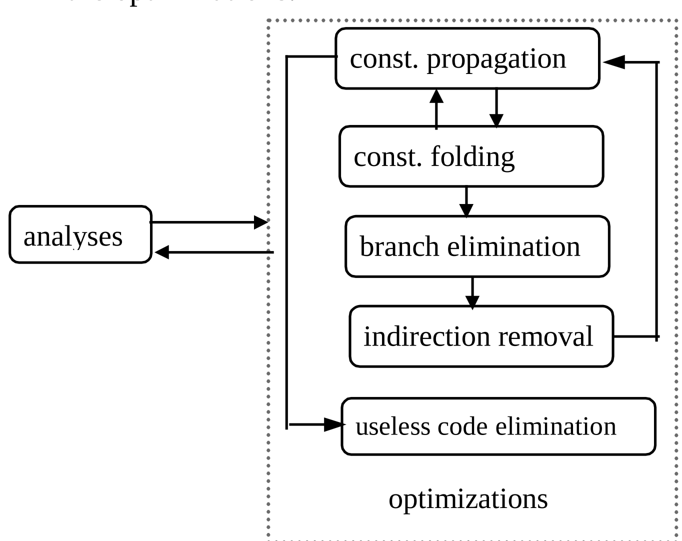


Figure 2: An example to illustrate the *phase ordering* problem

Assume pointer analysis and *use-def* analysis are performed in order to collect the necessary information for the above optimizations. Figure 2(c) describes the interaction among the analyses and the optimizations. In Figure 2(c), an arrow from an optimization

A to another optimization *B* represents that *A* may expose opportunity to *B*. Similarly, an arrow from an analysis to an optimization indicates that the analysis provides the necessary information for the optimization. An arrow from an optimization to an analysis represents that the optimization can refine the analyzed result. A cycle in Figure 2(c) indicates the difficulty in ordering the analyses and the optimizations, and to exploit the interaction without repeated analyses and optimizations.

1.3 Existing Approaches

In general, there are two approaches to implement a group of optimizations and to address the phase ordering problem: a *non-integrated* approach and an *integrated* approach. A non-integrated approach performs individual optimizations successively at the program level. It often implements different optimizations separately, each with its own special-purpose dataflow analysis. Some dataflow information, such as pointer alias information and interprocedural *Ref/Mod* information, are required for most of the optimizations. Thus, a dataflow analysis that collects the common useful information may be needed before doing any optimization. Figure 3(a) shows one way to transform the example program in Figure 2(a) into the optimized program in Figure 2(b), using a non-integrated algorithm which has a dataflow analysis to collect pointer alias information. As can be seen, it requires *ten* program traversals to achieve the optimized code. The analysis is repeated in order to collect the finer pointer alias information exposed by unreachable branch elimination. Optimizations, such as constant propagation and constant folding, are also repeated to take advantage of the new opportunity exposed by other optimizations.

Both GCC and SUIF-1 take a similar non-integrated approach. Moreover, their approaches to scalar optimizations are conservative comparing to the one we have just described because they do not make use of pointer alias and interprocedural *Ref/Mod* information. Thus, both of these two compilers have difficulties in transforming the program shown in Figure 2(a) into the optimized code given in Figure 2(b).

(a) A non-integrated approach

Program traversal	Action
1.	analysis
2.	constant propagation
3.	constant folding
4.	constant propagation (2)
5.	branch elimination
6.	analysis (2)
7.	indirection removal
8.	constant propagation (3)
9.	constant folding (2)
10.	useless code elimination

(b) An integrated approach without incremental updates

Program traversal	Action
1.	analysis
2.	constant propagation
	constant folding
	branch elimination
3.	analysis (2)
4.	indirection removal
	constant propagation (2)
	constant folding (2)
	useless code elimination

(c) Our integrated approach with incremental updates

Program traversal	Action
1.	analysis
2.	constant propagation
	constant folding
	branch elimination
	indirection removal
	useless code elimination

Figure 3: Comparing three approaches to implement a group of optimizations

A non-integrated approach has two main advantages. First, it provides the flexibility to order and to selectively repeat the analyses and optimizations in a high-level driver. Second, a special-purpose dataflow analysis algorithm is often simpler than a general-purpose dataflow analysis algorithm. For example, a simple constant propagation algorithm may only keep a table for the current available constants. It updates the table while handling assignment statements. It also invalidates all the constant in the table when it encounters an indirect assignment or function call statement if pointer-induced alias information and interprocedural modification information are not available.

However, the main problem of a non-integrated approach is its inefficiency. Performing optimizations in separated program traversals causes inefficiency for two reasons. First, if two optimizations do not affect each other, performing both optimizations in one program traversal reduces the time in traversing the program, but achieves the same optimized code. Second, if two optimizations interact each other, performing both optimizations in one program traversal and interleaving them at a statement level allows faster response to the interaction and has the potential to reduce the number of program traversals to obtain the same result. This can be illustrated as follows. Assume there are three statements in a program: *s1*, *s2* and *s3*. Also assume that performing an optimization A on statement *s1* exposes new opportunity to perform an optimization B on statement *s2*; performing an optimization B on statement *s2*, in turn, exposes new opportunity to perform an optimization A on statement *s3*. If we combine the two optimizations and traverse the statements in the order of *s1*, *s2*, and *s3*, then the program can be optimized as follows. We first visit statement *s1* and perform an optimization A on *s1*; Then, when we visit *s2* the new opportunity to perform optimization B on *s2* can be detected and the optimization is performed; Finally, when we visit statement *s3*, we can also detect the new opportunity to perform optimization A on *s3* and perform the optimization. This reduces the number of program traversals to one while still achieve the maximum optimized code.

Besides the fact that performing only one optimization in one program traversal is inefficient, the high level driver, which is used in an non-integrated approach to order and to repeat the analyses and the optimizations as we have just mentioned, is also a source of inefficiency. Because the optimal “pattern” to repeat the analyses and optimizations is most likely different from programs to programs, a general purpose driver which tries to accommodate all the possible “patterns” tends to repeat some optimization passes which are not needed for a particular program. Moreover, our experiments show that a non-integrated approach is more sensitive to the phase ordering problem than the integrated approach in the sense that altering the optimization order can reduce more optimization cases in the non-integrated approach than in the integrated approach (see Section 6.3).

Being sensitive to the optimization ordering also makes it difficult for the general-purpose driver to target for both high quality optimized codes and high efficiency.

An integrated approach, on the other hand, performs analysis to collect information for a group of optimizations, and to perform the optimizations in tandem. For example, both Wegman et. al. [WZ91] and Binkley [Bin94] combined constant propagation and unreachable branch elimination; Click and Cooper [CC95] combined constant propagation, unreachable branch elimination and global value numbering. Their work provides evidence that integrating related optimizations has a potential for a faster algorithm and a better result. As to the example in transforming the program in Figure 2(a) to the optimized program in Figure 2(b), an integrated approach can reduce the number of necessary program traversals from *ten* to *four* (see Figure 3(b)). However, in such an integrated approach, the analysis to collect the pointer alias information is still repeated to calculate the refined alias information after the unreachable branch elimination.

1.4 Our Approach

We take a more aggressive integrated approach to avoid the re-analysis in Figure 3(b) by incrementally updating the pointer alias information during the optimization. As shown in Figure 3(c), our approach can further reduce the number of program traversals to *two* and speed up the process to achieve the optimized version of code in Figure 2(b).

Our approach first performs a general-purpose interprocedural analysis on both pointer variables and non-pointer variables. The analyzed result is represented by an extended *static single assignment* (SSA) form which includes pointer-induced alias information, interprocedural *Ref/Mod* information and intraprocedural *use-def* information. It then performs a group of optimizations in tandem using the extended SSA form. Each optimization incrementally updates the extended SSA form to facilitate further analyses and optimizations. The final SSA form can be used by later parallelization and optimizations.

Such an approach integrates the scalar analyses and optimizations in three ways: (1) it performs a general purpose analysis that serves all the considered optimizations; (2) it interleaves the optimizations at the statement/expression level to exploit their interactions; and (3) it avoids re-analysis by updating the dataflow information during optimizations.

1.5 Contributions

The primary contributions of this work are as follows:

- An extended *static single assignment* (SSA) form. We describe our technique to incorporate pointer alias and interprocedural *Ref/Mod* information into a traditional SSA form, and to support its incremental updates. Our experiments show that, with an aggressive interprocedural program analysis, such an SSA form expands the size of the program representation by only a small factor.

- A new approach to analyze a program with pointers. We propose to partition the dataflow analysis according to the *pointer levels* of the data structures in the program. The *pointer level* of a variable is the maximum level of indirection to the variable. Our approach can improve the efficiency of flow-sensitive interprocedural analysis while does not sacrifice the quality of the analyzed result. We present our partitioned analysis technique using an improved SSA form to represent the analysis result. Our SSA generation algorithm is the first algorithm to create an SSA form without using an additional pointer analysis algorithm to pre-calculate pointer values.

- A new approach to integrate a group of scalar optimizations. We develop an algorithm to perform a group of scalar optimizations in tandem. Our approach can reduce the number of necessary program traversals and allow faster response to the interaction among the optimizations.

- An experimental evaluation of our partitioned analysis technique. We have implemented our partitioned analysis algorithm and demonstrated the feasibility and efficiency of the algorithm by performing extensive experiments on a large set of benchmark programs.

- An experimental evaluation of our integrated optimization algorithm. We have also implemented our algorithm to integrate optimizations. Our experiment demonstrates that our approach to integrate optimizations can improve the efficiency and effectiveness of performing a group of optimizations. Moreover, we show that, comparing to a non-integrated approach, an integrated approach is less sensitive to the *phase ordering* problem.

1.6 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we describe the program representation in our compiler. In Chapter 3, we describe our extensions to a naïve SSA form. These extensions not only allow us to implicitly represent pointer alias information and interprocedural *Ref/Mod* in the SSA form, but also allow us to incrementally update the SSA form during optimizations. In Chapter 4, we present our partitioned analysis algorithm, which creates an extended SSA form for the pointers one level at a time, and performs pointer analysis using the SSA form. The main goal of this partitioned approach is to improve the efficiency of flow-sensitive interprocedural dataflow analysis without sacrificing the precision of the analyzed results. In Chapter 5, we present our approach to integrate a group of optimizations using the extended SSA form. Each optimization incrementally updates the SSA form to allow faster response to the interaction among the optimizations. In Chapter 6, we present our experiments to evaluate our analysis algorithm as well as our optimization algorithm. Finally, we conclude with a summary of contributions and a discussion of future work.

CHAPTER 2

THE PROGRAM REPRESENTATION

We deal mainly with programs written in C in this dissertation. These programs may have general pointer usage, function pointers and recursive functions. The Agassiz Compiler reads in a source program, transfers the program into an intermediate representation (IR). We also add some other information, such as control-flow information and program call graph, to the IR. The additional information together with the IR provide a complete view of a program to the analyses and the optimizations described in this dissertation. The dataflow information collected in our analysis is annotated on top of the IR. In this chapter, we describe the IR and other additional information which provide a base for our analyses and optimizations.

2.1 Abstract Syntax Trees

We use abstract syntax trees (AST) [ASU86] to represent a source program. Figure 4 gives an example program and its AST representation. In the AST representation, operators and keywords are interior nodes. Operands are represented as children of the corresponding operators. Similarly, the components for the language construct corresponding to a keyword are represented as the children of the keyword node. This AST representation depicts the natural hierarchical structure of a source program. Moreover, the AST representation along with a symbol table which indicates the types of the variables provides all the necessary information to reconstruct the source program.

2.2 Expressions

We normalize some complicate C expressions to simplify our dataflow analysis. First, we transform an expression with side-effect operators into an equivalent expression without side-effect operators. For example, expression “i ++” is transferred into “i = i + 1”. This transformation makes the read and the write to *i* in the expression explicit. Second, we also transform an expression with function calls into its equivalent code without embedded function calls in expressions. For instance, expression “a + foo()” is transformed into “temp = foo(); a + temp”. This transformation can relieve the burden of modeling the effect of both computations and function calls in one expression, and simplify dataflow analysis. Lastly, the C programming language provides two dereference operators: “*” and “->”. We transform an expression with operator “->” into an equivalent expression with operator “*”. For example, expression “p->s” is transformed into an equivalent expression “(*p).s”. This transformation simplifies the handling of the same semantics represented in two different forms.

(a) An example program.

```
void foo(int p, q) {
    int a, b, c;
    a = p + q;
    if (c)
        b = p – q;
}
```

(b) The abstract syntax tree for the program.

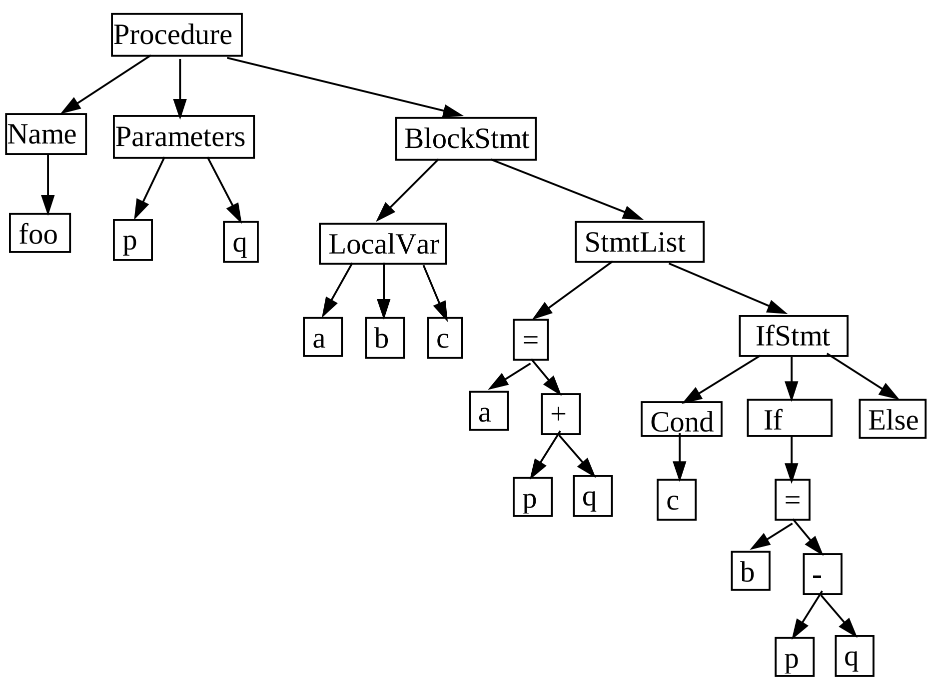


Figure 4: An example program and its abstract syntax tree representation.

While we eliminate the side-effect operators in our representation, we retain other high-level operators to facilitate our analyses and optimizations. For example, we retain the dereference operator "*" and the assignment operator "=". We also retains variable accesses, array and structure references, and complex array/structure access expressions, such as *s.a[1].f*, without dissecting them into simpler expressions using temporary variables. By doing so, we retain variable type, array size, array dimension, and structure field types information in the IR. Such information is critical for efficient dataflow analysis.

2.3 Statements

We classify the statements in a program into two categories: *simple statements* and *compositional statements*. Simple statements include assignments, function calls, *return* statements which cause exits from a procedure, *break* statements which cause earlier exits from a loop, *goto* statements, *setjmp/longjmp* statements. Notice that *continue* statements provided by the C programming language are eliminated in our IR, and a piece of code with *continue* statements is transformed into its equivalent representation without *continue* statements. There are five types of compositional statements in our IR: *block* statements which are sequences of statements in a nested scope, *do-loops*, *while-loops*, *do-while-loop* and *if* statements. Notice that *switch* statements provided in the C programming language are eliminated in our IR, and a program with *switch* statements is transformed into its equivalent representation without *switch* statements.

2.4 Intraprocedural Control-flow

In our program representation, a control-flow graph (CFG) [ASU86] represents the control-flow of a procedure. There are two types of edges in our control-flow graph: *implicit* edges and *explicit* edges. An implicit edge of a control-flow graph is implied by the semantics of a compositional control statement. Figure 5 shows the implicit control-flows for *if* statements (Figure 5(a)) and the three types of loops (Figure 5(b) and (c)). An explicit edge of a control-flow graph is caused by simple statements such as *break*, *return*, *goto*, and *setjmp*/*longjmp*. We represent these edges in a control-flow graph by annotations. More specifically, we annotate a loop with a list of *break* statements that may cause earlier exits from the loops. This annotation facilitates the dataflow analysis to calculate the dataflow values coming out of the loop by merging the dataflow values gathered at all *break* statements. Similarly, we also annotate a procedure with a list of *return* statements. The dataflow values at the *exit* node of the procedure are calculated by merging the dataflow values gathered at all *return* statements.

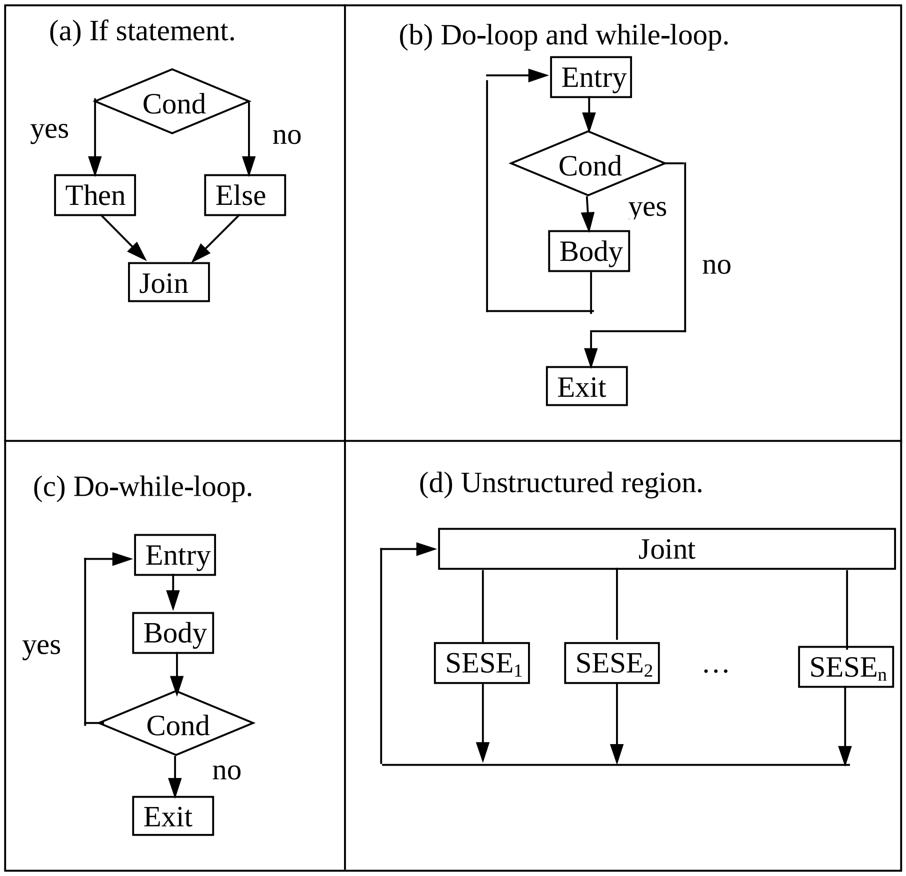


Figure 5: The control-flow structures for *if* statements, loops and *unstructured regions*.

In C programs, *setjmp* and *longjmp* may cause non-local control-flow. When *setjmp* is called, a data structure *jmp_buf* is passed to the library routine. If later on the *jmp_buf* is passed as a parameter to *longjmp*, the program execution will resume at the point where *setjmp* is called. We conservatively handle the nonlocal control-flow caused by *setjmp*/*longjmp* as follows:

- A call to *longjmp* is handled as a *return* statement in building the control-flow.
- If a procedure *foo* directly or indirectly calls *longjmp*, then after a call statement to *foo*, we insert a pseudo statement "if (unknown-condition) return" to indicate that the program execution may transfer from the *longjmp* call-site back to the *setjmp* call-site in the current call stack.
- If a procedure *bar* calls *setjmp*, then for each procedure *foo* (*foo* directly or indirectly calls *longjmp*) that *bar* calls, insert a pseudo statement "if (unknown-condition) goto the-statement-that-calls-setjmp" to indicate that the program execution may resume at the point where *setjmp* is called from the execution of *foo*.

Ideally, a program with *goto* statements can be transformed into an equivalent program without *goto* statements [Amm92]. For example, a *goto* statement which causes a cycle in a program control-flow may be transformed into a loop. Because eliminating *goto* statements may require code replication, we may want to keep some *goto* statements if the code bloating caused by the transformation to eliminate the *goto* statements becomes a problem. In this case, we need to represent the control-flow caused by *goto* statements.

We annotate a *goto* statement with its *goto* targets to represent the irregular control-flow caused by the *goto* statement. We also introduce the concepts of *unstructured region* and *structured region* to distinguish a piece of codes with or without *goto* statements. An *unstructured region* contains a set of single-entry/single-exit (SESE) block. The last statement of an SESE block may be a *goto* statement which jumps to the entries of other SESE blocks in the same unstructured region. Besides the last statement, other statements in an SESE block have nested structured regions or nested unstructured regions. All *goto* statements are enclosed in unstructured regions. We handle the control-flow of an unstructured region in a safe but relatively conservative manner as shown in Figure 5(d). A *structured region*, on the other hand, is a piece of code that does not directly contain *goto* statements.

2.5 Interprocedural Control-flow: Call Graph

A call graph of a program captures the interprocedural control-flow of the program and is defined as follows.

Definition 1:

> The call graph of a program is represented as G = <N, E, *main*>. The node set N contains one node for each procedure in the program. The edge set E is a subset of set N x N. An edge $<p_1, p_2>$ exists if and only if procedure p_1 calls procedure p_2. Procedure *main* is the entry of the call graph.

Without function pointers, the construction of a program call graph requires a traversal of all the call-sites in the program. For each call-site encountered during the program traversal, we simply add an edge from the caller to the callee if the edge has not been created yet. However, the presence of function pointers makes it difficult to construct the program call graph because it is difficult to determine precisely the callees of an indirect call using function pointers. We safely but conservatively calculate the potential callees of an indirection call as follows. First, we collect *all* the functions whose addresses are taken and assigned to pointers. Second, for each indirect call using a function pointer, we use the set of addressed functions as an approximation of all its potential callees. Lastly, the above approximation is refined under certain conditions. For example, if no functions have a variable number of parameters in the program, a potential callee must have exactly the same number of parameters defined by the function pointer type. Similarly, if no type casts are used in the program, the types of the parameters of a potential callee must be compatible with the types of the parameters defined by the prototype of the function pointer. After determining the potential callees for indirect calls, we can build the program call graph.

A procedure is *recursive* if it directly or indirectly calls itself. A procedure is *self-recursive* if it directly calls itself. The presence of recursive procedures complicates the program call graph. Without recursive procedures, a call graph is a direct acyclic graph. However, with recursive procedures, a call graph is a direct cyclic graph.

2.6 Program Traversal

Before we describe our methods of intraprocedural traversal and interprocedural traversal, we first introduce the concepts of *topological order* and *topological traversal*.

Definition 2:

> A *topological order* refers to an order of all the nodes in a direct acyclic graph [CLR90]. In a topological order, if there is an edge from *n1* to *n2*, then *n1* is ordered before *n2*. A *topological traversal* refers to a traversal of all the nodes in a direct acyclic graph according to their topological order.

Thus, a topological traversal is a method to traverse a direct acyclic graph in which we visit a node after all of its predecessors are already visited. Our scheme to traverse a procedure is a topological traversal along the simplified control-flow graph (CFG) of the procedure with all the backward edges removed from the graph. We call such a scheme a topological traversal of a procedure.

Definition 3:

> A *topological traversal of a procedure* refers to a traversal of the procedure using the simplified CFG without backward edges. It traverses all the CFG nodes in their topological order [CLR90].

The concept of strongly connected component is needed to describe our interprocedural traversal methods.

Definition 4:

> The *strongly connected components* (SCC) of a direct graph are the equivalent classes of its nodes under the "are mutually reachable" relation.

An SCC of a call graph may contain one or more procedures. When an SCC contains one procedure, the procedure can be either a non-recursive procedure or a self-recursive procedure. On the other hand, when an SCC contains more than one procedure, these procedures construct a cycle in the call graph. If we combine all the procedures in an SCC into a super node, then the call graph will become a direct acyclic graph.

We may traverse the procedures in a program in two possible orders: call order and reversed call order.

Definition 5:

A *call order* of all the procedures in a program corresponds to a topological order of all the SCCs in the program call graph. Procedures that are in the same SCC are ordered successively. On the other hand, a *reversed call order* of all the procedures in a program corresponds to a reversed topological order of all the SCCs in the program call graph.

CHAPTER 3

AN EXTENDED STATIC SINGLE ASSIGNMENT FORM

In this chapter, we first introduce a *static single assignment* (SSA) form proposed by Cytron et. al. [CFR+91]. We then discuss our motivation to extend such an SSA form. After that, we discuss our method to incorporate pointer alias information and interprocedural *Ref/Mod* information into the SSA form, and to support its incremental updates. Finally, we discuss related work and compare our SSA form with other SSA forms with pointer alias information.

3.1 Background

An SSA form is an efficient program representation which provides a sound base for powerful analysis and optimization [CFR+91]. Many efficient algorithms based on SSA forms have been developed for pointer analysis [Wil97, HH98, Lu98], optimization [WZ91, CC95, CCK+97] and parallelization [Wol92, GSW95, TP95].

3.1.1 Properties of SSA Forms

An SSA form has two important properties. First, each variable read is *directly* linked to one definition. In order to ensure this property, *ϕ-definitions* are inserted at program points where more than one definitions to a variable merge. A *ϕ-definition* has a *ϕ-function* on its right-hand side (RHS). The *ϕ-function* has one operand for each merged definition. This is in contrast to a *regular definition* whose RHS is not a *ϕ-function*. Second, each variable has exactly one definition. Variables are renamed to ensure that each variable has exactly one definition. The variable renaming is usually done by adding a version number as the subscript of the original variable. After the variable renaming, a read to a variable has the same version number as the definition to the variable that directly reaches the read. If a definition that directly reaches a read is a *ϕ-definition*, the *ϕ-definition* provides links to trace back all the *regular* definitions that indirectly reach the read.

Figure 6 gives an example program and its SSA form. In the program, there are two *regular* definitions that reach the read of the variable *b* at statement *s6*. One definition reaches the read via the *then-branch*, and the other definition reaches the read via the *else-branch*. In order to indicate that both definitions reach the read, we insert a *ϕ-definition* "$b_3 = \phi(b_1, b_2)$" at the merge point of the *then-branch* and the *else-branch*. This *ϕ-definition* has a *ϕ-function* to represent a merge of the two definitions to *b*. We also add a version number to each appearance of the original variable so that (1) each version of the original variable is defined only once, and (2) a read to a variable and the definition of the variable that *directly* reaches the read use the same version number. Thus, the *ϕ-definition* to *b* and the read of variable *b* at statement *s6* have the same version number.

(a) An example program.	(b) An SSA form for the program.
s1: c = …; s2: if (c) s3: b = …; s4: else s5: b = …; s6: … = b;	s1: c_1 = … s2: if (c_1) s3: b_1 = … s4: else s5: b_2 = … $b_3 = \phi(b_1, b_2)$ s6: … = b_3 *use-def* links implied by the SSA variables

Figure 6: An example program and its SSA form.

As can be seen from Figure 6, in an SSA form, the fact that a read to a variable and a definition to the variable use the same version number implies a *use-def* link and/or a *def-use* link between the read and the definition. In practice, many compilers do not actually implement the renaming of variables to avoid expanding the size of the symbol table. Instead, they maintain the SSA properties by explicitly represent these implicit *use-def* and/or *def-use* links between each use and its reaching definition [Wil97, SWG94]. Our implementation takes such an approach. However, this implementation does not preclude us from using the renaming notation in our discussion.

3.1.2 Representing Control-flow Joints

Different definitions to the same variable may be merged at joint nodes in the control-flow graph (CFG) [ASU86] joint nodes. Thus, one important step in transforming a program into its SSA representation is to determine all the CFG joint nodes where ϕ-functions are needed. An SSA form is *minimal* if ϕ-functions for variables are inserted only in the necessary control-flow joint nodes [CFR+91]. An SSA form with extraneous ϕ-functions beyond these nodes is still valid because it will not lead to incorrect optimization. However, the extraneous ϕ-functions add unnecessary overhead to both the representation and the optimization.

Cytron et. al. describe an efficient algorithm to construct the minimal SSA form for a program with arbitrary control-flow [CFR+91]. They first introduce a concept called *dominance frontiers* to describe the CFG joint nodes where ϕ-functions are needed for a variable. They then describe an algorithm to compute the dominance frontiers for each variable and to place the necessary ϕ-functions. Their algorithm requires several passes over the CFG and has a complexity of $O(R^3)$, where R is the program size. However, their experiment also shows that the complexity of the algorithm is usually linear to the size of a program in practice.

Brandis et. al. present a single-pass algorithm to generate SSA forms for well structured programs, i.e., programs that contain assignments and structure statements (such as *if* statements and loops with/without *break*/*continu*e statements) but do not contain *goto* statements [BM94]. Although their algorithm has the same worst-cast complexity as the one described by Cytron et. al., it is believed that the single-pass algorithm is simpler to implement and more efficient in real practice. Thus, such an algorithm is useful for languages which do not support *goto* statements. It can also be applied to compilers, such as McCAT [HDE+92], which lack the *goto* statement representation and requires a control-flow normalization pass to transform a program with *goto* statements into an equivalent program without *goto* statements. Because our Agassiz Compiler also transforms a program with irregular control-flow structures into an equivalent well-structured program, we apply a ϕ-function placement algorithm very similar to the one proposed by Brandis et. al.

3.1.3 The Advantage of SSA Forms

The advantages of using an SSA form have been discussed in literature [CFR+91, Joh94, Wol96]. Here, we highlight the advantages that are directly related to the use of an SSA form in our integrated approach to scalar analyses and optimizations.

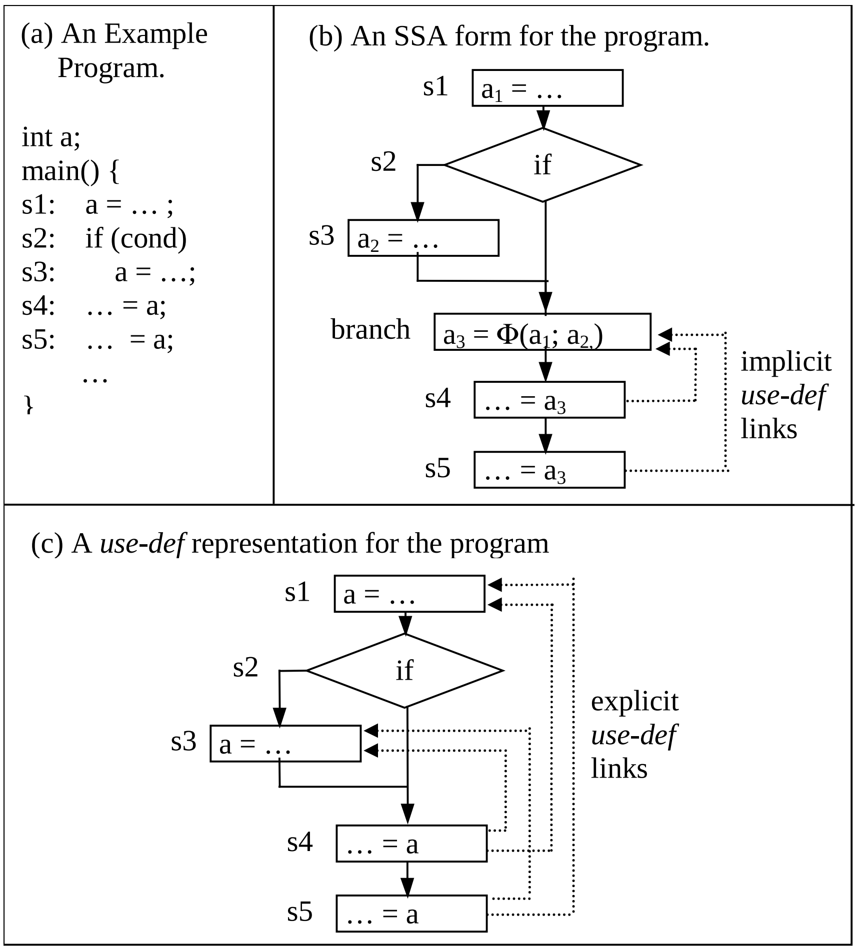


Figure 7: Comparing an SSA form with a *use-def* representation using an example program.

First, in comparison with a *use-def* (and/or *def-use*) representation, an SSA form provides a more compact program representation. This can be illustrated by Figure 7, which shows an example program, its SSA form and its *use-def* representation. Assume there are *n* reads to variable *a* after the *if* statement in the example program. The SSA form for the program has only *n* (implicit) *use-def* links linking to the ϕ-definition at the merge point between the *then-branch* and the *else-branch*. The *use-def* representation for the program, on the other hand, has *2n use-def* links: *n* links point to the definition created by statement *s1* and *n* links point to the definition created by statement *s3*.

Second, an SSA form also encodes control-flow information and supports incremental updates to the SSA from during optimizations, such as unreachable branch elimination. This can be illustrated by assuming that the compiler can determine that, in the example program given by Figure 7(a), the condition *c* of the *if* statement is *true* and the *else-branch* is unreachable. This optimization makes the definition created by statement *s3* overwrite the definition created by statement *s1*. Thus, the link from the use of *a* at *s4* to the definition of *a* at *s1* becomes a spurious *use-def* link. There are two ways to remove this spurious *use-def* link in the SSA form: (1) mark the ϕ-function operand for the *else-branch*, a_1, as invalid, or (2) replace the ϕ-function with its operand for the *then-branch*, a_2, so that the definition a_3 becomes "$a_3 = a_2$". However, such a refinement to the dataflow information cannot be performed on the *use-def* representation. The *use-def* representation cannot provide the information that the definition a_1 reaches the control-flow joint node via the *else-branch* while the definition a_2 reaches the joint node via the *then-branch*. Without this information, the compiler has to re-analyze the program in order to find out that, after the branch elimination, the definition a_2 overwrites the definition a_1. Our approach uses an SSA form and associates a predicate with each operand of the ϕ-function to support the removal of spurious *use-def* links during optimizations.

Lastly, in an SSA form, each variable is assigned only once. This has the advantage that an efficient flow-insensitive dataflow analysis using the SSA form of the program can achieve the same result as a less efficient flow-sensitive dataflow analysis using another representation of the program, assuming that the translation of the original program into the SSA form does not lose information [HH98].

3.2 Motivations for an Extended SSA From

We call an SSA form without pointer alias information and interprocedural *Ref/Mod* information a naïve SSA form. We extend the naïve SSA form for several reasons.

First, we want to incorporate pointer alias information into the SSA form. In the presence of pointers, an SSA form without pointer alias information may provide incorrect information for analyses and optimizations. This can be illustrated by Figure 8. In the figure, the facts that statement *s9* modifies *a* and statement *s10* may modify *b* or *c* are missing in the SSA form. As a result, the *use-def* links starting from statement *s11* are incorrect. This may mislead constant propagation to replace the reads to *a* and *b* at statement *s11* with constants 1 and 2 respectively. Thus, an SSA form has to incorporate pointer alias information to provide correct information.

(a) An Example Program.

```
int a, b, c, d, *p, *q;
main() {
s1:   p = &a;
s2:   if (cond)
s3:      q = &b;
s4:   else
s5:      q = &c;
s6:   a = 1;
s7:   b = 2;
s8:   c = 3;
s9:   *p = 4;
s10:  *q = 5;
s11:  d = a + b;
         …
}
```

(b) Its SSA form (without pointer alias).

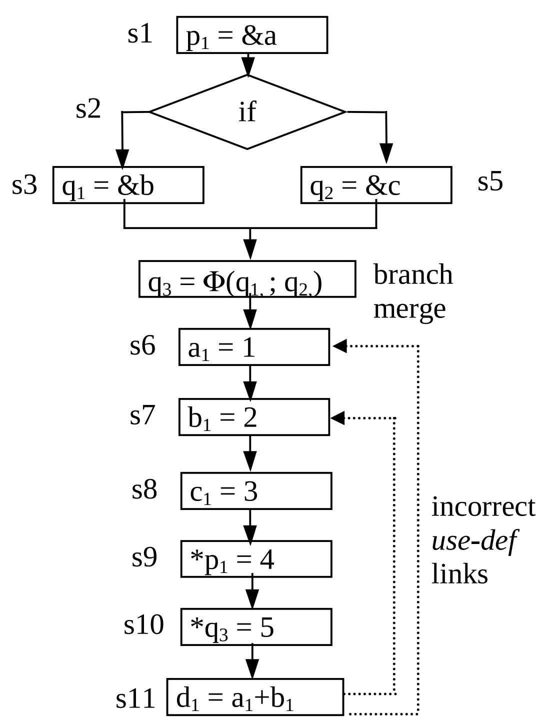


Figure 8: Missing pointer alias information may cause incorrect optimizations.

(a) An example program.	(b) An SSA form for the program (without *Ref/Mod* information).

int a, b, …;
s1: main() {
s2: a = …;
s3: b = 2;
s4: foo();
s5: c = b + 1;
s6: …;
}
	s1: main() {
	s2: a_1 = …;
	s3: b_1 = 2;
	s4: foo();
	s5: $c_1 = b_1 + 1$;
	s6: …;
	}

Figure 9: Missing interprocedural *Ref/Mod* information may cause incorrect optimizations.

Second, we want to incorporate interprocedural *Ref/Mod* information into an SSA form. Similar to pointer alias information, missing the interprocedural *Ref/Mod* information may also lead to incorrect optimizations. In Figure 9, assume procedure *foo* reads variable *a* and modifies variable *b*. Not representing this information in the SSA form may mislead optimizations to remove statement *s2* as useless code and replace the read of *b* at *s5* with a constant value *2*.

(a) Remove pseudo use-def links.	(b) Remove pseudo points-to values.	(c) Refine the *offset* values for pointers to arrays.

s1: if (c_1)
s2: b_1 = …
s3: else
s4: b_2 = …
s5: $b_3 = \phi(b_1, b_2)$
s6: … = b_3
…
	s1: if (c_1)
	s2: p_1 = &a
	s3: else
	s4: p_2 = &b
	s5: $p_3 = \phi(p_1, p_2)$
	s6: *p_3 = … ;
	…
		s1: p_1 = &a[c_1]
		s2: for (i = 0; i < 10; i++) {
		s3: p_1[i] = …
		s4: …
		}
		…

Figure 10: Examples for incrementally updating SSA forms during optimizations.

Lastly, we want to allow incremental updates to the SSA form during optimizations. Figure 10 gives three examples to explain why it is desirable to incrementally update an SSA form during optimizations so that more precise information exposed by an optimization can be used by later optimizations. In the example of Figure 10(a), if we can determine that the condition c_1 is *true,* we can eliminate the *else-branch*. This optimization can remove the spurious *use-def* links from statement *s6* to statement *s4*. Similarly, in the example of Figure 10(b), if we can determine that the condition c_1 is *true*, we can eliminate the *else-branch*. This optimization can remove spurious points-to relation "*p* points to *b*". As a result, statement *s6* creates a definition to *a*, this definition overwrites the previous value of *a*. In order to support the refinement of the dataflow information described in the above two examples, we need to represent how the *use-def* links and the points-to relation are related to the branches. As a third example, in Figure 10(c), we assume that pointer analysis can determine that pointer *p* points to array *a* with a variable offset *c*. If after the pointer analysis, constant propagation can determine that the read of *c* at statement *s1* has a constant value, we may want to refine the pointer offset information at statement *s3* and other statements inside the loop to allow more precise data dependence analysis.

3.3 An Extended SSA Form

We now describe our methods to incorporate pointer alias information and interprocedural *Ref/Mod* information into an SSA form, and to allow incremental updates to the SSA form during optimizations. Our SSA form has three main extensions to a naïve SSA form: representing implicit variable accesses using I-functions, representing weak-updates using ϕ-functions, and using predicates to enable incremental updates.

3.3.1 Representing Implicit Variable Accesses

Our first extension to a naïve SSA form is to incorporate indirect variable accesses into our SSA form. An implicit variable access refers to a variable which is read/written by an expression but does not textually appear in the expression. An expression with pointer dereferences or function calls may access variables that do not textually appear in the expression. We represent these implicit variable accesses by associating the expression with two I-functions: *Implicit Read* (I_R) and *Implicit Write* (I_W). These two I-functions have the general forms of $I_R(m_1, m_2, \ldots, m_n)$ and $I_W(m_1, m_2, \ldots, m_n)$, where an m_i (i = 1..n) represents a variable implicitly accessed in the expression. In this dissertation, we put the expression and its I_R and I_W representations into a pair of brackets "<>" and separate them with commas "," for clarity.

Example 1:

For an indirect assignment "*p = *q + 4", we assume (2) definitions a_0, b_0, p_0 and q_0 reach the assignment statement (1) both p_0 and q_0 may point to variables *a* and *b,* and. We incorporate indirect variable accesses into the above statement as "$<*p_0, I_W(a_1, b_1)> = <*q_0, I_R(a_0, b_0)> + 4$". This provides a compacted representation for the four possibilities represented by the original statement: $a_1 = a_0 + 4$, $a_1 = b_0 + 4$, $b_1 = a_0 + 4$, $b_1 = b_0 + 4$.

For function calls, the I_W function representation can be extended to incorporate the return values from the call. This extension is necessary for applications, such as interprocedural constant propagation, which require interprocedural values.

Example 2:

Assume a function call *"f()"* may write variables *a* and *b* and read variables *c* and *d*. The implicit variable accesses for the function call is represented as "$<f(), I_W(a_?=r\_a, b_?=r\_b), I_R(c_?, d_?)>$"[1], where *r_a* and *r_b* are the return values of *a* and *b* respectively.

3.3.2 Representing *Weak-Updates* [2]

The most important property of an SSA form is that each variable read is directly linked to exactly one definiton. In order to transform a program into a form with such a property, *ϕ-definitions* are inserted at program points where distinct values of a variable are merged. The right-hand side (RHS) of a *ϕ-definition* is a ϕ-function with a general form of $\phi(d_1; d_2; ...; d_n)$, where operand d_i (i=1..n) represents a definition that contributes to the merge.

[1] The question mark (?) associated with a variable indicates that the version of the variable is to be decided in a later step during the SSA generation. Similar annotation is also used in the remainder of this dissertation.

[2] *Weak Updates* [Ste95] are also called *Preserving* definitions [CCF94] or *MayDefs* [CCL+96].

Distinct values of a variable can be merged under two situations: at a CFG joint node and at a statement generating *weak-updates*. A weak-update refers to an assignment (definition) to a variable that may not overwrite the previous value of the variable. Thus, after the assignment, both the old value that reaches the assignment and the new value created by the assignment may reach a use of the variable. Although there exists a standard and well defined algorithm to place ϕ-functions at CFG joint nodes, there is no standard way to represent weak-updates. Different SSA forms may have different weak-update representations [CFR+91, BC92, CG93, CCF94, Ste95, CCL+96, Wol96, HH98].

Our second extension to a naïve SSA form is to extend the ϕ-function to represent the merged values caused by weak-updates. First, an assignment statement using pointer dereferences, reference parameters or array subscripted expressions may generate weak-updates. A ϕ-function for such an assignment statement has two operands, represented as $\phi(d_1; d_2)$. In this representation, d_1 is a definition that reaches the statement and may not be overwritten by the statement while d_2 is a new definition generated by the assignment. The number of ϕ-functions created for an assignment statement using a pointer dereference (or a reference parameter) equals to the number of variables that are pointed to by the pointer (or the parameter). On the other hand, we create only one ϕ-function for an assignment to an array element.

Example 3:

> Refer to the indirect assignment "*p = *q + 4" use in Example 1. Because pointer *p* has two points-to targets, we create two ϕ-functions for the statement: $a_2 = \phi(a_0; a_1)$ and $b_2 = \phi(b_0; b_1)$, where a_0 and b_0 are definitions reach the statement, and a_1 and b_1 are the new definitions generated by the statement.

Second, the maybe modification effect of function calls also creates weak-updates. A ϕ-function for a direct function call has two operands, represented as $\phi(d_1; d_2)$. In this representation, d_1 is a definition that reaches the function call and may not be overwritten by the function while d_2 is a new definition representing the function call's modification effect. The number of ϕ-functions created for a direct function call equals to the number of variables that may be modified by the function.

Example 4:

> Assume a function call *"f()"* may write two variables *a* and *b*. We create two ϕ-functions for the function call: $a_2 = \phi(a_0; a_1)$ and $b_2 = \phi(b_0; b_1)$, where a_0 and b_0 are definitions reach the function call, and definitions a_1 and b_1 are the new definitions representing the function's modification effect.

Third, indirect function calls using function pointers also create weak-updates. Considering the fact that an indirect function call may have *r* potential callees, a ϕ-function for such a function call has *r+1* operands, represented as $\phi(d_1; d_2; ...; d_{r+1})$. In this ϕ-function, d_1 is the definition that reaches the function call and may not be overwritten by the function call. Operand d_{i+1} $(i = 1 .. r)$ represents a new definition that is generated by the *i*th potential callee. Definition d_{i+1} $(i = 1 .. r)$ defines the same variable as d_1 does. If such a definition does not exist, $d_{i+1} = NULL$.

Example 5:

> Assume (1) function pointer *fp* may point to two procedures *fun-1* and *fun-2*, (2) *fun-1* may modify variables *a* and *b*, (3) *fun-2* may modify variable *b*, and (4) definitions a_0 and b_0 reach call statement *"(*fp)()"*. The extended SSA form for the function call is "<(*fp)(), $I_W(a_1, b_1, b_2)$>; $a_2 = \phi(a_0, a_1, \text{NULL})$; $b_3 = \phi(b_0, b_1, b_2)$". We avoid mixing the definitions generated by different callees to allow later refinement of the ϕ-functions with more precise function pointer values. For example, if further analysis can identify that *fun-1* is a spurious points-to target of the pointer *fp*, we can invalidate the second operands of the above two ϕ-functions.

3.3.3 Definition Graph

The concept of *definition graph* is needed to discuss our third extension to the naïve SSA form. Thus, we would like to introduce the concept of *definition graph* before we continue our discussion of the extended SSA form.

Definition 6:

A *definition graph* G is a direct graph $G = \{V, E\}$. It describes the hierarchical structure of a group of definitions. The node set V contains a node for each definition in the group. An edge $<d_1, d_2>$ is added to the edge set E if and only if (1) d_2 is a ϕ-definition, and (2) d_1 is an operand of the ϕ-function in d_2.

Example 6:

Figure 11 shows an example program, its extended SSA form and its definition graph. As can be seen, in Figure 11(b), two *regular* definitions p_0 and p_1 reach a read of p_2 at statement *s8*. Hence, both definition p_0 and definition p_1 can reach definition p_2 in the definition graph in Figure 11(c).

3.3.4 Predicates and *β-functions*

Our third extension to the naive SSA form is to associate a *predicate* with each ϕ-function operand to describe the condition under which the operand is valid. We call such ϕ-functions with predicates *β-function*s. Correspondingly, we also rename ϕ-definitions as *β-definition*s. Thus, we represent a merge of *n* definitions to a variable as $\beta((d_1, p_1); (d_2, p_2); \ldots (d_n, p_n))$, where p_i (i=1..n) is the condition under which d_i (i=1..n) is a valid operand of the β-function. A *true predicate* indicates that the corresponding operand definitely reaches the program point where the β-function is created. On the other hand, a *false predicate* indicates that the operand does not reach the program point. A *variable predicate* indicates that the operand conditionally reaches the program point.

The definition graph defined in Definition 6 can also be augmented with predicates. More specifically, if d_2 is a β-definition, and d_1 is an operand of the β-function in d_2 with a predicate p, we associate the predicate p with the edge $<d_1, d_2>$. Similarly, we also associate each I_R-function operand with a predicate to represent conditional implicit reads. However, we do not associate a predicate with each I_W-function operand. Instead, the condition under which a variable is implicitly written is represented by predicates in the corresponding β-function.

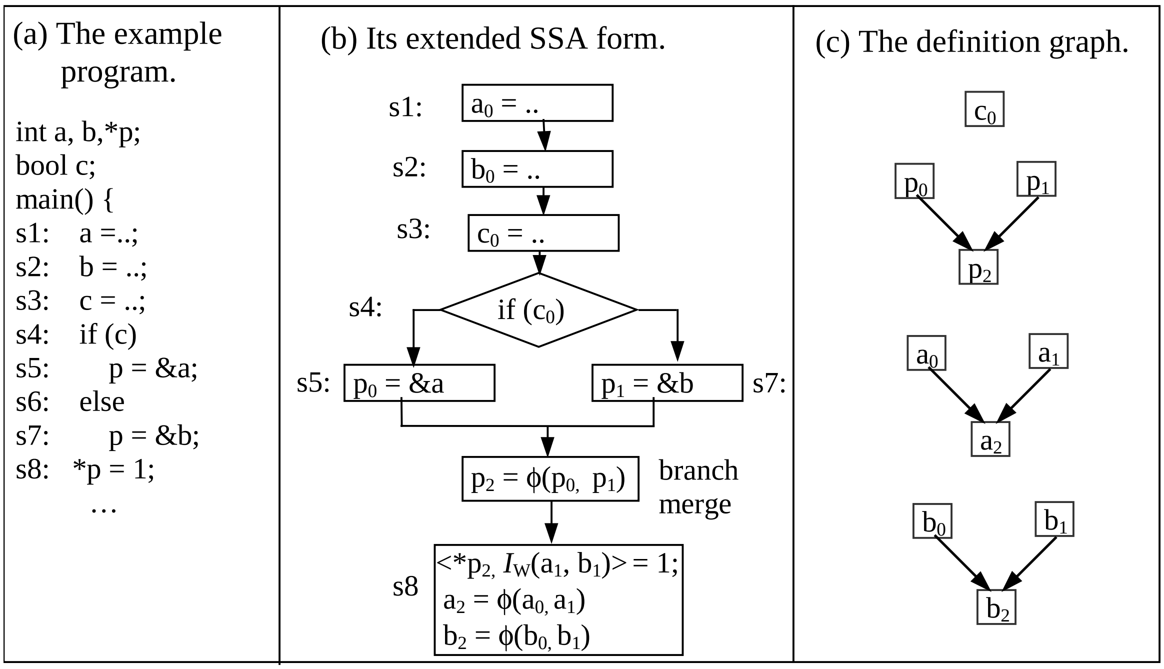


Figure 11: An example to illustrate definition graphs.

There are two main purposes in augmenting the SSA form with predicates. One purpose is to allow incremental updates to the SSA form during optimizations. Another purpose is to distinguish the dataflow values propagated into a procedure from different call-sites. In general, a *predicate* can be of any form, depending on the types of the β-functions and the implementation. In the remaining of this section, we introduce two forms of predicates that can effectively reflect the refinement of the dataflow information due to optimizations. The predicates to distinguish the dataflow values propagated into a procedure from different call-sites are discussed later in Section 4.5.2.

3.3.4.1 Branch Conditions as Predicates

The condition expression in a conditional branch statement determines which of the branches is reachable. This expression can be used as *predicate*s in β-functions.

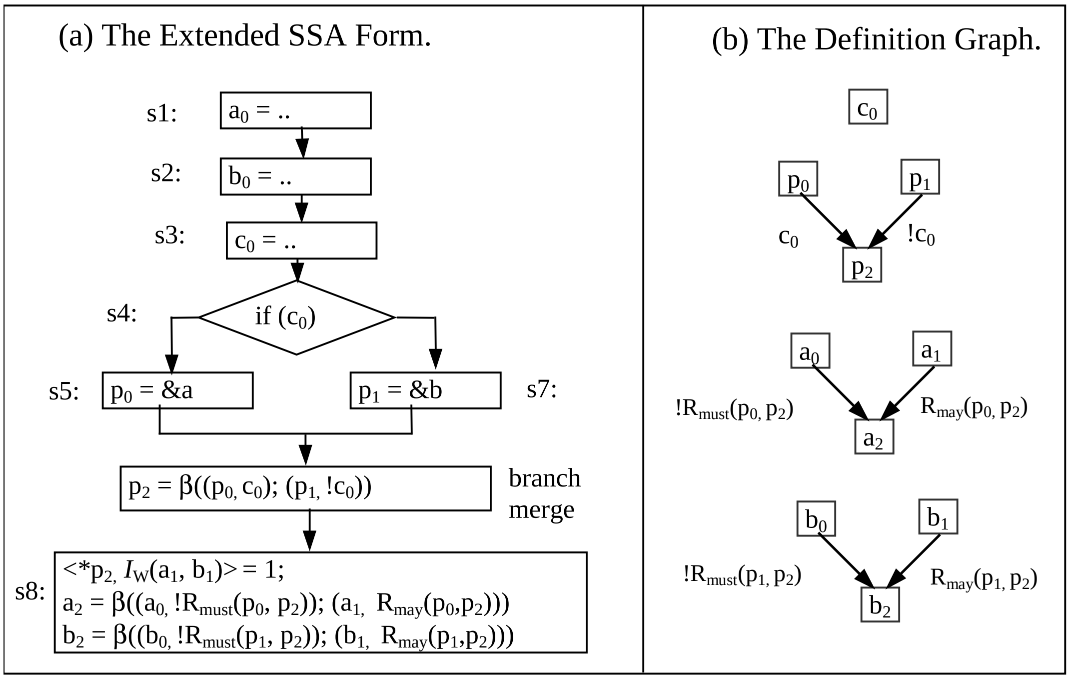


Figure 12: The extended SSA form (with predicates) for the program in Figure 11(a) and the definition graph for the definitions.

Example 7:

In Figure 12(a), we extend the ϕ-function "$p_2=\phi(p_0, p_1)$" with predicates as "$p_2=\beta((p_0, c_0); (p_1, !c_0))$". As can be seen, we use the *if-condition* c_0 in the predicates for the β-function operands. With further information on c_0, let's assume the compiler can determine that c_0 is *true*, then p_1 becomes an invalid operand of the β–function and p_2 has the value of p_0. This can remove the spurious *use-def* links from p_1 to any read of p_2. Thus, the β-function representation allows us to refine the dataflow information in the SSA form during unreachable branch elimination.

3.3.4.2 Predicates Defined by Reachability in Definition Graph

As we have seen in Example 7, optimizations may invalidate β-function operands. This can reduce the number of *regular* definitions that can reach the corresponding β-definitions in the definition graph and remove spurious *use-def* links. This refinement to the dataflow information can be effectively reflected by a change of value for two predicates representing definition graph reachability: *may reach* (R_{may}) and *must reach* (R_{must}).

Description	$R_{may}(d_i, d_j)$
No path from d_i to d_j.	*false*
There exists paths from d_I to d_j. All edges in the paths have *false* predicates.	*false*
There exists a path from d_i to d_j. At least one edge in the paths has a *true* predicate.	*true*

Figure 13: The truth table for predicate R_{may}.

Description	$R_{must}(d_i, d_j)$
No path from d_i to d_j.	*false*
There exists a path from d_i to d_j. All edges in the path have *true* predicates.	*true*
There exists paths from d_i to d_j. However, for any path from d_i to d_j, at least one edge in the path has a *false* predicate or a variable predicate.	*false*

Figure 14: The truth table for predicate R_{must}.

Definition 7:

For any two definitions d_i *and* d_j, the truth table for the predicates $R_{may}(d_i, d_j)$ and $R_{must}(d_i, d_j)$ are show in Figure 13 and Figure 14. The predicate $R_{may}(d_i, d_j)$ is *true* if and only if there exists a path from d_i to d_j in the *definition graph* and all the edges in the path has a predicate which is not *false* (*true* or a variable predicate). On the other hand, the predicate $R_{must}(d_i, d_j)$ is true if (1) the predicate $R_{may}(d_i, d_j)$ is *true* and (2) the predicates for all the edges in a path from d_i to d_j are *true*.

Example 8:

In Figure 12(a), statement *s8* "$*p_2 = 1$" generates definition a_1 because definition "$p_0 = \&a$" generated by s_5 may reach the use of p_2 in *s8*. This is represented by associating predicate $R_{may}(p_0, p_2)$ with the second operand a_1 in $\beta((a_0, !R_{must}(p_0, p_2)); (a_1, R_{may}(p_0,p_2)))$. On the other hand, if definition "$p_0 = \&a$" must reach the use of p_2, definition a_1 is a *killing* definition that kills definition a_0. Thus, predicate $!R_{must}(p_0, p_2)$ is associated with the first operand a_0 in $\beta((a_0, !R_{must}(p_0, p_2)); (a_1, R_{may}(p_0,p_2)))$ to represent the condition which a_0 is not killed by a_1. Similarly, the fact that statement *s8* conditionally generates definition b_1 which may or may not kill definition b_0 is represented by β-function $\beta((b_0, !R_{must}(p_1, p_2)); (b_1, R_{may}(p_1,p_2)))$. These representation allow us to refine the dataflow information during optimization. For example, assume that constant propagation can determine that c_0 is *true*. Both predicates $!R_{must}(p_0, p_2)$ and $R_{may}(p_1,p_2)$ become false. As a result, the first operand in β-function $\beta((a_0, !R_{must}(p_0, p_2)); (a_1, R_{may}(p_0,p_2)))$ and the second operand in β-function become invalid. Only definition a_1 reaches a use of *a* after the *if* statement. Likewise, only definition b_0 reach a use of *b* after the *if* statement.

3.4 Other Related Work

In this Section, we discuss how other SSA forms represent pointer alias information and how these SSA forms compare with our SSA form. We also discuss other SSA forms with predicates.

3.4.1 Location Factored SSA Forms Versus Assignment Factored SSA Forms

Some research has discussed how to incorporate pointer alias information into SSA forms [CFR+91, BC92, CG93, CCF94, Ste95, CCL+96, Wol96, HH98]. One important issue is how to represent the modification effect of an indirect assignment using pointer dereferences. As to the example program in Figure 8, we need to decide (1) how to represent the fact that statement *s9 must modify* , or *strongly updates* variable *a*; and (2) how to represent the fact that statement *s10 may modify*, or *weakly updates* variables *b* and *c*. As a result of these two indirect assignments, a read of *a* in statement s11 has a value 4 while a read of b in statement s11 is either 2 or 5 which the compiler cannot statically determine.

The above two types of modification, *must* modification and *may* modification, are directly related to the two types of aliasing relationship: *must-alias* and *may-alias*. Two symbolic names, such as **p* and *a* in Figure 8, are must-aliases in a statement if they refer to the same memory location in all execution instances of the statement. On the other hand, two symbolic names, such as **q* and *b* in Figure 8, are may-aliases in a statement if they refer to the same memory location only in some execution instances of the statement. If the pointer dereference in an indirect assignment is a must-alias of a variable, the indirect assignment can be treated as an assignment to the variable. For example, in Figure 8, statement *s9* "**p = 4*" can be treated as "*a = 4*" because "**p*" is a must-alias of "*a*". Complication occurs when the pointer dereference is a may-alias of other variables. In this case, there are two possible representations: *location factored* and *assignment factored* [Ste95].

A *location factored* SSA form and an *assignment factored* SSA form create different numbers of definitions for an indirect assignment using pointer dereferences. Assume an assignment may update *n* locations. A *location factored* SSA form creates *n* definitions for the assignment. Each definition has *def-def* links to daisy chain the definitions that reach the assignment but are not killed by the assignment [BC92, CCF94, CCL+96, HH98]. A definition which is not associated with any *def-def* link is a strong-update while a definition which is associated with *def-def* links is a weak-update. Figure 15(a) shows the location factored SSA form for the example program in Figure 8(a). As can be seen from the figure, the fact that statement s10 may modify *b* or *c* is represented by a weak-update to each of the two variables. With this SSA form, when a constant propagation algorithm comes to statement *s11*, it finds out that the strong-update a_2 has a value of 4 and replace the read of variable *a* in statement *s11* with 4. However, the constant propagation algorithm finds that definition b_2 is a weak-update. Thus, both definitions b_1

and b_2 may contribute to the read of *b* at *s11*. Because definitions b_1 and b_2 have different values, replacing the read of *b* at *s11* with a constant is prevented.

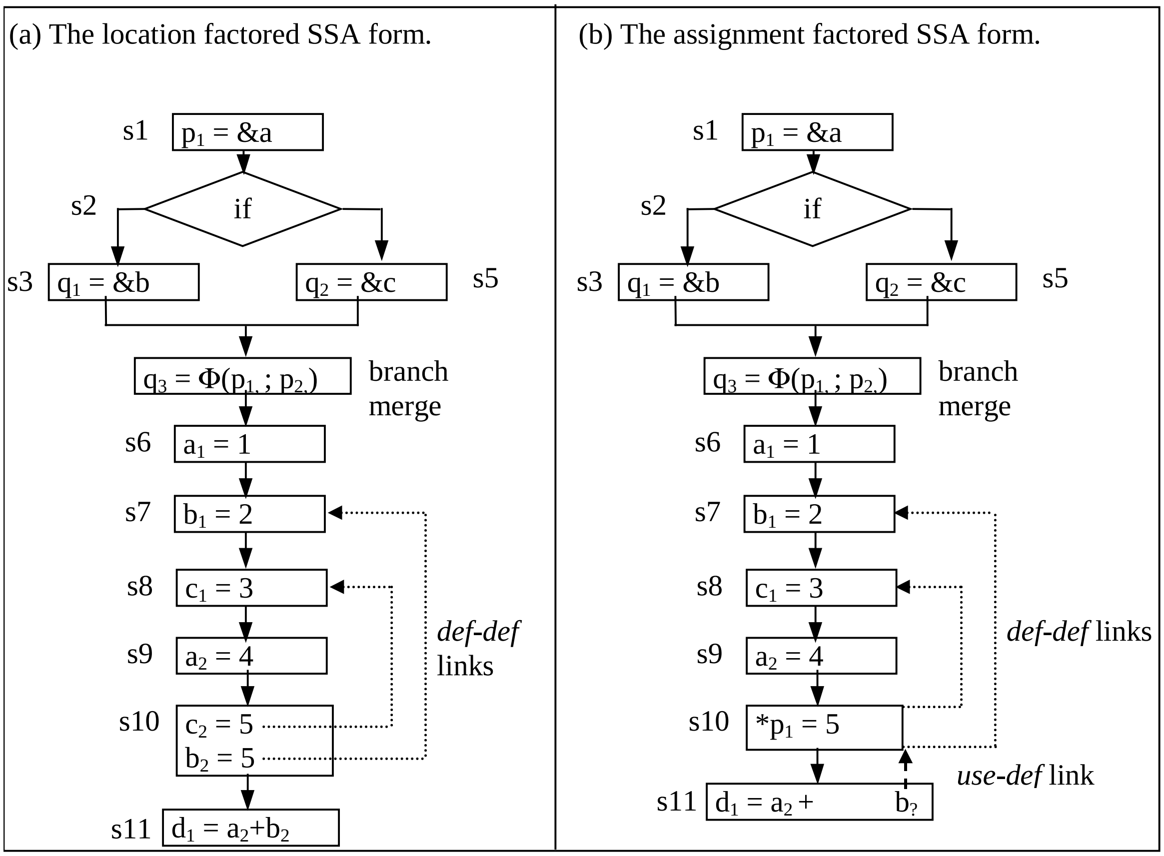


Figure 15: Comparing a location factored SSA form with an assignment factored SSA form using the example program in Figure 8(a).

An *assignment factored* SSA form, on the other hand, creates only one definition for the assignment [Ste95]. This definition has *n def-def* links pointing to the preserved definitions to each location. As shown in Figure 15(b), the fact that statement *s10* may modify variables *b* or *c* is represented by a weak-update which is associated with two *def-def* links. However, representing statement *s10* this way brings up a new problem, i.e., the *use-def* link for the read of *b* at *s11* cannot be represented by a version of variable *b*. This is because b_1 is the only version of b in the SSA form, and as we have already discussed, b_1 cannot provide links to all the definitions that reach the read. There are two possible solutions. One solution is to create an explicit *use-def* link to link the read of *b* to the definition at *s10* (see Figure 15(b)). Another solution is to create a virtual variable to represent all the memory locations accessible via *a*, *b*, and **p*, and replace the appearances of these symbolic names with versions of the virtual variable [Wol96]. In both solutions, there is a spurious *use-def* link from the read of *b* at *s11* to the definition c_1 created at *s8*. Burke and Choi discuss a method to avoid this spurious information by introducing a set call DMOD (definitely modified) to represent the variables that are definitely modified by each assignment statement [BC92].

Thus, the location factored SSA form and the assignment factored SSA form are different in their complexity and preciseness. The location factored SSA form tends to create more definitions than the assignment factored SSA form does. Assume, in the original program, *E* is the number of assignments and *V* is the number of variables. The number of definitions (not including ϕ-function for CFG joint nodes) in a *location factored* SSA form, in the worse case, is proportional to the product of *E* and *V*. The number of definitions in an assignment factored SSA form, on the other hand, equals to *E*. However, as we have shown in Figure 15(b), the assignment factored SSA form may introduce spurious *use-def* information by treating alias as a transitive relationship [Ste95] while the location factored SSA form can avoid the problem.

Our extended SSA form is a location factored SSA form. We create *n regular* definitions for an indirect assignment that may modify *n* locations. Moreover, we also create *n* β-definitions to provide *def-def* links to the definitions that reach the assignment but are not killed by the assignment. This location factored approach allows more accurate alias information in the SSA form and enables a refinement to the alias information in the SSA form. Although the number of definitions in our SSA form may potentially be quadratic in the size of the original program in the worse case, our experiments show that with an aggressive pointer analysis, the size of our extended SSA form has only a small increase over the size of the original program.

3.4.2 A Comparison with Hasti and Horwitz's SSA Form

Assume an indirect assignment may modify *n* memory locations. Our SSA form is similar to Hasti and Horwitz's SSA form in two ways. First, both SSA forms create a *regular* definition for each memory location that may be modified. Second, both SSA forms create a ϕ-definition (or β-definition) for each memory location that may be modified to represent the assignment's weak-update effect to the memory location.

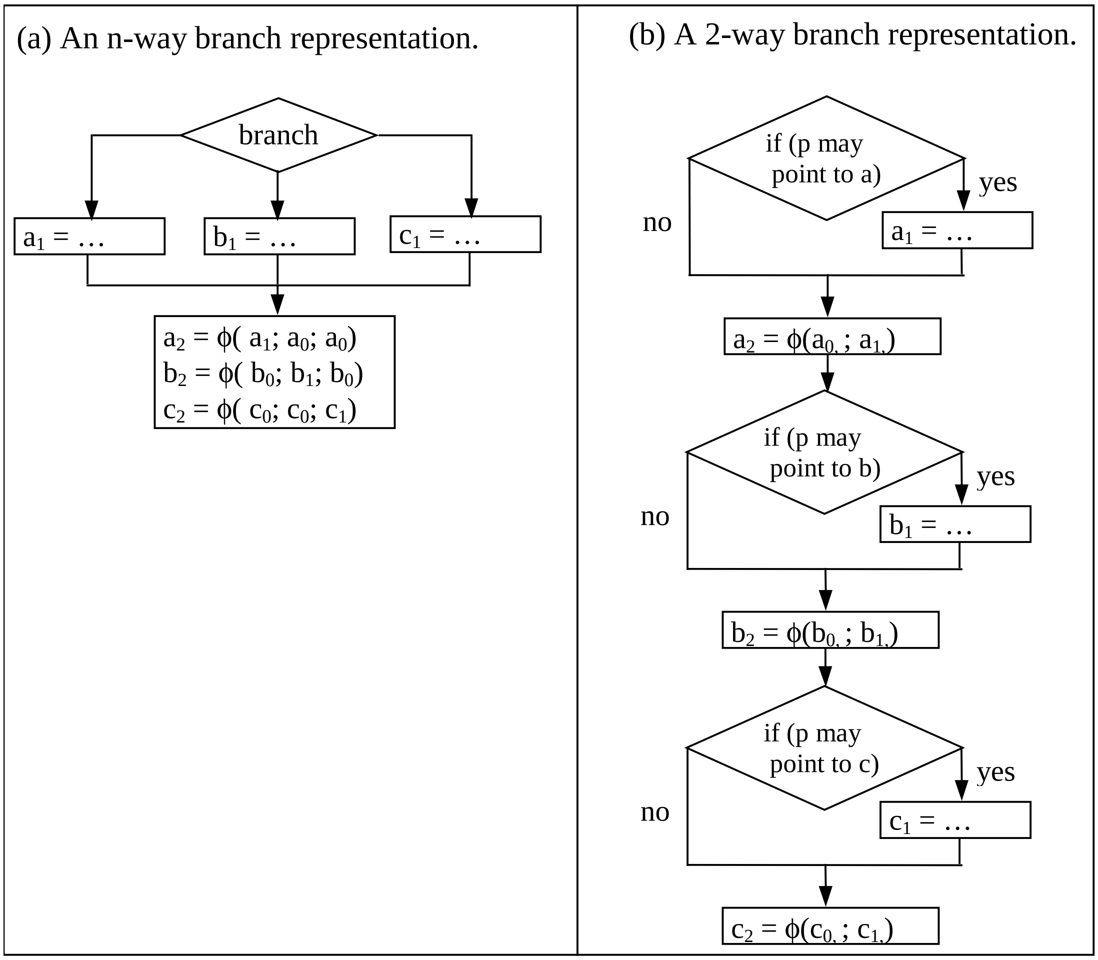


Figure 16: Comparing an n-way branch approach with a 2-way branch approach using indirect assignment "*p = …" (assumption: pointer *p* may point to *a*, *b*, and *c*; definitions a_0, b_0, and c_0 reach the indirect assignment).

However, the main difference between our approach and their approach is that they treat an indirect assignment as one *n*-way branches while we treat an indirect assignment as *n* 2-way branches. Because the number of operands in a ϕ-function (or β-function) for a branch statement equals the number of branches, the ϕ-functions in their SSA form have n operands while the β-functions in our SSA form have two operands. Figure 16 uses an example to illustrate this difference. In the example, we use an indirect assignment "*p = …" and assume that the pointer p have three points-to targets: a, b and c. Figure 16(a) handles the indirect assignment as one 3-way branches. It creates three ϕ-functions, each ϕ-function has three operands. On the other hand, Figure 16(b) handles the indirect assignment as three 2-way branches. It creates three β-functions, each β-functions has two operands. As can be seen, treating an indirect assignment as a sequence of *2*-way branches can reduce the number of ϕ-function (or β-function) operands and result in a more compact representation. Moreover, the branch conditions for the 2-way branches are directly related to the alias conditions. This makes it easy to determine the *predicate* for each β-function operand.

3.4.3 Partial SSA Forms Versus Full SSA Forms

Exhaustively accommodating pointer alias information into an SSA form may expand the size of the program representation, especially when a pointer analysis algorithm is not very precise and the result contains many spurious alias information. The overhead in the program representation can be reduced by incorporating only part of the pointer alias information into the SSA form. We call an SSA form with all the pointer alias information associated with each pointer dereference a *full* SSA form. On the other hand, we call an SSA form with only part of the pointer alias information associating with each pointer dereference a *partial* SSA form.

It is possible to incorporate only part of the pointer alias information into an SSA form. Here, we use Figure 17 to illustrate that by creating only the *live* definitions for indirect assignments, we can reduce the overhead caused by pointer aliasing. Figure 17(a) gives an example program, and Figure 17(b) gives a full SSA form of the program assuming that pointer p may point to variables a, b and c. In the full SSA form, it is not necessary to represent the fact that indirect assignment "*p_1 = …" may modify variables b and c. This is because these two definitions are not *live* in the sense that both variables b and c are redefined by statements *s3* and *s4* before they are used by later statements. Figure 17(c) shows a *partial* SSA form for the program which avoids the overhead to include the fact that the pointer *p* may point to *b* and *c* into the SSA form. Such a *partial* SSA form can provide correct *use-def* links for other analyses and optimizations, and is more efficient than a *full* SSA form.

Cytron et. al. expound on the issue to incorporate only the alias information that are relevant to the optimization being considered to avoid the overhead in creating a full SSA form [CG93]. Their experiments show that creating a *partial* SSA form which includes only the alias information that may affect constant propagation can reduce the memory usage and speed up the optimization process. However, their *partial* SSA form is specific to a certain optimization and may provide incorrect information for other optimizations.

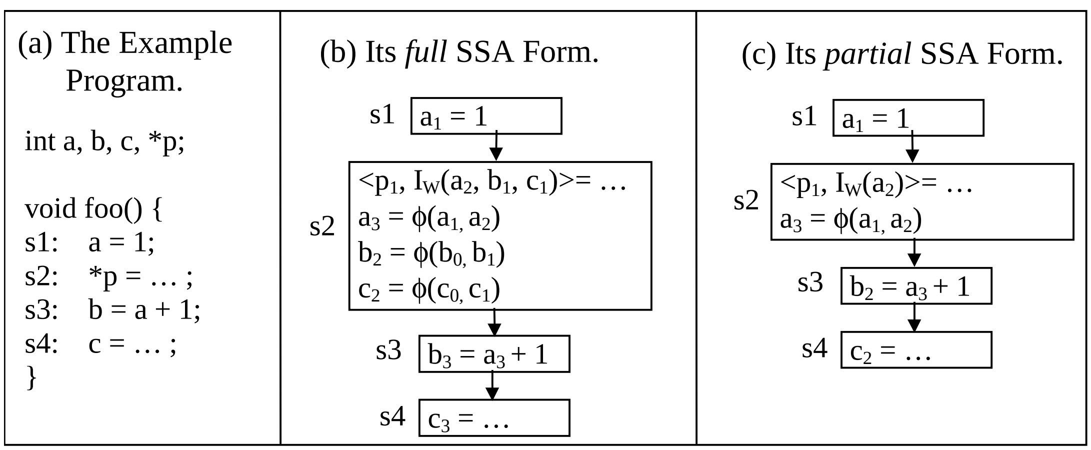


Figure 17: Comparing a full SSA form with a partial SSA form using an example program.

We use a *full* SSA form for two main reasons: (1) we are doing a group of optimizations; (2) our SSA form is also used by later parallelization and optimization phases. It should be noted that although we use a *full* SSA form, our technique to incorporate alias information can also be used to generate a *partial* SSA form.

3.4.4 SSA Forms with Predicates

The idea of augmenting a ϕ-function with predicates in our β-function is originated from the *gated SSA* approach [Hav94, TP95]. The *gated* SSA approach associates one predicate, i.e. a branch condition, with a ϕ-function created at the branch merge point. The *true* or *false* value of the predicate determines that the ϕ-function value should equal the operand from the corresponding branch. This gated SSA approach cannot directly represent a merge of more than two values, or a merge of two values whose valid conditions are not mutually exclusive. We extend the *gated SSA* approach by associating one predicate with each ϕ-function operand. This extension can effectively handle merges of multiple values caused by aliases and function calls, and to represent a merge of two values whose valid conditions are not mutually exclusive.

CHAPTER 4

PARTITIONING THE PROGRAM ANALYSIS USING POINTER LEVELS AND AN SSA FORM

In this Chapter, we propose a new partitioned approach to improve the efficiency of flow-sensitive interprocedural dataflow analysis. In contrast to existing partitioned techniques, our approach analyzes pointers at one level and handles one level of indirection at a time to avoid the complexity of handling multiple levels of pointers simultaneously and, thereby, produces a simpler and more efficient pointer analysis algorithm. We present our technique in the context of generating a *Static Single Assignment* (SSA) form [CFR+91] to represent the analyzed results for C programs. Our algorithm generates SSA forms without requiring an additional pointer analysis algorithm to pre-calculate pointer values. It has two main features. First, we avoid the expensive iterative algorithms used in other approaches by deferring the generation of an SSA form for pointers at a given level until all indirections that may access these pointers are known. Second, we collect values for pointers using SSA forms. Our flow-insensitive analysis algorithm using an SSA form can effectively achieve the same level of precision as a less efficient flow-sensitive analysis algorithm using another program representation does.

This chapter is organized as follows. We start with an introduction to some background in performing pointer analysis in Section 4.1. Section 4.2 provides a high-level description of our partitioned approach. Section 4.3 defines some terminology used in describing our algorithm. We present our intraprocedural algorithm in Section 4.4 and describe our interprocedural algorithm in Section 4.5. We also discuss how we handle some language specific features in Section 4.6. Finally, we discuss some other related work in Section 5.7.

4.1 Background

In a program with general pointer usage, it is difficult to determine the memory locations accessed by a pointer dereference. Without any knowledge about the pointer value, a compiler may assume that a pointer dereference may access any memory location in order to ensure the safety of the optimizations. This approach is taken by early versions of GCC. An improvement over this conservative approach is to assume that a pointer dereference may only access the addressed variables in the program [HH98]. This assumption is based on the assumption that only the variables whose addresses are taken via the address operator "&" can be accessed by pointer dereferences. This assumption is true in a program without using "raw" numbers to construct absolute addresses for variables. However, both approaches are too conservative and may lead to a loss of opportunity to perform parallelization and optimization. Collecting more precise pointer values via pointer analysis can improve the effectiveness of compiler parallelization and optimization.

The purpose of pointer analysis is to collect pointer values to be used for subsequent memory disambiguation. A pointer analysis algorithm has to first decide how to pass the collected information to the later parallelization and optimization phases in the compiler, i.e. the representation of the information. Moreover, a pointer analysis algorithm has to work out a fine line between the efficiency of the algorithm and the preciseness of the analyzed result.

4.1.1 Representing Pointer-Related Information

The representations for pointer related information can be divided into two broad categories: *points-to* representation [EGH94, WL95] and *alias pair* representation [Lan92, LR92, CBC93]. A points-to representation is usually storage-based. It first abstracts the set of all accessible *memory locations*[3] by a finite set of compile-time names. For example, the memory locations corresponding to a variable *p* may be abstracted by the name *p*; and all dynamic allocated memory locations may be abstracted by the name *heap*. Then, based on this abstraction, we describe the points-to relationship between a pair of compile-time names. For example, the assignment statement "*p = &a*" creates a points-to relationship between the compile-time names *p* and *a*, and can be represented by a points-to pair *PointsTo(p, a)*.

[3] Landi et. al. use another term *fixed-locations* [LRZ93].

An alias pair representation, on the other hand, does not use the abstraction of storage. Instead, it relies on symbolic names which provide ways to refer to the memory locations in a program. It also lists all the pair of symbolic names that may be aliases. A symbolic name can be a variable name, or a sequence of dereferences and structure field accesses. Two symbolic names are aliases if they may access the same memory location. For instance, after the assignment statement "*p = &a*", **p* and *a* access the same memory location. Thus, **p* and *a* are aliases, represented by an alias pair *Alias(*p, a)*. Because the "legal" symbolic names in a program can be unlimited in the presence of recursive data structures, a technique called *k-limiting* is often used to limit the number of symbolic names by merging the names beyond *k* dereferences away from a variable [LH88, HPR89, LR92, CWC90]. Both the points-to representation and the alias pair representation can be augmented with a flag to indicate whether the relationship *may* hold or *definitely* holds.

In general, points-to pairs provide a more compact representation for pointer-related information than that provided by alias pairs. For example, considering a program segment with two pointer assignments "pp = &p" and "p = &a", the pointer values created by the assignments can be represented by two points-to pairs: *PointsTo(pp, p)* and *PointsTo(p, a)*. However, similar information requires three alias pairs to represent: *Alias(*pp, p)*, *Alias(*p, a)* and *Alias(**pp, a)*.

Our algorithm uses the points-to representation for two main reasons. First, the points-to pairs directly represent the memory locations that may be accessed by dereferencing a pointer. Thus, it provides more direct information to generate location factored SSA forms (see Section 3.4.1). Second, as we have just discussed, the points-to pair representation is more compact and informative than the alias pair representation. A more detailed comparison of the points-to representation with the alias pair representation can be found in Section 4.7.2.

4.1.2 Flow-Insensitive Versus Flow-Sensitive

Similar to other dataflow analyses, a pointer analysis algorithm can be *flow-insensitive* or *flow-sensitive*. A flow-insensitive pointer analysis does not consider the intraprocedural control-flow while a flow-sensitive algorithm does. In other words, a flow-insensitive pointer analysis treats a program as a set of statements and computes a *program-wide* result. A flow-sensitive pointer analysis, on the other hand, treats a program as a sequence of statements and computes *statement-wide* results.

We use the example program in Figure 18(a) to further illustrate the difference between a flow-insensitive pointer analysis and a flow-sensitive pointer analysis. A flow-insensitive pointer analysis calculates two points-to pairs: *PointsTo(p, a)* and *PointersTo(p, b)* for the whole program. Other analysis algorithms which use this points-to information have to assume that both statement *s2* and statement *s4* may modify variables *a* and *b*. In contrast to a flow-insensitive algorithm, a flow-sensitive pointer analysis calculates one points-to pair *PointsTo(p, a)* for statements *s2*, and another points-to pair *PointsTo(p, b)* for statements *s4*. From this points-to information, other analysis algorithms can determine that statement *s2* only modifies variable *a* and statement *s4* only modifies variable *b*.

(a) One example program.	(b) Another example program.
s1: p = &a;	s1: p_1 = &a;
s2: *p = … ;	s2: * p_1 = … ;
s3: p = &b;	s3: p_2 = &b;
s4: *p = … ;	s4: * p_2 = … ;

Figure 18: Two example programs for comparing flow-insensitive pointer analysis with flow-sensitive pointer analysis.

Thus, a flow-insensitive pointer analysis is generally less precise but more efficient than a flow-sensitive algorithm. Many efficient flow-insensitive pointer analysis algorithms have been developed [BCC+94, Ste96a, Ste96b, HP97]. There are also a large number of flow-sensitive algorithms [CBC93, LRZ93, EGH94, Ruf95, Wil95, HP98] with a goal to collect more precise pointer values.

The most notable relationship between a flow-insensitive algorithm and a flow-sensitive algorithm is that if each variable in a program has only one definition, the two algorithms will achieve the same analysis result. For example, for the program in Figure 18(b), both pointer p_1 and pointer p_2 are assigned only once. A flow-insensitive algorithm computes two points-to pairs $PointsTo(p_1, a)$ and $PointsTo(p_2, b)$ for the whole program while a flow-sensitive algorithm computes the same information for each statement in the program. Hasti and Horwitz exploit this relationship between a flow-insensitive algorithm and a flow-sensitive algorithm and use the SSA form to improve the result of a flow-insensitive pointer analysis [HH98]. They believe that performing a flow-insensitive pointer analysis on an SSA representation of a program is more efficient than performing flow-sensitive pointer analysis on other representations of the same program. Our partitioned technique further extends their idea of using SSA forms to improve the efficiency and the precision of the analysis of a program with pointers.

4.1.3 Context-Insensitive Versus Context-Sensitive

Most of the recent published research work on alias analysis have focused on interprocedural techniques. These techniques can be classified into two broad categories: context-sensitive and context-insensitive. A context-insensitive approach produces a single approximation for a procedure to summarize its effect on all of its calling contexts while a context-sensitive approach distinguishes a procedure's effect in different calling contexts. A context-sensitive approach, in general, is more expensive but can produce more precise alias information.

The presence of pointers makes it difficult to perform context-sensitive interprocedural analysis. As shown in Figure 19(a), the procedure *foo*'s side effect depends on the input values of a pointer *pp*. The statements following the procedure calls, *s3* and *s6*, on the other hand, cannot be analyzed until the procedure's modification side effect is known. This posts a difficulty in ordering the analysis of the procedures in a program. Existing approaches often need to interleave the interprocedural analysis and the intraprocedural analysis as shown in Figure 19(b).

Currently, there are three major approaches to the context-sensitive analysis of programs with pointers. One approach uses tags to distinguish the interprocedural values from different calling contexts. Tags can be aliases reaching the call sites [LR92, CBC93, Ruf95, SL96] , or call strings which represent the call stack when the procedure is entered [CBC93, EGH94, HS94, SP81]. Another context-sensitive approach is to re-analyze a procedure each time when a call to the procedure is encountered during the analysis [EGH94]. This approach has an exponential complexity because the program invocation graph is potentially exponential to the number of procedures. Wilson and Lam proposed a *partially transfer function* approach to obtain context-sensitive results efficiently [WL95, Wil97]. Their approach is based on the observation that given a set of input values of the relevant pointers, a procedure's side effect can be partially summarized. Thus, they can group the calling contexts by their input alias patterns and perform one analysis to calculate the procedure's side-effect under an input alias pattern. Our approach achieves context-sensitivity by two means: tagging the interprocedural pointer values with call-chains and parameterizing the return values of the pointers modified by a procedure.

(a) An example program.

```
    int a, *p, *q, **pp;
    main () {
s1:     pp = &p;
s2:     foo();
s3:     *p = … ;
s4:     pp = &q;
s5:     foo();
s6:     *q = …;
}
void foo() {
s7:     *pp = &a;
}
```

(b) Interleaving the intraprocedural analysis with the interprocedural analysis.

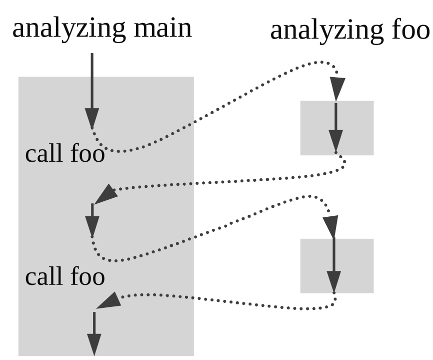


Figure 19: An example to illustrate that pointers complicate a context-sensitive analysis.

4.1.4 Partitioned Analysis

Most of the existing approaches to pointer analysis are either too conservative or too inefficient. Recent research suggests that analyzing all pointers simultaneously, as most of the existing approaches do, is the primary obstacle to efficient pointer analysis. To address the efficiency of pointer analysis, researchers have recently explored techniques to partition the pointer analysis into a group of subproblems [ZRL96, Ruf97, ZRL98].

There are several approaches to partition pointer analysis. A research group at McGill University applies different techniques to analyze heap storage (usually for recursive data structures) [GH95, GH96] and stack storage (usually for non-recursive data structures) [EGH94]. Similar to their approach, we also separate the analysis of recursive data structures and non-recursive data structures. We focus on the efficiency in analyzing non-recursive data structures. Zhang et. al. put the pointers that may contain the same value in the same group for the purpose of partitioning [ZRL96, ZRL98]. Ruf groups variables by their types (declared types or inferred types) and analyzes one variable group at a time [Ruf97]. A partitioned approach, in general, can improve efficiency due to several reasons:

- Separating subproblems with different characteristics allows the flexibility of applying an algorithm tailored to each subproblem [ZRL96]. For example, the analysis of recursive data structures and non-recursive data structures requires different techniques [EGH94, WL95].

- A more efficient analysis algorithm, which is not suitable for a whole-program analysis, may be used for each subproblem. For example, the analysis of a subproblem with only one level pointers has a polynomial complexity while a whole-program analysis which handles multiple levels of pointers simultaneously would have an exponential complexity [LR90, MLR^{+}93, PLR94, Ruf97].

- Smaller subproblems require less storage to analyze [Ruf97].

- Independent subproblems enable parallel analysis [Ruf97].

4.2 The Idea of Our Partitioned Approach

We propose a new method to partition the analysis of a program into a sequence of subproblems. Our approach is different from existing partitioned approaches in three ways. First, we use variable declared types in the partition and handle type casts as an exception. Our study of the SPEC benchmarks shows that type casts in those programs follow certain patterns, and we can handle most of those patterns without loss of precision. Moreover, the problem caused by type casts is not unique to our approach, but rather common to all approaches that use type information for purposes such as modeling run-time storage [WL95, YHR99]. Second, our partitioned technique is primarily for the analysis of non-recursive data structures, which constitute the majority of data structures in programs. We can use other existing techniques, such as the techniques described in [GH95, GH96], to analyze recursive data structures. Third, our method guarantees that each subproblem analyzes pointers at only one level and handles only one level of indirection. Handling one level of pointers at a time is an advantage because it avoids the complexity of handling multiple layers of dependencies due to multiple levels of pointers [LR90, WL95, Ruf95] and, thus, produces a simpler and more efficient algorithm.

(a) The example program.

```
int **pp, *p, *q,
    a, b;
s1:   p = &a;
s2:   q = &b;
s3:   pp = &p;
s4:  while (condition) {
s5:      *pp = &a;
s6:       *q = 3;
s7:      pp = &q;
s8:  }
       …
```

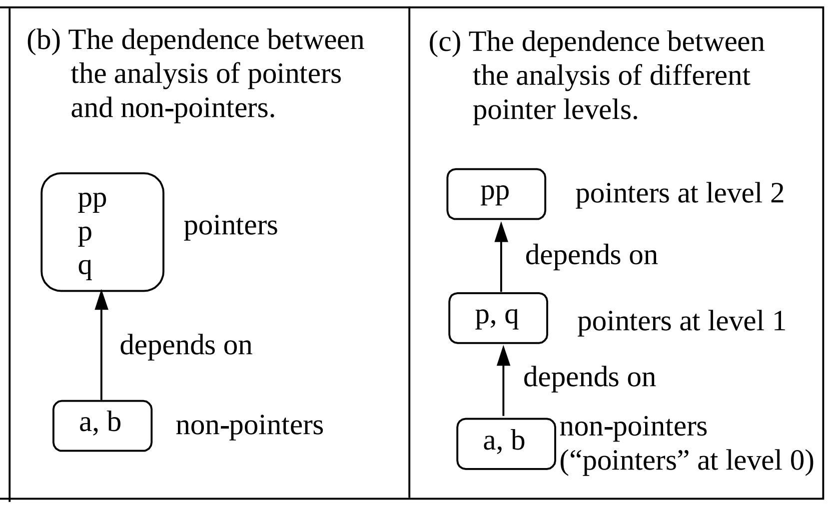


Figure 20: An example program and the dependence among the analysis of its variables.

We present our partitioned technique in the context of generating *Static Single Assignment* (SSA) forms [CFR+91] to represent the analyzed results. SSA forms have been used as an efficient program representation that provides a sound base for powerful optimization and parallelization. Many existing algorithms to generate an SSA form for a program require an additional pointer analysis algorithm to precalculate pointer values [CFR+91, CG93, CCF94]. For example, to construct an SSA form for the example program in Figure 20(a), an existing approach may use type information to separate variables into two groups: pointers and non-pointers, and assume that the analysis of non-pointers depends on the analysis of pointers (see Figure 20(b)). Thus, it generates an SSA form in two steps. The first step applies a pointer analysis algorithm to analyze all pointers simultaneously. The second step uses the precalculated pointer values to determine the variables that may be accessed via pointer dereferences and generates an SSA form for all variables.

This approach is inefficient for two main reasons. First, simultaneously analyzing all the pointers requires an inefficient iterative algorithm. How an indirect assignment using a pointer dereference, such as *s5* in Figure 20(a), may modify the values of other pointers depends on the values of the pointer being dereferenced (*pp*). Each time we discover a new fact about the pointer, we need to re-analyze the assignment. When the program being analyzed contains multiple procedures, each time we discover a new fact for a procedure, all the other procedures that either call the procedure or are called by the procedure need to be re-analyzed. Thus, a typical pointer analysis algorithm may iteratively analyzes the whole program several times. It also needs an extra analysis to the whole program to make sure that no new facts can be further discovered. Such an algorithm is inefficient and prohibitively expensive for large programs. Second, there may be some duplicated work between the two steps which makes the algorithms less efficient. For example, many existing pointer analysis algorithms are based on an iterative approach to construct an SSA form only for pointers [Wil97, HH98, Lu98]. If the first step applies such an algorithm and only passes the collected points-to values to the second step, the SSA form for pointers needs to be regenerated in the second step.

We address the above two problems by partitioning the generation of the SSA form using variable *pointer level*s. The *pointer level* of a variable is the maximum level of indirection to the variable. For example, in definition "*int **pp, *p*", the pointer levels for *pp* and *p* are 2 and 1 respectively (see Section 4.3.1). The key insight of our partitioned scheme is that, without type casts, dereferencing a pointer can only access other pointers at lower levels. Thus, the analysis of a pointer depends on, but does not affect, the analysis of other pointers at higher levels (see Figure 20(c)). We partition the variables in the program into three groups and order the analysis as follows. We first generate the SSA form for *pp*, and calculate its values using the SSA form. Then, when we move on to generate the SSA form for *p* and *q*, we know that the use of *pp* in statement *s5* may point to *p* or *q*. Thus, the SSA form generation for *p* and *q* does not require an iterative algorithm. After generating the SSA form for *p* and *q*, we use the SSA form to determine that the use of *q* in statement *s6* may point to *a* or *b*. Lastly, we construct the SSA form for *a* and *b*. The final SSA form for the program is a simple union of the above three SSA forms.

In general, our partitioned approach divides the process to construct the SSA form for a program with a maximum of *n* level pointers into *n+1* subproblems. The *i*th subproblem constructs the SSA form for the pointers at level *i* and non-pointer variables are handled as pointers at level *0*. Our algorithm constructs the SSA form for each pointer level, starting from the highest pointer level down to pointer level *0*. After generating the SSA form for a pointer level, we analyze the pointers at that level using the SSA form and pass the pointer values down to the analysis of the succeeding pointer levels. Ordering the analysis this way is based on the assumption that dereferencing a pointer can only access pointers at lower levels. This is true in programs without type casts.

Our approach to generate SSA forms is efficient for several reasons. First, each subproblem handles pointers at only one level to avoid the complexity of handling multiple levels of pointers simultaneously. Second, we avoid the expensive iterative approach used in other algorithms [Wil97, HH98, Lu98] by deferring the SSA generation for a pointer until after all the indirections that may access the pointer are known. Third, the analysis of each pointer level is performed on an SSA form. An efficient flow-insensitive analysis of the program in an SSA form can effectively achieve the same precision as a less efficient flow-sensitive analysis of the program in other intermediate representation [HH98]. Lastly, in contrast to existing SSA generation algorithms, our approach does not require an additional pointer analysis to precalculate pointer values, thus, avoids duplicating the SSA generation for pointers as we have just described.

Although we present our partitioned technique in the context of constructing SSA forms for C programs, this technique can also be applied to collect pointer values, or to create other program representation, such as program dependence web [BMO90], for other programming languages with general pointer usage.

4.3 Basic Concepts

4.3.1 Pointer Level

As we already mentioned, the *pointer leve*l of a data structure corresponds to the maximum level of indirect accesses to the data structure (for non-recursive data structure, without type casts). For example, in definition "*int **pp, a*", the pointer levels for variables *pp* and *a* are *2* and *0*, respectively. Notice that non-pointers are treated as pointers at level *0*. The variable p*s* in definition "*struct {int **pp, a;} *ps*" has a pointer level of *3* because there can be at most three level of indirect accesses to *ps*: **(*(*ps).pp)*. An indirect access to a variable, such as "**ps*" in the above example, is a *dereference* to the variable. We refer to a variable with pointer level *n* as an *n*-level variable. Without type casts, a dereference to an *n*-level variable accesses other variables with a pointer level of less than *n*.

The pointer-level of a non-recursive variable is defined by its declared type and can be calculated as follows:

- The pointer-level of an arithmetic type is *0*.
- The pointer-level of a type with the form "*T **" is $1 + x$, where *x* is the pointer-level of type "*T*".
- The pointer-level of a structure or a union equals to the maximum pointer-level of its field types.

In general, the pointer level for a variable of arithmetic type or a pointer of arithmetic type equals to the number of "*" in the variable definition. However, the pointer level for a structure variable or a pointer to a structure variable is more complicate to calculate, and usually does not equal to the number of "*" in the variable definition.

It should be noticed that our definition of the pointer level is different from Landi and Ryder's definition [LR90]. They simply refer to the number of "*" in a variable definition as the pointer level of the variable. Using their definition of pointer level, dereferencing a MemObj at level *l* may access other MemObjs at a level higher than *l*. For example, in definition "*struct {int **pp, a;} *ps, s*", MemObjs *ps* and *s* are at pointer level 1 and 0 respectively. Assume *ps* points to *s*, a dereference expression *(*ps).pp* access the *pp* field in *s*, which is at pointer level 2.

4.3.2 Representing Memory Locations

We use a *memory object* (MemObj) to represent a run-time memory location that stores information. There are two kinds of MemObjs: static MemObj and dynamic MemObj. A static MemObj represents a memory location created for a compile-time variable, or a field in a structure/union variable. A dynamic MemObj, on the other hand, represents memory objects generated by memory allocation statements (such as a malloc statement) at run-time.

A MemObj may be associated to one or more *types*. We call a MemObj of a single type a *regular* MemObj, and a MemObj of more than one type a *super* MemObj. Similar to the *type* of a variable, the type of a MemObj provides the information about the size of the MemObj and how the MemObj can be accessed.

We create MemObjs for scalar, array and structure variables according to the following rules:

- We create a MemObj to represent the memory location for a scalar variable. The type of the MemObj equals to the type of the scalar variable. For example, as shown in Figure 21, we create a MemObj with type "*int*" for variable *i* which is declared as "*int i*". We also use the variable name as the name of the MemObj.

- We create a MemObj to represent the memory location for an array of scalars. The type of the MemObj equals to the type of the array. As shown in Figure 21, we create a MemObj with a type "*int [10]*" for variable *a* which is declared as "*int a[10]*". This MemObj has a name "*a[]*".

- We create one or more MemObjs to represent the memory location for each field of a structure variable. If a field is a scalar or an array of scalars, we create one MemObj to represent the memory location for the field. If a field is a structure, we create a group of MemObjs to represent each field of the structure.

- We create a group of MemObjs to represent the memory location for an array of structures. Each MemObj represents an array of a field in the array of the structures.

variable declaration	MemObj name	type attribute
int i	i	int
int a[10]	a[]	int [10]
struct { int b; char c; } s;	s.b s.c	int char
struct { int b; char c; } sa[10];	sa[].b sa[].c	int [10] char [10]

Figure 21: Examples for representing memory locations for scalar, array and structure variables.

There are three ways to represent the memory location for a union variable. The first method is to create one MemObj for each field of the union variable. The second method is to group the fields in the union variable by the pointer levels of the field types, and create one MemObj to represent each field group. For example, the fields in union variable *u* in Figure 22 may have a pointer level of 1 or 0. Thus, we create two MemObjs for the variable: one super MemObj call “u.i+c” with types “int” and “char”, and one regular MemObj call “u.p” with a type “int *” (see Figure 22). Notice that we construct the name for a MemObj of a field group using the field names separated by “+”. The second method is to create one *super* MemObj for the union variable. The type attribute for the *super* MemObj consists of all the field types of the union variable. For instance, we create a *super* MemObj called “u” to represent the memory location for the union variable in Figure 22(a). This *super* MemObj has types “int”, “char” and “int *” as attributes.

There are two ways to represent dynamically allocated memory objects. One way is to create one regular MemObj to represent the memory objects allocated by a memory allocation statement. The type of the MemObj is derived from the type used in the type cast operation of the allocation statement. For example, the MemObj for the allocation statement “ p = (int *) malloc(sizeof(int) *10)” has type “int [10]”. If such a type cast operation does not exist, the MemObj has type “char []”. Another way is to create one *super* MemObj to represent all the dynamically allocated memory objects. The type attribute for the *super* MemObj is a collection of types derived from the type cast operators in all the allocation statements as we have just described.

variable declaration	MemObj name	type attribute
union { int i; char c; int *p; } u;	 u.i+c u.p	 int, char int *

Figure 22: An example for representing the memory locations for a union variable.

We use MemObjs instead of variables in our dataflow value representation because of two main reasons: (1) a dynamically allocated memory object does not correspond to any variable; and (2) a structure or a union variable may correspond to several memory objects.

Our MemObj representation allows us to distinguish between fields in a structure. However, we do not distinguish different elements in an array, nor different elements in an aggregate of recursive data structures.

4.3.3 Naming the Intermediate SSA Forms

As we have just described, our algorithm constructs the SSA form once pointer level at a time, starting from the highest pointer level n down to pointer level 0. The SSA form for a pointer level l contains the SSA representation for all the accesses to the l-level MemObjs in the program. It also explicitly represent the MemObjs accessed by each dereference to the l-level MemObjs. The program intermediate representation after constructing the SSA form for pointer level l is represented as SSA^l. It contains the SSA forms for all the pointer levels not lower than l.

4.4 Analyzing One Procedure

Figure 23 gives an overview of the algorithm to analyze a program with only one procedure. It partitions the SSA generation for a program with pointers at a maximum pointer level of n into $n+1$ subproblems. Each subproblem handles the MemObjs at a certain pointer level. Non-pointers are handled as pointers at level 0. Subproblems are handled from the highest pointer level down to pointer level 0.

For l = maximum-pointer-level down to 0, **do**:
  (1) Construct the SSA form for all the MemObjs at pointer level l.
    (1.1) **For** each statement that may access an l-level MemObj, **do**:
      (1.1.1) Analyze assignments.
      (1.1.2) Insert β-definitions at control-flow joint nodes.
      (1.1.3) Determine the version number for each appearance of an l-level MemObj.
  (2) Analyze the pointers at pointer level l using the SSA form.
  (3) Passing pointer values to the analysis of succeeding pointer levels.
    (3.1) **For** each dereference to an l-level MemObj, **do**:
      (3.1.1) Create I-functions that represent the variables implicitly accessed by the dereference.

Figure 23: The algorithm to analyze a program with one procedure.

Our algorithm is based on the assumption that dereferencing an *l*-level MemObj can only access MemObjs at pointer levels lower than *l*. Thus, there are two important properties in handling multiple levels of pointers: (1) the analysis of a pointer does not require the values of other pointers at lower pointer levels; and (2) the analysis of a pointer may require the values of other pointers at higher pointer levels for memory disambiguation. The first property makes it possible to analyze MemObjs one pointer level at a time, while the second property makes it necessary to order the analysis from the highest pointer level down to the lowest pointer level.

In order to analyze MemObjs at pointer level *l*, we first construct the SSA form for the MemObjs. We analyze the related assignments to create definitions that represent these statements' modification effects and explicitly represent the MemObjs that are implicitly accessed. We also insert β-functions at CFG joint nodes to represent merges of values. After creating the SSA form for the MemObjs, we collect pointer values using the SSA form. Finally, we pass the collected pointer values to the analysis of the succeeding pointer levels by recording the potentially accessed MemObjs for each dereference to an *l*-level MemObj. In the remainder of this section, we discuss some of these steps in more detail.

4.4.1 Analyzing Assignments

In order to analyze an assignment, we identify the *l*-level MemObjs that may be written by the assignment and create definitions to represent the modification effect. We first use the algorithm *Find_Level_MemObj* given in Figure 24 to identify the *l*-level MemObjs that may be written by the assignment. Then, we create a new definition for each MemObj that may be written by the assignment. If a new definition is a weak-update, we also create a β-definition to represent both the old definition that reaches the assignment and the new definition created by the assignment reach after the assignment. More specifically, we classify assignments into four categories:

- Scalar assignments. The left-hand side expression of a scalar assignment is a scalar variable or a structure/union field access. The algorithm *Find_Level_MemObj* returns a scalar MemObj for the expression. We create a new definition for the MemObj. This definition is a strong-update to the MemObj. Thus, no β-definition is created.

- Array assignments. The left-hand side expression of an array assignment is an access to an array element. The algorithm *Find_Level_MemObj* returns a MemObj representing all the elements in the array. We create a new definition for the MemObj. This definition is a weak-update to the MemObj and may not overwrite the previous definition to the MemObj. Thus, we create a β-definition, $d = \beta((d_1, true); (d_2, true))$, for the assignment to represent the merge of definitions. In the β-definition, d_1 is the definition to the MemObj that reaches the assignment and d_2 is the new definition created by the assignment.

- Aggregate assignments. The left-hand side expression of an aggregate assignment is a structure/union variable. The algorithm *Find_Level_MemObj* returns one or more MemObjs for the expression. The MemObjs represent fields in the structure/union variable. We create a definition for each MemObj returned from the algorithm *Find_Level_MemObj*. These definition are strong updates. Thus, no β-definition is necessary.

- Indirect assignments. The left-hand side expression of an indirect assignment is an indirect access using pointer dereferences. The algorithm *Find_Level_MemObj* returns one or more MemObjs for the expression. For each MemObj returned from the algorithm *Find_Level_MemObj*, if the pointer being dereferenced must point to the MemObj and the MemObj is not an array element, the assignment creates a strong-update to the MemObj. Otherwise, the assignment creates a weak-update to the MemObj. For each weak-update d_2 created by the assignment, we also create a β-definition, $d = \beta((d_1, p_1); (d_2, p_2))$, for the assignment to represent the merge of definitions. In the β-definition, d_1 is a definition that reaches the assignment and defines the same MemObj as d_2 does; p_1 represents the condition that d_1 is not killed by d_2 and p_2 represents the condition that d_2 is generated by the assignment.

```
//input: l – pointer level
//        exp – an expression
//output: a set of l-level MemObjs that may be accessed by the expression
Set_Of_MemObj Find_Level_MemObj (int l, Exp exp) {
    (1) all_obj = Find_MemObj(exp);
    (2) result = {};
    (3) For each MemObj m in all_obj, do:
          If (m has a pointer level l)
             result = result ∪ {m};
    (4) Return result;
}
//input: exp – an expression
//output: a set of MemObjs that may be accessed by the expression
Set_Of_MemObj Find_ MemObj (Exp exp) {
    (1) result = {};
    (2) Switch (exp)
       (2.1) Case (exp is a scalar variable or an array access):
              result = {the MemObj for the scalar variable or the array elements};
       (2.2) Case (exp is a structure/union variable)
              result = { all the MemObjs for the structure/union variable};
       (2.3) Case (exp has the form "str_exp.field_name",
                    where str_exp is a structure access and
                    field_name is a field of the structure/union):
             (2.3.1) tmp_result = Find_MemObj(str_exp);
             (2.3.2) For each MemObj m in tmp_result, do:
                        If (m is a MemObj for a field with name field_name
                           in a structure/union)
                           result = result ∪ {m};
       (2.4) Case (exp has the form of  "*sub_exp"):
              // The I-function is created by step (3.1.1)
              // in the algorithm given in Figure 23.
             (2.4.1) Get the I-function for exp;
             (2.4.2) For each MemObj m in the I-function operands, do:
                        result = result ∪ {m};
    (3) Return result;
 }
```

Figure 24: Calculating the *l*-level MemObjs accessed on the left-hand side of an assignment..

4.4.2 Handling CFG Joint Nodes

We illustrate our algorithm to insert β-definition to represent merges of definitions caused by each type of CFG joint nodes in Figure 25 and Figure 26. To simplify our discussion, we do not show the predicates for the β-functions in the algorithm description. It is trivial to add such predicates. Our method in creating β-definition for a certain pointer level requires a single pass over the program. This is similar to the algorithm discussed in [BM94].

Our algorithm uses a data structure *reach* with type *ReachDefinition* to record the definitions of the MemObjs that reach the current program point. For a MemObj *m*, *reach[m]* represents the definition of *m* that reaches the current program point. We use two procedures in our algorithm: procedure *Handle_Procedure_Joint* (Figure 25) and procedure *Handle_Joint* (Figure 26). Procedure *Handle_Procedure_Joint* first initializes the value of *reach[m]* with the definition that represents the initial value of MemObj *m*. It then passes data structure *reach* and all the statements in the procedure to procedure *Handle_Joint*. Procedure *Handle_Joint* maintains the value of *reach* and creates the β-functions for CFG joint nodes found in the input statements.

// input: proc – the IR for a procedure.
// output: the IR for the procedure with the new β-functions annotated to the IR.
void *Handle_Procedure_Joint(Procedure proc)* {
    (1) Initialize a data structure called *reach*.
        For each MemObj *m*, **do**:
        reach[m] = the definition that represents the initial value of the MemObj.
    (2) Assume *body* represents the statements in *proc*,
        call *Handle_Joint(reach, body)*.
}

Figure 25: The algorithm to insert β-functions in CFG joint nodes (procedure *Handle_Procedure_Joint*).

```
// input: reach – a data structure that records the reaching definition.
//                     For a MemObj m, reach[m] is the definition to m
//                     that reaches the current program point.
//          stmt_list – a list of statements that may contain CFG joint nodes
//                         which require β-functions.
// output: IR with the new β-functions.
void Handle_Joint(ReachDefinition reach, StmtList stmt_list) {
    For each statement stmt in stmt_list, do
        Switch (stmt type)
            (1.1) Case (stmt is an assignment):
                    Use the definitions created by stmt to update reach.
            (1.2) Cast (stmt is a break statement or a return statement)
                    Annotate the statement with a copy of reach to represent the
                    definitions that reach the statement.
            (1.3) Case (stmt is an if statement):
                    (1.3.1) else_reach = a copy of reach;
                    (1.3.2) Handle_Joint(reach, ThenPart(stmt));
                    (1.3.3) Handle_Joint(else_reach, ElsePart(stmt));
                    (1.3.4) Insert_Phi_For_If(stmt, reach, else_reach);
                    (1.3.5) Update reach so that it represents the definitions
                              that reach out of the statement.
            (1.4) Case (stmt is a loop):
                    (1.4.1) loop_reach = a copy of reach;
                    (1.4.2) Handle_Joint(loop_reach, LoopBody(stmt));
                    (1.4.3) Insert_Phi_For_Loop(stmt, reach, loop_reach);
                    (1.4.4) For each β-definition m_i = β(…) that is in the entry-node
                              and defines MemObj m, do
                              Assume m_j is the old definition to m that reaches the loop.
                              Replace all appearance of m_j with m_i in the loop body.
                    (1.4.5) Update reach so that it represents the definitions that
                              reach out of the loop.
            (1.5) Case (stmt is an unstructured region):
                    (1.5.1) keep_reach = a copy of reach;
                    (1.5.2) For each SESE block sese_block in stmt, do
                              array_reach[sese_block] = a copy of keep_reach;
                              Handle_Joint(array_reach[sese_block],
                                                BlockBody(stmt, sese_block));
                              Insert_Phi_For_Region(stmt, reach, array_reach);
                    (1.5.4) For each β-definition m_i = ϕ(…) that is in the region
                              entry-node and defines MemObj m, do
                              Assume m_j is the old definition to m that reaches the region.
                               Replace all appearances of m_j with m_i in the region.
                     (1.4.5) Update reach so that it represents the definitions that
                               reach out of the region.}
```

Figure 26: The algorithm to insert β-functions in CFG joint nodes (procedure *Handle_Joint*).

As shown in Figure 26, procedure *Handle_Joint* classifies the statements in the input statement list into five categories and handles statements in each category accordingly. It also recursively calls itself in order to handle nested control structures. The five categories of statements are: assignments, *break* or *return* statements, *if* statements, loops and *unstructured regions*. Of these five categories, we only need to insert β-functions for *if* statements, loops and *unstructured regions*. We now discuss our scheme to insert β-definitions in CFG joint nodes caused by the statements in these three categories.

If statements. Procedure *Insert_Phi_For_If(stmt, then_reach, else_reach)* creates β-definitions for an if-statement. Both parameters *then_reach* and *else_reach* have the type *ReachDefinition*. Parameter *then_reach* represents the definitions that reach the CFG joint node (see Figure 5(a)) from the *then-branch* while parameter *else_reach* represents the definitions that reach the CFG joint node from the *else-branch*. For each MemObj *m*, if *then_reach[m]* does not equal to *else_reach[m]*, we create a β-definition to represent a merge of two definitions: $m_{new} = \beta(then\_reach[m], else\_reach[m])$.

Loops. As shown in Figure 5(b) and Figure 5(c), there are two joint nodes for each loop: *entry-node* and *exit-node*. The β-definitions in the *entry-node* represent merges of values at the entry of the loop while the β-definitions in the *exit-node* represent merges of values when the loop exits. Procedure *Insert_Phi_For_Loop(stmt, reach, loop_reach)* creates β-definitions for both the *entry-node* and the *exit-node* of a loop. In the procedure, parameter *reach* represents the definitions that reach the entry of the loop statement while parameter *loop_reach* represents the definitions that reach the end of the loop body. For each MemObj *m*, if *reach[m]* does not equal to *loop_reach[m]*, we create a β-definition to represent a merge of the two definitions: $m_{new\_entry} = \beta(reach[m], loop\_reach[m])$. This β-definition is put in the *entry-node* of the loop.

The handling of the *exit-node*s for *do-while-loops* is different from the handling of the *exit-node*s for *do-loops* and *while-loops* because a *do-while-loop* has at least one iteration while a *do-loo*p or a *while-loop* may have zero iteration. Without early exits, the β-definitions put in the *exit-node* of a *while-loop* or a *do-loop* are very similar to the β-definitions put in its *entry-node*. More specifically, for each MemObj *m*, if *reach[m]* does not equal to *loop_reach[m]*, we put the following new β-definition in the exit-node of the loop: $m_{\text{new\_exit}}$ = β(*reach[m]*, *loop_reach[m]*). In order to handle loops with earlier exit conditions caused by *break* statements, we annotate each *break* statement with data structure *break_reach* which represents the definitions that reach the statement. In the presence of *break* statements, for each β-definition $m_{\text{new\_exit}}$ we have just created, we add operand *break_reach[m]* to the β-definition to represent the definition that reaches outside the loop via a *break* statement.

Similarly, we handle the *exit-node* of a *do-while-loop* according to whether the loop contains early exits. If the loop does not contain early exits caused by *break* statements, no β-definition is put in the *exit-node* because only the definitions that reach the end of the loop body reach out of the loop and merge of values is not needed in this situation. However, if the loop contains early exits, for each MemObj *m* which has different reaching definitions from different exits, we create one β-definition to represent the merge of these different definitions.

Unstructured regions. As we have seen in Section 2.4, an unstructured region consists of a set of SESE blocks. The end of a SESE block may have a *goto* statement that jumps into the beginning of other SESE blocks in the same unstructured region. We conservatively assume that there may be an edge from the end of a SESE block to the beginning of any other SESE blocks in the same unstructured region (see Figure 5(d)).

As shown in Figure 5(d), there is only one *joint-node* for an unstructured region. This joint node represents dataflow values merged from the entry of the unstructured region and the exits from each SESE block in the region. Procedure *Insert_Phi_For_Region(stmt, reach, array_reach)* creates β-definitions for the joint node. Parameter *reach* have type *ReachDefinition*. It represents the definitions that reach the entry of the unstructured region. Parameter *array_reach* is an array of *ReachDefinition*. Assume the unstructured region has *n* SESE blocks. Array *array_reach* has *n* element, each element represents the definitions that reach the end of an SESE block. For each

MemObj *m* that may be modified in an SESE block, we create a β-definition with the form of "m_{new} = β(d_0, d_1, … d_n)" to represent a merge of definitions. The β-function on the right-hand side of the β-definition has *n+1* operands. The first operand d_0 equals to *reach[m]*, while the i+1 operand d_i (i = i .. n) equals to *array_reach[i][m]*.

After inserting β-definitions in control-flow joint nodes to represent merges of values from different branches, we add a version number to each appearance of a MemObj so that each version of the MemObj is defined only once, and a read of the MemObj has the same version number as the definition that directly reaches the read.

4.4.3 Pointer Analysis Using an SSA Form

Figure 27 shows our working-list based algorithm to calculate the points-to values for each version of a MemObj (a version of a MemObj is also referred to as a definition to the MemObj). There are three major data structures used in the algorithm: *flag, points_to* and *work_list*. Data structure *flag* has one element for each definition to represent the status in calculating the value for the definition. An element in *flag* for a definition m_i is represented as *flag*[m_i]. Its value can be START, GOING or END. Data structure *points_to* also has one element for each definition to represent the value of the definition. An element in *points_to* for a definition m_i is represented as *points_to*[m_i]. Data structure *work_list* contains the working list. An element in *work_list* can be either a definition that corresponds to a version of a MemObj or an *action* to calculate the points-to values of a definition. An *action* to calculate the points-to values of a definition has the form "m_i = *MergeWith*(*list_def*)", in which *list_def* is a list of definitions. It merges the current value of m_i with the values of the definitions in *list_def* to calculate the new value of m_i.

We initially put all definitions that are used in dereference expressions in *work_list* because our goal is to calculate the points-to values for those definitions. Then, we remove an element from *work_list*, evaluate the element until *work_list* becomes empty. During the evaluation of a *work_list* element, we may add more elements to *work_list*. There are two types of *work_list* elements: definitions and actions. To handle the definitions in *work_list*, we classify the definitions into three categories: a simple

(1) **For** each definition to an *l*-level MemObj *m*, say m_i, **do**
    flag[m_i] = START;
    points_to[m_i] = {};
(2) *work_list* = {};
(3) Put all definitions that are used in a dereference expression in *work_list*.
(4) **While** (*work_list* is not empty), **do**
    (4.1) Remove an element *e* from *work_list*.
    (4.2) **If** (*e* is a definition m_i and flag[m_i] == START)
        Switch (m_i)
        (4.2.1) **Case** (the definition assigns the address of a variable *t* to m_i)
            points_to[m_i] = {*t*};
            flag[m_i] = END;
        (4.2.2) **Case** (the definition copies the value of t_j to m_i)
            Switch (*flag*[t_j])
            (4.2.2.1) **Case** END
                points_to[m_i] = *points_to*[t_j];
                flag[m_i] = END;
            (4.2.2.2) **Case** GOING
                points_to[m_i] = *points_to*[t_j];
                flag[m_i] = GOING;
                Append action "m_i = *MergeWith*({t_j})" to *work_list*
            (4.2.2.2) **Case** START
                Append definition t_j to *work_list*;
                Append definition m_i to *work_list*;
        (4.2.3) **Case** (the definition is a β-definition)
            Assume *tmp_list* contains all the definition in the β-function operands.
            Append action "m_i = *MergeWith*(*tmp_list*)" to *work_list*.
    (4.3) **Else if** (e is an action with form "m_i = *MergeWith*(*list_def*)")
        (4.3.1) *tmp_list* = {};
        (4.3.2) *old_value* = *points_to*[m_i]
        (4.3.2) **For** each definition t_j in *list_def*, **do**:
            If (*flag*[t_j] == END || *flag*[t_j] == GOING)
                points_to[m_i] = *points_to*[m_i] ∪ *points_to*[t_j];
            Else // *flag*[m_j] == START
                Append t_j to *work_list*;
            If (*flag*[t_j] == START || *flag*[t_j] == GOING)
                tmp_list = *tmp_list* ∪ { t_j };
        (4.3.3) **If** (*tmp_list* is empty)
                flag[m_i] = END
            Else if (*flag*[m_i] == GOING and
                all definitions in *tmp_list* have a flag GOING and
                old_value == *points_to*[m_i])
                flag[m_i] = END
            Else
                If (*flag*[m_i] == START *and points_to*[m_i] is not empty)
                    flag[m_i] = GOING
                Append action "m_i = *MergeWith*(*tmp_list*)" to *work_list*;

Figure 27: An algorithm to calculate pointer values using the SSA form.

definition that assigns an address of a MemObj to another MemObj, a copy definition that copies the value of a MemObj to another MemObj, and a β-definition. If we encounter a definition that assigns the address of a MemObj t to the definition, we set the points-to value of the definition to $\{t\}$, and set the *flag* of the definition to END (see step (4.2.1) in Figure 27). If we encounter a definition that copies the value of a MemObj t_j to another MemObj m_i, we first check the value of *flag*[t_j] and calculate m_i as follows (see step (4.2.2) in Figure 27):

- If *flag*[t_j] equals to END, the points-to value of t_j is already known. We copy *points_to*[t_j] to *points_to*[m_i] and set *flag*[m_i] to END.

- If *flag*[t_j] equals to GOING, the points-to value of t_j is partially known. We copy *points_to*[t_j] to *points_to*[m_i] and set *flag*[m_i] to GOING. We also add an action to *work_list* so that we can further add the new points-to values of t_j to the points-to set of m_i .

- If *flag*[t_j] equals to START, the points-to value of t_j is not known. We append t_j and m_i to *work_list*. This will lead to a calculation of the value for t_j and then a calculation of the value for m_i.

If we encounter a β-definition, we create an *action* to calculate the value of the definition from the values of the β-function operands. We put the action to *work_list* (see step (4.2.3) in Figure 27).

If the current *work_list* element is an action with the form "m_i = *MergeWith*(*list_def*)", we merge the current points-to value of m_i with the points-to value of other definitions on the definition list *list_def* to calculate the new value for m_i. If all the definitions in *list_def* have known points-to values, we finish calculating the points-to value for m_i. Otherwise, we put the rest of the definitions whose points-to values are not known into a new list call *tmp_list*, and append a new action "m_i = *MergeWith*(*tmp_list*)" to *work_list* (see step (4.3) in Figure 27).

The efficiency of the algorithm can be improved by ordering the definitions initially put in *work_list* so that if the value of m_i depends on the value of t_j, t_j is ordered before m_i. Such an ordering can avoid evaluating m_i before the value of t_j is ready. In the presence of cycles in *use-def* links, we need to repeatedly evaluate the MemObjs involved in the cycles. This repeated evaluation is limited only to the MemObjs in the cycles of the *use-def* links and is more efficient than the iterative algorithms used in other approaches which repeatedly analyze the whole program. The step (4.3.3) in Figure 27 can prevent the algorithm from falling into an infinite loop in the presence of cycles in the *use-def* links.

4.4.4 Passing Pointer Values to the Succeeding Pointer Levels

After collecting the pointer values for the *l*-level MemObjs, we construct an I-function for each dereference to an *l*-level MemObj. This I-function represents the MemObjs that are implicitly accessed (read/written) in the dereference expression. These implicitly accessed MemObjs are at a pointer level lower than *l*. Thus, with the annotated I-functions, the analysis of each succeeding pointer level has the information about what MemObjs may be accessed by a dereference to a higher level MemObj.

Recording the pointer values by associating each dereference with an I-function is efficient because the value of a pointer is recorded only at a program point where the pointer is dereferenced. We avoid recording unused pointer values.

4.4.5 An Example

We now use the example program in Figure 20(a) to illustrate the above algorithm. The maximum pointer level for the variables in the example program is 2. Thus, the analysis of the program is partitioned into three subproblems.

The first subproblem analyzes the variable at pointer level 2, i.e. variable *pp*. We first construct the SSA form for the variable as shown in Figure 28(a). We then use the working-list based algorithm to calculate the pointer values of the variable as follows:

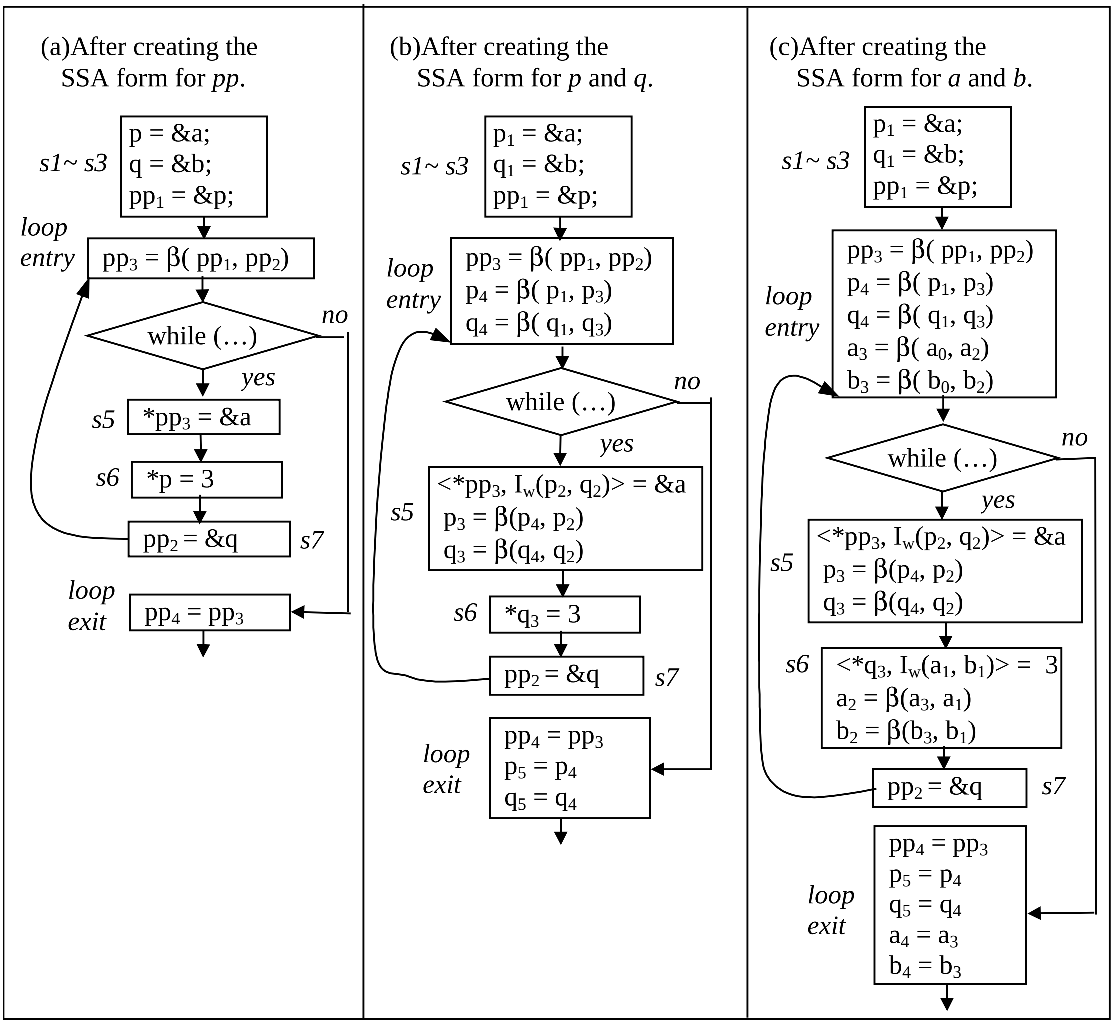


Figure 28: Partitioning the SSA form construction for the program in Figure 20(a) (Note: (1) We assume a_0 and b_0 represent the initial values of *a* and *b* respectively; (2) We use " $pp_4 = pp_3$" to represent that pp_4 has the same β-function on its RHS as pp_3 does. Similar notation is also used for p_5 and q_5.).

- At step (3) of the algorithm given in Figure 27, we initialize *work_list* to $\{pp_3\}$. Because pp_1 and pp_2 are not dereferenced, they are not put in *work_list*. We also initialize all elements in array *flag* with START, and all elements in array *points_to* with an empty set.

- We first remove definition pp_3 from *work_list*. We add definitions pp_1 and pp_2, and action "pp_3=*MergeWith*(pp_1, pp_2)" to *work_list* while evaluating pp_3. Thus, we have *work_list* = $\{pp_1; pp_2; pp_3=MergeWith(pp_1, pp_2)\}$.

- We then remove definition pp_1 from *work_list* and calculate that pp_1 points to variable *p*. After that, we have *work_list* = $\{pp_2; pp_3=MergeWith(pp_1, pp_2)\}$, $flag[pp_1] = END$, and $points\_to[pp_1] = \{p\}$.

- Next, we remove definition pp_2 from *work_list* and calculate that pp_2 points to variable *q*. After that, we have *work_list* = $\{ pp_3=MergeWith(pp_1, pp_2)\}$, $flag[pp_2] = END$, and $points\_to[pp_2] = \{q\}$.

step	*flag*				*points_to*				*work_list*
	q_1	q_2	q_3	q_4	q_1	q_2	q_3	q_4	
1	S	S	S	S					q_3
2	S	S	S	S					q_4, q_2 , q_3 =M(q_4, q_2)
3	S	S	S	S					q_2 , q_3 =M(q_4, q_2), q_1, q_3 , q_4 =M(q_1, q_3)
4	S	E	S	S		*a*			q_3 =M(q_4, q_2), q_1, q_3 , q_4 =M(q_1, q_3)
5	S	E	G	S		*a*	*a*		q_1, q_3 , q_4 =M(q_1, q_3), q_3=M(q_4)
6	E	E	G	S	*b*	*a*	*a*		q_3 , q_4 =M(q_1, q_3), q_3=M(q_4)
7	E	E	G	S	*b*	*a*	*a*		q_4 =M(q_1, q_3), q_3=M(q_4)
8	E	E	G	G	*b*	*a*	*a*	*a*, *b*	q_3=M(q_4), q_4=M(q_3)
9	E	E	G	G	*b*	*a*	*a*, *b*	*a*, *b*	q_4=M(q_3), q_3=M(q_4)
10	E	E	G	E	*b*	*a*	*a*, *b*	*a*, *b*	q_3=M(q_4)
11	E	E	E	E	*b*	*a*	*a*, *b*	*a*, *b*	

Figure 29: Calculating pointer values for variables at pointer level 1 using the SSA form in Figure 28(b) (S=START, E=END, G=GOING, M=MergeWith).

- Lastly, we remove action "$pp_3=MergeWith(pp_1, pp_2)$" from *work_list* and calculate that pp_3 may point to variables *p* or *q*. After that, we have *work_list* = {}, $flag[pp_3] = END$, and $points\_to[pp_3] = \{p, q\}$.

After calculating the pointer values for variable *pp*, we construct I-function "$I_w(p, q)$" for the expression $*pp_3$ at statement *s5*. This I-function represents that statement *s5* may write variables *p* and *q*.

The second subproblem analyzes the variables at pointer level 1, i.e. variables *p* and *q*. We first construct the SSA form for the variables as shown in Figure 28(b). We then calculate the pointer values for the variables using the working-list based algorithm. Figure 29 shows the values for the three main data structures *flag*, *points_to* and *work_list* at each step during the calculation. After calculating the pointer values, we know that the use of q_3 at statement *s6* may point to *a* or *b*. Thus, we associate I-function "$I_w(a, b)$" with expression $*q_3$. This I-function represents that statement *s6* may write variables *a* and *b*.

The third subproblem analyzes the variables at pointer level 0, i.e. variables *a* and *b*. We construct the SSA form for the variables as shown in Figure 28(c). Since variables *a* and *b* are non-pointers, there is no necessary to collect pointer values for these two variables. The final SSA form for the whole program is given in Figure 28(c).

4.5 Analyzing Multiple Procedures

4.5.1 Algorithm Description

Figure 30 gives an overview of the interprocedural algorithm to analyze a program with multiple procedures. This algorithm assumes simple source programs without some language specific features such as type casts, recursive data structures and recursive functions. The handling of these features is discussed in Section 4.6.

4.5.1.1 Bottom-up pass

The analysis of a given pointer level *l* starts with a bottom-up pass to construct the SSA form for the *l*-level MemObjs and to analyze the *l*-level MemObjs using the SSA form (see Figure 30 step (1)). This pass handles procedures in a reversed call order because the analysis of a subroutine requires knowledge about the reference and modification effect of all the other subroutines it calls.

SSA form generation. To generate an SSA form for the *l*-level MemObjs in the current subroutine *foo,* we first explicitly represent how the MemObjs are referenced and modified in the subroutine by analyzing assignments and subroutine calls. We have already described the algorithm to analyze assignments when we illustrate the intraprocedural algorithm (see Section 4.4.1). We analyze subroutine calls as follows:

For l = maximum-pointer-level downto 0, **do**
  //bottom-up pass
  (1) **For** each subroutine *foo* in a reversed call order, **do**
    (1.1) Construct the SSA form for the l-level MemObjs accessed by *foo*.
      (1.1.1) Explicitly represent how the MemObjs are referenced and modified by *foo*.
        (1.1.1.1) Analyze assignments.
        (1.1.1.2) Analyze function calls.
      (1.1.2) Insert β-definitions at control-flow joint nodes;
      (1.1.3) Determine the version number for each appearance of the MemObjs.
    (1.2) Analyze the l-level MemObjs accessed by *foo* using the SSA form.
      (1.2.1) Collect pointer values for the MemObjs;
      (1.2.2) Calculate *Ref(foo, l)* and *Mod(foo, l).*
  //top-down pass
  (2) **For** each subroutine *foo* in a call order, **do**
    (2.1) Calculate interprocedural related pointer values.
    (2.2) Propagate pointer values from *foo* to all the subroutines it calls.
    (2.3) Construct I-functions to explicitly represent the MemObjs that are implicitly accessed by each dereference to l-level MemObjs.

Figure 30: Overview of the interprocedural algorithm (assume source programs without some language specific features).

- For each call to another procedure *bar*, we apply *Mod(bar, l)* and *Ref(bar l)* to determine the MemObjs that may be referenced or modified in the subroutine call. If the subroutine call modifies a MemObj *m* with a return value *r*, we create a definition "*m* = *r*" and put the definition in the I_W-function that explicitly represents the subroutine call's modification effect. The return value *r* may be parameterized with the input values to the subroutine. Similarly, if the subroutine call references a MemObj *m*, we put *m* in the I_R-function that explicitly represents the subroutine call's reference effect.

- If a definition created in the above step is a weak-update, i.e. a definition that may not overwrite the old value of the defined MemObj, we create a β-definition to the MemObj that represents the merge of the old value and the new value.

Assume the current procedure is *foo*. To facilitate the calculation of *Ref(foo, l)* and *Mod(foo, l)* described later, for each MemObj *m*, we put a pseudo definition "*m=Input(m)"* to the *entry-node* of *foo*. This definition is called an *input definition* to *m*. Function "*Input(m)*" represents the input value of *m* to *foo*. We also put a *pseudo read* to *m* to the *exit-node* of *foo*. This *pseudo read* is associated with the exit value of *m*.

After representing how *l*-level MemObjs are referenced and modified by each statement in the subroutine, we insert β-definitions at control-flow joint nodes. We also add a version number to each appearance of a MemObj so that each version of a MemObj is defined only once and a read to a MemObj has the same version number as the definition to the MemObj that directly reaches the read (see Section 4.4.2).

Overall, the algorithm to generate the SSA form for a procedure in the interprocedural algorithm (see Figure 24 step (1.1)) is different from that in the intraprocedural algorithm (see Figure 23 step (1.1)) in two ways. First, the interprocedural algorithm needs to handle subroutine calls. Second, the interprocedural algorithm needs to put initial definitions to the *entry-node* of a procedure and pseudo reads to the *exit-node* of a procedure to facilitate the calculation of *Ref* and *Mod*.

Pointer analysis using an SSA form. The interprocedural pointer analysis algorithm is different from the intraprocedural pointer analysis algorithm described in Section 4.4.3. The main reason for the difference is that for the interprocedural pointer analysis, some pointer values are passed into the current procedure from other procedures that call the procedure. Theses pointer values are unknown until the top-down pass when we propagate pointer values from callers to callees. The interprocedural pointer analysis algorithm makes following changes to the intraprocedural algorithm described in Section 4.4.3:

- We add a data structure *finalize* to record the *actions* to calculate the interprocedural related pointer values during the top-down pass. For each definition m_i, if the value of m_i is interprocedural related, *finalize[m_i]* contains an *action* to calculate its value. Otherwise, *finalize[m_i]* is set to NOOP.

- We add one more action called *Input*. For each MemObj *m*, *input[m]* will copy the input value of *m.*

- If a definition m_i is an initial definition to MemObj *m*, we initially set *finalize[m_i]* to *Input(m)*. We also set *flag[m_i]* to END and set *points_to[m_i]* to an empty set.

- We initially put all definitions that are either used in a dereference expression or whose values are visible outside the procedure in *work_list*.

When the pointer analysis terminates, the value of definition m_i is in one of the following three types:

- The value of *finalize[m_i]* is NOOP and the value of *points_to*[m_i] is a set of MemObjs. In this case, the value of m_i is not interprocedurally related and is represented by *points_to*[m_i].

- The value of *finalize[m_i]* is *Input(t).* In this case, the value of m_i is interprocedurally related. It equals to the input value of MemObj *t* and cannot be represented as a points-to set until after the top-down pass described later.

- The value of *finalize[m_i]* is *MergeWith(list_def).* In this case, the value of m_i is also interprocedurally related. To calculate the value of m_i requires a union operation on *points_to[m_i]* and all the values of the other definitions in *list_def*. This calculation is not possible until the top-down pass when the values of all the other definition in *list_def* are known.

After calculating the value of the MemObjs, we calculate *Ref(foo, l)* and *Mod(foo, l),* i.e the *l*-level MemObjs referenced or modified in the current subroutine *foo*. A MemObj is referenced in *foo* if its input definition is read in the subroutine. On the other hand, a

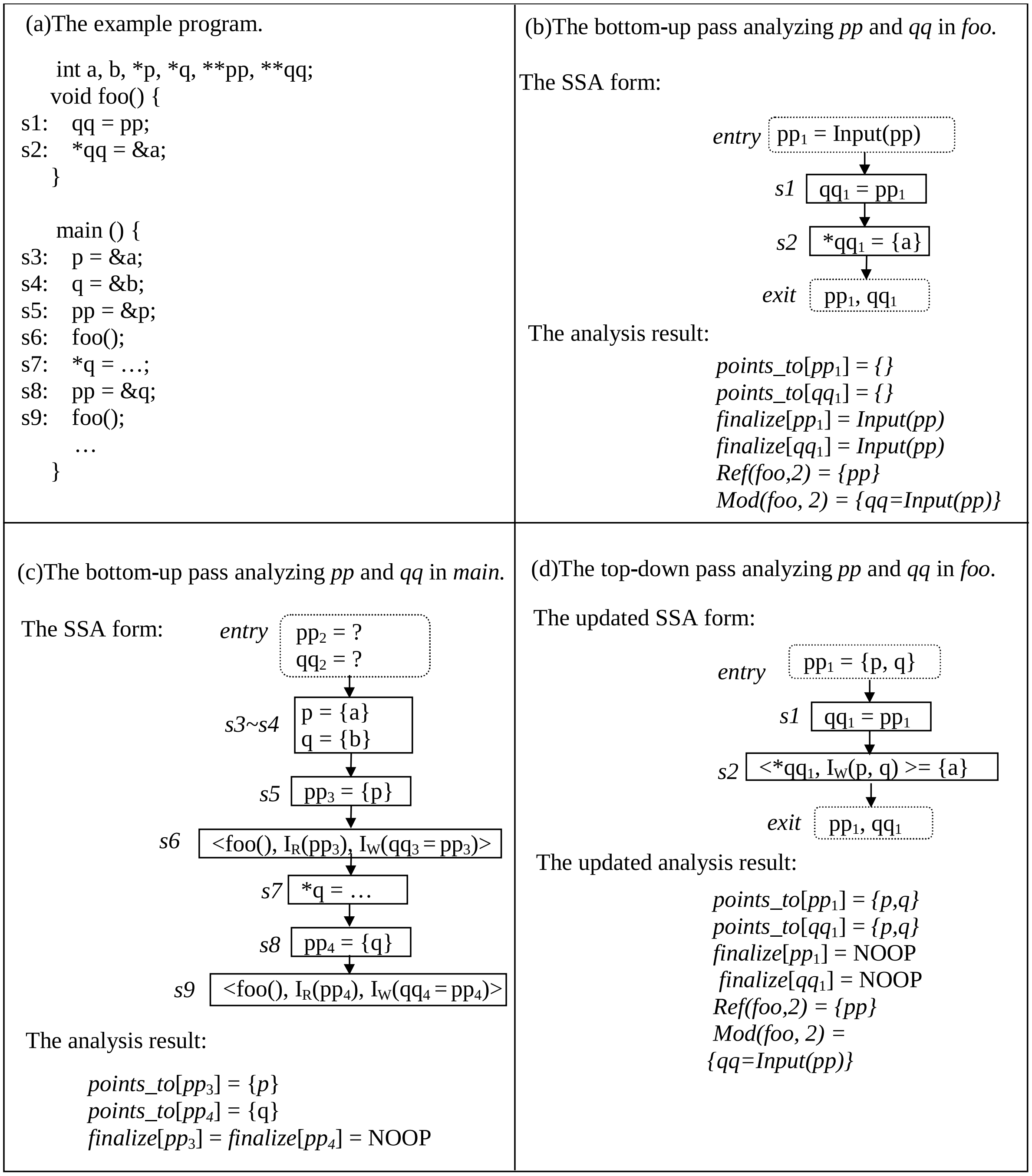


Figure 31: An example program and its intermediate SSA forms. (Note: we do not show the *exit-node* of *main*, because it is not necessary to calculate the *Ref/Mod* for main. We avoid putting an unused initial definition to the *entry-node* of a procedure)

MemObj *m* is modified by *foo* if the pseudo read to *m* in the *exit-node* links to a definition m_j which is not the *input* definition of *m*. If the exit value of a MemObj *m* is m_j and the value of m_j is not interprocedurally related, we put "*m=points_to[*m_j*]*" to *Mod(foo, l)*. Otherwise, if the exit value of a MemObj *m* is m_j and the value of m_j is interprocedurally related, we put action "*m=finalize[*m_j*]*" to *Mod(foo, l)*.

Example 9:

We now describe the bottom-up pass in handling the 2-level MemObjs *pp* and *qq* in the program in Figure 31(a). First, we construct the SSA form for *foo* and calculate the value of qq_1, *Ref(foo, 2)* and *Mod(foo,2)* using the SSA form (see Figure 31(b)). Then, we move on to construct the SSA form for *main*. During the construction of the SSA form, when we encounter the subroutine call in statement *s6*, we apply *Ref(foo,2)* and *Mod(foo, 2)* to determine the MemObjs referenced and modified by the call and associate $I_R(pp)$ and $I_W(qq=pp)$ with the call. Similarly, we also associate $I_R(pp)$ and $I_W(qq=pp)$ with the subroutine call in statement *s9*. After that, we rename the MemObjs by adding a version number to each appearance of *pp* and *qq*. Lastly, we calculate the pointer values for *main* using the SSA form (see Figure 31(c)).

4.5.1.2 Top-down Pass

There are three main tasks in the top-down pass: calculating interprocedurally related pointer values, propagating pointer values from callers to callees, and explicitly representing the MemObjs that may be accessed by each dereference to an *l*-level MemObj (see Figure 30 step (2)). For each procedure, we first perform the operations specified by data structure *finalize* to calculate the points-to sets of the interprocedural related MemObjs from the values input to the procedure. This is possible because we handle procedures in their call order. By the time we start to analyze a subroutine, its input values are already gathered and ready for use. We then traverse each statement in the procedure. If we encounter a subroutine call, for each *l*-level MemObj in the I_R-function associated with the call, we propagate the value of the MemObj from the caller to the callee. If we encounter a dereference to an *l*-level MemObj, we use the points-to set of the MemObj to determine the MemObjs that are implicitly accessed by the dereference, and create an I-function to represent the MemObjs that are implicitly accessed.

Comparing the interprocedural algorithm with the intraprocedural algorithm, the task performed in step (3) in the intraprocedural algorithm (see Figure 23) is performed in the top-down pass in the interprocedural algorithm.

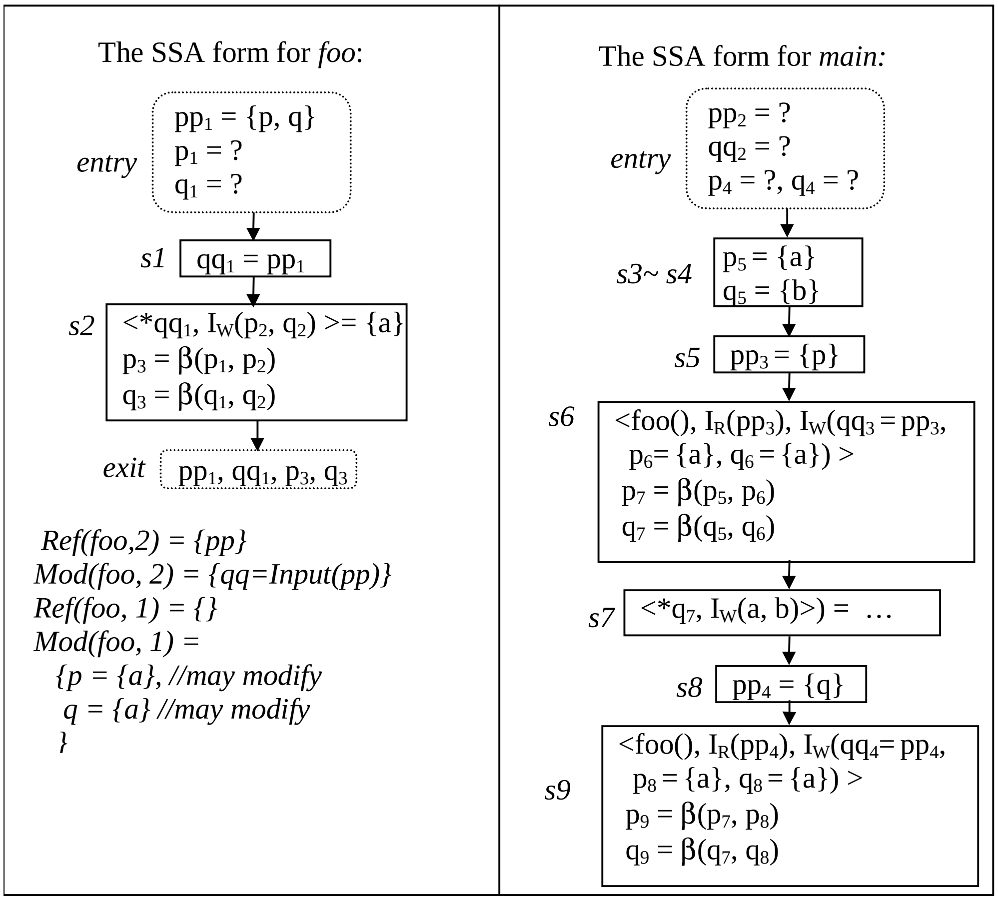


Figure 32: The SSA form for the program in Figure 31(a) (after analyzing *p* and *q*). (Note: we will show the predicates of the β-functions later.)

Example 10:

We now describe the top-down pass in handling the 2-level MemObjs *pp* and *qq* in the example program in Figure 31(a). We first traverse the SSA form for *main* (see Figure 31(c)). During the traversal, when we encounter the subroutine call in statement *s6*, we know that the subroutine call references MemObj *pp* from $I_R(pp_2)$ and propagate the fact "*pp* points to *p*" to callee *foo*. Similarly, in handling statement *s9*, we propagate the fact "*pp* points to *q*" to callee *foo*. Next, we traverse the SSA form for *foo* (see Figure 31(b)). We first calculate the interprocedurally related pointer values and found that both pp_1 and qq_1 have a points-to set $\{p, q\}$. We also set the value in data structure *finalize* to NOOP after calculating the interprocedurally related pointer values. Then, we associate $I_W(p, q)$ with expression $*qq_3$ in statement *s2* to represent that the statement may implicitly writes *p* and *q*. The top-down pass updates the SSA form and the analysis result for *foo* from Figure 31(b) to Figure 31(d).

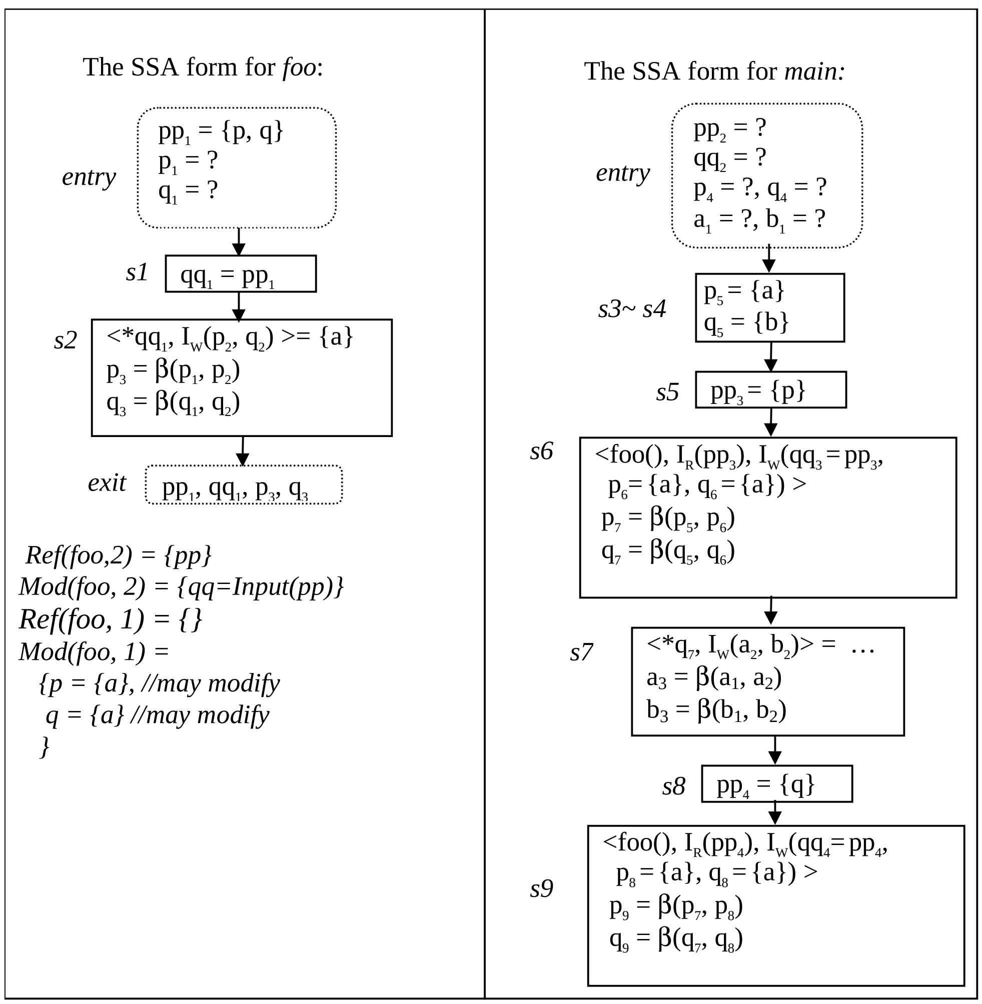


Figure 33: The final SSA form for the program in Figure 31(a). (Note: we will show the predicates of the β-functions later.)

Next, we repeat the above two-phase process to handle the MemObjs at pointer level *1*: *p* and *q*. In the bottom-up phase, we first analyze *foo* and create two β-functions for statement *s2* to represent that the assignment may modify both *p* and *q*. We also calculate *Ref(foo, 1) = {}* and *Mod(foo, 1) = {p={a}, q={a}}*. We then move on to analyze *main*. We associate $I_W(p=\{a\}, q=\{a\})$ with the procedure calls at statements *s6* and *s9*, and create the necessary β-functions to represent the weak-updates. In the top-down phase, we associate I-function $I_W(a, b)$ with the dereference expression in statement s7. After analyzing pointer level 1, the SSA forms for both *main* and *foo*, and the *Ref/Mod* for *foo* are given in Figure 32.

Lastly, we handle the MemObjs at pointer level 0: *a* and *b*. In the bottom-up pass, we find out that *foo* does not access *a* and *b*. We create β-functions for statement *s7* to represent its weak-update to *a* and *b*, and construct the SSA form for *main*. Nothing needs to be done in the top-down phase. The final SSA form for the whole-program is shown in Figure 33.

4.5.2 Context-Sensitivity

Context-sensitivity refers to the property that an algorithm distinguishes a procedure's effect in different calling contexts. We achieve context-sensitivity by two means. First, if a procedure modifies an *l*-level MemObj, the return value of the MemObj is parameterized by the values of the *l*-level MemObjs input to the procedure. We have seen in Example 9, procedure *foo* modifies MemObj *qq* with a return value *Input(pp)*. By notation *Input(pp)*, we defer incorporating the input value of *pp* until *Mod(foo, 2)* is applied to a specific call-site. Second, we tag interprocedural dataflow values with *path*s through the call graph to represent how each value is propagated into a procedure. We now describe our method to tag interprocedural dataflow values. Similar idea in tagging dataflow values have been discussed in [SP81, LR92, CBC93, EGH94, HS94, Lu98].

4.5.2.1 Predicates And Predicate Operations

We use a pair *(s, c)* to encode each *call-site*, where *s* is the callee procedure and *c* is an unique ID among different calls to the same callee. A *call-chain* is an order list of *(s, c)* pairs, represented as $(s_1, c_1)\text{->}(s_2, c_2)\ldots \text{->} (s_n, c_n)$. When such a call-chain is associated with a dataflow value, it represents that the value is first propagated into procedure s_1 via a call-site with ID c_1, then into procedure s_2 via a call-site with ID c_2, and so on. A *predicate* attached to a dataflow value is either *null* or the disjunction($\cup$) of a group of call-chains: $H_1 \cup H_2 \cup \ldots \cup H_n$ ($H_{i=1..n}$ is a call-chain). When the predicate is *null*, the dataflow value is not interprocedurally related. Otherwise, the group of call-chains describe the possible call graph paths under which the dataflow value is passed into a procedure.

For each procedure *foo*, a set of call-chains, represented as *AllCallChain(foo)*, describes all the possible ways to enter the procedure. The negative operator (!) is defined as follows.

Definition 8:

Given a call-chain H_1 of procedure *foo*, $!H_1 = \text{AllCallChain(foo)} - H_1$.

The conjunction operator ($\cap$) of any two call-chains of the same procedure are defined as follows.

Definition 9:

Assume H_1 and H_2 are two call chains of the same procedure *foo*.

$$H_1 \cap H_2 = \begin{cases} H_1 & (H_1 = H_2) \\ false & (otherwise) \end{cases}$$

The conjunction operator ($\cap$) and disjunction operator ($\cup$) follow the commutative law and associative law. Thus, the conjunction operator ($\cap$) of any two predicates $P^1 = H_1^1 \cup H_2^1 \cup \ldots H_n^1$ and $P^2 = H_1^2 \cup H_2^2 \cup \ldots H_m^2$ is defined as follows: $P^1 \cap P^2 = (H_1^1 \cap H_1^2) \cup (H_1^1 \cap H_2^2) \cup \ldots (H_1^1 \cap H_m^2) \cup \ldots (H_n^1 \cap H_m^2)$.

Besides the above operators, operators *Append*, *Returnable*, and *Detach* are also used to calculate the predicates for dataflow values. These three operators are defined as follows.

Definition 10:

Assume predicate P and a call-site (s,c).

$$Append(P,(s,c)) = \begin{cases} (s,c) \qquad (P = null) \\ \\ Append(H_1,(s,c)) \cup Append(H_2,(s,c))... \cup Append(H_n,(s,c)) \\ \qquad\qquad (P = H_1 \cup H_2 ... \cup H_n) \end{cases}$$

The operator to *append* a (s,c) pair to a call-chain $H = (s_1, c_1)\text{->} (s_2, c_2)\ldots \text{->} (s_n, c_n)$ is defined as follows:

$$Append(H,(s,c)) = \begin{cases} (s_1,c_1)-> (s_2,c_2)...-> (s_n,c_n)-> (s,c) \\ \qquad (procedure\ \ s_n\ \ calls\ s\ at\ a\ call-site\ with\ ID\ c) \\ error \qquad (otherwise) \end{cases}$$

Definition 11:

Assume predicate P and a call-site (s,c).

$$\mathrm{Re}turnable(P,(s,c)) = \begin{cases} true \qquad\qquad (P = NULL) \\ \mathrm{Re}turnable(H_1,(s,c)) \cup \mathrm{Re}rurnable(H_2,(s,c))... \\ \qquad\qquad \cup \mathrm{Re}turnable(H_n,(s,c)) \\ \qquad (P = H_1 \cup H_2 ... \cup H_{n)} \end{cases}$$

The operator to check whether a dataflow value with a call-chain $H = (s_1, c_1)\text{->} (s_2, c_2)\ldots \text{->} (s_n, c_n)$ is *Returnable* to a call-site (s, c) is defined as follows:

$$\text{Re}\,turnable(H,(s,c)) = \begin{cases} true & (c = c_n \ \ and \ \ s = s_n) \\ false & (otherwise) \end{cases}$$

Definition 12:

Assume predicate *P* and a call-site *(s,c)*, and operator *Returnable(P, (s,c))* return *true.*

$$Detach(P,(s,c))) = \begin{cases} Detach(H_1,(s,c)) \cup Detach(H_2,(s,c))... \\ \qquad \cup\, Detach(H_n,(s,c)) \\ \qquad (P = H_1 \cup H_2 ... \cup H_{n)} \\ Null \qquad (P = NULL) \end{cases}$$

The operator to *detach* a call-site *(s, c)* from a call-chain $H = (s_1, c_1)\text{->} (s_2, c_2)\ldots \text{->} (s_{n-1}, c_{n-1})\text{->}(s, c)$ is defined as follows:

$$Detach(H,(s,c)) = (s_1,c_1) -> (s_2,c_2)... -> (s_{n-1},c_{n-1})$$

4.5.2.2 Dataflow value with Predicates

Our initial description of the algorithm in Example 9 and Example 10 does not use predicates to tag dataflow values. As a consequence, the dataflow values represented by the resulting SSA forms are not precise. For example, the dataflow information represented by the SSA form in Figure 33 is not precise in several ways:

- It does not distinguish the fact that procedure *foo* modifies *p* when it is entered from the first call-site and modifies *q* when it is entered from the second call-site.
- The dataflow information “the function call at statement *s6* may modify *q* with a points-to set *{a}*” represented by the SSA form is spurious. Likewise, the dataflow information “the function call at statement *s9* may modify *p* with a points-to set *{a}* ” represented by the SSA form is spurious.

- The dataflow information "the function call at statement s6 *may* modify *p* with a points-to set *{a}*" represented by the SSA form could be refined as "the function call at statement s6 *must* modify *p* with a points-to set *{a}*". Similarly, The dataflow information "the function call at statement s9 *may* modify *q* with a points-to set *{a}*" represented by the SSA form would be refined as "the function call at statement s9 *must* modify *q* with a points-to set *{a}*".

- The dataflow information "statement *s7* may modify variable *a*" is spurious. The dataflow information "statement *s7 may* modify variable *b*" can be refined as "statement *s7 must* modify variable *b*".

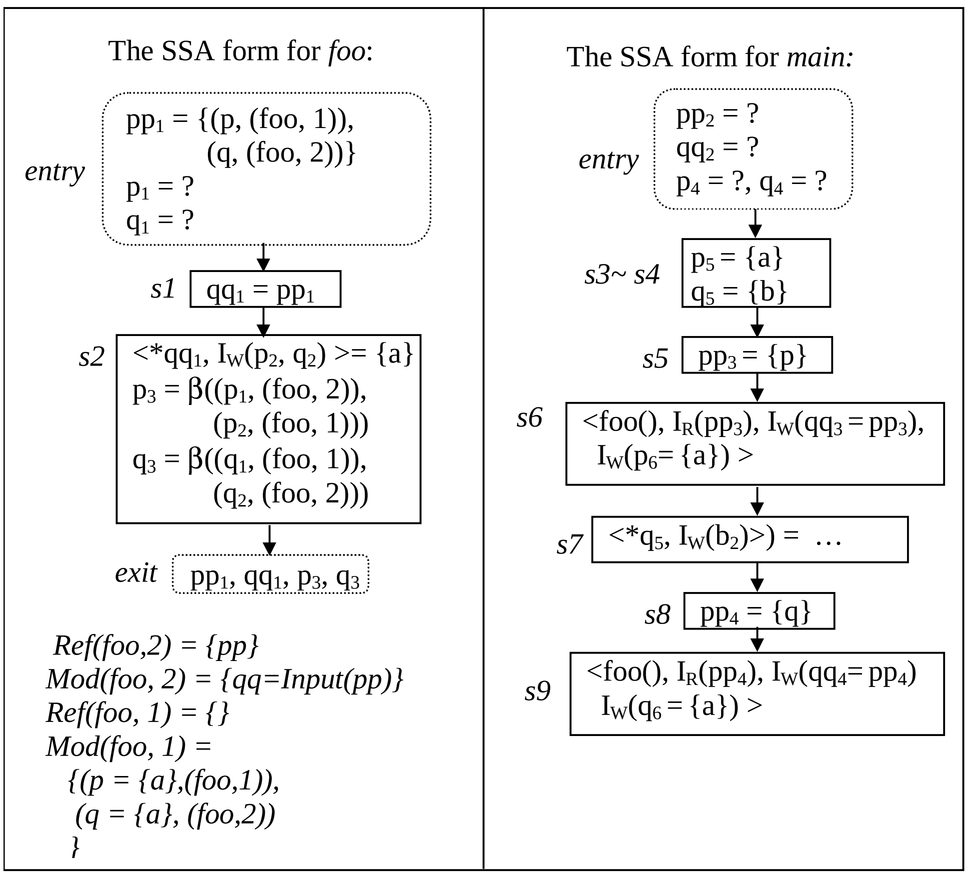


Figure 34: An improved SSA form of the SSA form in Figure 33.

Figure 34 shows an SSA form with more precise dataflow information comparing with the SSA form given in Figure 33. In order to achieve the SSA form in Figure 34, we need to tag the dataflow information with predicates and perform predicate operations during the SSA form generation. We illustrate how predicates are calculated and attached to dataflow information in the SSA form as follows.

Tagging forward propagating values. When interprocedural dataflow values are propagated from caller procedures to callee procedures, we attach the dataflow values with predicates to identify the source where the values are propagated from. For example, as shown in Figure 34, when value "*pp = {p}*" is propagated into *foo* from the first call site, we associate the value with predicate *(foo, 1)*. Similarly, we associate predicate *(foo, 2)* with value "*pp = {q}*". In general, if a MemObj which has a value *v* under predicate *P* is propagated into a procedure from a call site *(s,c)*, we perform operation *Append(P, (s, c))* to add *(s, c)* to the end of each call-chain in *P* (see Definition 10 in page 81).

Conditional modification. An indirect assignment using pointer dereferences may modify different MemObjs under different calling contexts. We associate each operand in the β-function for the indirect assignment with a predicate to represent the condition under which the MemObj is modified. For example, in the SSA representation for indirect assignment *s2*, $\beta((p_1, (foo, 2)); (p_2, (foo, 1)))$ represents that the indirect assignment modifies MemObj *p* only when the procedure is entered from the first call-site (see Figure 34). In this case, *s2* creates definition p_2 and overwrites definition p_1. In general, the fact that an indirect assignment may modify a MemObj under condition P^1 and must modify the MemObj under condition P^2 is represented by "$\beta((p_1, !\ P^2); (p_2, P^1))$". In the above representation, p_1 is an old definition to the MemObj that reaches the assignment and p_2 is a new definition to the MemObj created by the assignment.

A procedure may also conditionally modify some MemObjs. The conditional modification effect of a procedure is represented by associating a predicate with each return value. For example, the notation "*Mod(foo, 1)={(p={a}, (foo,1)), (q={a}, (foo,2))}*" in Figure 34 represents that procedure *foo* modifies *p* with return points-to value *{a}* when it is entered from the first call-site and modifies *q* with return points-to value *{a}* when it is entered from the second call-site.

Conditional reference. Similar to the representation of conditional modification, predicates may also be associated with each MemObj in I_R and *Ref* to represent conditional reference. For instance, if *qq* points to *q* under condition P^1 and points to *t* understand condition P^2, we represent the conditional reference for **qq* as "$<*qq, I_R((q, P^1), (t, P^2))>$".

The conjunction operator *($\cap$)* may also be used to calculate the predicate for conditional reference. For example, assume *qq* may point to *q* under condition P^1 and *q* may point to *b* under condition P^2, then **qq accesses *b* under condition $P^1 \cap P^2$.

Conditionally backward propagating values. If a procedure modifies a MemObj under a certain condition, we need to determine, for each call-site, whether the MemObj is actually modified. In general, we assume *Mod(foo, l)* indicates that a MemObj *m* is modified by *foo* with return value *v* under condition *P*. For each call site *(foo, c)*, we compare the predicate *P* with *(foo, c)* to determine whether *m* is actually modified by the procedure call using operation *Returnable(P, (foo, c))* (see Definition 11). If *m* is modified, *m* is added to the I_W-function associated to the procedure call. A new predicate *P′* is also added to the corresponding β-function operand. The predicate *P′* is calculated by operation *Detach(P, (foo, c)),* which detaches *(foo, c)* from the call-chains in *P* (see Definition 12). For example, *Mod(foo, 1)* indicates that "*p = {a}*" when *foo* is entered from the first call-site and "*q = {a}*" when *foo* is entered from the second call site. Thus, only value "*p = {a}*" is returned to the function call at statement *s6*. This avoids returning spurious value "*q = {a}*" to the statement.

Similarly, if *Ref(foo, l)* indicates that a MemObj *m* is referenced by *foo* under condition *P*, we compare the predicate *P* with a call site *(foo, c)* to determine whether *m* is actually referenced. If *m* is referenced, a pair *(m, P′)* is added to the I_R-function associated to the procedure call.

4.5.3 Complexity

The time complexity for the context-insensitive analysis of the l-level MemObjs is $O(s + t)$, where s is the complexity of constructing the SSA form for the MemObjs and t is the complexity of performing analysis on the SSA form. Both s and t has the complexity of $O(r_l^3)$, where r_l is the size of the subprogram containing accesses to the l-level MemObjs [CFR+91, HH98]. Variable r_l is the maximum of either the number of nodes or edges in the control-flow graph, or the number of (direct or indirect) accesses to the l-level MemObjs in SSA^{l+1} (the program intermediate representation after creating the SSA form for the $(l+1)$-level MemObjs, see Section 4.3.3). The number of accesses to the l-level MemObjs in SSA^{l+1} is not a constant. It depends on the analysis result for the MemObjs at a level higher than l. If there are u_1 dereference expressions to the higher level MemObjs that access l-level MemObjs and the analysis algorithm calculates an average of u_2 points-to targets for each higher level MemObj, SSA^{l+1} has a number of $u_1 \bullet u_2$ indirect accesses to the l-level MemObjs in addition to the direct accesses to the l-level MemObjs. Thus, the value of r_l depends of the analysis result for the MemObjs at pointer levels higher than l.

To achieve context-sensitivity, the SSA form generation and the analysis using the SSA form may perform predicate operations which add an additional complexity to the algorithm. The complexity of the predicate operations directly relates to the complexity of the call-chains used in the predicates. We set a parameter m to limit the maximum length of the call-chain in the predicate so that a call-chain contains only the most recent m call sites. This may cause a merge of the dataflow information propagated beyond the most recent m call sites. Let c denote the average number of call sites to a procedure, there can be up to c^m call chains in the predicates used in tagging a dataflow value. The conjunction operator ($\cap$) which is the most complicated predicate operator has the complexity of $O(c^{2m})$, or the square of the number of call chains.

Thus, the overall complexity of the context-sensitive algorithm in the worse case is $O((\Sigma_{l=0..n}\, r_l^3) \bullet c^{2m})$, where r_l $(l=0..n)$ is the size of the subprogram containing accesses to the *l*-level MemObjs.

4.6 Handle Some Language Specific Features

We now discuss our method to handle type casts, recursive data structures and recursive function calls. Our methods to handle function pointers and *setjmp/longjmp* have already been discussed in Chapter 2. In general, our partitioning technique is orthogonal to the techniques used in handling these language specific features and can accommodate existing techniques used in addressing these issues.

Type casting causes two main problems. First, if a high level pointer is assigned to a low-level pointer via type casts, dereferencing the low-level pointer can access the high level pointer. This causes errors to our analysis ordering. As shown in Figure 35, dereferencing a 1-level MemObj *p* accesses a 2-level MemObj *pp*. The analysis for pointer level 2 is incorrect because it ignores the fact that statement *s2* modifies *pp*. There are two methods to avoid this error. The first method "promotes" *p* to level *2*. This is possible only when *p* is consistently used as a pointer at level *2*. The second method is to set a pointer-level-violation flag to *pp* in a pre-analysis phase so that later analysis conservatively assumes that any dereference to *pp* may access any addressed MemObjs with a type compatible to *p*'s points-to type or *pp*'s points-to type. Second, the presence of type casts makes it necessary to explore data structure layouts. This problem is common to all approaches that use variable types to model the memory locations. Wilson and Lam handle type casts by using exact offsets and sizes [WL95]. The way we handle type casts is similar to the "*common initial sequence*" version of the three methods used by Yong et. al. [YHR99].

While our partitioned techniques mainly focus on non-recursive data structures, we can apply any existing analysis technique to analyze recursive data structures before analyzing the non-recursive data structures. However, our current implementation handles recursive data structure conservatively. It first collects all the addressed recursive data structures. Then, for each dereference expression which results in an access to a recursive data structure, such as *"(*p).next"* with *p* defined as *"struct S { struct S *next; …; } *p;"*, we create an I-function to represent the variables implicitly accessed by the expression. This I-function includes all the addressed recursive data structures of the compatible type. After that, we construct the SSA form for the recursive data structures.

```
int *p, **pp;
pp = …;
p = (int*) &pp;
*p = …; //modify pp!
… = pp;
```

an incorrect *use-def* link

Figure 35: An example program for type casts.

We modify our interprocedural algorithm to handle recursive functions. First, both the bottom-up pass and the top-down pass in Figure 30 need to repeatedly analyze the procedures in a call-graph cycle until the dataflow information reaches a stable point. The number of re-analysis to each procedure is not larger than *r*, which is the number of procedures in the call-graph cycle. This is because there are at most *r* edges in a cycle containing *r* procedures, and it takes at most *r* step for a dataflow value to be propagated from a procedure to another procedure along the edges in the cycle. Second, we also need to modify our predicate representation to accommodate call graph cycles. More specifically, we modify the call-site encoding *(s, c)* so that *s* represents a strong connected component that contains the callee procedure instead of the callee itself. We also assign the same call-site ID (–1) to all the calls to a recursive function. This causes a merge of all the calling contexts to the same recursive function, and avoids an unlimited length of call-chains used in tagging the dataflow value.

4.7 Other Related Work

There have been a large number of published approaches on pointer analysis, and our work draws heavily from the contribution of previous work. In this section, we start with a comparison of our partitioned approach with two other partitioned approaches. Section

4.7.2 compares the points-to representation used in our approach with the alias pair representation. Section 4.7.3 describes Hasti and Horwitz's work on using SSA form to improve pointer analysis. Finally, in Section 4.7.4, we describe techniques in analyzing recursive data structures.

4.7.1 Comparing with Existing Partitioned Pointer Analysis Approaches

A partitioned approach divides all the data structures in a program into groups so that each subproblem analyzes data structures in a group. These data structure groups, say g_i $(i=1..n)$, have two important properties:

- These data structure groups do not intersect each other. That is, given two groups g_i and g_j, $g_i \cap g_j$ is not *empty* if and only i equals to j.
- Given two groups g_i and g_j where $i<j$, the analysis of the data structures in g_i *does not* depend on the analysis of the data structures in g_j, while the analysis of the data structures in g_j *may* depend on the analysis of the data structures in g_i. Thus, we can order the analyses and analyze g_i before analyzing g_j.

Two other partitioned approaches are most relevant to our partitioned pointer analysis. Zhang et. al. [ZRL96, ZRL98] use the assignments that assign pointer values to estimate wether two object names may contain the same value. They put the object names that may contain the same value into a group. Two object names, *a* and *b*, may contain the same value under one of the following two situations:

- There is an assignment "$a = b$" which directly assigns the value of one object to another object.
- There are a group of assignments, "$a = c_1$, $c_1 = c_2$, …, $c_{n-1} = c_n$, $c_n = b$", which indirectly assign the value of one object to another object.

After dividing the object names into groups, Zhang et. al. use a relation called *prefix relation* to infer the dependence among the analysis of different groups. An object name *a* is a prefix of another object name *b*, if adding field accesses (.field) and/or pointer dereferences (*) to *a* can result in another object name *b*. If an object name *a* is a prefix of another object *b*, then the analysis of *b* depends on the analysis of *a*. Thus, the analysis of the group containing *b* depends on the analysis of the group containing *a*. For example, object name *p* is a prefix of object names **p* and *(*p).f*. The value of *p* is needed in order to calculate the values of **p* and *(*p).f*. Thus, the analysis of **p* and *(*p).f* depends on the analysis of *p*.

With this partitioning scheme, Zhang et. al. focus on how a partitioned approach provides the flexibility of appling a flow-sensitive algorithm to analyze some subproblems and a flow-insensitive algorithm to analyze other subproblems to improve the efficiency while not sacrificing too much precision. They call a partitioned approach which applies different algorithms to different subproblems *combined analysis*.

Ruf presents another partitioned approach [Ruf97]. He first puts variables with the same type (declared type or inferred type) into a group. He then infers the dependence relation among the analysis of different variable groups by the *pointed-to-by* dependence among types. Ruf's scheme focuses on the benefit of reducing peak storage usage. He measures storage usage in pointer analysis by counting points-to pairs. He first counts the number of points-to pairs in the whole-program (u_1), and the maximum number of points-to pairs among all the subproblems (u_2). He then uses the value of u_1 - u_2 as an estimate of the reduction in peak storage usage because a non-partitioned algorithm needs to record u_1 points-to pairs while a partitioned algorithm needs to record u_2 points-to pairs at a time.

Without type casts, assigning the value of a variable to another variable is allowed only when the two variables are of the same type. Thus, Zhang et. al.'s partitioned method can usually results in finer subproblems comparing with Ruf's partitioning method. In contrast to these two partitioned schemes, we put variables with the same pointer level into a group. We also estimate the dependence among the analysis of different variable groups based on the fact that, without type casts, dereferencing a variable can only access other variables at lower pointer levels. Thus, the analysis of a pointer level depends on the analysis of its higher pointer levels. Because the pointer level of a variable is a derived attribute of the type of the variable, variables with the same type have the same pointer level. Comparing with our approach, both Zhang et. al. and Ruf's approach can usually results in finer subproblems. Our work differs from their work in two aspects. First, we demonstrate how our partitioning strategy allows a simpler and more efficient algorithm. Second, we provide an integrated algorithm for both pointer analysis and SSA generation while they only perform pointer analysis.

4.7.2 Comparing Points-to Representation with Alias Pair Representation

The points-to representation is used by many existing pointer analysis algorithms because of its advantages over the alias pair representation. First, comparing with the alias pair representation, the points-to representation is more compact and more informative [EGH94]. This can be illustrated by Figure 36. In the figure, after assignment *s2*, there are two points-to pairs: *PointsTo(x, y*) and *PointsTo(y, z*), but four alias pairs: *Alias(*x, y)*, *Alias(**x, *y)*, *Alias(*y, z)*, *Alias(**x, z*). Second, it is easier to compute points-to pairs than to compute alias pairs. The points-to pairs created by an assignment only depend on the assignment itself. The alias pairs created by an assignment, on the other hand, not only depend on the assignment, but also depend on the types of the involved variables and the current alias information. For example, the assignment "*y = &z*" creates one points-to pair *PointsTo(y, z)*. The same assignment creates one alias pair *Alias(*y, z)* if *y* and *z* are defined as "*int *y, z*" and neither *y* nor *z* is involved in other alias pairs before entering the assignment. However, as shown in Figure 36, when *y* is involved in other alias pairs the assignment may create two alias pairs: *Alias(*y, z)* and *Alias(**x, z)*. Third, the points-to representation can avoid some spurious alias information in the alias pair representation. For the example in Figure 36, points-to pair *PointsTo(y, z)* created by statement *s2* is killed by statement *s3*. However, only one of the two alias pairs created by statement *s*2 is killed by statement *s3*, and the alias pair (*Alias(**x, z)*) becomes spurious alias pair after statement *s3*. Lastly, for pointers to arrays, the points-to representation can be extended to include the *offset* information to support array data dependence test [JH94] while the alias pair representation cannot.

(a) An Example Program.	(b) Points-to pairs after each assignment.	(b) Alias pairs after each assignment.
int **x, *y, z, w; main() {		
s1: x = &y;	s1: PointsTo(x, y)	s1: Alias(*x, y); Alias(**x, *y)
s2: y = &z;	s2: PointsTo(x, y) PointsTo(y, z)	s2: Alias(*x, y); Alias(**x, *y) Alias(*y, z); Alias(**x, z)
s3: y = &w; }	s3: PointsTo(x, y) PointsTo(y, z) PointsTo(y, w)	s3: Alias(*x, y); Alias(**x, *y) Alias(**x, z) Alias(*y, w); Alias(**x, w)

Figure 36: Comparing the points-to representation with the alias pair representation (from [EGH94]).

However, there are some advantages in using the alias pairs which entices some researchers to choose the alias pair representation instead of the points-to representation. Most noticeably, an alias pair directly describes the relationship between a pair of symbolic names in a program and is suitable for applications such as conflict/dependence analysis. For example, in order to determine whether it is safe to re-order two statements "**p = 1; a = *q*; ", we need to know whether a write to **p* in the first statement conflicts with a read to **q* in the second statement. In the alias pair representation, **p* and **q* may access the same memory location if and only if there exists an alias pair *Alias(*p, *q)*. However, in the points-to representation, **p* and **q* may access the same memory location if and only if the points-to targets of *p* and the points-to targets of *q* intersect. Assume *p* has *n* points-to targets and *q* has *m* points-to targets, the complexity in determining whether **p* and **q* are alias is *mn*, or, the product of *m* and *n*. Moreover, an alias pair is more suitable to describe the relationship between two elements in a collection of recursive data structures. For such a data structure collection, it is difficult to assign a name to each element, and is too conservative to call them by a single name, such as *heap*. Thus, it is very difficult to describe the relationship of two accesses to a recursive data structure via the memory locations they may touch. The alias pair representation avoids this difficulty and directly describes the relationship of the symbolic names involved in accessing the recursive data structure. Fortunately, describing accesses to recursive data structure this way can provide the necessary information for data dependence tests. Figure 37 shows an example program with recursive data structures. In order to determine whether the loop is parallelizable, we need to know whether "*p.data*" has a data dependence with "*p->next.data*". The alias pair representation provides direct information for such a data dependence test because there is a data dependence between "*p.data*" and "*p->next.data*" if and only if there exists an alias pair *Alias(p, p->next)*. Furthermore, using points-to pairs may infer spurious alias information which does not present in an alias pair representation. This can be illustrated by Figure 38. The figure shows two different program pieces with the same points-to pairs. From these points-to

```
struct RDS {
    int data;
    struct RDS *next;
} *head, *p;

…
p = head;
while (p <> NULL) {
    p.data = … ;
    …
    p = p->next;
}
```

Figure 37: An example program with accesses to recursive data *structures*.

(a) An example program.

```
int *p, *q, a, b;
main() {
    if (cond)
        p = &a;
        q = &b;
    else
        p = &b;
        q = &a;
}
```

(b) Another example program.

```
int *p, *q, a, b;
main() {
    if (cond)
        p = &a;
        q = &a;
    else
        p = &b;
        q = &b;
}
```

(c) The points-to pairs after the *if-statement*.

```
PointsTo(p, a)
PointsTo(p, b)
PointsTo(q, a)
PointsTo(q, b)
```

Figure 38: The points-to representation may infer spurious aliases.

pairs, we may infer that *p and *q are aliases, which is not true for the program in Figure 38(a).

4.7.3 Using SSA Form in Pointer Analysis

Hasti and Horwitz describe a method to use SSA form to improve pointer analysis [HH98]. There are two important ideas in their approach. First, given a safe approximation of the points-to values of all the pointers in a program, we can explicitly represent the memory locations accessed by pointer dereferences and construct an SSA form for the program. Second, if a program is already transformed into an SSA form using safe but not precise points-to values, performing flow-insensitive pointer analysis on the SSA form may improve the preciseness of the points-to values used in generating the SSA form. Because a variable in an SSA form corresponds to a definition to a variable in the source program, the points-to value collected for each SSA variable by a flow-insensitive algorithm can be mapped back to a specific definition point for a source variable, thus resulting in a precise flow-sensitive solution for the source program.

We now briefly describe Hasti and Horwitz's algorithm. In order to create an SSA form for the program in Figure 39(a), they first collect all variables whose addresses are taken. For example, in assignment "*p* = *&a*", the address of variable *a* is taken and assigned to variable *p*. They also assume a pointer may point to any addressed variables with compatible types. For instance, in Figure 39(a), a pointer *p* with type "*int* *" may point to all addressed variables with type "*int*", i.e. *a* and *b*. If the program being analyzed does not have type casts nor absolute addresses for variables, this assumption about points-to value is safe because a pointer cannot point to a variable whose address is not taken. With the estimated pointer values, they *normalize* the program with pointer dereferences into an equivalent program without pointer dereferences by explicitly representing the potential memory locations that may be accessed by each pointer dereference(see

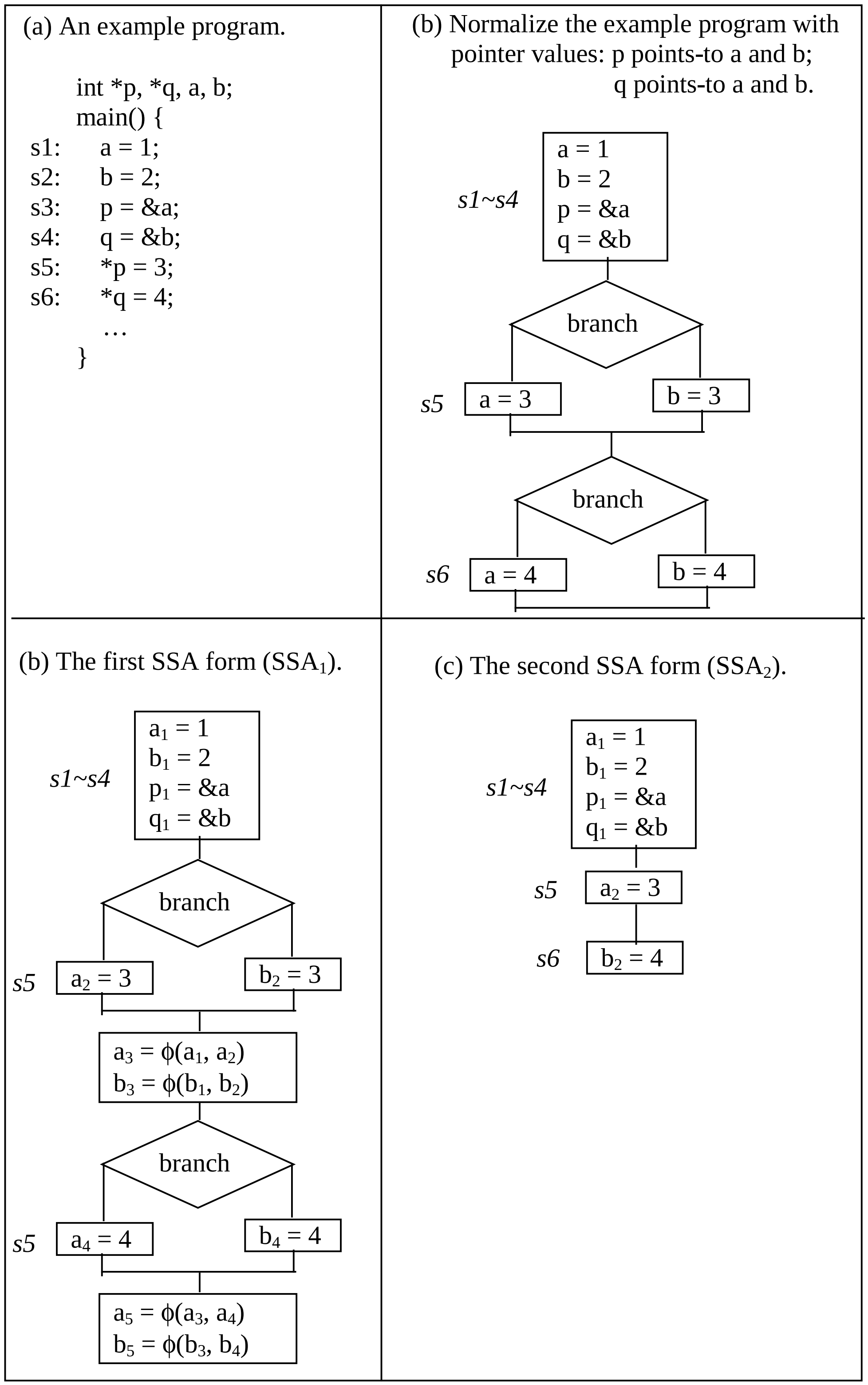


Figure 39: An example program and its SSA forms created by Hasti and Horwitz's pointer analysis algorithm.

Figure 39(b)). After that, they construct the first SSA form, say SSA_1, for the program (see Figure 39(c)). Then, they perform a flow-insensitive pointer analysis using SSA_1 and collect more precise points-to values: *p* definitely points to *a*, and *q* definitely points to *b*. With the finer points-to sets, they re-normalize the program and construct a second SSA form, SSA_2 (see Figure 39(d)). The above steps in performing pointer analysis on an SSA form, normalizing the program and constructing a new SSA form can be repeated until a stable point is reached. However, the algorithm could be stopped after generating one or more versions of SSA forms. Any version of SSA forms generated by the algorithm can provide a safe approximation of the use-def information in the program while the last version of SSA form provides the best approximation.

There are some similar ideas between Hasti and Horwitz's algorithm and our approach. First, both approaches make use of the fact that performing a flow-insensitive analysis on a program in an SSA form can effectively achieve the same result as performing a flow-sensitive analysis on a program in other representations. Second, both approaches use points-to values to normalize a program with pointer dereferences into an equivalent program without dereferences in order to construct an SSA form for the program. However, our approach is more efficient. Their approach starts with a very imprecise assumption about pointer values and repeatedly constructs several SSA forms for the whole program. Our approach, on the other hand, makes use of the dependence among the analysis of pointers at different pointer levels and constructs one SSA form for the whole program.

4.7.4 Analyzing Recursive Data Structures

A recursive data structure is a data structure which has a field to directly or indirectly point to another data structure of the same type. The presence of recursive data structures makes it possible to link together heap memory objects which are dynamically allocated. Figure 40 gives an example recursive data structure type *tree_node* and a possible graph constructed out of the dynamic allocated data structures of type *tree_node*. A dynamically allocated storage does not have a given name, and there may be potentially unlimited number of dynamically allocated storage. This makes it difficult for a compiler to assign a static name to each dynamically allocated storage. It also makes it difficult to distinguish different elements in an aggregate of recursive data structures and to describe the relationship among the pointers pointing to these structures.

Our partitioned approach mainly addresses the analysis of non-recursive data structures. It does not address the problems of recursive data structures. While we currently use a conservative algorithm to handle recursive data structures in our implementation, our framework provides the flexibility to apply any existing, more precise algorithms to handle recursive data structures. Here, we briefly summarize some previous techniques in analyzing recursive data structures. These techniques are presented in the context of handling heap storage because recursive data structures are usually involved in heap storage.

```
struct tree_node {
    int data;
    struct tree_node *left;
    struct tree_node *right;
};
```

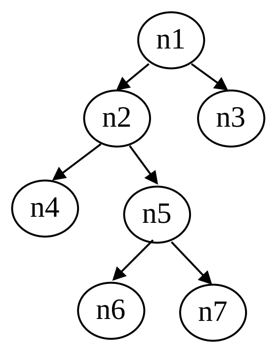


Figure 40: A recursive data structure type and an aggregate of its data structures.

Storage graph analysis. Many early research abstract the structure of heap by variant *storage graph*s [LH88, HPR89, CWC90]. For example, a storage graph may contain one node for each variable and one node for each heap object. Edges in the graph are labeled by pointer fields. In addition, nodes in the graph are labeled by either a variable name or an access path describing how to reach the node via a variable node. Because the number of heap objects can be potentially unlimited, to keep the number of nodes in the storage graph finite, a technique called *k-limiting* is used. In a *k-limiting* graph, no node has a label longer than *k*. This can indirectly limit the number of nodes in the storage graph. With such a storage graph, determining whether two symbolic names are aliases or not needs to traverse the graph using the symbolic names as access paths. Two symbolic names are aliases if and only if the two access paths lead to the same node in the storage graph.

Symbolic access path analysis. Some approaches calculate aliases of recursive data structures using symbolic access paths [Hen90a, Hen90b, LR92, CBC93, Deu94]. Deutsch describes a general symbolic access path as a string of the form $e_1e_2 \ldots e_n$ [Deu94]. An element e_i (i = 1..n) can be the name of a structure field, the name of a variable or a dereferencing operator "*" or "->". An element e_i (i = 1..n) can also be an expression of the form B^k, where *k* is an integer and *B* is an access path. The variable *k* is called a coefficient variable. It represents that *B* is repeated *k* times in the access path.

A symbolic access path can represent both direction and length information. For example, an access path with the form "$p(\text{->left})^i$" (i = 1..n) for the data structures in Figure 40 represents paths with an exact direction *left* and an approximated length (one or more).

With such a representation, a pointer analysis algorithm can keep track of pairs of access paths that may be aliases [LR92, CBC93, Deu94]. For example, Deutsch uses a pair of access paths ($p(\text{->next})^i$, $q(\text{->next})^{2i+1}$) to represent the *i*th element of list *p* and the (*2i+1*)th element of list *q* are aliases [Deu94]. Hendren and Nicolau's algorithm does not keep track of pairs of access paths that may be aliases. Instead, they use a *path matrix* to keep track of the path between two pointer variables [Hen90a, Hen90b]. For any two pointer variables *p* and *q*, a path matrix element M[*p, q*] contains all possible paths which can reach *q* from *p*.

Connection analysis. Ghiya and Hendren use connection analysis to approximate the relationship between a pair of pointer variables [GH95, Ghi95]. Two pointer variables *p* and *q* are connected if they point to the same aggregate data structures. They use a *connection matrix* to represent this pair-wide connection information. A connection matrix *C* is a boolean matrix. For any two variables *p* and *q*, *C[p, q]* is either *true* or *false*. A true value of *C[p, q]* represents that the two variables are connected.

Connection analysis provides a simple way to identify aliases at a coarse level. Given two symbolic access paths starting from variables *p* and *q* respectively, the two access paths are estimated to be aliases if *C[p,q]* is true. In other words, if two variables may reach the same memory location, any pair of paths starting from the two variables are estimated to be aliases.

Shape analysis. Some approaches try to analyze the shape of an aggregate data structure [Ghi95, GH96, SRW96]. Ghiya and Hendren distinguish three types of shapes: tree, direct acyclic graph (DAG) and cyclic graph[Ghi95, GH96]. Sagiv et. al. describe a shape analysis with a focus on handling memory deallocation. Shape information of an aggregate data structure can be used to improve alias information. For example, if the aggregate data structure pointed to by a variable *p* is a *tree* (see Figure 40), then *p->left* and *p->right* are not aliases.

Language-based Approaches. It is difficult for a compiler to get accurate alias information on recursive data structures, and current approaches to analyze recursive data structures only work reasonably well for a small class of data structures, such as acyclic lists and trees. It is believed that a lack of appropriate data structure declaration in a language is the primary obstacle to further improve the analysis result for recursive data structures. Thus, a mechanism for programmers to provide hints about the properties of data structures to compilers is needed to improve the analysis result.

Laurus provides a language annotation for programmers to specify the *acyclicness* of a data structure in Lisp programs [Lar89]. Hummel, Hendren and Nicolau [HHN92a, HHN92b] develop an approach called *abstract description of data structure* (ADDS). In their approach, the language provides some keywords such as *uniquely*, *forward*, *backward*, *independent* and *dependent*. A programmer can associate these key words with a pointer field in the declaration of a recursive data structure to specify the dimension of the data structure and the direction in traversing a particular pointer field.

The ADDS description language is simple and its descriptive ability is limited by the small set of keywords. Hummel, Hendren and Nicolau later develop a more powerful description language called *abstract specification of alias properties* (ASAP) to specify data structure properties[HHN94a]. Each ASAP specification consists one or more axioms, which are provided by programmers or predefined by a language that supports higher-level data structures. They also develop a framework to perform pointer analysis on programs with ASAP description using *theorem proving* [HHN94b, HHN94c].

The problem of such a language-based approach is that, for program correctness, the programmer needs to guarantee that any data access in the program does not cause violation to the properties specified by the ASAP description.

CHAPTER 5

INTEGRATING OPTIMIZATIONS USING AN SSA FORM

In this chapter, we first discuss a group of optimizations that we intend to implement in the Agassiz front-end compiler. We then discuss the interactions between the analyses and the optimizations. Finally, we present an algorithm to integrate the optimizations using an SSA form.

5.1 The Considered Optimizations

The optimizations that we intend to implement in the Agassiz Compiler are as follows:

Constant folding. Constant folding refers to an optimization to evaluate an expression whose operands are all constant values, and replace the expression with the result. Constant folding can improve the efficiency of the source code, and expose information for analyses such as array dataflow analysis and data dependence test. Constant folding, in its simple form, does not require any dataflow information.

Constant propagation. Constant propagation is an optimization which, given an assignment "$x = c$" where c is an integer value, replaces later uses of x with the constant value c if intervening instructions do not change the value of x.

Indirection removal. Indirection removal refers to an optimization which replaces a dereference expression "$*p$" with a direct variable access such as "a". Such an optimization is possible only when the compiler can detect that pointer p definitely points-to variable a. Indirection removal is an important optimization to programs with pointer dereferences for two reasons. First, it can reduce the number of indirect accesses due to pointer dereferences, thus reducing the number of memory accesses in the program. Because memory accesses are very expensive, indirection removal can effectively reduce the program execution time. Second, if a back-end compiler does not have any access to pointer alias information, reducing the number of pointer dereferences in a program can prevent the back-end compiler from making conservative assumption about the pointer dereference, thus, may improve the effectiveness of back-end optimizations.

Unreachable branch elimination. Unreachable branch elimination is to eliminate the branch that will never be executed at run time. Typically, if the branch condition is of a constant value, either the *then-branch* or the *else-branch* will not be executed at run time and can be deleted. Deleting the part that will not be executed not only reduces the code size but also improves the preciseness of the dataflow information.

Useless code elimination. If the definitions generated by an assignment are never used, removing the assignment is an optimization called useless code elimination, or dataflow dead code elimination. Eliminating useless code can reduce the code size.

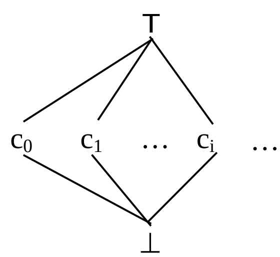
Non-zero iteration loop recognition. Non-zero iteration loop recognition is to determine whether a loop will be executed at least one iteration or not. Recognizing that a loop has at least one iteration can lead to two optimizations. First, we can transfer a *while-loop* to a *do-while-loop*. A *while-loop* requires two jump statements to implement while a *do-while-loop* requires only one jump statement. Second, recognizing that a loop has at least one iteration allows us to eliminate the bypassing edge from the loop *entry-node* to the loop *exit-node* in the program CFG. This can refine the dataflow information by invalidating the operands corresponding to the edge in the β-functions at the loop *exit-node*.

Most of the above optimizations need to determine whether a definition is of constant and calculate its constant value. An SSA form can provide the necessary information to determine the constant value of a definition. This makes it possible to integrate the optimizations using the SSA form.

5.2 Calculating Values for Definitions

Determining whether a definition has a constant value and calculating the constant value are fundamental to the above optimizations. This inference process can be described by a standard 3-level lattice and its meet operator $\cap$ [WZ91, CC95] (see Figure 41).

As shown in Figure 41, the lattice consists of a top element $\top$, an infinite collection of constants c_i (i=0..n), and a bottom element $\perp$. The top element $\top$ represents "no information" about whether the value is constant or not. The bottom element $\perp$ represents that the value is "non-constant". The meet operator $\cap$ defined in Figure 41 is used to calculate the value of a β-function. The value of a β-function equals to the *meet* of all of its operands. In Figure 41, notation "$(c_0 = c_1)?\ c_0 : \perp$" means "if c_0 equals to c_1 then c_0 else $\perp$". The meaning of this meet operator is as follows: if we have "no information" ($\top$) on an operand, then the result equals to the other operand; The meet of two constants is a constant only when they are equal; If one operand is "non-constant" ($\perp$), the result is also "non-constant" ($\perp$).



$\cap$	$\top$	c_0	$\perp$
$\top$	$\top$	c_0	$\perp$
c_1	c_1	$(c_0 = c_1)?\ c_0 : \perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Figure 41: A standard 3-level lattice and its meet operator.

There are three possible situations when we calculate the value of a definition. First, if the definition has an expression on its right-hand side (RHS) and all the operands of the expression are constants, then evaluating the expression calculates the value for the definition. Second, if the definition has an expression on its RHS and some operands of the expression are variables. We need to first evaluate the definitions that reach the use of the variables to determine whether they are of constant values. If all the definitions are constants, we can also calculate the constant value of the current definition. Third, if the definition is a β-definition with a β-function on its RHS, the value of the definition is a *meet* of all the operands of the β-function. In the last two situations, evaluating whether a definition has a constant value requires knowledge about whether other definitions used by the definition are of constant values.

We describe three approaches in calculating the value of the definitions using an SSA form as follows.

5.2.1 An Iterative Approach

Many existing optimization algorithms, such as the constant propagation algorithm described in [WZ91, CC95], use a working-list based iterative approach to determine the constant values for definitions. Figure 42 gives an example of such an iterative algorithm. As can be seen, the algorithm initially assumes "no information" for all definitions and place all definitions on the working-list. It then removes one definition from the working list at a time, and evaluates the value of each definition. If the value of a definition is changed from "no information"($\top$) to a constant value, or from a constant value to "non-constant"($\bot$), we need to re-evaluate the values of other definitions that depend on that definition by putting them on the working list. The algorithm terminates once the working-list becomes empty.

(1) Initialize all definitions to $\top$.
(2) Place all definitions on working list *w*.
(3) **While** *w* is not empty, **do**:
- (3.1) Remove a definition d_i from working list *w*.
- (3.2) Evaluate the right-hand side of the definition d_i.
- (3.3) **If** the value of d_i is changed, for each definition d_j
  - (3.3.1) **If** d_j is not in working list *w* and its right-hand side expression (or β-function) reads d_i, place d_j on *w*.

Figure 42: A working list based iterative algorithm to calculate values for definitions.

```
s1:  a1 = 1 + 1;        => a1 = 2
s2:  b1 = a1 + 1;       => b1 = 3
s3:  c1 = b1 + 1;       => c1 = 4
```

Figure 43: An example program to illustrate the working-list based algorithm.

One drawback of this algorithm is its complexity because each definition may be put on the working list and evaluated more than once. Another drawback of this algorithm is that it does not properly order the evaluation of the definitions and thus may require more iterations than necessary. This can be illustrated by the example program in Figure 43. If the working list is initialized as {a_1, b_1, c_1} in step (2) of the algorithm, then the definitions are evaluated in the order of a_1, b_1, c_1 and each definition is evaluated once. On the other hand, if the working list is initialized as {c_1, b_1, a_1}, then the definitions are evaluated in the order of c_1, b_1,a_1, b_1, c_1 and both definition b_1 and definition c_1 are evaluated twice.

5.2.2 A Demand-Driven Approach

A demand-driven approach exploits the fact that evaluating the value of a definition requires knowledge about the values of other definitions, and recursively calls a utility to evaluate those definitions [SWG94, Sto95] (see Figure 44). In Figure 44, we associate a flag with each definition. This flag can help determining cycles in *use-def* links. Initially, the flag for each definition is set to *START*, which represents that the definition has not been evaluated yet. This flag is set to *GOING* when the definition is being evaluated by utility *EvaluateDefinition*. After finishing the evaluation of the definition, the flag is set to *END*. If during the evaluation process we need the value of a definition with a flag value of *GOING*, a cycle in *use-def* links is detected.

The detection of cycles in *use-def* links can be illustrated by the example program in Figure 45. In the example, the evaluation of definition i_3 demands the value of i_2. The evaluation of definition i_2 demands the value of j_3, which demands the value of j_2. It, in turn, demands the value of i_3. In order to avoid infinite iterations in the presence of cycles in *use-def* links, the algorithm in Figure 44 returns “non-constant” ($\perp$) when the value of a definition with *GOING* flag is demanded. This conservative assumption prevents us from discovering that a definition involved in a *use-def* link cycle has a constant value. For example, we cannot detect that i_2, i_3, j_2 and j_3 have a constant value *1*. However, in real practice, this conservative assumption only causes a small loss of optimization opportunity.

```
main () {
  (1) For each definition d, do
      value[d]  = T;
      flag[d] = START;
  (2) For each definition d, do:
      If (flag[d] == START) EvaluateDefinition(d);
}
EvaluateDefinition(Definition d) {
  (1) flag[d] = GOING;
  (2) assume d's righ-hand side is e;
      Switch (type(e))
      (2.1) Case constant c:
            value[d] = c;
      (2.2) Case arithmetic operations with all constant operands:
            value[d] = the result from the evaluation of the operations;
      (2.3) Case arithmetic operations with some variable operands:
            (2.3.1) For each operand op_i, do
                If (op_i is a constant c)
                  operand-value[op_i] = c;
                Else if (op_i is a read to definition d_j)
                  If (flag[d_j] == START)
                      EvaluateDefinition(d_j)
                      operand-value[op_i] = value[d_j]
                  Else if (flag[d_j] == GOING)
                      operand-value[op_i] = ⊥;
                  Else if (flag[d_j] == END)
                      operand-value[op_i] = value[d_j]
          (2.3.2) value[d] = the result from the evaluation of the operations using
                      the operands in operand-value;
      (2.4) Case β-function:
            (2.4.1) For each β-function operand op_i, do
                If (op_i is a constant c)
                  operand-value[op_I] = c;
                Else if (op_i is a read to definition d_j)
                  If (flag[d_j] == START)
                      EvaluateDefinition(d_j)
                      operand-value[op_i] = value[d_j]
                  Else if (flag[d_j] == GOING)
                      operand-value[op_i] = ⊥;
                  Else if (flag[d_j] == END)
                      operand-value[op_i] = value[d_j]
          (2.4.2) value[d] = ∩ all β-function operands recorded in operand-value;
  (3) flag[d] = END;
}
```

Figure 44: A demand-driven algorithm to calculate values for definitions.

Comparing with the iterative algorithm in Figure 42, the demand-driven algorithm avoids the problem of evaluating a definition several times, thus has a better efficiency. If for some reasons, we need the values for some, but not all, definitions, the demand-driven approach has the advantage of evaluating only the relevant definitions.

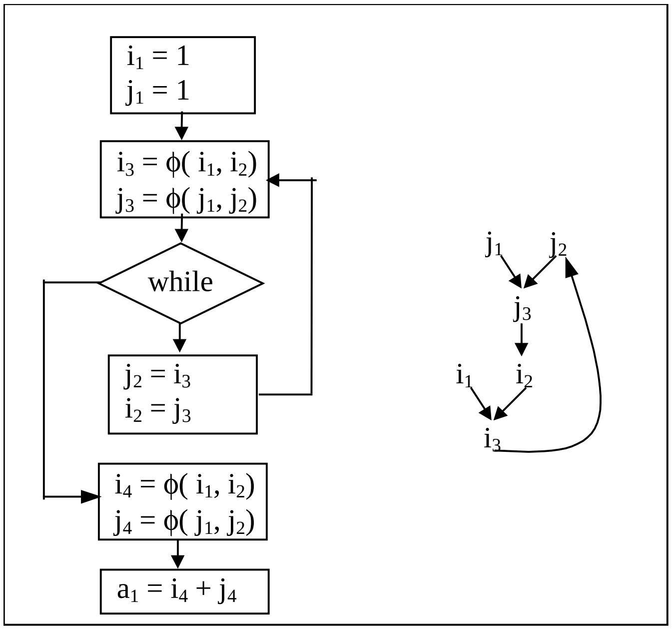


Figure 45: An example program and a *use-def* link cycle in the program.

Similar to the iterative algorithm, the demand-driven algorithm in Figure 44 does not make any assumption about the ordering of the definitions in step (2) of procedure *main*. This makes it difficult to predict the maximum number of recursive calls in the run-time call stack. For instance, if step (2) of procedure *main* processes the definitions in Figure 43 in the order of c_1, b_1, a_1, procedure *main* first calls *EvaluateDefinition*(c_1). During the execution of *EvaluateDefinition*(c_1), *EvaluateDefinition*(b_1) is recursively called to calculate the value of definition b_1. The execution of *EvaluateDefinition*(b_1) further recursively calls *EvaluateDefinition*(a_1). Thus, there can be up to three calls to procedure *EvaluateDefinition* in the run-time call stack. However, if step (2) of procedure *main* processes the definitions in the order of a_1, b_1, c_1, there is at most one call to *EvaluateDefinition* in the run-time call stack. The value of the definition a_1 is first calculated by calling *EvaluateDefinition*(a_1) without demanding the values of other definitions. Then, *EvaluateDefinition*(b_1) is called to calculate the value of definition b_1. The execution of *EvaluateDefinition*(b_1) does not need to recursively call *EvaluateDefinition* because the needed value of definition a_1 is already calculated. After that, when *EvaluateDefinition*(c_1) is called to calculate the value of definition c_1, the needed values of the definitions a_1 *and* b_1 are already known. Thus, by properly ordering the definitions processed in step (2), we can reduce the length of the recursive call-chains in the run-time stack, and improve the efficiency of the algorithm.

5.2.3 An Improved Demand-Driven Approach

The program CFG can be used to order the evaluation of the definitions and to avoid the unnecessary complexity in the above two approaches. Before we discuss how the evaluation of the definitions can be ordered, we first introduce the concepts of *forward use* and *backward use* to distinguish the two possible ways that definitions are used.

Definition 13:

> In a topological traversal of a program (see Definition 3 in Page 18), if a definition created by a statement is used by a *succeeding* statement, we say the definition is *forward used*. The use of the definition is called a *forward use*.

Definition 14:

> In a topological traversal of a program (see Definition 3 in Page 18), if a definition created by a statement is used by a *preceding* statement, we say the definition is *backward used*. The use of the definition is called a *backward use*.

We use the example program in Figure 45 to further illustrate these two concepts. A topological traversal of the example program visits the definitions in the following order: i_1, j_1, i_3, j_3, j_2, i_2, i_4, j_4, a_1. In the program, two definitions of i (i_1 and i_2) reach a read of i in assignment "$j_2 = i_3$" (i_3). Of these two definitions, i_1 is forward used by the assignment while i_2 is backward used by the assignment.

A backward use is always accompanied by a forward use. That is, if there is a definition, say v_x, backward reaches a read to variable *v*, then there must be another definition to variable *v* forward reaches the read to variable *v*. Thus, the number of forward uses in a program exceeds the number of backward uses.

Therefore, the algorithm described in Figure 44 can be improved by ordering the definitions processed in step (2) in procedure *main* as follows: given two definitions d_1 and d_2 generated by statements *s1* and *s2* respectively, if s1 is processed before s2 in a topological program traversal, we order d_1 before d_2. Ordering the definitions this way, by the time we calculate the value of a definition, the values of the other definitions forward used by the definition are already calculated and ready for use. We only need to calculate the values for the definitions backward used by the definition, if any. Thus, this ordering can improve the efficiency of the algorithm.

5.3 Integrating Optimizations

The main purpose to perform a group of scalar optimizations in tandem is to improve the efficiency and the effectiveness of the optimizations. For example, both Wegman et. al. [WZ91] and Binkley [Bin94] show that combining constant propagation and unreachable branch elimination can find more optimization opportunities. Click and Cooper [CC95, Cli95] theoretically proved that combining the dataflow analysis framework for constant propagation, unreachable branch elimination and global value numbering has a potential to find more optimizations.

There are two main issues related to integrated optimizations. First, unifying a group of optimizations requires a common intermediate representation (IR) which can provide the necessary information for the optimizations. Both the approaches in [WZ91] and [CC95, Cli95] use an SSA form while the approach in [Bin94] uses a representation called *program dependence graph*. Second, it is also necessary to update the IR during optimizations. In general, an optimization may affect an IR in two ways: *damage* and *refine*. An optimization *damage*s an IR if after the optimization, the information in IR becomes incorrect and may lead to unsafe optimizations. In this case, the IR must be updated during the optimization. None of the optimizations considered in [WZ91, Bin94, CC95, Cli95] can damage IR. On the other hand, an optimization refines an IR if after the optimization, the information in IR is still valid in the sense that it will not lead to unsafe optimization. However, updating the IR to reflect the change caused by the optimization can provide more precise information and lead to more optimization opportunity. For example, unreachable branch elimination can refine IR by reducing the number of spurious *use-def* links and spurious aliases. The approach in [Bin94] removes the spurious *use-def* links in their program dependence graph but does not remove the spurious alias information. However, neither the approach in [WZ91] nor the approach in [CC95, Cli95] removes these spurious aliases and *use-def* links from their SSA forms during the optimizations.

We integrate a group of optimizations (see Section 5.1) using the extended SSA form described in Section 3.3. We apply *all* optimizations on each statement before we move on to the next statement in the program. This is in contrast to a non-integrated approach which successively applying *one* optimization at a time on *all* the statements in a procedure. Our work in integrated optimizations is different from the previous work in three ways. First, we refine the alias information as well as the *use-def* information in our SSA form during optimizations to expose more optimization opportunity. Second, we use a demand driven algorithm to calculate the values of the definitions, as opposed to the approach in [WZ91] and [CC95, Cli95] which combines the dataflow equation describing the optimizations and uses an iterative algorithm to solve the dataflow equation. Third, we provide extensive experiment results to demonstrate the efficiency and effectiveness of integrating a group of optimizations for a set of benchmark programs.

5.4 Interactions Affect Ordering

The interactions among the optimizations affect how we order the optimizations to improve the efficiency and the effectiveness. However, as we have seen in Figure 2(c), the interactions among the optimizations could be very complicated and may contain cycles. This makes it difficult to order the optimizations according to their interaction. In this section, we do three things to facilitate our optimization ordering:

- Classify the interactions among optimizations according to the program traversal order and clarify how different types of interactions can have different impacts on the optimization ordering.
- Incorporate *forward uses* (see Definition 13 in Page 108) and *backward uses* (see Definition 14 in Page 108) in optimizations.
- Instead of considering how performing one optimization on *any* statements affects performing another optimization on *any* statements, we consider how performing one optimization on *one* statement affects performing another optimization on its *preceding* statements and *succeeding* statements in a topological program traversal.

We first classify the interactions among the optimizations as follows.

Definition 15:

Assume we traverse the statements in a program via their topological order (see Definition 3 in Page 18). Let *A* and *B* denote two optimizations. *A self enable*s *B* if performing A on an expression (or a statement) exposes opportunity to perform B on the *same* expression (or statement). *A forward enable*s *B* if performing *A* on a child-expression e_c (or a statement s_i) exposes opportunity to perform *B* on its *parent-expression* e_p (or *succeeding* statement s_{i+}) [4]. *A backward enable*s *B* if performing *A* on a parent-expression e_p (or a statement s_i) exposes opportunity to perform *B* on its *child-expression* e_c (or *preceding* statement s_{i-}).

These interactions have different impacts on how we order the optimizations for a better result. First, if *A self enable*s *B*, ordering *A* before *B* allows *B* to take advantage of the new opportunity exposed by *A*. Second, the *forward enable* interaction does not affect the optimization ordering under our program traversal scheme. Finally, re-processing a

[4] Assume child-expressions are handled before handling their parent-expressions.

statement is necessary to take advantage of the new optimization opportunity created by the *backward enable* interaction. This re-processing can be achieved by indiscriminately repeating the optimizations on the entire procedure as in most current practice. In our framework, we try to take advantage of this information by putting the affected statements in separate queues, and repeating the optimizations selectively only on the statements in the queues.

The definitions that reach a variable read play an important role in the interactions among optimizations. First, an optimization may turn a non-constant value of a definition into a constant value, thus making it possible to determine a read to that definition to be a constant. Second, an optimization may invalidate some *β-function* operands and expose the opportunity of determining a read directly linked to the corresponding *β-definition* a constant. Third, an optimization may also remove a *use-def* link by replacing a variable read with its value and make the definition unnecessary.

A definition may be *forward used* and *backward used*. *Forward uses* and *backward uses* play different roles in optimization because a *backward use* is often accompanied by a *forward use*, a read involving *backward use*s has low probability of having a constant value comparing with a read involving only *forward use*s.

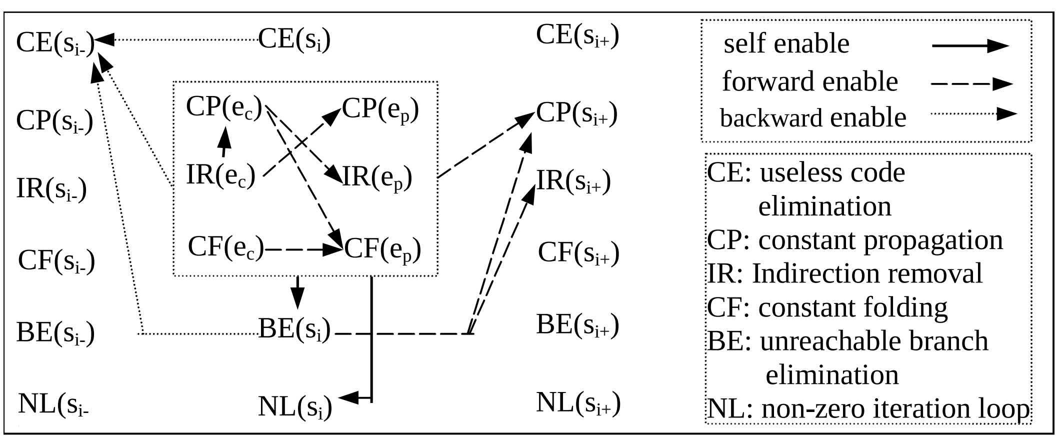


Figure 46. Optimizing statement s_i exposes new opportunities for optimizing its succeeding statement s_{i+} and its preceding statement s_{i-}

```
//Given a procedure that is already transformed into an extended SSA form,
//perform a group of optimizations on the procedure and incrementally update the
//dataflow information on the SSA form.
void OptimizeOneProcedure() {
    For each statement s_i in a topological traversal, do
      (1) useless code elimination (CE):
          If  (s_i is an assignment and all the definitions generated by
               the statement is not used)
              Delete(s_i).
      //handle expression-level optimizations:
      (2) For each expression e_j in a bottom-up traversal of each expression in s_i, do
         (2.1) indirection removal (IR):
               If (e_j has a form of "*p_x")
                  eliminate all the invalid operands in the I_R-function
                  associated with "*p_x";
                  If (a_y is the only valid operand in the I_R-function) and
                     (the MemObj has a compile-time name a)
                      replace "*px" with "a_y" in the SSA form;
                     RemoveOneUse(p_x).
         (2.1) constant propagation (CP):
                //need to augment the algorithm  EvaluateDefinition
               //defined in Figure 44 to evaluate only the valid definitions.
               If (e_j is a read to definition a_y)
                  EvaluateDefinition(a_y);
                  If (value[a_y] is constant c)
                      replace e_j with c;
                      RemoveOneUse(a_y).
         (2.3) constant folding (CF):
               If (e_j is an operation with two constant operands, say op(c1, c2))
                  replace e_j with the result from the operation;
      (3) unreachable branch elimination (BE):
          If (s_i is an if-statement and the condition expression is a constant value)
            invalidate the corresponding operand for the unreachable branch
            in all the β-functions at the merge point;
            For each statement s_j in the unreachable branch, do
               Delete(s_j);
      (4) non-zero iteration loop recognition (NL):
           If (s_i is a while-loop and its exit condition is FALSE before entering the loop)
              invalidate the corresponding operands in all the β-functions
              at the exit node of the loop.
    FurtherRemoveUselessCodes().
}
```

Figure 47: The algorithm to optimize a procedure.

Considering the optimizations involving only forward uses, Figure 46 shows how the optimizations of a statement s_i affect the optimizations of its preceding statement s_{i-} and its succeeding statement s_{i+}. In the figure, an edge from $A(s_x)$ to $B(s_y)$ indicates that performing an optimization *A* on a statement s_x may expose opportunities for an optimization *B* on a statement s_y. Assume child-expressions are handled before handling their parent-expressions. For expression-level optimizations, we also investigate how the optimizations of a child-expression affect the optimizations of its parent-expression at statement s_i. We construct this interaction graph by analyzing the optimizations. We use this figure as our guidance to order the optimizations.

5.5 The Optimization Algorithm

Figure 47 shows our algorithm to optimize a procedure, assuming the input program is already transformed into an extended SSA form. Figure 48 shows the utility functions used in the algorithm. As can be seen, we visit all the statements in a procedure in their topological order, and apply *all* optimizations on each statement before we move on to the next statement in the program. Optimizations are ordered and interleaved at the statement level. We put useless code elimination in the beginning because there is no need to optimize an useless statement. We order the remaining optimizations according to their *self enabling* interactions depicted in Figure 46 so that if an optimization *A* self enables another optimization *B*, *A* is ordered before *B*.

Our method to traverse the program and order the optimizations allows us to exploit the *forward enabling* and *self enabling* interactions among optimizations. In order to remove the useless statements *backward enable*d by the optimizations, we create a queue *UselessCode* to keep the assignments that generate unused definitions. An assignment is put into the queue whenever we remove a use of a definition and detect all the definitions generated by the assignment are unused. We re-process the statements in the queue to further remove the useless codes after finishing a procedure traversal. In the set of optimizations we are considering, the only *backward enabled* optimization is useless code elimination. Hence, only one queue needs to be created. For more general cases, several queues may be needed for selective reapplication of the *backward enabled* optimizations.

```
//Delete a useless statement from the SSA form.
//The definitions generated by the statement are all unused.
//Remove all the corresponding use-def links for the definitions read in the statement.
void Delete(s_i) {
    For each definition, say a_x, read by s_i, do
        RemoveOneUse(a_x);
        Delete s_i from the SSA form;
}

//Remove a specific use-def link to a definition a_x.
void RemoveOneUse(a_x) {
    Remove a specific use-def link of a definition a_x;
    Assume definition a_x is generated by assignment s;
    If (definition a_x is no longer used) and (all the other definition generated
       by s are no longer used)
       Put s_i  in queue UselessCode;
}

//Traverse queue UselessCode to further remove useless codes.
void FurtherRemoveUselessCode() {
    For each statement s_j in queue UselessCode, do
      Delete(s_j).
}
```

Figure 48: The utility functions used by the algorithm in Figure 47.

5.6 An Example

We now illustrate the integrated algorithm with the example program shown in Figure 49. The extended SSA form for the program is shown in Figure 50(a). The integrated algorithm traverses the statements in their topological order. When we visit statement s_3, we handle expression "$a_1 + b_1$" in a bottom-up fashion. Thus, we first handle expression "a_1" and perform constant propagation on the expression. We then visit expression "b_1" and replace the expression with a constant value 2. Finally, we come to the expression "1 + 2" which is a new representation for the original expression "$a_1 + b_1$" and perform constant folding on the expression. As a result, the final SSA form for statement s_3 is "$c_1 = 3$" (see Figure 50(b)).

```
int a, b, c, *p, *q;
main() {
s1:    a = 1;
s2:    b = 2;
s3:    c = a + b;
s4:    if (c > 2)
s5:        p = &a;
s6:        q = &b;
s7:    else
s8:        p = &b;
s9:        q = &a;
s10:   *p = *q + 4;
s11:    c = a + b;
 //other statements that use variables a, b, c, p, and q.
         …
}
```

Figure 49: An example program to illustrate the integrated optimization algorithm.

During the traversal of statement s_4, we perform constant propagation and constant folding on the expression "$c_1 > 2$" and replace the expression with a value *true*. This allows us to determine that the *else-branch* of the *if* statement is unreachable. Hence, we can delete statements *s8* and *s9* from the extended SSA from. We also invalidate the second operand in the β-functions of definitions "$p_3 = \beta((p_1, c_1 > 2); (p_2, !(c_1 > 2)))$" and "$q_3 = \beta((q_1, c_1 > 2); (q_2, !(c_1 > 2)))$". After that, we visit statement s_{10} and determine that predicate $R_{must}(p_1, p_3)$ is *true* and predicate $R_{may}(p_2, p_3)$ is *false*. We can invalidate the first operand in $\beta((a_1, !R_{must}(p_1, p_3)); (a_2, R_{may}(p_1, p_3)))$ and the second operand in $\beta((b_1, !R_{must}(p_2, p_3)); (b_2, R_{may}(p_2, p_3)))$. As a result, the expression "$*p_3$" only generates one definition a_2. This definition kills definition a_1. We can replace the expression "$*p_3$" with "a_2". Likewise, we also replace the expression "$*q_3$" with "b_1". We further replace the expression "b_1" with a constant 2 by constant propagation.

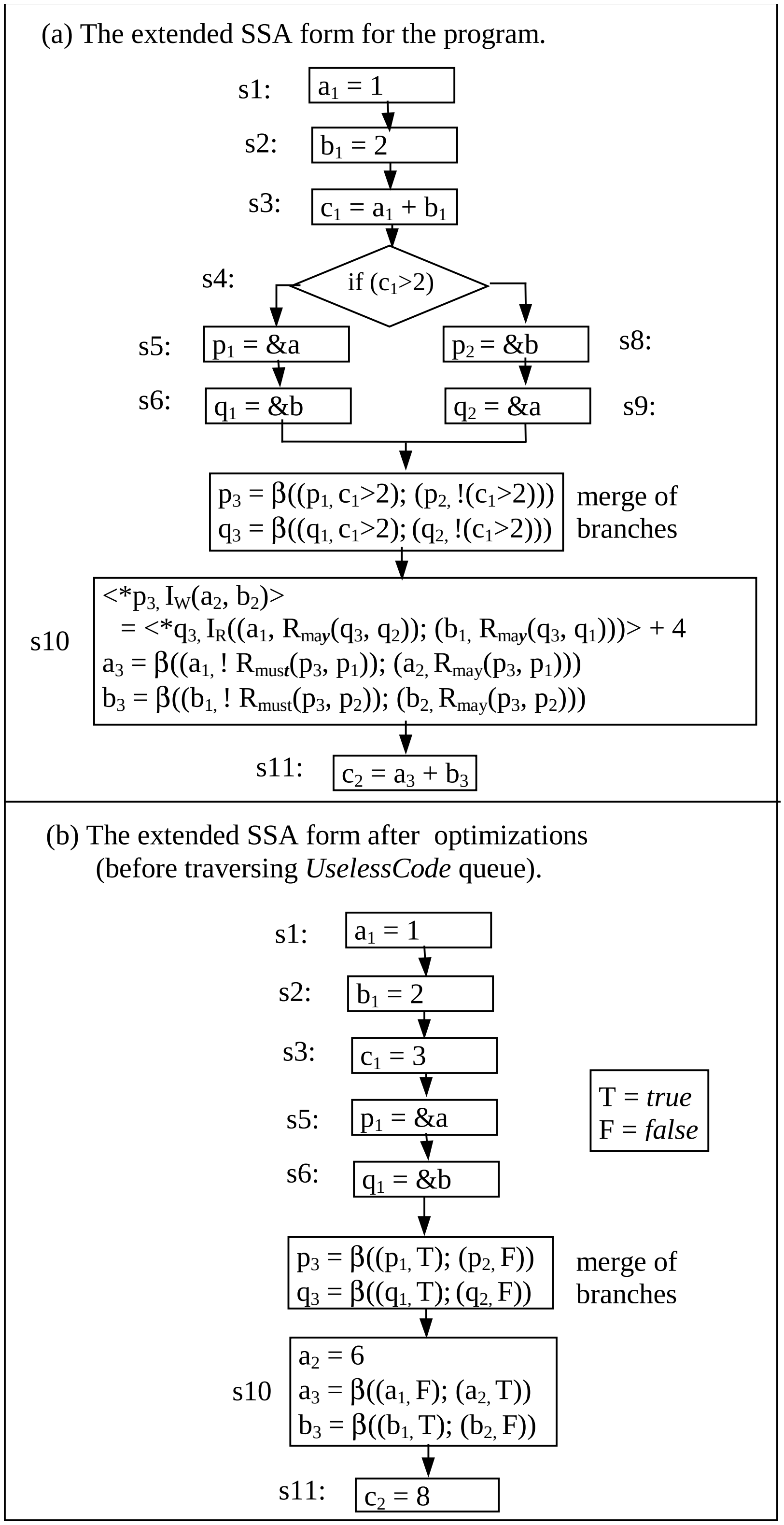


Figure 50. The un-optimized and optimized SSA forms for the program in Figure 49.

When we come to the parent expression “2 + 4” (transformed from the expression “*q_3+ 4”), we perform constant folding and replace the expression with a constant 6. When we come to the last statement s11, we find that the only *regular definition* reachable to a_3 is a_2. The only *regular definition* reachable to b_3 is b_1. Thus, constant propagation and constant folding turn s11 into “c_2 = 8”. The final extended SSA form after these optimizations is shown in Figure 50(b).

During the above optimizations, when we remove a forward use of a definition generated by a statement, we check whether the assignment that generates the definition may become useless and put the new useless assignment in the *UselessCode* queue. In this example, we will have s_1, and s_3 in the queue. Finally, we visit the queue and further delete the useless statements s_1 and s_3.

As can be seen from the example, optimizations can provide more precise information on aliases and *use-def* chains. This new information can be used to incrementally update the extended SSA form and to expose further optimization opportunity.

5.7 Related Work

Our approach to integrated optimizations addresses several issues. The first issue is how to explore the interactions among the optimizations and order the optimizations to take advantage of their interactions. The second issue is how to perform a group of optimizations in tandem. The third issue is how to incrementally update the underlying intermediate representation during the optimizations. In this section, we discuss some other related work which also addresses these issues. In Section 5.7.1, we present Wegman and Zadeck, Click and Cooper’s work in performing a group of optimizations using SSA forms. In Section 5.7.2, we briefly discuss Choi et. al. and Sastry et. al.’s work on incrementally updating an SSA form during optimizations. In Section 5.7.3 we introduce several approaches to integrate optimizations using other intermediate representations. Finally, we discuss the framework developed by Whitfield and Soffa to explore the interactions among optimizations in Section 5.7.4.

5.7.1 Integrated Optimizations Using SSA Forms

The main purpose of Wegman and Zadeck's work is to improve constant propagation by considering the effect of unreachable branch elimination [WZ91]. They describe a *sparse conditional constant* (SCC) propagation algorithm, which combines constant propagation with branch elimination. Their algorithm operates on an SSA form and the control-flow graph of a program and works as follows. There are two working lists used in the algorithm: *FlowWorkList* and *SSAWorkList*. The *FlowWorkList* contains edges in the control-flow graph which are determined to be executable. The *SSAWorkList*, on the other hand, contains *def-use* edges in the SSA form. A *def-use* edge links a definition to a variable read that uses the definition.

Initially, the edges starting from the *entry-node* of a program is put on *FlowWorkList*, and *SSAWorkList* is empty. The algorithm proceeds to remove an edge from either one of the two working lists and evaluate the destination node until both working lists become empty. The evaluation of the destination node is as follows:

- If the node contains ϕ-functions, they calculate the new values of the ϕ-functions. If the new value of a ϕ-function is different from its old value, they put the SSA edges starting from the ϕ-function on the *SSAWorkList*. If the destination node has only one outgoing control-flow edge, they put the edge on *FlowWorkList*.
- Otherwise, the node contains an expression that controls a conditional branch. In this case, they evaluate the expression and put the outgoing control-flow edges corresponding to the reachable branches on *FlowWorkList*.

When a ϕ-function is evaluated, the meet operator is applied only to the ϕ-function operands that correspond to the control-flow edges marked as executable. By not considering the value from the unreachable control-flow edges, they combine constant propagation with unreachable branch elimination.

Click and Cooper [CC95, Cli95] present a theoretical framework to describe combined optimizations and to determine whether combining a group of optimizations will get a result that is better than applying each optimization separately. They treat an optimization as a dataflow analysis problem that corresponds to a set of dataflow equations and combine the optimizations by combining the set of equations for each optimization. They also point out that if the optimizations being combined are interdependent, then in the combined equations, the equations originated from one optimization may refer to the facts derived from the equations originated form another optimization, and vice-versa. As an example of this theoretical method to describe combined optimizations, they present a framework to combine constant propagation, value numbering and unreachable-code elimination. They also show that these optimizations interact with each other and combing these optimization can yield a better optimized code than performing each optimization successively on the whole program.

Our work on integrated optimization differs from both Wegman and Zadeck's work and Click and Cooper's work in several ways. First, we calculate values for definitions in a demand-driven fashion while they use an iterative dataflow analysis framework. Second, we implemented our approach and provide experiment results to demonstrate the efficiency and effectiveness of our integrated optimization approach on a group of C programs in real practice. We are not aware of any implementation of Wegman and Zadeck's approach. Click and Cooper's experiments with a group of FORTRAN programs demonstrate that their integrated approach can achieve a better optimized code but may take a longer time than performing the optimizations separately. One possible reason that their integrated optimization approach may take a longer time is because they integrate optimization based on an iterative dataflow analysis algorithm in which combining several optimizations may lead to more complicated dataflow equations which need more iterations to solve. Lastly, we address the issue of incrementally updating the SSA form during optimization. Neither Wegman and Zadeck's work nor Click and Cooper's work addresses this issue.

5.7.2 Updating SSA Forms During Optimizations

Both Choi et. al. [CSS96] and Sastry et. al. [SJ98] have addressed the issue in maintaining the correctness of the SSA form during code transformation. Choi et. al. present an algorithm to arbitrarily insert and delete a use/definition of a variable in an SSA form. They also present a group of algorithms to update the SSA form during loop-oriented transformation such as loop interchange and loop fusion. Sastry et. al. also discuss how to incrementally update the SSA form when a definition is inserted during register promotion.

As we have seen that an optimization may affect an SSA form in two ways: *damage* and *refine* (see Section 5.3) An optimization damages an SSA form if after the optimization, the information in the SSA form becomes incorrect and may lead to unsafe optimizations. On the other hand, an optimization refines an SSA form if after the optimization, the information in the SSA form is still valid and will not lead to unsafe optimization. Because the optimizations considered in our integrated approach do not damage the information in the SSA form. We mainly address the issue of how to refine the SSA form during optimizations. In contrast to our work, both Choi et. al.'s work and Sastry et. al.'s work addresses the issue of how to maintain the correctness of the SSA form during optimizations.

Currently, the optimizations integrated in our approach do not include loop-oriented optimizations. In order to extend our approach to include loop-oriented optimizations, we need to maintain the correctness of the SSA form during optimizations using algorithms such as those described by Choi et. al.

5.7.3 Integrate Optimizations Using Other Intermediate Representations

Many other researches also notice that many optimizations require similar information from variables and expressions, and thus, using a common program representation for all optimizations can speed up the optimization process [Joh94, Bin94]. In this section, we briefly describe two other intermediate program representations.

Johnson performs scalar optimizations on a *dependence flow graph* (DFG) and incrementally updates the DFG during optimization [Joh94]. A DFG is similar to an SSA form in that it integrates data dependence and control dependence into the program representation. However, in contrast to our SSA form, a DFG can exploit the sparsity of dataflow information. It partitions a program into *single-entry single-exit* (SESE) regions and uses program structure tree (PST) to capture the hierarchical structure of the SESE regions. This makes it possible to avoid propagating dataflow values through SESE regions which do not affect the values, and thus improve the efficiency of the algorithm.

Binkley uses *system dependence graph* (SDG) as a common program representation to support optimizations [Bin94]. A SDG represents three types of information: control dependence, data dependence, and interprocedural call and return information. These three types of information are also available in our SSA form. However, Binkley's data dependence information is more detailed than our data dependence information because he explicitly represents loop-carried data dependence. Moreover, Binkley incrementally updates the SDG for each performed optimization according to a set of graph rewriting rules specifically for each optimization. This is in contrast to our method to incrementally update the SSA form by changing the value for the predicates in the β-functions.

In addition, our SSA form is different from the above two representations in that we address issues of explicitly representing information, such as alias and interprocedural *Ref/Mod*, in a program representation.

5.7.4 A Framework to Explore the Interactions among Optimizations

Whitfield and Soffa develop a framework to explore, both analytically and experimentally, the *enabling* and *disabling* interactions among optimizations [WS97]. There are two major components in their framework. The first component is a specification language called *Gospel*. Gospel provides the language construct to express the conditions needed to safely apply an optimization and the actions required to change the code to implement the optimization. The Gospel specifications for a group of optimizations can be used to reason about the interactions among optimizations as follows. First, the conditions needed to safely apply an optimization are used to establish the enabling and disabling properties of the optimization. Second, through the enabling and disabling conditions, the interactions among the optimizations can be determined. The optimizations which can create the enabling conditions for an optimization enable the optimization. On the other hand, the optimization which can create the disabling conditions for an optimization disables the optimization. As we have seen in Section 5.4, the enabling and disabling interactions among optimizations can guide us to order the optimizations.

The second component in Whitfield and Soffa's framework is a tool call *Genesis*. It takes Gospel specification as input and analyzes the Gospel specification. It also generates codes that perform pattern matching to check the condition for performing an optimization and call the necessary primitive routines to perform the optimization if the conditions are met. Genesis also provides an interface to specify the order of the optimizations. Thus, Genesis greatly reduces the programming effort to implement a group of optimizations in any order. By applying the optimizers generated by Genesis to the benchmark programs, they experimentally examine the interactions among the optimizations perform on a program and the effectiveness of different optimization orders.

The optimizations considered in Whitfield and Soffa's approach include both scalar optimizations and loop-oriented optimizations. In contrast to their approach, we analytically examine the interactions for a group of scalar optimizations and use the interactions to guide our ordering of the optimizations. Their framework can be used to experimentally examine the interactions for a group of optimizations in real programs.

CHAPTER 6

EXPERIMENTAL EVALUATION

We have implemented our integrated scalar analyses and optimizations algorithm in the Agassiz Compiler. In this chapter, we experimentally evaluate our algorithm on a set of benchmark programs. Our experiments were carried out on a 400MHz Intel Pentium II machine with 256MB main memory and 2GB swap space running Linux. We compiled our program using *GCC* with option “-O2”. Execution times were obtained by inserting Unix system call *getrusage* in the program and averaging over three runs of the program.

The remainder of this chapter is organized as follows. Section 6.1 describes some relevant characteristics of the benchmark programs used in our experiments. Section 6.2 presents experimental data to evaluate our partitioned approach on programs with general pointer usage. Section 6.3 presents experimental data to evaluate our integrated optimization approach.

6.1 Benchmark Program Description

Our benchmark suite contains a total of 16 programs: six from SPEC95, six from SPEC92 and four from a pointer intensive benchmark suite [Austin]. Table 1 presents some important characteristics of these programs that affect the complexity of our analysis. These programs are ordered in increasing numbers of source lines. In the table, following the columns that describe the name and the source of each program, the third column lists the number of lines in each program. The number of user-defined procedures (including main) and the number of calls to all of these procedures are shown in the next two columns. The sixth column reports whether each program contains recursive data structure or not. The last column gives the maximum pointer level of each program reported by our algorithm. We partition the analysis of a program with a maximum of *n* pointer level into *n+1* subproblems.

Program	Source	# of lines (k)	# sub.	# calls	rec. DS	max.PL
alvinn	spec92 cfp	0.3	8	7	no	1
anagram	[Austin]	0.6	15	22	no	3
ks	[Austin]	0.8	13	16	yes	2
026.compress	spec92 cint	1.5	14	83	no	2
129.compress	spec95 cint	1.9	18	38	no	1
ft	[Austin]	2.2	27	47	yes	2
eqntott	spec92 cint	3.5	59	554	yes	2
yacr2	[Austin]	4.0	51	156	no	2
ear	spec92 cfp	5.2	68	140	no	3
li	spec95 cint	7.4	355	1183	yes	3
sc	spec92 cint	8.5	147	1010	yes	2
espresso	spec92 cint	14.8	314	1636	yes	3
m88ksim	spec95 cint	19.9	239	1161	yes	4
go	spec95 cint	29.2	372	2099	no	2
ijpeg	spec95 cint	31.2	271	837	yes	5
vortex	spec95 cint	67.2	627	5622	yes	5

Table 1: The benchmark programs and some important characteristics of the programs (rec. DS = recursive data structure, max. PL = maximum pointer level) .

6.2 Evaluating the Partitioned Analysis Algorithm

In this section, we evaluate our partitioned analysis algorithm to construct extended SSA forms for programs with general pointer usage. First, we measure the time to perform pointer analysis and non-pointer analysis. Next, we measure the average number of targets for all pointer dereferences in each program, and use it as a metrics to evaluate the precision of the pointer analysis result. We also present data to evaluate our extended SSA form with explicit pointer alias and interprocedural *Ref/Mod* information and evaluate our two methods to achieve context-sensitivity. Finally, we use some data from a most recent published work on pointer analysis to calibrate our pointer analysis algorithm.

6.2.1 Analysis Times

Table 2 presents the analysis time for our analysis algorithm with the parameter *m* (the maximum call-chain length, see Section 4.5.3) set to 1. For each program, we show the time to perform pointer analysis, the time to perform non-pointer analysis, and the sum of these two. As can be seen from the table, except programs *go* and *vortex*, the remaining programs spend more time on pointer analysis than on non-pointer analysis. This indicates that, for most of the programs in our benchmark suite, pointer analysis is more complicated and expensive than non-pointer analysis. Except *vortex*, the time to perform pointer analysis ranges from about one second for *alvinn* to about five minutes for *ijpeg*. The time to perform non-pointer analysis ranges from about one second for *alvinn* to about four minutes for *go*. We believe both our pointer analysis and non-pointer analysis are efficient for these programs. We spend about 52 minutes to analyze both the pointers and the non-pointers for *vortex*. This is significantly more time consuming than the analysis of other programs. We find some characteristics, such as a large invocation graph, large number of MemObjs, indirect function calls using function pointers and recursive function calls, which make the analysis time-consuming. However, we are not able to quantify how each of these complicated language features contributes to the time of the analysis.

Program	pointer	non-pointer	*total*
alvinn	0.81	0.68	1.49
anagram	2.11	1.07	3.18
ks	3.29	1.38	4.67
026.compress	4.08	2.80	6.88
129.compress	1.48	1.42	2.90
ft	3.29	1.34	4.63
eqntott	40.77	18.31	59.08
yacr2	8.03	5.21	13.24
ear	5.65	4.68	10.33
li	117.41	17.94	135.35
sc	104.96	77.96	182.92
espresso	209.76	81.43	291.19
m88ksim	168.06	163.66	331.72
go	56.72	217.90	274.62
ijpeg	316.25	66.35	382.60
vortex	1131.71	2008.71	3140.42
average	135.90	166.93	302.83

Table 2: The analysis time (in seconds).

6.2.2 Analysis Results

We use the average number of target objects (*AvgNumTarget*) for pointer dereferences to evaluate the precision of the pointer analysis results. Similar metrics are also used by other researchers [EGH94, Ruf95, Wil97, HH98]. At run time, a pointer dereference should access at least one memory object. Thus, one is the lower bound for *AvgNumTarget*. An analysis with an *AvgNumTarget* close to one means that the algorithm is precise. However, a larger *AvgNumTarget* may indicate either a reduced precision due to the algorithm, or the pointer dereferences actually access more than one memory object on average at run time. In our approach, the *AvgNumTarget* for indirect reads (writes) also corresponds to the average number of operands in the I_R-functions (I_W-functions).

Table 3 shows the *AvgNumTarget* for pointer dereferences in terms of *AvgNumTarget* for indirect reads (in column two) and *AvgNumTarget* for indirect writes (in column four). Similar to the data we have just seen in Section 6.2.1, these data are also for our context-sensitive algorithm with the parameter *m* (the maximum call-chain length) set to one. As can be seen, each pointer dereference only accesses one target for programs *alvinn, ks, 026.compress* and *yacr2*. This indicates that our algorithm can precisely analyze these programs. For the other programs except *li*, the *AvgNumTarget* ranges from two to twelve. For program *li*, our analysis determines that the *AvgNumTarget* for indirect reads is more than 155 and the *AvgNumTarget* for indirect writes is more than 145.

There are two methods to estimate whether an *AvgNumTarget* larger than one is due to a reduced precision of the analysis algorithm or the fact that pointer dereferences actually access more than one memory object on average at run time. First, we can measure the loss of precision due to our limited level of context-sensitivity. Because we set parameter *m* (maximum call-chain length) to one, we may lose precision from merging pointer values propagated beyond the most recent call-site in the call-chains. Such a loss of precision can be measured by comparing the *AvgNumTarget* in Table 3 with the *AvgNumTarget* achieved by setting the parameter *m* to a larger number. If increasing the value of parameter *m* can achieve a smaller number for *AvgNumTarget*, we know that the large number for *AvgNumTarget* in Table 3 is due to a loss of precision caused by the limited level of context-sensitivity in the algorithm. We will perform this comparison in Section 6.2.3. Second, we can also compare our analysis results with results from other algorithms. If other approaches can achieve a smaller number for *AvgNumTarget* for the same program, a large number for *AvgNumTarget* in Table 3 indicates a loss of precision in our approach. We will perform this comparison in Section 6.2.4.

Program	Indirect reads		Indirect writes		β-fun. for ind. (%)	β-fun. for call (%)
	ANT	%	ANT	%		
alvinn	1.0	12.9	1.0	7.9	7.6	5.1
anagram	1.9	5.2	1.0	4.8	9.1	2.6
ks	1.0	0.1	1.0	0.8	13.6	8.1
026.compress	1.0	11.0	1.0	6.6	23.7	1.0
129.compress	1.1	5.7	1.1	11.0	24.1	2.4
ft	1.4	1.5	1.4	2.1	12.2	1.5
eqntott	2.0	4.2	1.5	5.6	26.2	1.0
yacr2	1.0	9.2	1.0	11.3	8.4	0.4
ear	7.5	9.4	6.3	12.4	9.8	11.1
li	155.6	1.2	145.9	2.8	4.3	4.1
sc	2.2	2.0	1.7	1.3	11.9	0.2
espresso	3.9	7.7	2.5	10.4	26.8	1.6
m88ksim	6.2	2.7	3.6	6.3	29.9	1.9
go	2.6	0.3	2.0	0.6	39.7	1.9
ijpeg	1.4	5.1	1.4	8.3	4.9	1.9
vortex	12.3	8.7	8.9	7.3	35.5	0.1
average	12.63	5.43	11.33	6.22	17.98	2.81

Table 3: Statistics about the precision of the analyzed result and the SSA form with explicit alias and *Ref/Mod* information (ANT = average number of targets).

We also examine how explicitly representing alias and *Ref/Mod* information in an SSA form may expand the size of the program representation from two aspects. First, we show the percentage of reads (writes) which are indirect reads (writes) in column three (five). If a program has a large percentage of indirect reads (writes) and the analysis result has a large *AvgNumTarget* for the indirect reads (writes), the SSA form may expand the original program by a significant factor. Interestingly, this worse case happens rarely and does not happen in our entire benchmark suite. Second, we give the percentage of definitions that are *β-definitions* which represent the weak-updates caused by indirect writes using pointer dereferences (in column six) and procedure calls (in column seven). As can be seen from the numbers in these two columns, the percentage of *β-definitions* for indirect writes and procedure calls in each program is at most 42 (in program *go*) and is usually very small. Overall, with an aggressive interprocedural pointer analysis, explicitly representing alias and *Ref/Mod* information in an SSA form only expands the programs in our benchmark suite with a very small factor.

6.2.3 Evaluating the Two Methods to Achieve Context-Sensitivity

Context-sensitivity refers to the property that an algorithm distinguishes a procedure's effect in different calling contexts. As we have discussed in Section 4.5.2, we achieve context-sensitivity by two means: (1) parameterizing the return values of a MemObj modified in a procedure; and (2) tagging the interprocedural dataflow values with call-chains. We examine the effectiveness of these two methods.

	% of return values that are parameterized
alvinn	0.00
anagram	47.17
ks	33.33
026.compress	46.05
129.compress	31.94
ft	45.90
eqntott	44.16
yacr2	55.50
ear	65.22
li	47.22
sc	53.97
espresso	40.79
m88ksim	37.00
go	21.60
ijpeg	50.17
vortex	38.80
average	41.17

Table 4: Parameterize procedures' modification effect (*Mod*) to pointers.

We first measure the effectiveness of parameterizing the modification effect of each procedure. Table 4 shows the percentage of return values that are parameterized. If an *l*-level MemObj is modified by a procedure, and the return value of the MemObj is a function of the values of other *l*-level MemObjs input to the procedure, we say the return value of the MemObj is parameterized. Parameterizing the return value is a context-sensitive representation because the return value of a MemObj depends on the input values when a procedure is called. As can be seen from the table, except program *alvinn*, the percentage of parameterized return values range from 20 for *go* to 65 for *ear*. This indicates that the values of a significant portion of the MemObjs depend on the input values of other pointers when the procedure is called. The maximum pointer level of program *alvinn* is one, and no *1*-level global MemObjs are modified by the procedures in *alvinn*. That explains why no return values in *alvinn* are parameterized.

We also measure how the maximum call-chain length used in tagging the dataflow information (i.e., parameter *m*) affects the efficiency of the analysis algorithm and the precision of the analyzed result. Without any limitation, a call-chain used in tagging the dataflow information can be as long as the depth of the program call graph. With the limit set by parameter *m*, a call-chain used in tagging the dataflow information can only contain the most recent *m* call-sites in the call graph path. This may cause a merge of the dataflow values propagated beyond the most recent *m* call-sites. In general, a version of our algorithm with a larger value of *m* may achieve a more precise result. However, a more precise analysis does not necessary require more analysis time because a less precise algorithm may cause more spurious alias and *Ref/Mod* and may expand the size of the SSA form which could lead to more complex analysis (see Section 4.5.3).

We examine how parameter *m* affects the algorithm by comparing the analyzed results and the analysis times using three different values of the parameter *m*. We first set *m* to zero, thus, no call-chains are used in tagging the dataflow value. The then set *m* to one and use the most recent call-site in tagging the dataflow value. Finally, we do not enforce any limit to the maximum call-chain length used in tagging the dataflow values.

	AvgNumTarget (m = 0)	AvgNumTarget (m=1)	AvgNumTarget (m=unrestricted)
alvinn	1.0	1.0	1.0
anagram	1.7	1.7	1.7
ks	1.0	1.0	1.0
026.compress	1.0	1.0	1.0
129.compress	1.1	1.0	1.0
ft	1.4	1.4	1.4
eqntott	1.9	1.8	1.8
yacr2	1.0	1.0	1.0
ear	8.2	7.1	7.1
li	153.2	152.0	152.0
sc	2.3	2.1	2.1
espresso	3.3	3.3	3.3
m88ksim (*)	9.0	5.3	NA
go	14.7	2.4	1.0
ijpeg	1.6	1.4	1.4
vortex (*)	15.0	11.6	NA

Table 5: The effect of the parameter *m* (*: for programs *m88ksim* and *vortex*, we are not able to complete the analysis and collect the statistics in one day with an unrestricted *m*.).

Table 5 compares the analyzed results using the three different values for *m*. The analyzed result for each program is represented by the average number of targets (*AvgNumTarget*) of all pointer dereferences, including indirect reads and indirect writes, in the program. The table indicates that the benefit of increasing the value of parameter *m* varies from programs to programs. In seven out of 16 programs (*alvinn*, *anagram*, *ks*, *026.compress*, *ft*, *yacr2* and *espresso*), they achieve the same the same precision in their analyzed results. This may suggest that (1) parameterizing the return values can achieve a precise context-sensitive solution for these programs; or (2) most of the dereferenced pointers in these programs are not interprocedurally related; or (3) most of the subroutines in these eight programs are called with the same point-to values. For the remaining nine programs, setting *m = 1* can improve the precision of the analyzed results. This indicates that tagging the dataflow values with call-chains can improve the level of context-sensitivity for these programs. For program *go*, the algorithm with an unrestricted *m* can achieve an

AvgNumTarget of 1, a significant improvement over the algorithm with $m = 1$. However, except *go*, setting $m > 1$ does not have noticeable for most programs.

	m=0		m=1		m=unlimited	
	pointer	non-pointer	pointer	non-pointer	pointer	non-pointer
alvinn	0.80	0.66	0.81	0.68	0.80	0.70
anagram	2.07	1.07	2.11	1.07	2.07	1.07
ks	3.29	1.38	3.29	1.38	3.30	1.38
026.compress	3.99	2.80	4.08	2.80	4.21	2.81
129.compress	1.48	1.40	1.48	1.42	1.50	1.42
ft	3.28	1.34	3.29	1.34	3.30	1.35
eqntott	39.02	18.34	40.77	18.31	44.23	18.32
yacr2	7.92	5.17	8.03	5.21	8.21	5.31
ear	5.55	4.65	5.65	4.68	5.56	4.70
li	117.32	17.94	117.41	17.96	117.41	17.96
sc	102.81	77.89	104.96	77.96	132.74	77.95
espresso	190	69.52	209.76	81.43	21181.47	508.83
m88ksim (*)	156.66	164.12	168.06	163.66	NA	NA
go	56.65	241.7	56.72	217.90	792.96	7145.3
ijpeg	312.71	66.28	316.25	66.35	54179.30	128.33
vortex (*)	952.25	14210.8	1131.71	2008.71	NA	NA

Table 6: The effect of the parameter *m* on the analysis time (in seconds).

Table 6 shows the effect of the parameter *m* on the analysis time. For each program, we separate the time to perform pointer analysis and the time to perform non-pointer analysis. As can be seen from the table, for most programs, the analysis time with $m = 1$ is slightly longer than the analysis time with $m = 0$. This indicates that for these programs, (1) the impreciseness caused by setting $m = 0$ does not significantly expand the size of the program representation, and (2) the computation for call-chains with $m = 1$ only adds a small overhead. However, for program *vortex*, the analysis time with $m = 1$ is shorter than the analysis time with $m = 0$. This indicates that the impreciseness caused by setting $m = 0$ significantly expands the size of the program representation for vortex. The time to handle this imprecise dataflow information is significantly longer than the time to perform predicate calculation with $m = 1$. Furthermore, for the large programs such as *espresso*, *m88ksim*, *go*, *ijpeg*, and *vortex*, increasing the value of *m* significantly increases the complexity and dramatically increases the analysis time.

Overall, the data in Table 5 and Table 6 indicates that, in our choosed benchmark suite, using an unrestricted *m* is not necessary beneficial. Limiting *m* to a small number, such as one, is a practical way to handle large programs efficiently without sacrificing very much the precision of the analysis result. Our approach allows users to set an appropriate value for *m* which works the best for each particular program.

6.2.4 Calibrate Our Pointer Analysis with Yong et. al.'s Analysis Results

It is difficult to precisely compare two different algorithms because of the difference in their implementation and other environments. We use Yong et. al.'s analysis results [YHR99] to roughly calibrate our pointer analysis algorithm. Their experiment data were collected on an Ultra 10 with 256MB memory and 1.1G swap space while our experiment data were collected on a 400MHz Intel Pentium II machine with 256MB main memory and 2GB swap space. Besides the difference in machine environment, there are two major differences between the two algorithms. First, in contrast to our algorithm, their algorithm is flow-insensitive and context-insensitive [5]. Second, their algorithm only collects points-to values while our algorithm not only collect points-to values but also constructs SSA forms to represent the *use-def* information for all pointers.

		pointer analysis time (sec)	AvgNumTarget
li	Our Alg.	117.41	152.0
	Yong's	> 7 hours	>230
m88ksim	Our Alg.	168.06	5.3
	Yong's	> 540	>15
go	Our Alg.	56.72	2.4
	Yong's	0.64	NA

Table 7: Calibrate our pointer analysis algorithm with Yong et. al.'s analysis results [YHR99] for three large programs.

[5] I would like to thank one of the authors in [YHR99], Suan Hsi Yong, for answering my questions and verifying this information.

Table 7 compare our pointer analysis with Yong et. al.'s pointer analysis [YHR99] on three large programs. As far as the analysis time is concerned, our algorithm is more efficient in analyzing *li* and *m88ksim*, but less efficient in analyzing program *go*. The difference in the analysis time for program *li* is the most significant. We spent less than two minutes to analyze program *li* while their algorithm spent more than seven hours. Because we perform flow-sensitive analysis with a moderate level of context-sensitivity while they perform flow-insensitive context-insensitive analysis, our analysis result is in general more precise than their analysis result. Overall, we believe the fact that our approach can reduce the analysis time for *li* from seven hour in their approach to less than two minutes is very encouraging.

6.3 Evaluating the Integrated Scalar Optimizations

In this section, we evaluate our algorithm to integrate a group of optimizations. We first measure the time to perform the optimizations and the number of times the optimizations can be applied for each program. We then compare the efficiency and effectiveness of our integrated approach with that of a non-integrated approach. Lastly, we show that the optimizations can reduce the size of the SSA forms.

6.3.1 Optimization Times

We measure the time to perform the group of optimizations using our integrated optimization algorithm and present the data in the second column of Table 8. Our integrated optimization algorithm traverses the whole program once, and interleaves the optimizations at the statement/expression level (see Section 5.1). As indicated in the table, the time to perform an integrated optimization on a program ranges from half a second for *alvinn* to about six minutes for *vortex.* Except *vortex*, the times to perform the optimizations for the other programs are less than 0.8 minutes. On average, we spend 0.6 minutes to optimize one program.

The third column in Table 8 gives the ratio between the time to optimize a program and the time to perform analysis and create an extended SSA form for the program (given in Table 2). As can be seen, the small programs tends to be larger than the ratio for the large programs. This is because that, as applications become larger, the time to perform dataflow analysis grows faster than the time to perform optimizations grows. On average, the optimization time is about 21 percent of the analysis time. Thus, performing integrated optimizations only add a small portion of time overhead to the whole compilation process.

	optimization time (seconds)	ana. time / opt. time
alvinn	0.51	0.34
anagram	0.59	0.19
ks	1.06	0.23
026.compress	1.58	0.23
129.compress	1.97	0.68
ft	1.10	0.24
eqntott	8.92	0.15
yacr2	3.19	0.24
ear	2.72	0.24
li	11.30	0.08
sc	29.28	0.16
espresso	39.09	0.13
m88ksim	36.80	0.11
go	45.13	0.16
ijpeg	46.27	0.12
vortex	354.45	0.11
average	36.50	0.21

Table 8: Optimization time (in seconds).

6.3.2 Counting the Number of Optimizations

We count the number of cases that we apply an optimization to a program and present the data in Table 9. In the table, we use six columns to show the number of cases each of the six optimizations was applied. The last column of the table summarizes the total number of optimization cases applied to each program while he last row of the table shows the average number of cases each optimization was applied to all the programs.

As indicated in the last row of the table, the majority of optimization cases are from constant folding and constant propagation. These two optimizations together constitute 79 percent of the total optimization cases. This indicates that constant folding and constant propagation are two most important optimizations in the Agassiz front-end compiler. We also have a large number of cases from useless code elimination. We identify three sources that may create useless codes. First, performing optimizations such as constant propagation and indirection removal may create useless codes. Second, we find in many cases, programmers wrote useless codes in the benchmark programs. Third, the cleanup phase in our compiler may also create useless code.

	CE	IR	CP	CF	BE	NZL	*Total*
alvinn	2	0	1	15	0	23	41
anagram	7	0	3	1	1	6	18
ks	0	0	2	3	0	2	7
026.compress	15	0	40	29	5	5	94
129.compress	17	14	56	28	1	12	128
ft	41	0	9	5	1	7	63
eqntott	23	61	129	410	0	9	632
yacr2	1	0	12	24	0	6	43
ear	15	59	83	44	0	7	208
li	1	0	7	16	0	4	28
sc	51	11	51	627	2	9	751
espresso	732	0	90	344	2	112	1280
m88ksim	104	81	229	2263	15	99	2791
go	151	3	97	3732	3	103	4089
ijpeg	69	114	170	1152	23	101	1629
vortex	1010	181	1880	451	8	9	3539
average	140	33	179	572	4	32	959

Table 9: Optimizations performed on each programs (CE=useless code elimination, IR=indirection removal, CP=constant propagation, CF=constant folding, BE=branch elimination, NZL=non-zero iteration loop recognition).

The last column of the table indicates that the number of optimizations applied is different from program to program. About 50 percents of the total number of optimizations are applied to two programs: *go* and *vortex*. On the other hand, the number of optimizations applied to some programs such as *ks*, *anagram* and *li* is very small.

6.3.3 Comparing with a Non-integrated Approach

In this section, we compare our integrated approach with a non-integrated approach. The non-integrated approach performs each optimization separately. That is, it performs one optimization over the whole program at a time. During the optimization, it also incrementally updates the extended SSA form to avoid re-analysis. Both the integrated approach and the non-integrated approach use the same optimization order.

	speed up
alvinn	2.86
anagram	2.73
ks	2.91
026.compress	2.46
129.compress	1.14
ft	2.81
eqntott	2.74
yacr2	2.46
ear	2.79
li	2.78
sc	2.03
espresso	2.68
m88ksim	2.20
go	2.00
ijpeg	2.70
vortex	1.61
average	2.43

Table 10: The integrated approach is faster (speed up = $T_{non\text{-}integrated} : T_{integrated}$).

6.3.3.1 The Integrated Approach is Faster

Our first experiment compares the efficiency of the integrated approach and the non-integrated approach. Table 10 shows the ratio of the time to perform the optimizations for each program (see Table 8 for the absolute time to perform the optimizations). As expected, the integrated approach is faster than the non-integrated approach for all programs. Moreover, for 14 out of 16 programs, the speed up is between two and three. On average, our integrated approach has a speed up of 2.43.

6.3.3.2 The Integrated Approach Finds More Optimizations

We also compare the effectiveness of these two approaches by counting the number of optimization applied in each approach. Table 11 shows the ratio between the number of optimizations applied in our integrated approach and the number of optimization applied in the non-integrated approach. As shown in the table, for 14 out of 16 programs, our approach finds more optimization cases. On average, the number of optimization cases found in our approach is 1.48 times that found in the non-integrated approach.

	opt-ratio
alvinn	1.03
anagram	1.00
ks	1.00
026.compress	1.41
129.compress	1.48
ft	4.43
eqntott	1.32
yacr2	1.26
ear	1.10
li	1.03
sc	1.50
espresso	1.07
m88ksim	2.40
go	1.29
ijpeg	1.17
vortex	1.23
average	1.48

Table 11: The integrated approach finds more optimizations.

6.3.3.3 The Integrated Approach is Less Sensitive to the Phase Ordering Problem

Finally, we investigate how the optimization order affects the effectiveness of the integrated approach and the non-integrated approach. We use two ways to order the optimizations: one is the *normal order* described in Section 5.5 and used in the above two experiments; the other is the *reversed order* which keeps the useless code elimination as the first in the order, but reverses the order of the remaining five optimizations. $NumOpt_{NormOrder}$ is the total number of optimization cases using the *normal order* while $NumOpt_{ReversedOrder}$ is the total number of optimization cases using the *reversed order*. The difference between $NumOpt_{NormOrder}$ and $NumOpt_{ReversedOrde}$ for both the integrated approach and the non-integrated approach are shown in the second and third column of Table 12.

	$NumOpt_{NormOrder} - NumOpt_{ReversedOrder}$	
	Integrated	Non-integrated
alvinn	10	11
anagram	1	3
ks	0	0
026.compress	0	27
129.compress	0	8
ft	*42*	*3*
eqntott	1	53
yacr2	0	4
ear	4	66
li	0	2
sc	0	18
espresso	1	163
m88ksim	29	72
go	2	45
ijpeg	7	175
vortex	3	256
average	6	57

Table 12: The integrated approach is less sensitive to the optimization orders.

As indicated in the table, for both the integrated approach and the non-integrated approach, the *reversed orde*r performs worse than the *normal order*. Furthermore, except program *ft*, reversing the optimization order causes more optimization opportunities to be lost in the non-integrated approach than in the integrated approach. For program *ft*, reversing the optimization order losss more optimization opportunity in the integrated approach than in the non-integrated approach. However, reversing the optimization order in both approaches finds the same number of optimizations for the program. On average, reversing the optimization order loses six optimization cases in the integrated approach but 57 optimization cases in the non-integrated approach. This indicates that our integrated approach is less sensitive to the optimization ordering than the non-integrated approach.

6.3.4 The Effect of the Optimizations

Performing optimizations can reduce the size of the resulting SSA form. For example, useless code elimination can remove the unused definitions, and constant propagation can reduce the number of *use-def* links. Some optimizations can improve the preciseness of the dataflow information and reduce the size of the SSA form. For instance, branch elimination can remove the spurious *use-def* links to improve the preciseness of the *use-def* information and reduce the size of the SSA form. We measure the improvement on the SSA forms by counting the number of *use-def* links. Table 13 shows the ratio between the numbers of *use-def* links for the un-optimized SSA form and the optimized SSA form. As indicated in the table, the percentage of *use-def* links reduced by the optimizations ranges from zero for *li* to 5.1 for *129.compress*. On average, the optimizations can reduce the number of *use-def* links by 0.8 percents for all the programs in the benchmark suite.

	use-def link ratio (un-opt/opt)
alvinn	1.006
anagram	1.013
ks	1.001
026.compress	1.008
129.compress	1.051
ft	1.007
eqntott	1.008
yacr2	1.002
ear	1.004
li	1.000
sc	1.003
espresso	1.005
m88ksim	1.005
go	1.002
ijpeg	1.007
vortex	1.007
average	1.008

Table 13: The optimizations can reduce the number of *use-def* links represented by the SSA forms.

	instruction ratio (un-opt/opt, gcc –O0)	instruction ratio (un-opt/opt, gcc –O2)
alvinn	1.006	1.005
anagram	1.015	1.000
ks	1.007	1.000
026.compress	1.024	1.010
129.compress	1.044	1.046
ft	1.005	1.009
eqntott	1.005	1.001
yacr2	1.001	1.001
ear	1.010	1.005
li	1.000	1.000
sc	1.009	1.004
espresso	1.033	1.004
m88ksim	1.024	1.027
go	1.008	1.001
ijpeg	1.011	1.005
vortex	1.021	1.017
average	1.014	1.008

Table 14: The optimizations can reduce the number of assembler instructions generated for the benchmark programs.

Performing optimizations can also reduce the size of the program being compiled. We estimate this effect by comparing the number of assembler instructions for the un-optimized programs versus the optimized programs as follows. We first turn *off* the optimizations in the Agassiz Compiler and generate the un-optimized source program for each benchmark program. We then turn *on* the optimizations in the Agassiz Compiler and generate the optimized source for each benchmark program. After that, we feed the above two versions of each benchmark program to the GCC compiler to generate assembler instructions. The second column of Table 14 shows the instruction ratio for the two versions when the GCC back-end optimizations are turned *off*. The third column of the table shows the instruction ratio for the two versions when the GCC back-end optimizations are turned *on*. As can be seen, without using GCC back-end optimizations, the optimizations performed in the Agassiz front-end compiler can reduce the number of instruction by an average of 1.4 percents. When the GCC back-end optimizations are used, the optimizations performed in the Agassiz front-end compiler can reduce the number of instruction by an average of 0.8 percents. The reason that using GCC back-end optimizations may reduce the effect of the Agassiz front-end optimizations is that some optimizations that are performed by the Agassiz front-end optimizations can also be performed by the GCC back-end optimizations.

It should be noted that Table 14 shows only the effect of the scalar optimizations. It does not include the effect of the analyzed result on later phases in the compiler. Currently, we are utilizing the analyzed result to enhance the analyses and the optimizations for parallelization. We are also exporting the analyzed result to our back-end compiler via a standard interface called *high-level information* (HLI) [CTS+98] to enhence the back-end optimizations and instruction scheduling. With the analyzed result utilized by the compiler, we are expecting more significant improvement.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

We have presented our technique to improve the efficiency and effectiveness of scalar analyses and optimizations. We have also implemented this technique in the Agassiz Compiler and demonstrated its feasibility and effectiveness by experiments.

We make contributions to the development of pointer analysis algorithm in three ways. First, we develop a scheme to improve the efficiency of flow-sensitive interprocedural pointer analysis. Second, we demonstrate the feasibility and efficiency of the scheme by performing extensive experiments on a set of benchmark programs. These benchmark programs have a number of lines ranging from several hundred to sixty-seven thousand and have variant complicated language features. Third, we also examine some key questions in performing pointer analysis and provide answers for them. Our implementation allows us to parameterize the level of context-sensitivity and to study the effect of context-sensitivity on the efficiency of the algorithm and the preciseness of the analysis result. We find that for most of the benchmark programs, context information can improve the precision of pointer analysis. However, context information is extremely expensive to compute in terms of time and space. We also find that limiting the context-sensitivity to a moderate level is necessary to handle large program efficiently.

By developing a scheme to integrate pointer analysis and SSA generation, we also make contributions to extend the use of SSA forms to a wider range of programming languages. Until now, SSA forms have been primarily used for programs without general pointer usage because of the complexity to generate SSA forms for programs with general pointer usage. Many existing approaches separate pointer analysis from the SSA generation. It is not practical to use an expensive pointer analysis algorithm as a prelude of SSA generation. Using conservative analysis results produced by an efficient but less precise pointer analysis algorithm may expand the size of the SSA form by a factor of v, which is the number of variables in the program. Our approach to integrate pointer analysis and SSA generation can improve the efficiency of the SSA generation as well as the precision of the resulting SSA forms. Our experiments using a set of benchmark programs validate this claim.

We also present an application of the resulting SSA forms to integrate a group of scalar optimizations. In contrast to existing approaches to integrate optimizations, our approach not only performs these optimizations in tandem at the statement level, but also incrementally updates the SSA forms to allow a faster response to their interactions. Our experiment demonstrates that our approach to integrated optimizations can improve the efficiency and effectiveness in performing a group of optimizations. Moreover, we show that, comparing with a non-integrated approach, an integrated approach is less sensitive to the *phase ordering* problem.

Despite the above efforts, much work remains to be done. In the remainder of this Chapter, we discuss some of our thoughts on what need to be done to make scalar analyses (including pointer analysis and traditional dataflow analysis) even more efficient and effective. We discuss some applications of our analysis results in Section 7.1 and discuss future work in Section 7.2.

7.1 Applications of the Analysis Result

The result of our integrated scalar analyses and optimizations is an extended SSA form. It consists of pointer alias information, interprocedural *Ref/Mod* information and intraprocedural factor *use-def* information. While the information provided by our SSA forms can potentially enable many compiler optimizations, we need to utilize the information in our compiler to translate such potential into real performance improvement. Currently, our Agassiz compiler uses the analysis result in both its front-end parallelizing compiler and its back-end optimizing compiler.

The Agassiz front-end parallelizing compiler uses the extended SSA form to enhance some traditional analyses and optimizations for parallelization. These analyses and optimizations include induction variables identification, loop invariant code motion, privatizable scalar identification, and data dependence test. With the pointer alias and interprocedural *Ref/Mod* information represented by the extended SSA form, we avoid making conservative assumption about pointer aliasing and interprocedural modification effect and perform more aggressive analyses and optimizations. Moreover, we can also perform more precise data dependence test for array accesses using pointers.

Our research group have also developed a standard interface called *high-level information* (HLI) to export useful information, such as pointer alias and interprocedural *Ref/Mod* information from a front-end compiler to a back-end compiler [CTS+98]. We are using the HLI approach to export our analysis results to the Agassiz back-end compiler in order to improve back-end optimizations and instruction scheduling.

Utilizing the analysis results not only can help us measure how the collected information influences other analyses and optimizations, but also can provide feed back to improve the design of pointer analysis. For instance, by investigating how the analysis results are used by other compiler phases, we may be able to develop a heuristics to determine how precise an analysis should be to provide the right amount of information. This heuristics can help us avoid performing an expensive analysis if the improvement cannot be utilized by other analyses and optimizations.

7.2 Future Work

In this section, we discuss several areas related to the work in this dissertation that we have not been able to explore. These areas could be useful starting points for future research.

Probability for maybe points-to targets. The points-to representation used in this dissertation can be extended to include the probability that a pointer may point to a target. This information can be used to calculate the probability the two memory accesses maybe dependent, which is especially useful to optimize a program for architectures supporting speculated execution. For instance, the Superthreaded Architecture [TY96] provides hardware support for run-time data dependence checking. In this model, the compiler speculates on *maybe* data dependencies between threads without generating explicit data synchronization instructions to enforce them. An incorrect speculation is detect at run time if a thread writes to a memory location after a later thread reads from the same memory. When an incorrect speculation happens, the later thread and all of its succeeding threads have to be squashed and then re-executed in order to satisfy the data dependence. Because the cost to squash and re-execute can be very high, it is desirable to speculate only on the data dependence with low probability. As another example, the IA-64 architecture provides an *advanced load* instruction to allow speculative execution of a load instruction before a store instruction which may access the same memory location [WL99]. At run time, if hardware detects that the store instruction writes the same memory location, a data dependence is violated. In this case, the load instruction has to be re-executed. Again, because re-executing the load instruction is expensive, the data dependence probability information can guide compilers to only speculate a load before a store when they have low probability to be dependent.

In some situations, the compiler can estimate the probability of the collected points-to targets. For example, if a pointer is assigned different points-to targets via different branches in a branch statements, a branch probability is a good estimate of the probability of each points-to target. However, it requires further research to investigate how such probability is calculated in more general cases.

Recursive data structure analysis. The pointer analysis algorithm discussed in this dissertation mainly addresses the analysis of non-recursive data structures. For recursive data structures, we currently use a very conservatively approach and cannot distinguish different elements in an aggregate of recursive data structures. However, we found that, in order to parallelize some benchmark programs, more precise dataflow information about different elements in an aggregate of recursive data structures is needed. Thus, handling recursive data structures using a more precise algorithm is a much needed enhancement to our current implementation.

Array SSA forms. Our SSA form treats an array as a single object, and thus, is not able to provide array element level dataflow information. Knobe and Sarkar introduce an array SSA form to capture dataflow information at the array element level [KS98]. Extending our approach to incorporate such an array SSA form can provide better information for parallelization.

Pointer analysis for object-oriented programming languages. Object-oriented programming languages, such as C++ and Java, have becoming more and more popular. These languages also support pointers. Thus, in order to generate high-quality executable codes for these languages, a compiler may need to collect high-quality pointer information.

The required pointer analysis for C++ and Java could be different in several aspects. First, as an object-oriented programming language, both C++ and Java support virtual functions. Similar to an indirect call using function pointers in C, a call to a virtual function may also have several potential callees. Determining the potential callees for an indirect call using function pointers requires an analysis of all the statements that may affect the value of the function pointer. An upper bound of the callee set contains all the addressed procedures in the program being analyzed. However, the potential callees for a virtual function call are determined by the type of the current object. An upper bound of the callee set contains all the member functions of the same name defined in the class hierarchy. Thus, in contrast to a function pointer analysis in C, C++ and Java may need type analysis and class hierarchy analysis. Second, both C++ and Java provide exception handling, which causes irregular control-flow of the program and complicates the analysis. Third, in a Java program, a pointer can only point to the beginning of an object. This restriction can undoubtedly simplify pointer analysis. Lastly, unlike C and C++, Java is a strong-typed system. Thus, handling arbitrary type casts is not an issue in the analysis of Java programs.

BIBLIOGRAPHY

[Amm92] Zahira Ammarguellat. A Control-Flow Normalization Algorithm and Its Complexity. In *IEEE Transactions on Software Engineering*, Vol 18, NO. 3, March 1992.

[Austin] Todd Austin. Pointer-Intensive Benchmark Suite. http://www.cse.ogi.edu/~taustin.

[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Publishing Company, pages 610-618, 1986.

[BC92] Michael Burke and Jong-Deok Choi. Precise and Efficient Integration of Interprocedural Alias Information into Data-Flow Analysis. In *ACM Letters on Programming Languages and Systems*, Vol 1, No 1, March 1992, Pages 14-21.

[BCC+94] Michael Burke, Paul Carini, Jong-Deok Choi and Michael Hind. Efficient Flow-Insensitive Alias Analysis in the Presence of Pointers. *IBM Research Report RC* 19546, 1994.

[Bin94] David Binkley. Interprocedual Constant Propagation Using Dependence Graphs and a Data-Flow Model. In *Proceedings of the International Conference on Compiler Construction*, pages 374-388, 1994.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass Generation of Static Single-Assignment Form for Structural Languages. In *ACM Transactions on Programming Languages and Systems*, Vol 16. No 6, November, 1994.

[BMO90] R. Balance, A. Maccabe and K. Ottenstein. The Program Dependence Web: a Representation Supporting Control-, Data- and Demand- Driven Interpretation of Imperative Languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257-271, June 1990.

[CBC93] Jong-Deok Choi, Michael Burke and Paul Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proceedings of the 20th Annual SCM Symposium on Principles of Programming Languages*, pages 232-245, Jan. 1993.

[CC95] Cliff Click and Keith D. Cooper. Combing Analyses Combing Optimizations. *ACM Transactions on Programming Languages and Systems*, Vol.17, No.2, pages 181-196, March 1995.

[CCF94] Jong-Deok Choi, Ron Cytron and Jeanne Ferrante. On the Efficient Engineering of Ambitious Program Analysis. *IEEE Transaction on Software Engineering*, pages 105-113, February 1994.

[CCL+96] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the International Conference on Compiler Construction*, pages 253-267, 1996.

[CCK+97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 1997, pages 273-286

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pages 451-490, Oct. 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient Accommodation of May-Alias Information in SSA Form. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36– 45, June 1993.

[Cli95] Clifford N. Click. Combining Analyses, Combining Optimizations. Ph.D. Thesis, Rice University, 1995.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. The MIT Press. Cambridge, Massachusetts, London, England, 1990.

[CSS96] Jong-Deok Choi, Vivek Sarkar and Edith Schonberg. Incremental Computation of Static Single Assignment Form. In *Proceedings of the 1996 International Conference on Compiler Construction*. Linkoping, Sweden, April 1996.

[CTS+98] S. Cho, J.-Y. Tsai, Y. Song, B. Zheng, S. J. Schwinn, X. Wang, Q. Zhao, Z. Li, D. J. Lilja, and P.-C. Yew. High-Level Information - An Approach for Integrating Front-End and Back-End Compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 345-355, Auguest 1998.

[CWC90] J.D. Chase, M. Wegman and F.K. Carini. Analysis of Pointer and Structure. In *Proceedings of the Conference on Programming Language Design and Implementation*, vol 25(6) of SIGPLAN Notices, pages 296-310, Jan. 1993.

[Deu94] Alain Deutsch.Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Symposium on Programming Languages Design and Implementation*, pages 230-241, Jun. 1994.

[EGH94] Maryam Emami, Rakesh Ghiya and Laurie J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN'94 Symposium on Programming Languages Design and Implementation*, pages 242-256, Jun. 1994.

[GH95] Rakesh Ghiya and Laurie J. Hendren. Connection Analysis: A practical interprocedural heap analysis for C. In *Proceedings of the e 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1-15, Jan. 1996.

[Ghi95] R. Ghiya. Practical techniques for interprocedural heap analysis. *Master thesis, McGill University*, 1995.

[GSW95] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Transaction on Programming Languages and Systems*, Vol 17 No 1, pages 85-122, January 1995.

[Hav94] Paul Havlak. Interprocedural Symbolic Analysis. *Phd thesis, Rice University*, 1994.

[Hen90a] L. Hendren. Parallelizing Programs with Recursive Data Structures. *PhD thesis, Cornell University*, 1990.

[Hen90b] L. Hendren. Parallelizing Programs with Recursive Data Structures. In *IEEE Transactions on Parallel and Distributed Systems*, 1(1), pages 35-47, Jan. 1990.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Auguest 1992

[HH98] Rebecca Hasti and Susan Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proceedings of the ACM SIGPLAN'98 Symposium on Programming Languages Design and Implementation*, pages 97—105, 1998

[HHN92a] J. Hummel, L. Hendren and A. Nicolau. Abstract Description of Pointer Data Structures: An Approach for Improving the Analysis and Optimization of Imperative Programs. In *ACM Letters on Programming Languages and Systems*, 1(3), pages 243-260, Sep. 1992.

[HHN92b] J. Hummel, L. Hendren and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249-260, Jun. 1992.

[HHN94a] J. Hummel, L. Hendren and A. Nicolau. A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structure. In *Proceedings of the 8th International Parallel processing Symposium*, pages 208—216, Apr. 1994.

[HHN94b] J. Hummel, L. Hendren and A. Nicolau. A General Data Dependence Test for Dynamic, Pointer-Based Data Structures. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 218-229, Jun. 1994.

[HHN94c] J. Hummel, L. Hendren and A. Nicolau. A Framework for Data Dependence Testing in the Presence of Pointers. In *Proceedings of the 23rd Annual International Conference on Parallel Processing*, pages 216-224, Aug. 1994.

[HP97] Michael Hind and Anthony Pioli. An Empirical Comparison of Interprocedural Pointer Alias Analyses. *IBM Research Report RC* 21058, 1997.

[HP98] Michael Hind and Anthony Pioli. Accessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In Giorgio Levi, editor, *Lecture Notes in Computer Science*, 1503, pages 57-81, Springer-Verlag, 1998. Proceedings from the 5th International Static Analysis Symposium.

[HPR89] S. Horwitz, P. Pfeiffer and T. Reps. Dependence Analysis for Pointer Variables. In *Proceeding of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 24(7), pages 28-40, Jun. 1989.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient Computation of Interprocedural Definition-Use Chains. In *ACM Transactions on Programming Languages and Systems* Vol.16, No.2, pages 175-204, Mar. 1994.

[JH94] Justiani and Laurie J. Hendren. Supporting Array Dependence Testing for an Optimizing/Parallelizing C Compiler. In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94 (Peter A. Fritzson, ed.), no. 786 in Lecture Notes in Computer Science, Edinburgh, Scotland, pp. 309-323, Springer-Verlag, April 7-9,1994.

[Joh94] Richard C. Johnson. Efficient Program Analysis Using Dependence Flow Graphs. *PhD thesis, University of Cornell*, 1994.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA Form and Its Use in Parallelization. In *Proceedings of the ACM SIGPLAN Conference on Principles of rogramming Languages*, pages 107-120, 1998.

[Lan92] William Landi. Interprocedural Aliasing in the Presence of Pointers. *PhD thesis*, *The State University of New Jersey at Rutgers*, 1992.

[Lar89] James R. Larus. Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors. *PhD thesis, University of California, Berkeley*, 1989.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting Conflicts Between Structure Access. In *Proceedings of the SIGPLAN'88 Conference on Programming LanguageDesign and Implementation*, pages 21-34, Jun. 1988.

[Lu98] John Lu. Interprocedural Pointer Analysis for C. *PhD thesis, University of Rice*, 1998.

[LR90] William Landi and Barbara G. Ryder. Pointer-induced Aliasing: A Problem Classification. In *Proceeding of the Eighteenth Annual ACM Symposium on Principles of rogramming Languages*, pages 93-103, Jan. 1990.

[LR92] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235-248, Jun. 1992.

[LRZ93] William Landi and Barbara G. Ryder and Sean Zhang. Interprocedural Modification Side Effect Analysis With Pointer Aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56-67, Jun 1993.

[MLR+93] Thomas Marlowe and William Landi and Barbara Ryder and Jong-Deok Choi and Michael Burke and Paul Carini. Pointer-Induced Aliasing: A Classification. In *ACM SIGPLAN Notices*, pages 67-70, Sep. 1993.

[Muc97] Steven S. Muchnick. Advanced Compiler Design And Implementation. Morgan Kaufmann Publishers, Inc. 1997.

[PLR94] H. D. Pande, W. Landi and B. G. Ryder. Interprocedural Def-Use Associations for C Systems with Single Level Pointers. In *IEEE Transactions on Software Engineering*, Vol 20(5), pages 382-403, May 1994.

[Ruf95] Erik Ruf. Context-Insensitive Alias Analysis Reconsidered. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 13-31, Jun. 1995.

[Ruf97] Erik Ruf. Partitioning Dataflow Analyses Using Types. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 15-26, Jan. 1997.

[SJ98] A. V.S. Sastry and Roy D.C. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 15-25, Jun. 1998.

[SL96] Patrick W. Sathyanathan and Monica S. Lam. Context-Sensitive Interprocedural Analysis in the Presence of Dynamic Aliasing. In *Proceedings of the First SUIF Compiler Workshop*, pages 1-20, 1996.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis : Theory and Applications*, pages 189-236. Prentice Hall, 1981.

[SRW96] M. Sagiv, T. Reps and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, Jan. 1996.

[Ste95] Bjarne Steensgaard. Sparse Functional Stores for Imperative Programs. In *Proceedings of the SIGPLAN '95 Workshop on Intermediate Representations*, Jan. 1995.

[Ste96a] Bjarne Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and

Unions. In *Proceedings of the 1996 International Conference on Compiler Construction*, Apr. 1996.

[Ste96b] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[Sto95] Eric Stoltz. Intermediate Compiler Analysis via Dereference Chaining. *PhD thesis, Oregon Graduate Institute of Science & Technology*, 1995.

[SWG94] Eric Stoltz, Michael Wolfe and Michael P. Gerlek. Constant Propagation: A Fresh, Demand-Driven Look. In *Proceedings of the ACM Symposium on Applied Computing*, 1994.

[TP95] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. Proceedings of the 9th ACM International Conference on Supercomputing, page 144, 1995.

[TY96] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proceedings of the Int'l Conf. on Parallel Architectures and Compilation Techniques,* October 1996.

[Wil97] Robert P. Wilson. Efficient Context-Sensitive Pointer Analysis for C Programs. *PhD Thesis, Stanford University*, 1997.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1-12, Jun. 1995.

[WL99] Youfeng Wu and Yong-fong Lee. Comprehensive Redundant Load Elimination for IA-64 Architecture. *The 12th Workshop on Languages and Compilers for Parallel Compilers*, 1999.

[Wol92] M. Wolfe. Beyond Induction Variables. Proceedings of the SIGPLAN'92 Symposium on Programming Langages Design and Implementation. SIGPLAN Notice 27, pages 162-174, July 1992.

[Wol96] Michael Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, 1996.

[WS97] Deborah L. Whitfield and Mary Lou Soffa. An Approach for Exploring Code Improving Transformations. *ACM Transaction on Programming Languages and Systems*, Vol 19, No 6, pages 1053-1084, Nov. 1997.

[WZ91] Mark N. Wegman and Frank Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, Vol.13, No 2, pages 181-210, 1991

[YHR99] Susan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementtation*, 1999.

[ZRL96] Sean Zhang, Barbara G. Ryder and William Landi. Program Decomposition for Pointer Aliasing: A Step towards Practical Analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, Oct. 1996.

[ZRL98] Sean Zhang, Barbara G. Ryder and William Landi. Experiments with Combined Analysis for Pointer Aliasing. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*, Jun. 1998.

[ZTZ+99] B. Zheng, J. Y. Tsai, B. Y. Zang, T. Chen, B. Huang, J. H. Li, Y. H. Ding, J. Liang, Y. Zhen, P. C. Yew and C. Q. Zhu. Designing the Agassiz Compiler for Concurrent Multithreaded Architectures. *The 12th Workshop on Languages and Compilers for Parallel Compilers*, 1999.