The Dissertation Committee for Teck Bok Tok

certifies that this is the approved version of the following dissertation:

Removing Unimportant Computations in Interprocedural Program Analysis

Committee:

Calvin Lin, Supervisor

Kathryn S. McKinley

James C. Browne

Greg Lavender

Samuel Z. Guyer

Removing Unimportant Computations in Interprocedural Program Analysis

by

Teck Bok Tok, B.Sc.(Hons.), M.Sc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

To my wife, Gim Gee

Acknowledgments

I am indebted to many people for helping me become who I am today. First, I would like to thank my supervisor, Calvin Lin, for his constant support, guidance, patience, and encouragement throughout the years. He has enthusiastically invested the time and energy in my research.

I would like to thank my committee members for all their insightful feedbacks and suggestions: Kathryn McKinley, James C. Browne, Greg Lavender, and Samuel Guyer, whose Broadway compiler was indispensable in my research. I also wish to express my gratitude to the Department and its Faculties and Staff, for an excellent graduate program, conducive learning environment, and the opportunity bestowed on me.

I would also like to thank my colleagues, whom I worked with closely and whose assistance proved to be invaluable. I greatly enjoy and appreciate: Adam Brown, Walter Chang, Ben Hardekopf, Maria Jump, Alison Norman, and former colleagues, Samuel and Daniel Jiménez.

I also owe much to my family and extended family, whose confidence and support has enabled me to focus and strive.

Last but not least, to my wife and best friend, Gim Gee, whose confidence, support, patience, and understanding helped me survive the trials and tribulations, and whose love and company without which I cannot imagine.

Teck Bok Tok

The University of Texas at Austin

August 2007

Removing Unimportant Computations in Interprocedural Program Analysis

Publication No. _____

Teck Bok Tok, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Calvin Lin

Existing solutions to program analysis problems often trade between scalability and precision. We propose new techniques to improve scalability of interprocedural high-precision program analyses without sacrificing precision. The key insight is that there is a high volume of unimportant computations in these analyses. Our way to improve analysis time is by identifying and reducing these computations.

We identify unimportant work in three classes of program analyses: flow-sensitive dataflow analysis, context-sensitive dataflow analysis, and reachability-based analysis. We propose new algorithms to reduce this inefficiency. We are interested in analysis problems and programs that involve pointers, because they

are used in many modern languages. Their presence sometimes complicates the analysis problems, and they sometimes raise the problem size due to large points-to sets in the program.

Our solutions include an improved worklist management algorithm used in flow-sensitive interprocedural analysis that drastically reduces the amount of work placed on the worklist. We introduce Relevance-Based Context Partitioning, a new algorithm that groups contexts together in a way such that only important information will be computed precisely, and that the number of contexts to analyze is considerably smaller. For the class of reachability-based analysis, the way to improve scalability is to reduce the graph used in the analysis. The nodes in the graph represent dataflow facts, while the edges represent flow functions. The analysis problem is then reduced to a reachability problem. Compared to existing efficient algorithms, we present two new algorithms that significantly reduce the size of the graphs. The first idea uses the graph to represent data dependences instead of the more indirect control dependences that are typically used. The second idea identifies nodes that can be eliminated, because they do not contribute to solving the analysis problem.

We evaluate our ideas by applying our algorithms to five error-checking analyses and by comparing results against state-of-the-art algorithms, using a large suite of 19 open-source programs. We are able to improve performance significantly. Our new algorithms for flow-sensitive analysis achieve approximately 2× speedup, on average. For context-sensitive analysis, our technique sometimes also allows the new algorithm to solve cases that previously could not be solved due to run out of memory. For other cases, the average speedup is 7.0× and can be more than 200× in some cases. Finally, our new algorithms for reachability-based analysis can produce graphs that are just 6% the size of the original graphs. Consequently, more benchmarks can be analyzed successfully, and the average analysis time speedup is

as high as 5.5× faster.

Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xvi**

List of Figures **xviii**

Chapter 1 Introduction **1**

1.1 Observation 6

1.2 Solutions 8

1.2.1 Defining Unimportant Computations 9

1.3 Evaluation 11

1.4 Contributions 12

1.5 Outline 13

Chapter 2 Algorithm Space **15**

2.1 Basic Terms 15

2.1.1 Analysis Algorithm and Analysis Problem 16

2.1.2 Lattice and Precision 16

2.1.3 Computation Granularity and Precision 17

2.1.4 Soundness 18

2.2 Properties of Program Analyses 19

2.2.1 Scope of Analysis 20
2.2.2 Flow-Sensitivity 21
2.2.3 Context-Sensitivity 24
2.2.4 Memory Model 27
2.3 Selected Subset in Analysis Space 30
2.3.1 Complexity 31
Chapter 3 System Design and Evaluation Methodologies 33
3.1 System Design 33
3.1.1 Overall System Architecture 33
3.1.2 Analysis Framework 34
3.1.3 Precision Dimensions 36
3.1.4 Memory Model 37
3.1.5 Definitions and Uses 38
3.1.6 Pointer Analysis 40
3.1.7 Client Dataflow Analysis 41
3.1.8 Precision and Accuracy 41
3.1.9 Library Routines 42
3.2 Implementations 43
3.2.1 Program Representation 43
3.2.2 Dominance Relation and Reaching Definitions 44
3.3 Evaluation Methodologies 48
3.3.1 Metrics 48
3.3.2 Client Analysis Problems 49
3.3.3 Benchmarks 51
3.3.4 Platform 51

Chapter 4 Efficient Flow-Sensitive Interprocedural Dataflow Analysis **54**

4.1 Introduction 54

4.1.1 Contributions 57

4.2 Analysis Framework 57

4.3 $\mathcal{DU}$: Worklist Management 58

4.3.1 Structure of a Worklist Algorithm 59

4.3.2 Naive Worklist Algorithms 60

4.3.3 Worklist Algorithm Using Intraprocedural Def-Use Chains . . 60

4.3.4 Dynamic Def-Use Computation 60

4.3.5 Bundles 62

4.3.6 Handling Interprocedural Def-Use Chains 64

4.3.7 Full Version of Algorithm $\mathcal{DU}$ 65

4.3.8 Exploiting Loop Structure 66

4.4 Evaluation 68

4.4.1 Benchmarks and Metrics 68

4.4.2 Setup 68

4.4.3 Empirical Lower Bound Analysis 69

4.4.4 Results 70

4.5 Conclusion 73

Chapter 5 Relevance-Based Context Partitioning **76**

5.1 Introduction 76

5.1.1 Contributions 78

5.2 Our Solution 79

5.2.1 What is in a Contour? 82

5.2.2 Algorithm Overview 82

5.2.3 Analyzing a Procedure 83
5.2.4 Computing the Partitioning Vector 85
5.2.5 Choosing a Contour 89
5.2.6 Contour Refinement 90
5.2.7 Reusing Contours Context-Insensitively 92
5.2.8 Precision and Accuracy 93
5.2.9 Application to Client-Driven Pointer Analysis 94
5.2.10 Implementation 95
5.3 Evaluation 96
5.3.1 Methodology 96
5.3.2 Context-Sensitive Analysis: RBCS versus FSCS 97
5.3.3 Client-Driven Analysis: RBCS-CD versus CD 101
5.3.4 RBCS versus PTF 104
5.4 Coupled Analyses 105
5.5 Conclusion 108
Chapter 6 Reachability-Based Analysis 110
6.1 Introduction 110
6.1.1 Contributions 112
6.2 IFDS Baseline Algorithm 115
6.2.1 IFDS Problems 116
6.2.2 The Original IFDS Algorithm 116
6.2.3 Practical Issues and Modifications 118
6.2.4 Effects of Pointers on IFDS Analysis 122
6.3 Sparse IFDS 124
6.3.1 Motivation 124

6.3.2 Algorithm . . . 125
6.4 Variable-Pruning IFDS . . . 126
6.4.1 Algorithm . . . 127
6.4.2 Difference from Sparse IFDS . . . 128
6.4.3 Difference from Demand IFDS . . . 130
6.4.4 Analysis Configurations and Feedback-Based Analysis . . . 133
6.5 Evaluation . . . 135
6.5.1 Methodology . . . 136
6.5.2 Effects of Pointers on IFDS Analysis . . . 138
6.5.3 Results of Sparse IFDS . . . 140
6.5.4 Results of Variable-Pruning IFDS . . . 141
6.5.5 Results of Feedback-based Variable-Pruning IFDS . . . 146
6.5.6 Detail Comparisons . . . 150
6.5.7 Comparing Best IFDS Algorithm with Dataflow Analysis . . . 157
6.6 Cycle Elimination . . . 157
6.6.1 Motivation . . . 158
6.6.2 No Speedup: Too Much, Too Late . . . 159
6.6.3 Early Detection: Too Little, Not Enough? . . . 160
6.7 Conclusion and Future Work . . . 162
Chapter 7 Related Work **164**
7.1 Worklist . . . 164
7.1.1 Other Approaches . . . 166
7.2 Sparse Analysis . . . 167
7.3 Context-Sensitive Analysis . . . 167
7.3.1 Partitioning Contexts . . . 167

7.3.2 Other Context-Sensitive Analyses 169
7.3.3 Context-Loss Problem 170
7.3.4 Coupled Analysis 170
7.4 Reachability-Based Analysis 171
7.4.1 Relation to Other Analyses 171
7.4.2 Variable Pruning 172
7.5 Cycle Elimination 174
Chapter 8 Conclusion **175**
8.1 Contributions 175
8.2 Future Work 176
Bibliography **177**
Vita **200**
Index **201**

List of Tables

2.1 Analysis properties and examples of their possible values. 20

3.1 Some features in the analysis framework. 34
3.2 Properties of the benchmarks. 52
3.3 Properties of the benchmarks, continued. 53

5.1 Number of distinct flow values per variable in the client analyses. . . 96
5.2 Relevance-Based Context-Sensitive analysis vs. FSCS analysis, part 1. 98
5.3 Relevance-Based Context-Sensitive analysis vs. FSCS analysis, part 2. 99
5.4 Relevance-Based Context-Sensitive analysis vs. FSCS analysis, part 3. 100
5.5 Relevance-Based Context Sensitive with Client-Driven analysis vs. Client-Driven analysis, part 1. 102
5.6 Relevance-Based Context Sensitive with Client-Driven analysis vs. Client-Driven analysis, part 2. 103
5.7 Relevance-Based Context Sensitive with Client-Driven analysis vs. Client-Driven analysis, part 3. 104
5.8 Results of simulations of one PTF on selective programs. 106

6.1 Setups for applying Variable-Pruning (VP) to IFDS analysis. 134

6.2 Effects of various new IFDS algorithms on exploded graphs. 142
6.3 Effects of using feedbacks in Variable-Pruning IFDS algorithm. . . . 150
6.4 Comparing different Sparse, Variable-Pruning, and Feedback IFDS algorithms. 151
6.5 Percentages of loop nodes in flow graphs and exploded graphs. . . . 159

List of Figures

3.1 Analysis framework: Algorithm to analyze a procedure. 35

3.2 Node numbering on a dominator tree. 46

4.1 A loop example. 55

4.2 An example CFG. 59

4.3 Initial version of algorithm $\mathcal{DU}$. 61

4.4 Maximum and average ratio $r = C/(B+1)$, in log scale. 63

4.5 Efficient R_{bundle} that uses bundles. 63

4.6 Full version of algorithm $\mathcal{DU}$. 66

4.7 Three loop examples. 67

4.8 Performance results of $\mathcal{DU}$ and its variant, on CI pointer analysis (part 1). 71

4.9 Performance results of $\mathcal{DU}$ and its variant, on CI pointer analysis (part 2). 72

4.10 Performance results of $\mathcal{DU}$ and its variant, on CS pointer analysis (part 1). 74

4.11 Performance results of $\mathcal{DU}$ and its variant, on CS pointer analysis (part 2). 75

5.1 Partitioning call sites. 78
5.2 Call graph example. 80
5.3 RBCP: Analysis on a procedure. 84
5.4 RBCS Analysis Algorithm. 86
5.5 Example of computing the Partitioning Vector. 88
5.6 Example illustrating the need for contour refinement. 91

6.1 Constructing exploded graph. 117
6.2 Constructing exploded graph. 121
6.3 Effect of pointer analysis precision on IFDS analysis. 123
6.4 Constructing sparse exploded graph. 125
6.5 Variable-Pruning IFDS Algorithm. 128
6.6 Difference between Sparse IFDS algorithm and Variable-Pruning IFDS algorithm. 129
6.7 Difference between Demand IFDS algorithm and Variable-Pruning IFDS algorithm. 132
6.8 Effects of pointer analysis on IFDS algorithms. 139
6.9 Performance results of *Sparse-IFDS* relative to *IFDS*. 141
6.10 Results of Variable-Pruning IFDS against baseline *IFDS*. 143
6.11 Results of Variable-Pruning IFDS relative to Demand IFDS algorithm. 145
6.12 Results of Feedback-based Variable-Pruning IFDS algorithm, relative to baseline algorithm. 147
6.13 Results of Feedback-based Variable-Pruning IFDS algorithm, relative to non-Feedback Variable-Pruning IFDS algorithm. 148
6.14 The two best IFDS algorithms relative to Client-Driven analysis, using total analysis time. 157

6.15 Breakdown of IFDS analysis time into initialization, graph construction, and reachability analysis. 161

Chapter 1

Introduction

The objective of static program analysis is to gather information at compile-time regarding the run-time behavior a program [100]. The gathered information is used for many purposes, including program understanding, verifying that it follows specifications, improving its performance and finding bugs and security vulnerabilities. These applications are used to increase productivity, raise competitiveness of products, and improve the public image of institutions.

Examples of program analysis techniques include type analysis, dataflow analysis, and reachability-based analysis. Type analysis is used in polymorphic programming languages to discover possible run-time type information in order to facilitate optimizations [82, 61, 129, 135, 31]. The purpose of dataflow analysis is to (statically) gather global information about possible sets of values calculated at various points in a program or how a program manipulates its data [100]. Among other applications, it is used to discover program code that is never executed (dead code analysis) and how each assignment affects other statements (reaching definition analysis). Reachability analysis is a technique that encodes an analysis problem as

a graph where the node represents possible program states at statements, and the problem is reduced to figuring out what nodes are reachable from nodes representing the initial program state [121].

Given the large space of analysis algorithms, two qualities are typically used to choose an algorithm: *precision* and *scalability*. These are relative qualities so that it is possible to compare two algorithms using each quality. We informally describe each quality below.

Precision

The precision of an analysis measures how closely its results reflect the run-time behavior of the input program. The exact meaning of precision deserves detailed explanation, which we will discuss in Chapter 2. At this time, we will relate it to the abstraction model used by an analysis.

The abstraction model provides the compiler with a means to statically represent the potentially unbounded run-time program states. The model has a set of attributes, which include how the program and flow values are represented. The possible values of each attribute offer different granularities of detail for that particular attribute so that the set of chosen values for each affects how closely, or precisely, the states generated by the model resemble the run-time program states. Consequently, each attribute is also referred to as a *precision dimension*, and the chosen attribute values constitute the precision of the analysis. Note that we use the term "dimension" because it has been previously used in the literature [130], even though the dimensions are not necessarily orthogonal.

A precise analysis tends to be more useful than one that is imprecise. For example, an imprecise analysis can miss optimization opportunities, which can lead

to inferior products. Another analysis that identify security vulnerabilities that do not exist in reality can cause programmers to waste valuable time doing manual inspections. Therefore, an analysis should be as precise as possible, but only if the additional precision can actually improve results. For example, if the objective is to find all zero integers and only multiplications are allowed, then the analysis only needs to model integers as zero or non-zero, rather than calculating possible actual values (too precise).

Scalability

Scalability refers to the analysis' ability to handle increasing amounts of work due to growing program size. Assuming that the analysis has to inspect the entire program, an ideal algorithm has an analysis time that grows linearly with the input program size. Since many algorithms use iterative approaches to compute fix-point solutions for non-trivial problems, linear algorithms are typically not available.

As modern programs are large and complex, with no trend of getting any smaller, scalable analysis is necessary to complete analysis on large programs using limited resources. Without any aid from the analysis, programmers must spend many man-hours (or days) to manually inspect large programs [15].

One common method to measure scalability is to compute the theoretical worst-case complexity of the algorithm. This method is flawed in that the worst-case scenario may be uncommon, and in practice, the average analysis time is not as expensive. Two algorithms may have the same asymptotic complexity, but in practice, one of them is significantly faster for most real programs. Therefore, it is just as important to evaluate and compare algorithms by using a large suite of large benchmarks.

Precision vs. Scalability

For some analysis problems such as computing intraprocedural liveness analysis and dead code analysis, there exist algorithms that are both very precise and very efficient [70]. However, there are two reasons why these two qualities are at odds for problems such as pointer analysis:

- The analysis focuses on tracking properties of data structures that span procedure boundaries, sometimes spanning the entire program. Using an intraprocedural analysis would require making very conservative assumptions at call sites that eventually hurt analysis precision. The alternative, *interprocedural* or whole-program analysis, is more precise. However, common programming practices, such as modular programming, encourage separation of concerns, making interprocedural analyses more complex and time-consuming.

- The analysis uses models to represent the data structures and operations in the programs, as well as states of program executions. A sophisticated model is sometimes necessary for precise analysis, but it is usually more complex and expensive. For example, complex graphs are necessary to accurately capture the points-to relations at each statement. Using an imprecise model can lead an analysis to report bogus results such as many false positive bugs—bugs that do not actually exist in the programs—so that programmers have to waste many hours inspecting the analysis report.

Faced with such reality, analysis algorithms are compelled to choose either precision or speed. For example, consider the problem of finding Remote Access vulnerability in programs—errors that allow a remote hacker to gain control of sensitive components in programs. A fast analysis may produce many false positives

that overwhelm programmers, while a more precise analysis producing fewer or no false positives can take hours to complete [54].

Because modern programs are large and complex, the tension between precision and scalability is critical. In addition, large programs also tend to have a higher demand for precise analysis, because: (1) many slow programs are large programs, and using precise techniques, such as interprocedural analysis, can expose more optimization opportunities such as library-level optimizations [51]; and (2) large programs have higher tendencies to contain bugs and security holes.

The Pointer Factor

The widespread use of pointers in modern programs makes the task of program analysis even more challenging. Many analysis problems need pointer information so pointer analysis needs to be solved first, but the pointer analysis problem is known to be undecidable. If an imprecise pointer analysis is used, the output often contains larger points-to sets that lead to spurious flow values in the other analyses. As a result, it appears that tradeoffs between precision and scalability in analyzing modern programs are inevitable.

Existing Algorithms

Many existing efficient algorithms scale well to large programs, but they often suffer from one or more limitations:

- They improve scalability by arbitrarily sacrificing precision in one or more dimensions. An example is the way some algorithms address context-sensitivity which is a precision dimension. A context-sensitive analysis respects the semantics of procedure call-return: for each procedure, the analysis distinguishes

arguments from different contexts in which the procedure is executed; the analysis also ensures the return value at a call site is not improperly propagated to other call sites. Such an analysis is expensive due to the exponential number of contexts. A context-insensitive analysis is less precise because it merges all contexts while analyzing a procedure, but it is also linear with respect to the number of procedures. The k-CFA approach [138] is a family of algorithms seeking compromise: each choice of a fixed k induces an algorithm that merges different contexts, so that the greater k is, the better precision and higher complexity the algorithm is. The problem with this approach is that because an arbitrary k is chosen and used throughout analysis, the predetermined level of precision is not sufficient all the time.

- Other approaches may be precise in one dimension but not in another. For example, the use of Binary Decision Diagrams data structure (BDD) enables context-sensitive analysis to scale very well [146, 157], but these algorithms are flow-insensitive. To date, it is not clear if this tradeoff between precision dimensions is due to the inherent limitation of the technique or that the limitation will be overcome in due time.

1.1 Observation

What is needed is a more principled approach that, while being applicable to multiple dimensions of precision, improves analysis scalability without sacrificing precision. We observe that many precise algorithms are inefficient, because they perform many *unimportant computations*. These computations come in two forms: (1) the same computations are repeated many times without any new updates, and (2) considerable unimportant or irrelevant information is computed precisely when their

precision does not affect the output quality of the final analysis result. Section 1.2.1 will further discuss this concept of unimportant computations.

The notion of unimportant computation is not new, but no previous work specifically studies unimportant computations as a general problem in program analysis. All previous work uses the notion implicitly, and there is no formal definition.

Both types of unimportant computations can be found in many dimensions of precisions. We briefly give two examples:

Flow-sensitive analysis: In a flow-sensitive analysis that uses a worklist, the way the worklist is managed directly affects how frequently the same computations are repeated. For example, when the Hind and Pioli algorithm [67] is applied to the `nn` program (approximately 36K lines of C), we find that only 3% of the basic block visits are useful—the others repeat same computations without any flow value updates.

Context-sensitive analysis: In context-sensitive analysis, a procedure is analyzed separately for its different calling contexts. While such analysis is more precise because it distinguishes the differences among contexts, there often exist substantial similarities among the contexts. By exploiting the similarities, much of the same information need not be computed repeatedly. This insight is the basis of previous work such as Partial Transfer Function (PTF) [153] and the use of BDD's [146, 157].

Unimportant computations also exist in other forms of program analysis. For example, in a reachability-based analysis, a graph is used where the nodes represent dataflow values at statements, and edges represent flow functions. Analysis of a graph is reduced to reachability of the nodes [121]. We find that as much as 94% of

the nodes are redundant, so a significant fraction of computations during reachability analysis is not important.

1.2 Solutions

The high-level goal of this dissertation is to improve the scalability of precise interprocedural program analysis without sacrificing precision. We achieve this goal by identifying, understanding, and reducing the high volume of unimportant computations in high-precision analyses. This class of high-precision analysis algorithms is defined later in Chapter 2.

This dissertation has three components. The first two components focus on two dimensions of precision: flow-sensitivity and context-sensitivity. These components present techniques targeted to program with pointers. The third component focuses on a reachability-based analysis. In all three components, we will define and identify sources of unimportant work in existing techniques, and we will measure or estimate them using appropriate metrics. We will propose new algorithms that will reduce or eliminate unimportant computations. We will then measure how much that elimination helps reduce consumption of space and time resources. In some cases, we will show that some analyses that could not complete with previous techniques (because they run out of memory) can now complete with our new techniques.

The first component of our study improves the basic worklist algorithm used in flow-sensitive interprocedural dataflow analysis. We introduce a technique that drastically reduces the amount of work placed on the worklist. Our technique is similar to that proposed by others [117, 118, 147], but ours is the first that applies to programs that require pointer analysis.

The second component reduces computations of unimportant and duplicate information in context-sensitive dataflow analysis. Our technique works when a pointer analysis is combined with a client analysis—the latter depends on pointer information provided by the former. Based on information provided by the client, our technique groups similar contexts together in a way such that only important information will be computed precisely, and the number of contexts to consider is substantially reduced. Consequently, the pointer analysis scales better to large programs.

The third component improves a reachability-based analysis by removing unimportant nodes from the graphs. These removed nodes either do not carry useful information or do not contribute to the solution of the analysis problem.

This dissertation demonstrates that the identification and removal of unimportant computations effectively improves the scalability of precise program analyses on large programs. Even though we will consider only two dimensions of precision in dataflow analyses, and one reachability-based analysis, our experience can benefit improvements in other dimensions or other forms of program analyses as well.

1.2.1 Defining Unimportant Computations

We consider *unimportant computations* to be those that do not contribute to the *accuracy* of the analysis results. Here we provide definitions of these two terms.

The two terms accuracy and precision both describe analysis qualities, but they are different in their usages. The main problem with the term precision is, for some analyses, such as pointer analysis, there is no universally accepted metric that measures precision. To cope with this problem, we look at the "effectiveness" of pointer analysis when it is used for other applications, which leads us to the

definition of accuracy.

The meaning of the term accuracy depends on the analysis or combination of analyses. For the case of a single analysis, the term describes the analysis' primary output—the output that meets the objective of the analysis. For example, the output of a dead code analysis is the set of statements that are never executed, while the output of a tool that finds security vulnerabilities is a list of security vulnerabilities. The term provides a way to compare two analyses that solve the same problem. For example, if the set of dead codes computed by algorithm X is sometimes larger and never smaller than that computed by algorithm Y, and all computed dead codes are in fact dead in reality, then algorithm X is more accurate then algorithm Y.

The definition is extended to a combination of two analyses. If analysis X depends on analysis Z and the output of the combined analyses Z+X is the output of X, then the accuracy of the combined analyses is the accuracy of X. Similarly, the accuracy of Z with respect to the combined analyses is also the accuracy of X.

For example, suppose two pointer analysis algorithms P and Q are used independently by a dead code analysis D, and there is no difference in the output by D; then as far as D is concerned, the two pointer analyses are considered equally accurate. If P sometimes causes D to produce a larger set of dead code but never smaller, and all outputs are in fact dead code in reality, then P is more accurate than Q.

We will use the concept of accuracy to define unimportant computation. It is one that fits one or both characteristics as follows:

1. Repeated computations: Consider an example of flow-sensitive analysis. During each visit to a program statement, there is a computation to evaluate a set

of flow values. If between two consecutive computations on the same statement, the output flow values do not change, then the second computation is a repeated computation. It is redundant because removing it will not cause any effect on the final analysis result of the entire program.

We extend this definition to other precision dimensions. For example, we can also find repeated computations on procedures in a context-sensitive analysis, when consecutive computations on a procedure produce same output.

2. Computing unimportant information precisely. The precision of some information has no effect on the accuracy of an analysis. Therefore, computing this information precisely only adds overhead (space and/or time) to the analysis.

One reason the idea of unimportant computations has received little or no attention in the past is due to the confusion between accuracy and precision. The distinction is, in a nutshell, between what we want to get in the end and how we get there.

1.3 Evaluation

We evaluate our ideas by implementing new algorithms that eliminate or reduce unimportant computations, by applying them to a set of non-trivial analyses problems and by analyzing an extensive suite of large benchmarks. These problems and benchmarks allow us to measure and compare precision, accuracy, and scalability. We measure the volume of unimportant computations and performance improvement provided by our new algorithms, using analysis time, memory usage, accuracy, and other metrics where appropriate. We evaluate our new algorithms by comparing with outputs produced by existing to-date algorithms. In particu-

lar, part of the comparisons is to verify that our new algorithms are not sacrificing accuracies.

We use a pointer analysis and five error-checking analyses [54] that use the pointer analysis. These five analyses are important analysis problems, including the identification of Format String Vulnerability [104] and File State errors—interprocedural analyses that generally yield better accuracy with higher precision.

We use the Broadway compiler [52] because its infrastructure allows us to adjust precision of analyses, thus enables us to perform a wide range of experiments.

Additional details on the evaluation methodologies can be found later in Section 3.3.

1.4 Contributions

This research makes the following contributions:

1. We identify and eliminate the source of repeated computations in flow-sensitive analysis by using a new worklist-management algorithm that removes a considerable amount of redundant computations. Our technique differs from previous work [117, 118, 147], because ours is the first that applies to programs that require pointer analysis. When we apply our algorithm to a pointer analysis, the analysis time is reduced by half on average with up to 10× speedup.

2. We present Relevance-Based Context Partitioning, a principled approach to context-sensitive analysis that limits the number of contexts to be analyzed. Our technique is applicable when a pointer analysis is combined with another analysis that requires pointer information. Based on information gathered during analysis, the technique groups similar contexts together in such a way

that only important information are computed precisely. This approach significantly reduces the amount of computations on unimportant and duplicate information, avoiding the high cost of context-sensitive analysis. Without our technique, only seven out of our 19 benchmarks successfully complete; with the new technique, all benchmarks successfully complete. Among those seven benchmarks, the average speedup is 7.0× and is more than 200× in the best case. The technique also uses 4× less memory.

3. We present a sparse algorithm and a pruning algorithm that reduce the graph used in a reachability-based analysis. These two techniques are orthogonal, and the combined algorithm reduces graph size to just 6% of the original graphs on average, which leads to speedups of 5.5× on average.

All our new algorithms are designed to replace existing to-date high-precision algorithms, so that the output qualities of the analyses are not sacrificed. All performance evaluations of the new algorithms are based on comparing with those existing algorithms.

1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 describes the space of algorithms for program analysis and specifies the class of algorithms this dissertation focuses on. Chapter 3 describes a taxonomy of the compiler system used by all analyses in our experiments, implementation details, and evaluation methodologies. It includes definitions and terminologies. The next chapter presents our new worklist algorithms and their results. Chapter 5 presents Relevance-Based Context Partitioning. Chapter 6 presents our new algorithms that improve IFDS analyses.

Chapter 7 reviews related work, and finally, Chapter 8 wraps up with a summary and conclusion.

Chapter 2

Algorithm Space

This dissertation proposes techniques that improve the scalability of existing precise analysis algorithms without sacrificing precision. In order to evaluate our work, it is necessary to understand the space of analysis algorithms and the subset of precise algorithms in this space. This in turn also requires defining basic terminologies such as analysis precision. This chapter begins with defining the basic terms. Section 2.2 presents the space of analysis algorithms by categorizing their different properties. Section 2.3 specifies a subset in this space that is the focus of this dissertation. It also briefly compares the complexity of existing algorithms in this subset with that of our new algorithms.

2.1 Basic Terms

This section defines various basic terms such as lattice, precision, and soundness, which are useful later when we describe different analytic properties. There are two aspects of precision, which we explain in Sections 2.1.2 and 2.1.3. Section 2.1.4 then defines soundness.

2.1.1 Analysis Algorithm and Analysis Problem

An *analysis* is an algorithm applied to a given abstraction of a program, with the objective of statically gathering information regarding the run-time behavior of the program.

An *analysis problem* is the result sought from an analysis. For example, a dead-code analysis aims to find out code fragments that are never executed. Other examples include the liveness analysis, the reaching definition analysis, and the pointer analysis.

2.1.2 Lattice and Precision

In this section, we describe what a lattice is and explain how it is used to define precision.

Dataflow analysis is a type of program analysis that uses an algebraic structure called a *lattice*. The elements in a lattice are flow values representing abstract properties of variables, expressions, or other entities in a program. Specifically, the lattice consists of a partially ordered set L and a meet operator $\sqcap$. The partial order L has a finite height with a unique upper bound $\top$ called *top* and a unique lower bound $\bot$ called *bottom*. The meet operator has the property that for any $x, y \in L$, there exists a unique $z \in L$ such that $x \sqcap y = z$. If $x \neq z$, then x is greater than z, and hence, x is higher than z in the lattice.

The meet operation $x \sqcap y$ acts as a merge operation on the abstract properties x, y. When $x \neq y$, the result of the merge represents a less *precise* value than either x or y. Thus for any two elements x, y in the partial order L, if x is greater than y (x is higher than y in the lattice), then x is more precise than y.

To give a specific example, suppose x and y represent an even and an odd

integer, respectively, and they are at the same height of a lattice; then the merge result $z = x \sqcap y$ represents any integer, which is less precise than either x or y.

An analysis problem may be solved by multiple algorithms that differ in the way flow values are computed: they can assign different values to each variable, or they can differ in the way values are propagated from one statement to the next. We say that algorithm A is more precise than algorithm B if (1) algorithm A computes more precise values than algorithm B for some non-empty set of input programs, (2) both compute same values for other input programs, and (3) algorithm A never computes less precise values.

2.1.3 Computation Granularity and Precision

The lattice used by an analysis only defines one aspect of precision. The other aspect depends on the granularities used by the analysis algorithm when computing flow values. This section briefly explains the relationship.

For instance, suppose algorithm A computes only one flow value x_0 for each variable x, while algorithm B computes one value x_i for the same variable and for each statement i ($i > 0$). Algorithm B is more precise because it distinguishes value of x for different statements, while the value x_0 computed by algorithm A is typically the less-precise merged value of all x_i ($i > 0$). This example of granularity is called flow-sensitivity, which we will discuss further in Section 2.2. Other forms of granularities are also possible, such as context-sensitive, which distinguish flow values based on procedure calling contexts.

The different possible granularities not only affect analysis precision but also complexity. They are analysis properties that define some dimensions of program analysis taxonomy. Note that these properties are not orthogonal, but we are using

the term "dimension" since it has been used in the literature. We will discuss these properties further later in Section 2.2.

For the rest of the dissertation, unless otherwise stated, the term precision shall have the meaning with respect to lattice values as defined in the previous section.

2.1.4 Soundness

The notion of precision, defined in previous sections, provides a way to compare flow values computed by an analysis. What is still lacking is a way to relate analysis results to the run-time behavior of a program. For example, an algorithm that always uses the top value $\top$ is the most precise by definition, but it does not correctly reflect run-time program behavior unless the program computes nothing. This section explains how the notion of soundness is used to fix this problem.

A program analysis is *conservative* or *sound* if it uses estimation techniques that always err on the safe side with respect to the program's concrete semantics. Depending on the problem, a sound analysis may over- or underestimate the solution. For example:

- In liveness analysis, if a variable is live but the analysis reports that it is not, then the analysis is not sound. Otherwise the analysis is sound. This is true even if a variable is not live but the analysis reports that it is, in which case the analysis *overestimates* the set of live variables.

- In available-expression analysis, if an expression is not available but the analysis reports that it is, then the analysis is not sound. Otherwise, the analysis is sound. This is true even if an expression is available but the analysis does not report that it is, in which case the analysis *underestimates* the set of available

expressions.

- In an analysis that searches for security vulnerabilities, a *false negative* is an error that the analysis fails to find. A *false positive* is an error that the analysis finds but actually does not exist. A sound analysis may overestimate the set of errors in a program, so the result may contain some false positives but never any false negatives.

To ensure soundness, the analysis often has to use less-precise flow values using a lattice (defined previously in Section 2.1.2). In the extreme, an analysis that by default always uses the bottom value $\perp$ is sound and fast but is also probably too imprecise to be useful. Therefore, it is important to design sound algorithms that are as precise as computationally feasible.

For the rest of this dissertation, we will focus only on sound analysis and ignore all unsound algorithms. Consequently, we will only be comparing algorithm precision in all our discussions.

2.2 Properties of Program Analyses

The space of analysis algorithms can be described by the dimensions of a program analysis taxonomy. The dimensions are not orthogonal. Each algorithm in this space can be identified using a value with respect to each dimension, and these values constitute the properties of the algorithm.

It is beyond the scope of this dissertation to rigorously identify all analysis dimensions. Instead, we identify only the subset relevant to our work. Table 2.1 summarizes the dimensions we identified, their possible values, and examples of previous work. The subsequent subsections will describe in detail each property,

including a motivation for the need of the property, and how the possible values affect precision and cost.

Analysis Dimension	**Examples of Possible Values**
Scope of analysis	Intraprocedural [99, 118, 3, 48]; Interprocedural [98, 93, 49]
Flow-sensitivity	Flow-insensitive [140, 84, 33, 85, 7]; Flow-sensitive [16, 147, 24, 17]; Subset [54]; Local; Region-based
Context-sensitivity	Context-insensitive [137, 138, 108, 58, 79, 24, 84] Context-sensitive [136, 80, 153, 121, 47, 21] Bottom-up; Top-down; Subset [54]; Context merge; Object sensitive [95]
Memory model-array	Singleton; Complex [91, 14, 45]
Memory model-field	Field-sensitive; Field-insensitive; Field-based [110]
Memory model-heap	Singleton; Allocation site; Allocation context
Memory model-heap layout	Shape analysis [81, 68, 20, 64, 44, 43, 134, 151, 122, 26, 72, 88, 57, 123, 124]

Table 2.1: Analysis properties and examples of their possible values. For dimensions such as flow-sensitivity and context-sensitivity, the example values are not exhaustive because these dimensions have many possible values.

2.2.1 Scope of Analysis

There are two types of analyses that define the scope of analysis: intra- and interprocedural. An intraprocedural analysis tracks properties local to a procedure, so that each procedure is analyzed in isolation. Assuming no pointers and reference parameters, then computing liveness of local variables is one such analysis. Other examples include dead code analysis, available expression analysis, partial redundancy elimination, and register allocation. Intraprocedural analysis is simple because the analysis at the procedure does not grow with program size. It is also sufficiently precise for analysis problems such as the liveness analysis and other

examples mentioned above, in the sense that an interprocedural analysis will not improve their precisions.

On the other hand, programs with pointers and reference parameters allow data structures to span across procedures. Intraprocedural analysis then has to make conservative assumptions about effects of procedure calls on flow values. Consequently, the analysis becomes very imprecise.

To avoid making such conservative assumptions, these programs require interprocedural analysis that will apply the analysis results of a procedure to the analysis of another. Generally, interprocedural analysis tends to be more expensive; for example, programs with large call graphs and many recursive calls cause the analysis to take a longer time to reach a fix-point.

2.2.2 Flow-Sensitivity

Flow-sensitivity is a property of program analysis concerned with whether control-flow plays a role in the analysis. This section explains the differences between a flow-sensitive and flow-insensitive analysis, and explores other options.

An analysis computes flow values for variables, expressions, or some well-defined entities; without loss of generality, we assume that the analysis is computing solutions for variables. For each variable, the analysis can either compute the statements for which a value holds, or it can compute a value that holds for the entire program. These two choices are known as *flow-sensitive* and *flow-insensitive* analyses, respectively. They and other options are granularities of details called *levels of flow-sensitivity*. These details have direct impact on the precision and complexity of analysis. The following example illustrates the difference:

```
x = 5;
y = x;
x = 8;
```

A flow-sensitive analysis will determine that the variable `x` has value 5 at the second statement and a value 8 at the last statement. Consequently, it also determines that the variable `y` has value 5 after the second statement. On the other hand, a flow-insensitive analysis cannot determine a unique value for `x` so that both `x` and `y` will have an unknown value at all statements.

Flow-sensitive analyses can be further divided into two types, depending on how the dependences between flow values are represented. For example, dependences between two flow values x, y can be represented as pairs of attributes in the form $[(x = a \wedge y = b)$ or $(x = c \wedge y = d)]$. Jones and Muchnick refer to this type of analysis as *relational-attribute analysis* [73]. In contrast, the analysis may choose to ignore the dependences and represent the values as $[(x = a \vee x = c)$ and $(y = b \vee y = d)]$; this type of analysis is called *independent-attribute analysis*. The second form trades-off precision for efficiency

Although a flow-insensitive analysis is generally less precise, it is sufficient for certain applications. For example, the experiments by Hind and Pioli [67] and by Stocks et al. [141] suggest that a flow-sensitive analysis improves little or no precision on some programs compared to a flow-insensitive analysis, because the flow values are seldom changed. On the other hand, evidence from other previous work [145, 54] shows that flow-sensitive analysis is crucial to achieve precise results.

In terms of complexity, flow-sensitive analysis is computationally more expensive. A relational-attribute analysis can have exponential complexity [102] due to the possibly large set of dependent attributes, while a flow-insensitive analysis can be as fast as almost linear [140]. For example, between a flow-sensitive and

a flow-insensitive pointer analysis, currently only the latter can scale to programs with millions of lines of code [60].

Fortunately, flow-sensitivity is also a spectrum of which the two ends are the two options described above. A non-exhaustive list of levels of granularities includes:

- Flow-sensitive (FS): a flow value is computed for every variable in each statement.
- Flow-insensitive (FI): a single flow value is computed for every variable for the entire program.
- Subset: flow values for a subset of variables are computed flow-sensitively, while those of the remaining variables are computed flow-insensitively.
- Local: flow values for all variables are computed flow-sensitively within a segment of code, but they are computed flow-insensitively outside the segment. For example, for each variable x, the values $x_i, i = 1 \ldots n$ are computed for the n statements of a segment of code (one value per statement), and a single value x_0 is computed for all statements outside the segment.
- Region-based: a single flow value is computed for every variable in each region (such as a procedure), but different value can be computed for different regions. For example, for each variable x, a single value x_p is computed for each procedure p.

Note that except for FS and FI (which are the two ends of the spectrum), other options are not necessarily mutually exclusive. For example, Subset and Local can be combined so that flow values for a subset of variables are computed flow-sensitively within a segment of code, whereas flow values for other variables or for outside the segment are computed flow-insensitively.

This flexibility in the spectrum proves to be important in designing fast and precise algorithms. For example, the Client-Driven algorithm [54] can determine the subset of variables that are essential to solving a given problem, so that they are analyzed flow-sensitively while other variables are analyzed flow-insensitively.

2.2.3 Context-Sensitivity

This section explains the differences between precision and complexity among the different types of context-sensitivity.

Context-sensitivity is another property of program analysis concerning the use of calling contexts of a procedure to compute solutions. The use of calling contexts helps to improve analysis precision in two possible ways: the bottom-up context-sensitivity receives the attention of most previous work, while the top-down context-sensitivity is discussed less frequently. These two options and others are occasionally referred to as the *levels of context-sensitivity*. We will now explain their differences.

Bottom-up context-sensitivity focuses on eliminating propagations along unrealizable paths. A path is *realizable* if at every return-edge, the callee returns to the caller that invokes it. One way to ensure only realizable paths is to independently analyze each procedure for each of its calling contexts, where a context is the chain of call sites starting from a call in the root procedure. This method has exponential costs. Many methods have been proposed to reduce the cost, such as the Reps et al.'s algorithm [121], the use of Binary Decision Diagrams [150, 157], and the use of Partial Transfer Functions [153].

Top-down context-sensitivity aims at preventing the *context-loss problem*, which is imprecision in the callee due to the merging of flow values from different

calls. Consider the following example:

```
main() {
  a=2; b=3;
  foo(a,b);
  foo(b,a);
}
foo(x,y) {
  z=x+y;
  printf("%d", z);
}
```

Based on the two calls to `foo`, the parameters `x, y` can have values 2 and 3, so that the variable `z` can hold values 4, 5, or 6. However, if the two calls are analyzed separately, then `z` can only have value 5. One way to prevent the context-loss problem is to analyze each call separately. Previous work, such as Reps et al.'s algorithm, cannot prevent such a problem.

In contrast, a context-insensitive analysis can propagate flow information along unrealizable paths, and it also does not prevent the context-loss problem. It is therefore less precise, but it is also less expensive in space and time, because it does not distinguish calling contexts when analyzing a procedure.

There are two reasons why a context-insensitive analysis may be chosen instead of its context-sensitive counterpart. First, context-sensitive analysis is expensive due to the exponential number of possible contexts, while in a context-insensitive analysis, the number of contexts is the same as the number of procedures. Hence, a context-insensitive analysis is more feasible for large programs. Second, a context-insensitive analysis provides sufficient precision for certain applications, such as intraprocedural versions of liveness analysis, reaching definition analysis, and constant propagation.

Nevertheless, context-sensitive analysis is sometimes preferred, as it evidently

improves analysis precision [145, 156, 54, 83]. Other results also suggest a context-sensitive analysis is not necessarily prohibitively expensive [141].

Like flow-sensitivity, context-sensitivity is also a spectrum, so that there are other levels of context-sensitivity besides the options described above. A non-exhaustive list of context-sensitivities includes:

- Fully context-sensitive (CS): every invocation context is analyzed separately.
- Context-insensitive (CI): every invocation of a procedure is analyzed using same context information.
- Bottom-up: excludes unrealizable paths in the propagation of flow values.
- Top-down: analyzes invocation contexts in such a way that prevents context-loss problem.
- Subset: every invocation of a subset of procedures is analyzed separately, while other procedures are analyzed context-insensitively.
- Context merge: partition invocation contexts of each procedure so that contexts in each group are analyzed using same context information. This technique is the focus of our Relevance-Based Context-Sensitive analysis, which we will describe in Chapter 5.
- Object sensitive: in object-oriented languages, each method is analyzed separately for the different receiver and parameter types [95].

Note that except for CS and CI (which are the two ends of the spectrum), other options are not necessarily mutually exclusive. For example, Subset, Bottom-up, and Top-down can be easily combined. Such flexibility proves to be important in

designing fast and precise algorithms. For example, the Client-Driven algorithm [54] uses the Subset context-sensitivity to improve scalability of pointer analysis while maintaining a high level of precision.

2.2.4 Memory Model

In order to approximate the run-time behavior of a program, static program analysis needs to estimate the layout and/or contents of memory locations, including that of heap-allocated variables. For example, it has to find out the possible values of pointer and reference variables in order to compute liveness analysis or reaching definitions. This section explains the need for a memory model and how the various options influence analysis precision and cost.

The memory locations used by a program consist of local variables, global variables, heap variables, and derivative locations such as array elements and fields of aggregates. At runtime, a program can request an unlimited number of memory locations (bounded only by available virtual memory). Since resolution of all memory references requires execution, it must use an abstraction model.

We assume the model uses an abstract object to represent one or more variables. The level of detail in this model influence the precision of the analysis. For example, if a single abstract is used to represent all heap variables, then assignment to any pointer dereference `*p` will modify the value of another pointer `*q`. A more precise memory model will discover that the two pointers have disjoint points-to sets, hence avoiding the superfluous update.

Since programs use different types of variables, the memory model has different options that affect these variables:

- Arrays:

The simplest way to represent array elements is to use a single abstract object (Singleton) to represent all elements of an array. In this model, operations on any element are assumed to apply to all elements of the same array. A more precise model is needed to parallelize a program or to verify correctness, such as ensuring array accesses are in bounds. Such model uses additional techniques to compute the array indices or properties of array elements [91, 14, 45]. Previous work on these techniques did not discuss their complexities.

- Field-Sensitivity:

  Modern programming languages offer aggregate data structures, such as C's `struct` and `union`. There are at least three options on how to deal with them. A field-sensitive analysis treats each field of a `struct` as if it is a separate variable. Such details are useful for many programs whose aggregate fields carry useful information for the analysis. A field-insensitive analysis discards field information so that a single variable is used to represent the aggregate. This lack of details may be appropriate if, say, the analysis only needs to focus on scalar variables. A field-based analysis [110] treats all instances of a field as a single variable.

  These options can be useful for different purpose. For example, a thread escape analysis [154] can use a hybrid approach that focuses on fields of the `Runnable` type in Java and ignore fields of other types.

  Among these three options, field-sensitive analysis is the most precise but is also the most expensive in space. Previous work reports faster analysis time when field-insensitivity is used [110], though no complexity comparison is reported. Interestingly, based on our own experience, a field-insensitive analysis can be either more or less expensive in time. It is more expensive

when it artificially introduces dependences between fields so that the analysis takes longer to converge to a fix-point.

For weakly-typed languages where there is no explicit type declaration for variables, Ramalingam et al. [115] present a fast technique to discover aggregate structures.

- Heap Model:

  In the simplest heap model, a single abstract object (Singleton) is used to represent all heap-allocation variables. This simple model is suitable for analyses that are uninterested in any property of any heap variables. A more precise model is to use an abstract object for each allocation statement (Allocation site) in the program. This model can be improved further in a context-sensitive analysis: if the procedure containing an allocation statement is analyzed context-sensitively, then the new model uses one abstract object per context for that allocation statement (Allocation context). This high level of detail is sometimes necessary in order for the analysis to be precise [54, 105]. For example, after detailed examination of the behavior of an analysis that searches for security vulnerabilities [54], we conclude that the precise heap model is necessary to avoid false positives.

- Heap Layout:

  For programs that extensively use heap-allocated data structures such as linked lists or trees, the above models are not precise enough for applications such as proving a tree is acyclic. These applications need a model such as that provided by shape analysis [81, 68, 20, 64, 44, 43, 134, 151, 122, 26, 72, 88, 57, 123, 124], which is a collection of techniques used to statically determine

layout properties of heap-allocation data structures. Existing techniques vary in their applicability to different types of data structures and in their precision. Many of these techniques are also very expensive, up to doubly-exponential in analysis time.

2.3 Selected Subset in Analysis Space

The previous section has defined a space of program analyses. This section explains how and why this dissertation chooses a subset of algorithms to focus on. It also briefly explains how our new techniques improve existing algorithms.

The space of algorithms contains a set of highly-precise analyses. Previous work on analyses such as security vulnerabilities analysis provides evidence that these properties are essential to achieve high precision [54]. Program slicing, an important and useful tool for software engineering, is another analysis example where flow- and context-sensitivity are very important for precise results [144].

It is this subset in the space of algorithms that this dissertation focuses on: we aim to improve the scalability of these analyses while preserving high precision. This subset can be described by the analysis properties. We recognize that sometimes, for specific analysis problems and input programs, modifying an analysis property does not affect the precision provided by the analysis. Therefore, we focus on algorithms that provide the same level of precision as an algorithm $\mathcal{A}$ with the following properties:

- Flow-sensitivity: The analysis $\mathcal{A}$ is flow-sensitive, with no restriction on whether relational attributes are used.

- Context-sensitivity: The analysis $\mathcal{A}$ provides at least bottom-up context-

sensitivity.

- Memory model: The analysis $\mathcal{A}$ is field-sensitive but does not distinguish elements in an array. In addition, separate abstract objects are used to represent heap variables allocated by different statements and calling contexts.

2.3.1 Complexity

This dissertation aims to improve the scalability of analysis algorithms with the chosen properties. Here we briefly state the best-known complexity of previous algorithms with the chosen set of properties, and explain how our new algorithms improve existing work.

The complexity associated with flow-sensitivity depends on the analysis problem: the analysis time can be as high as exponential time [102]. The experiments in this dissertation consist of pointer analysis and analyses that search for security vulnerabilities. If a context-insensitive analysis is used, then the analysis has asymptotic complexity of PSPACE-Complete [18].

Context-sensitive analysis is also generally exponential in time due to the number of possible contexts. Recently, the use of Binary Decision Diagrams [150, 157] enables analysis to grow with the number of procedures, but the technique is not yet shown to be efficient for flow-sensitive analysis.

Complexity for the choice of memory model is as follows:

- The simple model to represent array elements adds constant time and space overhead for each array.

- The space overhead due to field-sensitive analysis is linear with respect to the number of fields per aggregate. While this translates to additional time

overhead, as mentioned before, it is unclear exactly how this model affects how fast the analysis can converge to a fix-point.

- Since the heap model is tied to the number of contexts, the space and time overhead is directly dependent on the adopted level of context-sensitivity.

Our new algorithms do not improve theoretical worst-case complexities of existing algorithms. In practice, however, our experiments show significant improvements. This is because our techniques rely on certain features found in many real programs so that the pathological worst-case scenarios seldom occur. Our three techniques and their experimental results are presented in Chapters 4–6. To summarize, our algorithms enable many analyses on large programs to complete successfully when previous techniques run out of memory. For other programs, we achieve many times speedup.

Chapter 3

System Design and Evaluation Methodologies

3.1 System Design

This chapter describes the architecture and implementation of our compiler system. We focus on describing the important unique features of our system, as well as defining terms used in the rest of the thesis. These features and terminologies are important in the evaluation of our contributions in this thesis.

The second part of this chapter describes some aspects of our evaluation methodology that are common to many of our experiments.

3.1.1 Overall System Architecture

We evaluate our ideas by performing all our experiments using the Broadway infrastructure [56, 51, 53, 52]. At the core of the compiler is an iterative dataflow analysis framework. It performs interprocedural and whole-program analysis, with features

including an integrated pointer analysis with configurable *precision* (Section 3.1.3). If a *client analysis* is specified, it is also performed together with the pointer analysis (Section 3.1.7). Table 3.1 provides a summary of the default behavior of important features. We choose these default settings because they generally help us achieve more precise results. The next few subsections provide details for other features.[1]

Feature	**Default Setting**	**Other Options**
pointers representation	points-to sets using storage shape graph	unification-based [140]
flow-sensitivity	configurable	
context-sensitivity	configurable	
assignments	uni-directional	bi-directional
memory model	see Section 3.1.4	
aggregate model	field-sensitive	field-insensitive, field-based [110]
pointer arithmetic	limited to pointers within an object	(assume program is correct)
arrays	all elements represented by a single node	
libraries	uses Broadway annotations	

Table 3.1: Some features in the analysis framework: Their default behavior and other available options.

3.1.2 Analysis Framework

We assume an input program with a *root procedure* (e.g., the procedure `main` in C programs). Each procedure is represented by a control flow graph (CFG) where each node is a basic block. Figure 3.1 shows the pseudocode that highlights the basic and essential features of the framework that drive the analysis on the input program. As we discuss our work in later chapters, we will add more details or add routines to this generic version. The code is recursive and iterative:

1. Starting with the root procedure, the routine `analyzeProc` processes each procedure by using a worklist of basic blocks, one worklist per procedure.

[1]Note that Guyer is the principle designer of the compiler, and his thesis provides many other details [52]. This chapter highlights several important compiler features relevant to the subjects of this dissertation.

```
analyzeProc(proc) {
  worklist = cfgNodes(proc) {
  while worklist ≠ ∅ do {
    blk = remove_front(worklist);
    for stmt s in blk do {
      apply ptr analysis on s;
      if ∃ client analysis
        apply client analysis on s;
      if s is CALL then
        analyzeCall(proc, s);
    }
    if changed then
      addWorklist(worklist);
  }
}
```

```
analyzeCall(proc, call) {
  proc' = callee(call);
  if proc' ∉ callstack then {
    callstack = callstack ‖ proc';
    import(proc, proc');
    if needReanalysis then
      analyzeProc(proc');
    export(proc', proc);
    callstack = callstack − proc';
  } else {
    import(proc, proc');
  }
}
```

Figure 3.1: Analysis framework: Algorithm to analyze a procedure.

2. When a block is removed from the worklist, the analysis processes the block's statements in consecutive order.

3. Each statement is processed according to the type of operations. Pointer analysis is first performed on the statement, followed by a client analysis, if any is specified (see Section 3.1.7). If any update by either analysis is detected, additional blocks are added to the worklist.

4. If the statement is a procedure call, `analyzeCall` is invoked, which prepares to analyze the called procedure:

   - A stack is maintained to keep track of the current chain of calls.
   - Flow values are imported from caller to callee.
   - `analyzeProc` is called to begin analysis on callee.
   - After analysis on callee is complete, flow values are exported back to caller.
   - The stack is updated by removing the callee.

The exception to the above steps is when recursion is detected by stack inspection, in which case analysis on the recursive call is skipped, except to export the known side effects of the callee to the current caller. This export is necessary to achieve soundness in the analysis of the recursive procedures.

5. Analysis on a procedure completes when its worklist is empty, at which point the framework collects a list of updated flow values that need to be propagated back to the caller. It then backtracks to the call site and resumes analysis on the caller.

3.1.3 Precision Dimensions

The analysis framework provides configurable precision modes. As we mentioned earlier, context-sensitivity and flow-sensitivity are two dimensions of precision.

In context-sensitive (CS) analysis, the framework analyzes each procedure independently for each of its calling contexts. On the other hand, in context-insensitive (CI) analysis, flow information is merged at procedure entries, before analysis on the procedure proceeds. After CI analysis on the procedure completes, its list of updated flow values are collected, which are then propagated back to the callers. The same happens in CS analysis, except the values are propagated only to the calling call site. Context-sensitive analysis is more precise but also more expensive in space and time.

Broadway also allows a subset of procedures to be analyzed context-sensitively, while others are analyzed context-insensitively. The flexibility gives rise to a spectrum of context-sensitivity, which is sometimes described as different *levels of context-sensitivity*. For convenience, when a procedure P is analyzed context-sensitively, we say P is a *context-sensitive procedure*; otherwise, we call P a *context-insensitive*

procedure.

In flow-sensitive (FS) analysis, the analysis computes solutions of each variable at each statement. In flow-insensitive (FI) analysis, only one solution is computed for each variable by merging all its flow values and storing only the merged value. Flow-insensitive analysis is generally less precise, but it consumes less space and time.

Broadway also allows a subset of variables to be analyzed flow-sensitively. This flexibility gives rise to different *levels of flow-sensitivity*. For convenience, if a variable is analyzed flow-sensitively, we call it a *flow-sensitive variable*; otherwise, we call it a *flow-insensitive variable*.

Broadway always performs *sound* analysis. That is, for any dataflow fact that occurs due to an actual execution of the input program (e.g., "pointer p points to variable x"), the result of the analysis must contain that fact.

3.1.4 Memory Model

This section explains how our compiler models the memory utilized by an program, i.e., the abstraction used to represent the local and global variables and objects allocated on the heap.

The system uses a data structure, called `memoryBlock` to represent program variables. The data structure is used to store the flow values computed on the variables. The memory model determines how the system creates new instances of `memoryBlock` for the variables.

Two factors in the model are the type of variables and context-sensitivity used in the analysis. There are three types of variables. The first consists of local variables and formal parameters, the second is the global variables, and the third is

the heap variables, i.e., abstract representations for memory locations allocated on the heap. The default treatment of each type of variable is as follows. For the local and formal parameters, assuming the variable is local to procedure P, then there is only one instance of `memoryBlock` if P is a context-insensitive procedure. Otherwise, when P is a context-sensitive procedure, there is one instance of `memoryBlock` for each calling context of the procedure. Each global variable has one `memoryBlock` for the entire analysis. Lastly, the heap variables are treated like a local variable, local to the procedure that contains the allocation site.

These treatments are the default behavior of the memory model, and we call it the default memory model. Other options are:

- *context-insensitive memory model*: only one instance of `memoryBlock` for each local variable or each allocation site, regardless of whether the enclosing procedure is context-sensitive.

- *context-insensitive heap model*: only one instance of `memoryBlock` for each allocation site, regardless of whether the enclosing procedure is context-sensitive.

Sometimes, a model that uses fewer `memoryBlocks` can lead to less precise results. Therefore, unless otherwise stated, we will use the default model for all experiments in this thesis.

3.1.5 Definitions and Uses

The dataflow analysis needs a way to keep track of dataflow values. The obsolete way to do this is to store all "incoming" flow values at a statement in an "IN" set. Operations in the statement cause some side effects on these flow values, and the net result is then stored in an "OUT" set, which is then propagated to the successor statements. This method is highly inefficient because the IN/OUT sets can be large

for analyses such as pointer analysis. In addition, the side effect of each statement is often small, so there is no difference in most flow values between the IN and OUT sets of a statement.

Broadway uses the following data structures to store and propagate flow values. Given a statement d containing modification to a variable, we use a data structure called "*def*", D, to represent the modification. Similarly, assume another statement u containing reference to the same variable, we use a data structure *"use"*, U, to represent the reference. If there is no other definition between the statements d and u, then a *use-def chain* is created. Each chain is a pointer, stored in the use U, pointing to the def D. Section 3.2.2 explains how use-def chains are computed.

Flow values for a variable are stored at the defs. To obtain the value at a use site, the use-def chain is used to retrieve the flow value. Broadway uses factored Static Single Assignment (SSA) [32] forms for all variables, including heap variables. SSA has well-understood properties: every use U has a unique definition D. To enforce this property, defs are also created at basic block entries and, in the case of context-insensitive analysis, at procedure entries. These definitions serve to merge flow values from different incoming paths to the blocks or procedures. These defs are also know as ϕ-functions and interprocedural ϕ-functions, respectively. For example, assuming a basic block has k predecessors, each ϕ-function has the form

$$D \quad = \quad \phi(U_1, U_2, \ldots, U_k) \tag{3.1}$$

where D is the def, and each U_i is a ϕ-use, whose purpose is to store the pointer to the def coming in from the ith predecessor.

Since we only compute a single (merged) flow value for each flow-insensitive variable, the system creates only one def and one use for the variable, with a use-def

chain between the two. Any update to the variable leads to update of this unique def, so that any reference to the variable will lead to the flow value stored in this def.

Sometimes it is also useful to compute def-use chains. Such chains are pointers to uses, and they are stored in each def. Chapter 4 describes how def-use chains develop efficient worklist algorithms.

3.1.6 Pointer Analysis

Broadway contains an integrated pointer analysis. The different levels of context- and flow-sensitivity gives rise to a spectrum of pointer analysis algorithms. These algorithms differ in time and space complexity. Among all possible instances are four fixed-mode algorithms:

- *FICI*: all procedures are analyzed context-insensitively, and all variables are analyzed flow-insensitively.
- *FSCI*: all procedures are analyzed context-insensitively, and all variables are analyzed flow-sensitively.
- *FICS*: all procedures are analyzed context-sensitively, and all variables are analyzed flow-insensitively.
- *FSCS*: all procedures are analyzed context-sensitively, and all variables are analyzed flow-sensitively.

Note that the heap model can be varied independently of these precision modes.

Generally, the *precision mode*, or *precision policy*, of the algorithm directly affects the quality, or precision, of the output. Given two sound pointer analyses with the same memory model (see Section 3.1.4), if the results of algorithm A contains

larger points-to sets than that computed by algorithm B, we say the output of algorithm B is more precise, and hence algorithm B is more precise. For example, among the four fix-mode pointer analyses, generally the FICI pointer analysis is the most efficient but is also the least precise, while the FSCS pointer analysis is the most expensive and most precise.

3.1.7 Client Dataflow Analysis

If a client analysis is specified, Broadway will perform the pointer analysis and client analysis side by side, as described in Section 3.1.2. We call it a client analysis because it requires solutions to the pointer analysis.

Broadway has a few built-in client analyses, such as constant propagation and liveness analysis. It also provides a lightweight annotation language [55] that can be used to define new dataflow analyses and actions, such as transforming the program or reporting an error [53].

These actions are triggered by the results of the analysis. Specifically, the client analysis must define a set of *queries* at specific locations in the program (e.g., at all calls to a certain library routine). Each query checks if a specific dataflow fact, or a combination of dataflow facts, is valid at that location. When the query is evaluated to be true, the corresponding action or actions are triggered.

Section 3.3.2 will describe some client dataflow analyses that we use in our study.

3.1.8 Precision and Accuracy

The quality of the output produced by the client analysis depends on the output quality of the pointer analysis, as well as the precision mode used by the client

analysis. By default, unless otherwise stated, we will assume the client analysis uses the same precision policy as the pointer analysis.

Section 1.2.1 defines the meaning of accuracy for combinations of analyses, such as a pointer analysis and a client analysis. For example, for the same analysis problem that finds security vulnerabilities in programs containing pointers, a combined analysis that reports no errors in a program is more accurate than another that reports multiple errors (assuming that both are sound analyses).

In a combined analysis, a change in precision policy does not always have the same visible effect on the analysis accuracy for all input programs. For example, compared to using a FICI pointer analysis, accuracy might be improved if a FSCI pointer analysis is used on one program but not another. In other words, any effect on the accuracy depends on whether the right precision level of pointer and client analyses is used for a given specific program. This observation leads to two outcomes:

- When two analysis algorithms with different precisions yield no difference in the client analysis output accuracy, then the algorithm with lower time and space complexity is desired.

- Automatically searching for the "optimal" precision policy is the motivation behind the Client-Driven analysis [54], as well as the *Relevance-Based Context-Sensitive* algorithm we developed, which will be presented later in Chapter 5.

3.1.9 Library Routines

Broadway does not analyze the source code of library routines. Instead, it relies on the same built-in annotation language [55] mentioned earlier to model the behaviors of these routines. At each call to a library routine, Broadway interprets the

annotations to update the state of the analyses, such as applying the side effects of the routines at the call sites. The advantage of using the annotations are (1) we do not have to deal with the library source code, and (2) we can accurately model the behavior of client analyses in these routines.

For example, consider a call `strcpy(dst,src)`: the pointer analysis will create uses for each variable that `src` points to, and create defs for each variable that `dst` points to; if a client analysis is present, then its associated property value is passed on from the source strings `*src` to the target strings `*dst`.

3.2 Implementations

This section briefly summarizes some data structures used internally in the Broadway compiler. At the end of the section, we also provide a list of known limitations.

3.2.1 Program Representation

Broadway uses a data structure called a *location tree* to represent program locations. The tree is constructed from three types of nodes:

- Statement locations represent statements in the program.
- Basic block locations represent basic blocks; one of their main purposes is to represent control merge points in the procedures. The children of a basic block location are the statement locations corresponding to the statements in the block.
- Procedure locations represent procedures; these locations also serve to represent interprocedural merge points at procedure entries. The children of a

procedure location are the basic block locations representing the blocks in the procedure.

The above tree data structure can represent a procedure and is adequate for context-insensitive analysis: a forest of trees is used to represent the program, one tree per procedure. To support context-sensitive analysis, we need two changes: (1) allow each statement location to have a procedure location as a child, and (2) allow multiple instances of procedure locations per procedure. The idea is that since we want to analyze each procedure independently for each of its calling contexts, we use a separate procedure location instance for each context. The parent of each procedure location is the statement location that represents the calling context. From each clone of the procedure location is the location subtree consisting of clones of the basic block locations and statement locations. Assuming there is no recursion, the entire program is now represented by a single location tree, with the root procedure as the root of the tree.

The data structure is flexible enough to support other levels of context-sensitivity too. For example, when only a subset of procedures is context-sensitive, we always clone the procedure location (and subtree) for each context-sensitive procedure, and we allow only one copy of this procedure location (and subtree) for each context-insensitive procedure.

3.2.2 Dominance Relation and Reaching Definitions

By definition, a statement s dominates another statement t in the same procedure if every path from the procedure entry to t must pass through s. Dominance relations are used in the computation of use-def chains (see Section 3.1.5), which must satisfy the following properties (def-use chains share the same properties):

- Because we are using SSA form, for every use-def chain $\langle u, d \rangle$, the def d must dominate the use u.

- There must be no other def between the d, u pair; i.e., there exists no other def that dominates u and is itself dominated by d. In other words, d is unique if it exists. We say that d is the *nearest reaching definition* of u.

To compute the nearest reaching definition for a given use, Broadway first needs an efficient representation of the dominance relation, and it needs another data structure that facilitates fast lookup: from among a list of definitions, which is the nearest reaching definition for a given use? These two data structures are explained in the remainder of this section.

Dominator Tree

Broadway first builds a dominator tree [87, 4], where the nodes are the CFG basic block nodes. A node m in the tree is an ancestor to node n in the tree if and only if m dominates n. Figure 3.2(a,b) shows two examples of dominator trees for two procedures.

On each dominator tree, we can assign to each node n a preorder number and a postorder number, denoted $min(n)$ and $max(n)$, respectively. These numbers are assigned by performing a depth-first traversal and incrementing a counter each time we move either up or down the tree. Figure 3.2(b) shows the numbers assigned to the example dominator trees. The number to the left of each node is its min number, and the number to the right of each node is its max number. These numbers turn out to be efficient representations for dominance relation. Specifically, given two distinct statements m and n, m dominates n, denoted by $DOM(m, n)$, if and only

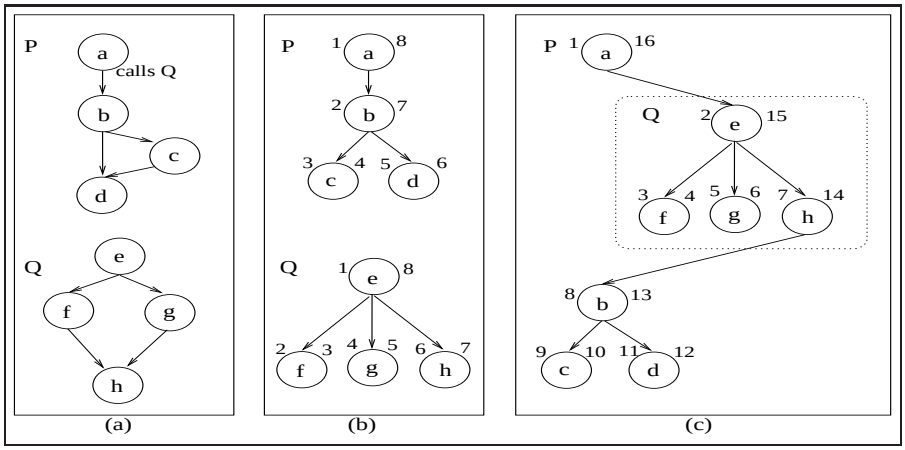


Figure 3.2: Node numbering on a dominator tree. (a) Control flow graphs for two procedures and (b) their respective dominator trees, with node numbering. (c) Node numbering on the expanded dominator tree: node a makes a call to procedure `Q`.

if the following condition holds:

$$DOM(m,n) \quad \text{iff} \quad min(m) < min(n) \ \wedge max(m) > max(n). \tag{3.2}$$

In reality, the numbering on dominator trees is inadequate for Broadway analysis, for two reasons. First, we are interested in dominance relations, not just between basic blocks, but also between statements. Second, the dominator trees cannot capture interprocedural dominance relations.

To overcome these problems, Broadway uses *expanded dominator trees*. Each such tree is like a normal dominator tree, except that the nodes are the nodes from a location tree (see Section 3.2.1) that represent statements, basic blocks, and procedures.

In addition, just as a location tree can span procedure boundaries, an expanded dominator tree can also have nodes belonging to different procedures. The expanded tree can be built by logically composing the dominator trees of the dif-

ferent procedures. Specifically, if a procedure `P` calls procedure `Q` at a call site, and `Q` is to be analyzed context-sensitively, then a clone of the dominator tree of `Q` is attached to that of `P` so that (1) the node for the call site becomes the parent of the root node in the dominator tree of `Q`, (2) the exit node in dominator tree of `Q` becomes the parent of the return node for the call site, and (3) the edge between the call and return nodes of the call site is removed.

The same node numbering algorithm is then used to label the nodes in the expanded tree. Figure 3.2(c) shows the resulting expanded tree when the node a contains a call to procedure `Q`. After the nodes are numbered, the Invariant (3.2) also applies to the nodes in the expanded tree.

Broadway builds the location tree and numbers the dominator trees on the fly during analysis: as the analyzer is processing statements, new nodes are created if they are not yet created, and the *min* number is assigned using a counter. The *max* number is assigned by a postorder portion of the depth-first traversal, whenever analysis reaches the end of a basic block that does not dominate any others.

Nearest Reaching Definition

Using the dominator tree, Broadway stores all definitions of a variable in a list satisfying the following invariant [152]: let d_i be the i^{th} definition in the list, and let n_i be the statement associated with d_i; then the *min* of the statements are stored in decreasing order:

$$\begin{aligned} & \forall i < j: \quad min(n_i) > min(n_j) \\ \Rightarrow \quad & \forall i < j: \quad \neg DOM(n_i, n_j) \end{aligned} \tag{3.3}$$

To find the nearest reaching definition that strictly dominates a statement

m, we perform the following steps:

1. Perform a binary search to obtain the minimum i such that d_i in the list satisfies $min(n_i) < min(m)$.

2. Perform a linear scan from i to the end of the list to find the first d_i such that $max(n_i) > max(m)$.

For the resulting d_i, $DOM(n_i, m)$. Without the binary search, a linear search alone (starting from $i = 1$) can still find the correct result if the $DOM(n_i, m)$ test is used, because by Invariant (3.3), the first n_i that dominates m must also be the nearest. The first step, therefore, is to help speed up the search.

Whenever a new def is created, the list is updated by inserting the new def into the list. The above two steps are used to determine where in the list to insert the new entry, thereby preserving the invariant.

3.3 Evaluation Methodologies

This thesis contains many experiments with diverse goals. This section describes common elements in these experiments: the metrics, analysis problems, benchmark programs, and hardware platform.

3.3.1 Metrics

The goal of this dissertation is to improve the performance of dataflow analysis by reducing redundancy. We focus on three primary metrics to measure performance: analysis time, memory usage, and accuracy. Measuring client accuracy has the advantage of helping us determine if the pointer analysis is effective with respect to the dataflow problem.

Other static metrics, such as points-to sizes or alias sets, are less effective in providing clear comparisons. For example, the sizes of points-to sets of two analyses may be the same even when they disambiguate each pointer dereference to different sets and have different effects on the client analysis. Therefore, a better approach is to focus on measuring the effects on the client analysis.

Among our three components of the dissertation, none have the ability to improve accuracy over other existing algorithms. Therefore, analysis time becomes the focus, after we verify that the new algorithms do not degrade accuracy.

3.3.2 Client Analysis Problems

We use error-checking tools as clients to a pointer analysis and use the reported errors as the metric to measure accuracy: when two analyses report different numbers of errors, the one with the smaller number is more accurate. This interpretation is possible because all our analyses are *sound* but *incomplete*. They are sound because if a program contains an error, then the client analysis will report it. They are incomplete because they can sometimes report *false positives*, or false alarms, which are reported errors even though the program does not actually contain them. Therefore, an analysis with fewer false positives is more accurate.

We will use the five error detection analysis problems used previously to evaluate the Broadway compiler [54]. These are realistic, non-trivial errors that could cause significant damage. They are *typestate* analysis problems [142], that involve tracking properties of data structures that cross procedure boundaries, and they therefore require precise interprocedural analysis. Automatic detection of these errors is, therefore, an important and challenging problem. The five errors we detect are:

- File-Access: make sure files are open when accessed;
- Format String Vulnerability (FSV): make sure format strings do not contain untrusted data [104];
- Remote-FSV: enhanced version of FSV that determines when vulnerabilities are remotely exploitable;
- Remote Access Vulnerability (Remote-Access): make sure a remote hacker cannot control sensitive procedures, such as execution of other programs;
- FTP-Behavior: make sure a program cannot be tricked into reaching and returning contents of arbitrary local files.

Each of these problems defines a set of queries, where a query is a system or library call in the program that may be the manifestation of an error (such as accessing a closed file). The dataflow analysis will evaluate each query, which will yield either a true or false answer; when the answer is positive, the client analysis will report an error.

One more note about how we count reported errors: for context-sensitive analyses, a query may be evaluated many times using different contexts. If a query is evaluated to be positive under different contexts, the system will generate reports for these errors separately, but it will only count them as one error. This is necessary because if we count them as multiple errors, then a context-sensitive analysis will easily count more errors than a context-insensitive analysis, leading to the misleading impression that the context-insensitive analysis is more accurate.

3.3.3 Benchmarks

We use 19 open-source C programs for our experiments, which range from 2K–69K lines of code, as listed in Table 3.2. These are the same set of benchmark programs used by Guyer [52], except we add the largest program `sendmail`. These benchmarks are realistic programs, some of which are system tools or daemons. Some of these programs are known to contain certain errors. We use these moderately-large programs to "stress-test" the scalability of our analyses, while at the same time verify no precision loss due to our new algorithms.

These 19 programs and five analysis problems give us 95 combinations of program and problem cases. The number of queries has a significant impact on the performance, more so on some classes of algorithms, such as the Demand IFDS algorithm (see Chapter 6). For this reason, we also tabulate the number of queries, shown in Table 3.3.

3.3.4 Platform

The experiments are conducted on a 1.7GHz Pentium 4 with 2GB of main memory, running Linux 2.6.17.4. Our system is implemented entirely in C++ and compiled using the GNU g++ compiler version 3.3.4.

Program	Description	LOC	Procs	Stmts	CFG nodes	Call sites
stunnel 3.8	Secure TCP wrapper	2K	42	2,067	511	417
pfingerd 0.7.8	Finger daemon	5K	47	3,593	899	545
muh 2.05c	IRC proxy	5K	84	4,711	1,173	666
muh 2.05d	IRC proxy	5K	84	4,921	1,245	669
pureftpd 1.0.15	FTP server	13K	116	10,772	2,537	1,180
crond (fcron-2.9.3)	cron daemon	9K	100	11,252	2,426	1,249
apache 1.3.12 (core only)	Web server	30K	313	16,717	3,933	1,727
make 3.75	make	21K	167	18,787	4,629	1,855
BlackHole 1.0.9	E-mail filter	12K	71	20,227	4,910	2,850
openssh client 3.5p1	Secure shell client	38K	441	21,601	5,084	4,504
wu-ftpd 2.6.0	FTP server	21K	183	22,185	5,377	2,869
wu-ftpd 2.6.2	FTP server	22K	205	23,130	5,629	2,946
named (BIND 4.9.4)	DNS server	26K	210	23,405	5,741	2,194
privoxy 3.0.0	Web server proxy	27K	223	23,615	5,765	3,364
openssh daemon 3.5p1	Secure shell server	50K	601	28,877	6,993	5,415
cfengine 1.5.4	System admin tool	34K	421	38,232	10,201	6,235
sqlite 2.7.6	SQL database	36K	386	43,489	10,529	3,787
nn 6.5.6	News reader	36K	493	47,058	11,739	4,104
sendmail 8.11.6	Mail server	69K	416	67,773	15,153	7,573

Table 3.2: Properties of the benchmarks. Lines of code (LOC) are given before preprocessing. Statistics do not include those inside library routines. The programs are sorted according to the number of CFG nodes. Table 3.3 shows more properties regarding the benchmarks.

Program	File Access	FSV	Remote FSV	Remote Access	FTP Behavior
stunnel 3.8	11	11	11	3	8
pfingerd 0.7.8	104	102	102	7	94
muh 2.05c	16	15	15	8	12
muh 2.05d	16	15	15	9	12
pureftpd 1.0.15	14	18	18	46	9
crond (fcron-2.9.3)	225	198	198	27	216
apache 1.3.12 (core only)	113	105	105	11	109
make 3.75	189	124	124	15	180
BlackHole 1.0.9	476	534	534	106	436
openssh client 3.5p1	87	121	121	4	69
wu-ftpd 2.6.0	54	349	349	50	19
wu-ftpd 2.6.2	54	348	348	53	19
named (BIND 4.9.4)	460	597	597	13	446
privoxy 3.0.0	20	110	110	14	15
openssh daemon 3.5p1	78	98	98	14	59
cfengine 1.5.4	982	1503	1503	101	970
sqlite 2.7.6	77	115	115	5	73
nn 6.5.6	201	157	157	46	132
sendmail 8.11.6	1003	1134	1134	55	952

Table 3.3: Properties of the benchmarks, continued from Table 3.3. The table shows the number of queries in the programs used by the five analysis problems (see section 3.3.2).

Chapter 4

Efficient Flow-Sensitive Interprocedural Dataflow Analysis

4.1 Introduction

Flow-sensitive analysis is important for problems such as program slicing [144] and error checking [42, 54]. While recent work with BDD's has produced efficient algorithms for solving a variety of flow-insensitive analyses [150, 157], these techniques have not translated to flow-sensitive problems. Other techniques, such as demand interprocedural analysis [70], do not apply to pointer analysis. Thus, the most general technique for solving flow-sensitive problems continues to be iterative dataflow analysis. Existing iterative dataflow analysis algorithms work well within a single procedure, but they scale poorly to interprocedural analysis, because they spend too much time unnecessarily reanalyzing parts of the program.

At issue is the manner in which worklists are managed, which can greatly affect the amount of work—and unimportant computations—performed during each iteration. The most basic algorithm maintains a worklist of basic blocks for each procedure. Basic blocks are repeatedly removed from the worklist and applied with flow functions. If any changes to the flow values occur when a block is processed, we say the block visit is useful, all reachable blocks are added to the worklist. This basic algorithm becomes extremely inefficient when used for interprocedural analysis: when reanalyzing a block containing procedure calls, the algorithm may revisit all of the called procedures, even though many of them may not require reanalysis. In other words, the same computations are repeated needlessly. Extensions to this basic algorithm, such as Hind and Pioli's priority queue approach [67], which considers the structure of the control flow, also suffer from this problem of useless work. For example, when the Hind and Pioli algorithm is applied to the `nn` program (approximately 36K lines of C), we find that only 3% of the basic block visits are useful—the others do not update any flow values.

```
p = &x;
while (cond) {
  y = x;
  *p = 7;
  p = &z;
}
y = z;
```

Figure 4.1: A loop example.

In this chapter, we present a new algorithm for interprocedural iterative dataflow analysis that is significantly more efficient than previous algorithms. The algorithm exploits data dependences to reduce the number of times that blocks are revisited. The algorithm builds on an insight from previous work on intraprocedural algorithms: def-use chains can be used to directly identify those blocks affected

by flow value updates [147]. This goal, however, is complicated by the fact that the computation of def-use chains is itself an expensive flow-sensitive computation, particularly in the presence of pointers. The example in Figure 4.1 shows why: the first time through the loop, "*p" refers to x and, therefore, implies a def-use chain to the statement above it. The second time through the loop, however, "*p" refers to z, which implies a def-use chain to the block following the loop.

Our algorithm solves this problem by computing data dependences on the fly, along with precise pointer information, while solving the client dataflow analysis problem. The key to our approach is that as the pointer analysis computes the uses and defs of variables, it builds a network of use-def and def-use chains; the use-def chains enable fast lookup of flow values, while the def-use chains are used to narrow the scope of reanalysis when flow values change. Initially, the framework visits all basic blocks in a procedure to compute a first approximation of (1) the pointer information, (2) the data dependences, and (3) the client dataflow information. Subsequent changes in the flow values at a particular def only cause the corresponding uses to be reanalyzed. More importantly, our system incorporates new dependences into the analysis as the pointer analysis discovers them: changes in the points-to sets cause reevaluation of pointer expressions, which in turn may introduce new uses and defs and force reevaluation of the appropriate parts of the client analysis problem. Occasionally, we find pairs of basic blocks connected by large numbers of def-use chains. For these cases, we have explored a technique called *bundling*, which groups these def-use chains, so they can be efficiently treated as a single unit.

4.1.1 Contributions

This chapter makes the following contributions. First, we present a metric allowing us to compare the relative efficiency of different worklist algorithms. Second, we present a new worklist management algorithm, which significantly improves efficiency as measured by our metric. Third, we evaluate our algorithm by using it as the dataflow engine for an automated error checking tool [54]. We compare our algorithm against a state-of-the-art algorithm [67] on a large suite of open-source programs. We show that improved efficiency translates into significant improvements in analysis time. For our set of 19 open-source benchmarks, our algorithm improves efficiency by an average of 500% and improves analysis time by an average of 55.8% when compared with the Hind and Pioli algorithm. The benefits of our algorithm increase with larger and more complex benchmarks. For example, the `nn` benchmark sees an order of magnitude improvement in efficiency, which translates to a 90% improvement in analysis time.

This chapter is organized as follows. Section 4.2 briefly describes the analysis framework. Section 4.3 presents our worklist algorithm, $\mathcal{DU}$, that enables sparse analysis, and a variant, $\mathcal{DU}_{loop}$, that exploits loop structures. Section 4.4 presents our empirical setup and results. We conclude in Section 4.5. We review related work later in Section 7.1.

4.2 Analysis Framework

This section reviews the background regarding our dataflow analysis framework, including details about how we efficiently compute reaching definitions using dominance information. Some additional details can be found in earlier in Chapter 3.

We assume an iterative-based whole-program flow-sensitive pointer analysis

that uses a worklist for each procedure, where each worklist maintains a list of unique CFG blocks. An alternative is a single worklist of nodes from a supergraph [103], eliminating procedure boundaries, but we believe that such a large worklist would be too expensive.

Our algorithm requires accurate def-use chains. (Recall that Section 3.1.5 describes the properties of defs and uses.) Since definitions are created on the fly during pointer analysis, we need to update chains whenever a new definition is discovered. To perform such updates efficiently, we assume SSA form for all variables, including heap objects. SSA has well-understood properties: every use u has a unique reaching definition d, and d must dominate u if u is not a *phi*-use. These properties, together with dominance relations (described below), allow us to quickly determine if a newly-discovered definition invalidates any existing def-use pairs. Finally, to merge flow values at different call sites, the system uses interprocedural ϕ-functions at procedure entries.

Our system does not use IN/OUT sets to propagate flow values [24], because their use would mandate a dense analysis: any update on a node would force all of its successors to be revisited. Our sparse analysis instead uses dominance information to efficiently retrieve flow values across use-def chains. Section 3.2.2 explains how we efficiently compute the reaching definition of any given use of a variable.

4.3 $\mathcal{DU}$: Worklist Management

Our algorithm is based on the following idea: use def-use chains to identify those blocks that may be affected by the most recent updates, thereby exploiting the sparsity of the analysis. To compute def-use chains in the presence of pointers, we present $\mathcal{DU}$, a worklist algorithm coupled with pointer analysis. This algorithm can

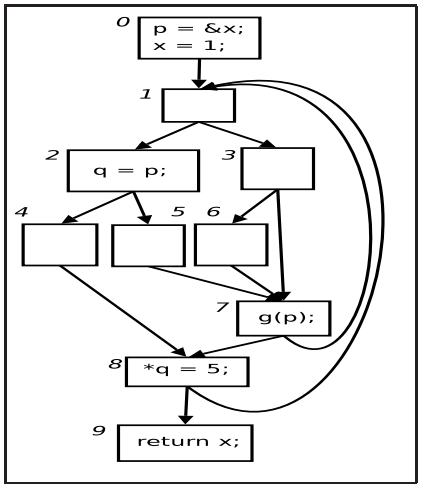


Figure 4.2: An example CFG.

exploit both intra- and interprocedural def-use chains.

To simplify our presentation, we start off with a naive, inefficient version and gradually add details to build our full version at the end of this section. We will use the program in Figure 4.2 as a running example.

4.3.1 Structure of a Worklist Algorithm

The left box of Figure 4.3 provides a high-level description of a generic worklist algorithm. It maintains a queue of CFG blocks, initially set to include all blocks in reverse post-order. The pointer analysis retrieves and analyzes one block from the worklist. The pointer analysis then identifies the set of *changes*, which is the set of variables whose flow values have been updated. The algorithm then uses a function R to compute and add to the worklist the blocks that will be revisited. The worklist may then be reordered, as we discuss in Section 4.3.8. The entire process is repeated until the worklist becomes empty. Different implementations differ in the computation of R and in the worklist reordering.

4.3.2 Naive Worklist Algorithms

The behavior of the function R is crucial to the worklist efficiency. If we do not know which blocks are affected by the changes in the last block visit, then we must conservatively return all the reachable blocks of the given block n. We refer to this version as R_{reach}, shown in the right box of Figure 4.3. Considering the example in Figure 4.2, suppose we have just revisited the loop header (block 1), where a new ϕ-function for variable x is created. R_{reach} will return blocks 2–9, for a total of 8 blocks.

4.3.3 Worklist Algorithm Using Intraprocedural Def-Use Chains

R_{reach} is easy and cheap to compute, but it adds too many blocks. We introduce R_{DU}, shown in the right box of Figure 4.3. This function iterates over the set of variable changes, retrieves their last definitions in the block, and obtains their use sites in the procedure. The blocks containing these use sites are returned and added to the worklist. For now, assume that only intraprocedural def-use chains are used. In the example of Figure 4.2, only two blocks (7 and 8) are returned by R_{DU}, so R_{DU} is more efficient than R_{reach}.

4.3.4 Dynamic Def-Use Computation

Def-use chains are computed on the fly as new pointer information is discovered, so the worklist algorithm needs to be aware that some defs may temporarily have no uses. As we shall see, the solution requires a new form of communication between the pointer analysis and the worklist algorithm.

New definitions are created at indirect assignments, function calls, and ϕ-functions. There are three cases to consider: (i) a new def leads to a new ϕ-function,

```
Initially:
WL = reverse_post_order(CFG)

Main loop:
while WL ≠ ∅ do
  block n = remove_front(WL);
  var_changes = visit_block(n);
  if var_changes ≠ ∅ then
    more = R(n, var_changes);
    merge(more, WL);
```

```
R_reach(n, var_changes) {
  return reachable_blocks(n);
}
R_DU(n, var_changes) {
  for v ∈ var_changes do
    d = last_def_of(v,n);
    for u ∈ uses(d) do
      add(block_of(u), result);
  return result;
}
```

Figure 4.3: Initial version of algorithm $\mathcal{DU}$. The function R computes what blocks need to be added to the worklist. The first version, R_{reach}, simply returns reachable blocks from block n. R_{DU} uses def-use chains to compute the blocks affected by the latest variable updates during the last block visit.

(ii) a new def resides between an existing def-use pair, and (iii) a new def temporarily has no reaching definition.

Consider case (i). SSA form requires that whenever a new definition d is created, a ϕ-function is also created at dominance frontiers. Because pointer information is not yet available, many ϕ-functions cannot be computed in advance.[1] Therefore, after d is created, the algorithm must make sure that the dominance frontiers are eventually revisited, so that the ϕ-functions can be created.

Cases (ii) and (iii) are similar because any existing use below the new def d may need to update its reaching definition. Such situations often occur in the presence of loops when a use is visited before its reaching definition is created. In the example of Figure 4.2, if a new ϕ-function for p is created at the loop header, we need to make sure that block 7 is revisited, even if the new def has no known use yet.

There are two possible solutions. The first method identifies those uses that need to be revisited by simply searching through existing def-use chains and through

[1] Short of exhaustive up-front creation.

existing uses without defs (it only needs to inspect those chains whose def is the nearest definition above d). The second solution handles (iii) as follows: whenever a use u without a reaching definition is discovered, statements above u are marked if they are merge points or if they contain indirect assignments or function calls. Later when d is discovered at one of these statements, u is revisited.

The first method can be quite expensive, while the second method does not handle case (ii). We have found that combining the two is cost effective. We use the second method on case (iii) by marking only loop headers, and we use the first method otherwise. This combination works well in practice, most likely because uses that initially have no reaching definition typically occur in loops, so marking and inspecting loop headers is sufficient. Because in practice there is usually a small, fixed number of loop headers in any procedure, the overhead due to the markings is small.

4.3.5 Bundles

One problem with R_{DU} is that it can be expensive to follow def-use chains if there are many def-use chains connecting the same two basic blocks. We can measure the extent of this problem as follows. Define C to be the number of variables whose flow values change after analyzing a given basic block. Define B to be the number of unique basic blocks containing uses of these C variables. If the ratio $r = C/(B+1)$ is large, then there is a large amount of redundancy in the dependence information represented by the def-use chains (the $+1$ term prevents division by zero). Figure 4.4 shows the maximum and average values of this ratio for the benchmarks used in our later experiments. We omit the minimums, which are all close to zero. We see that the average ratios hover between two and ten, while the maximums are two orders

of magnitude larger. One reason for the large maximums is the large number of global and heap variables defined at merge points near the end of procedures, which leads to large values of C with no further uses in the procedure ($B = 0$).

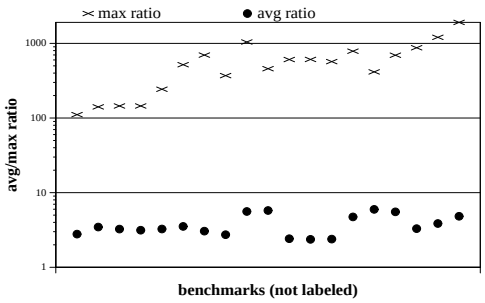


Figure 4.4: Maximum and average ratio $r = C/(B + 1)$, in log scale. The set of benchmarks is explained in Section 4.4. The high ratios indicate potential for high overhead due to def-use chains.

To handle the cases where the value of r is large, we define a *bundle* $\langle D, U \rangle$ to be the set of all def-use chains whose definitions and uses share the blocks D and U, respectively. A bundle is used as follows (see Figure 4.5). After analyzing a block n, all bundles of the form $\langle n, u \rangle$ are retrieved. R_{bundle} then iterates through these bundles: for each bundle containing a variable in the *changes* set, the u stored in the bundle is added to the worklist. When there is no bundle ($B = 0$), no overhead will be incurred even if there is a large number of changes.

```
R_bundle(n, changes) {
  bundles = set of bundles {⟨n, *⟩};
  for b ∈ bundles do
    if b contains var∈ changes then
      let b = ⟨n, u⟩;
      add u to result;
  return result;
}
```

Figure 4.5: Efficient R_{bundle} that uses bundles.

Our experimental results with an earlier implementation of our $\mathcal{DU}$ algorithm

shows that bundles are quite effective for reducing analysis time. Our results also show that bundles can consume considerable space. Given the space overhead of bundles and the bimodal distribution of r values, we use a simple heuristic to apply bundles selectively. This heuristic compares C to a threshold defined as some factor of the size of the basic block in question (as defined by the number of statements in the block).

Because we have not yet tuned the selective use of bundles for the current implementation of our worklist algorithms, the results shown later in this chapter do *not* use bundles. We expect to see improved results once this tuning has been completed.

4.3.6 Handling Interprocedural Def-Use Chains

Our system allows def-use chains to cross procedure boundaries, which typically occurs when a procedure accesses global variables or accesses variables indirectly through pointers. The framework treats these variables as if they were inputs or outputs to the procedure but not explicitly mentioned in the formal parameters. During interprocedural analysis, these def-use chains can be used to further improve worklist efficiency.

A procedure input is a variable that has a use inside a procedure and a reaching definition inside a caller. When reanalyzing a procedure due to changes to procedure inputs, we revisit only the affected use sites—which are often a subset of the procedure's blocks—because we know which inputs' flow values have changed. To identify these changed flow values, we use interprocedural ϕ-functions, which merge flow values at procedure entries. As before, these ϕ-functions are created on the fly.

A procedure output has a definition inside the procedure with some use inside a caller. The output can export a new variable, for example, a heap allocated object, or it can export a side effect on an input. We use information regarding the procedure output to help manage the worklists of the callers: if there is change in flow value in an output variable, the worklist of each caller marks the sites that need to be revisited. For this idea to work, we require a departure from the usual way worklists are used.

In many existing algorithms, analysis is performed one procedure at a time; analysis of a procedure P is started by placing all of its blocks on its worklist. To exploit interprocedural def-use chains, we no longer initialize the worklist to all blocks, except when the procedure is analyzed for the first time. Instead, a procedure P's blocks are marked to identify callers of P that change P's inputs and to identify callers of P affected by P's outputs.

In conjunction with a call graph worklist, this strategy allows us to exploit sparsity at the granularity of the procedure level. Thus, a procedure need not appear in the call graph worklist if its corresponding worklist is empty.

4.3.7 Full Version of Algorithm $\mathcal{DU}$

Figure 4.6 presents our full algorithm. It first computes the reverse post-order, *rpo*, of the procedure, which is used as the worklist if the procedure is analyzed for the first time. Otherwise, the inputs are processed, searching for those with new flow values, so that their use sites are put in the worklist. Those blocks marked for reanalysis are placed on the worklist, which is then sorted according to *rpo*.

The main loop is the same as that of Figure 4.3. After the loop, all outputs with changed flow values are gathered, and the callers' call sites are processed.

Initially:

```
if analyze proc for 1st time then
  rpo = reverse_post_order(CFG);
  WL = rpo;
  marked = ∅;
else
  WL = process_proc_inputs();
  merge(marked, WL);
  sort WL according to rpo;
```

Main loop:

```
while WL ≠ ∅ do
  n = remove_front(WL);
  changes = visit_block(n);
  if changes ≠ ∅ then
    more = R_bundle(n, changes);
    merge(more, WL);
```

Finally:

```
// worklist done; export variables.
outputs = vars to export;
if outputs ≠ ∅ then
  add(exit_block(), outputs);
```

Addition interface:

```
add(e, changes) {
  bundles = set of {⟨e, ∗⟩};
  for b ∈ bundles do
    if b contains var∈
    changes then
      let b=⟨e, u⟩;
      p=proc_of(u);
      p_marked=marked_set(p);
      add u to p_marked;
}
```

Figure 4.6: Full version of algorithm $\mathcal{DU}$, when it considers both intra- and inter-procedural def-use chains. Note that we also use bundles to export variables.

During this final stage, bundles can again be used in the `add` routine to avoid looping through all the variables in *changes*. We assume there is a definition for each output variable at the callee's exit block e. Each bundle of the form $\langle e, u \rangle$, therefore, has a use site in a caller. We can then mark this use site in the caller's *marked* set, enabling the caller to reanalyze it later.

4.3.8 Exploiting Loop Structure

By always adding blocks to the rear of the worklist, our $\mathcal{DU}$ algorithm ignores loop structure, which would seem to be a mistake because the CFG structure seems to be closely related to convergence. For example, Kam and Ullman [74] show that for certain types of dataflow analyses, convergence requires at most $d + 3$ iterations, where d is the largest number of back edges found in any cycle-free path of the CFG. Thus, it seems desirable to exploit knowledge of CFG structure when ordering the worklist, which is precisely what Hind and Pioli's algorithm does [67], although their

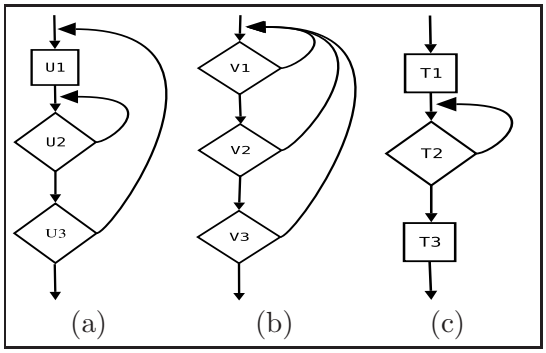


Figure 4.7: Three loop examples: a simple loop, a nested loop, and a loop with multiple back-edges.

algorithm does not distinguish different types of loops.

To understand the complexities that arise from handling different types of loops, consider two types of loops. First, in a nested loop (Figure 4.7(a)), which loop should we converge first? Second, in a loop with multiple back edges (Figure 4.7(b)), which back edge should get priority, i.e., after visiting V2 in the figure, should V1 be revisited before or after V3? After exploring many different heuristics, we evaluate a minor variant of our $\mathcal{DU}$ algorithm that ignores inner loops and uses a round-robin schedule for each loop. This algorithm, $\mathcal{DU}_{loop}$, does not try to converge an inner loop, because the loop will be revisited when trying to converge the outer loop. The round-robin schedule ensures all blocks in a loop are visited before any block is revisited.

In general, we believe that exploiting loop structure alone is not enough to yield significant improvement—we need to also account for data dependences in loops. Unfortunately, these dependences can be indirect. For example, in Figure 4.7(c), we have implicitly assumed that there is only a forward dependence from block T2 to T3. However, a backward, indirect dependence from block T3 to T2 can exist via a sequence of interprocedural def-use chains, so that a change in T3

could force T2 to be revisited. This phenomenon reduces the effectiveness of any techniques that try to exploit loop structures.

4.4 Evaluation

In this section we present our empirical setup and results.

4.4.1 Benchmarks and Metrics

Our experiments use 19 open-source C programs (see Table 3.2, page 52), which—except for `sendmail`—were used in previous work [54]. In addition to measuring analysis time and memory usage, we define metrics to evaluate the efficiency of worklist algorithms.

1. *Basic block visitation*, or *BB-visit*, is the number of times blocks are retrieved from the worklists and analyzed.

2. *Basic block changes*, or *BB-change*, is the number of basic block visitations that update some dataflow information. *BB-change* is a measure of useful work.

3. *Worklist efficiency*, $\mathcal{E}$, is the percentage of basic block visitations that are useful, i.e., the ratio *BB-change/BB-visit*.

4.4.2 Setup

We implement our worklist algorithms using the Broadway compiler system, described earlier in Chapter 3. The compiler employs an interprocedural pointer analysis that computes points-to sets for all variables. The system supports both context-sensitive (CS) and -insensitive (CI) analyses. To handle context-sensitivity

correctly, the $\mathcal{DU}$ algorithm is modified to mark a block for reanalysis under specific contexts. We will use the default heap model in Broadway; use one abstract heap object per allocation site in CI mode, and one object per allocation context in CS mode.

To evaluate our worklist algorithm, we need to choose a pointer analysis algorithm. Because the characteristics of the pointer analysis will affect the performance of our worklist algorithm, we present results for pointer analysis algorithms representing two extreme points, CI and CS.

We compare our algorithms against a priority-queue worklist. This algorithm assigns a unique priority to each block in a CFG, and uses R_{reach}. Procedure exits always have lowest priority, so loops are always converged first. This algorithm is similar to that used by Hind and Pioli [67], except we do not use IN/OUT sets. When we tried using IN/OUT sets, the compiler ran out of memory for many of the larger benchmarks.

4.4.3 Empirical Lower Bound Analysis

To explore how much room there is for further improvement, we empirically estimate a lower bound as follows. First, we execute $\mathcal{DU}$ to produce a trace of block visitations where dataflow information is updated, so the length of the trace is *BB-change*. We then re-execute the analysis, visiting blocks using the trace. Ideally, this second execution should yield 100% efficiency. In practice, we do not get 100% efficiency because, due to implementation details, the compiler has to visit additional blocks to ensure state consistency between useful visits. We measure this second execution to approximate a lower bound,[2] which on subsequent graphs is labeled as "bound."

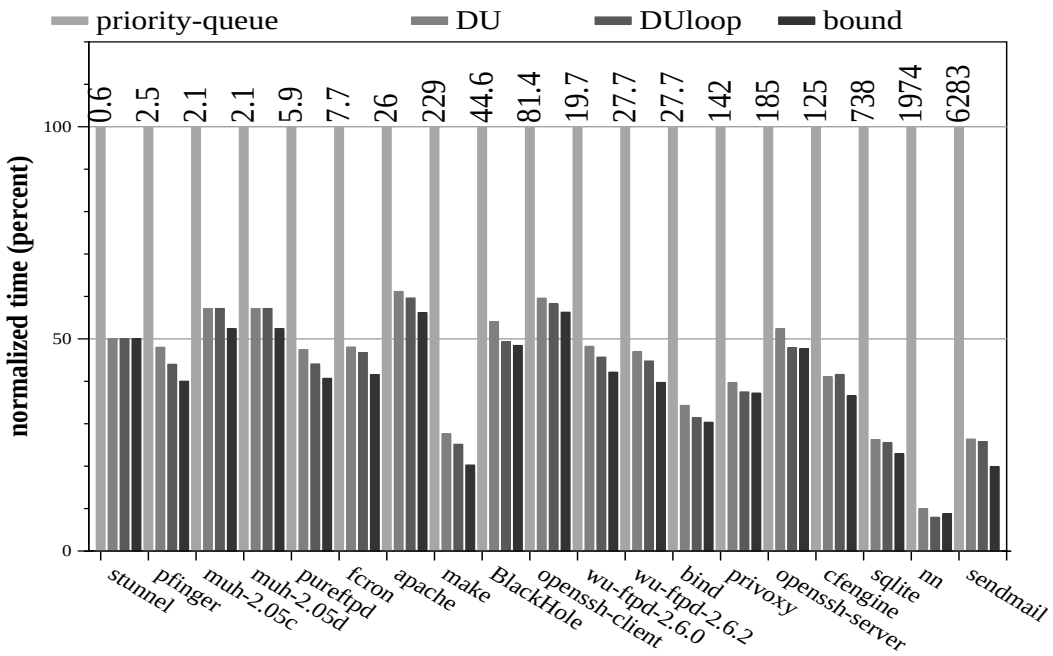
[2]Note that a better ordering of the visits in the first execution may lead to an even lower bound.

4.4.4 Results

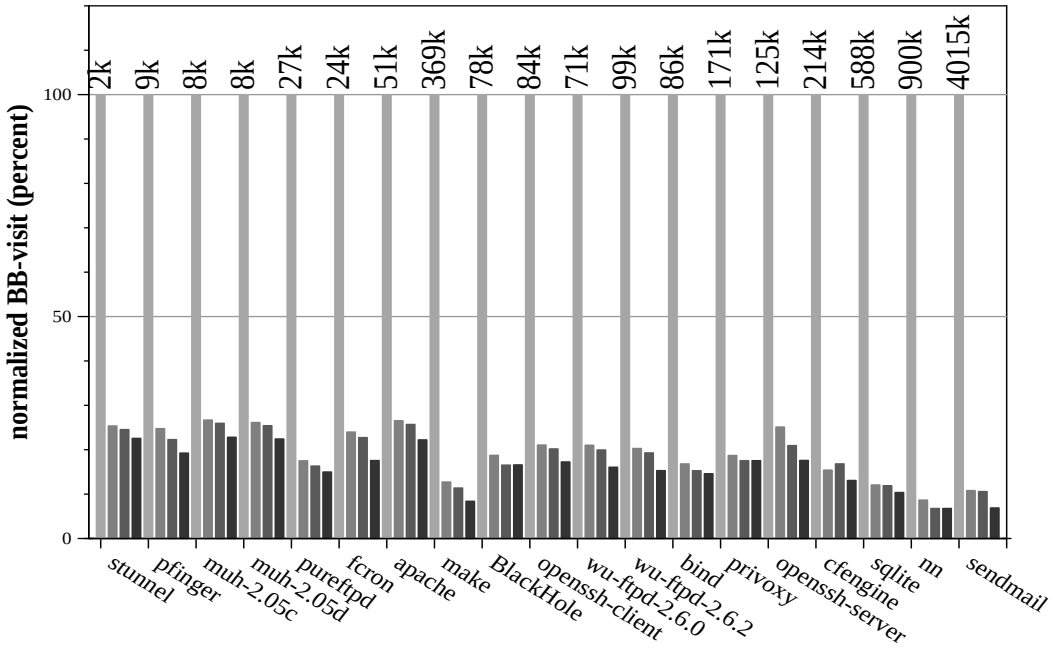
We first consider the behavior of our worklist algorithms in conjunction with CI pointer analysis. Each graph in Figure 4.8–4.9 shows the performance of $\mathcal{DU}$, $\mathcal{DU}_{loop}$, and our empirical lower bound normalized against our baseline, which uses Hind and Pioli's priority queue. The benchmarks are listed in order of increasing size, so we see that $\mathcal{DU}$ significantly reduces analysis time, with an average reduction of 56% (2.2× speedup), and larger benchmarks tend to benefit the most. For example, $\mathcal{DU}$ analyzes `sendmail` 74% (3.8×) faster than the baseline. We also see that $\mathcal{DU}_{loop}$ only improves upon $\mathcal{DU}$ by a few percentage points and that the main source of improvement is the increased efficiency. For example, for the large benchmarks, the efficiency of the baseline is just a few percent, but for $\mathcal{DU}$ it is in the 30-60% range. The cost of this reduced analysis time is a modest increase in memory usage. Finally, we see that there, theoretically, is still room for increased efficiency.

Figure 4.10–4.11 shows similar results for context-sensitive pointer analysis. Results are only shown for benchmarks that complete under the baseline. The benefit of $\mathcal{DU}$ is larger for CS mode than CI mode, because the large number of contexts exacerbates any inefficiency in the worklist. For example, $\mathcal{DU}$ improves the analysis time of `wu-ftpd-2.6.2` by approximately 80% (5× speedup), while in CI mode, its improvement is only approximately 53%. These results are encouraging. We also see that the memory overhead of our algorithms increases under CS mode.

We have repeated our experiments with five different error-checking clients (Section 3.3.2, page 49). These are interprocedural analyses that generally yield better precision with flow-sensitivity. We run each client concurrently with the pointer analysis, and the results generally follow the same pattern as those in Figures 4.8–4.11, so we omit these to conserve space.

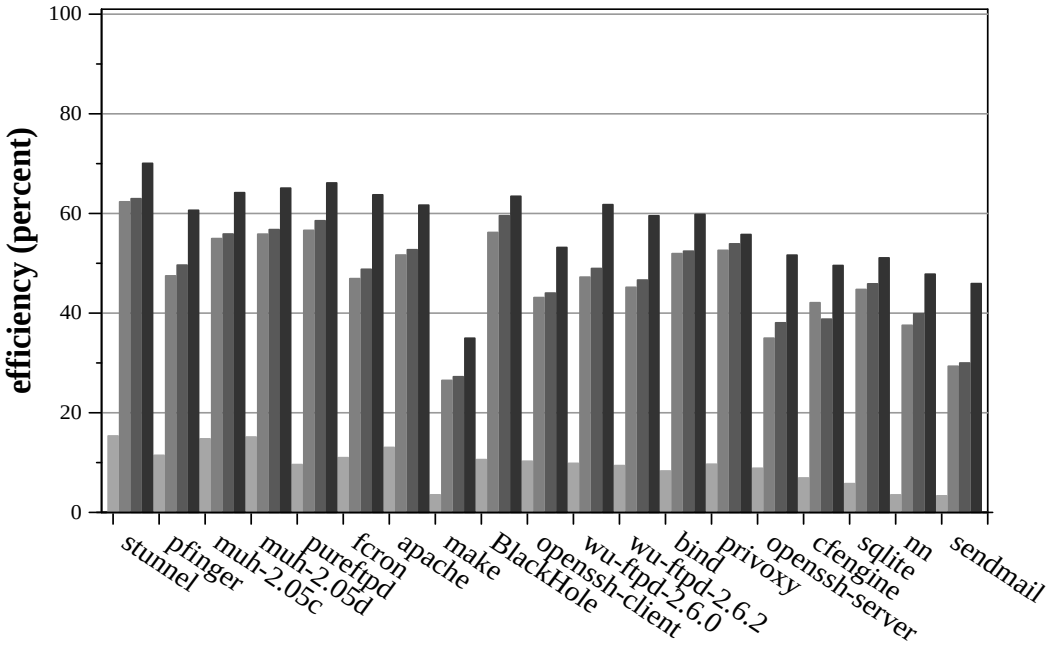


(a) Normalized analysis time.

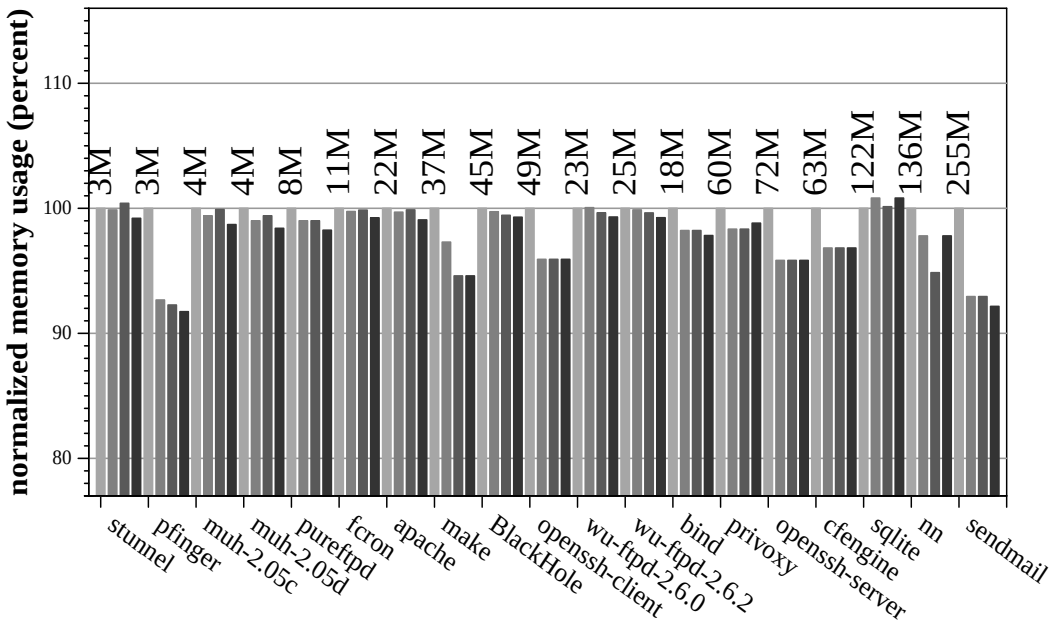


(b) Normalized BB-visits.

Figure 4.8: Performance results of $\mathcal{DU}$ and its variant, on CI pointer analysis (part 1).



(c) Efficiency.



(d) Normalized mem

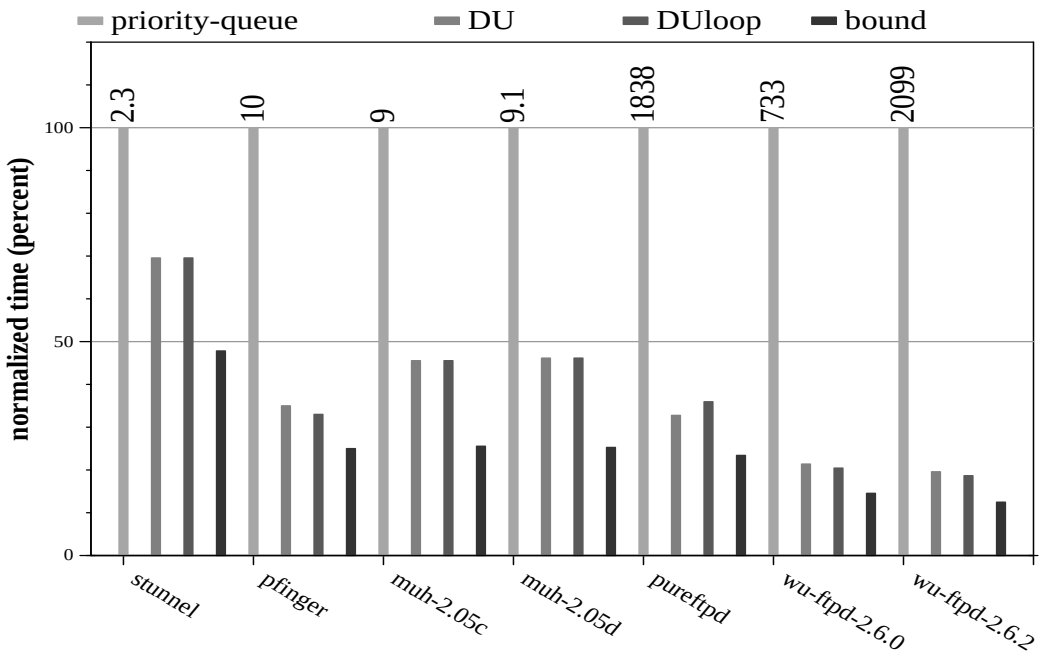
Figure 4.9: Performance results of $\mathcal{DU}$ and its variant, on CI pointer analysis (part 2).

4.5 Conclusion

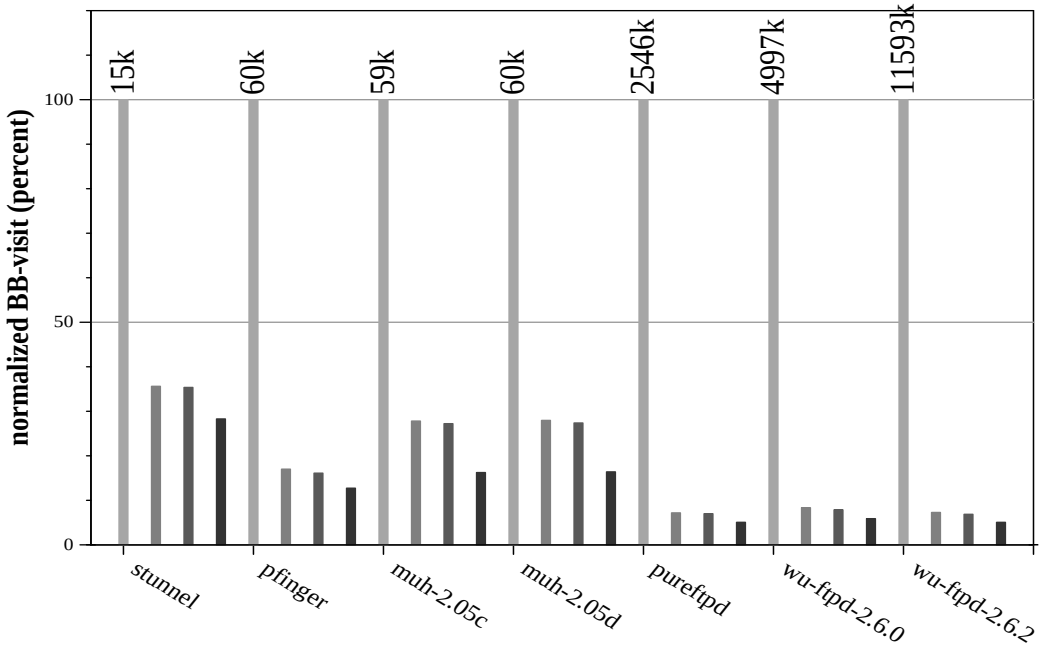
The ability to accurately analyze large programs is becoming increasingly important, particularly for software engineering problems such as error checking and program understanding, which often require high precision such as interprocedural flow-sensitive analysis. This chapter shows that by tuning the worklist to avoid repeating same computations needlessly, dataflow analysis can be made much more efficient without sacrificing precision.

We have implemented and evaluated a worklist algorithm that utilizes def-use chains. When compared with previous work, our $\mathcal{DU}$ algorithm shows substantial improvement, reducing analysis time for large programs by up to 90% for a context-insensitive analysis and by up to 80% for a context-sensitive analysis. The $\mathcal{DU}$ algorithm works well because it avoids a huge amount of unnecessary work, eliminating 65% to 90% of basic block visitations. We have also explored methods of exploiting CFG structure, and we have found that exploiting loop structure provides a small benefit for most of our benchmarks.

An empirical lower bound analysis reveals that there is room for further improvement. More study is required to determine whether some technique that considers both CFG structure and its interaction with data dependences can lead to further improvement.

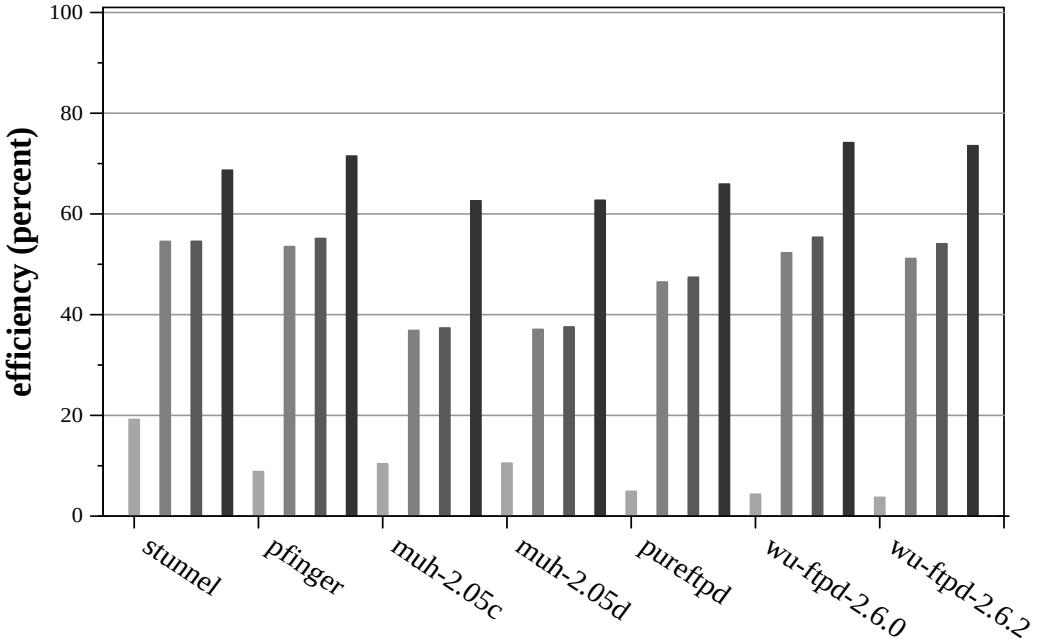


(a) Normalized analysis time.

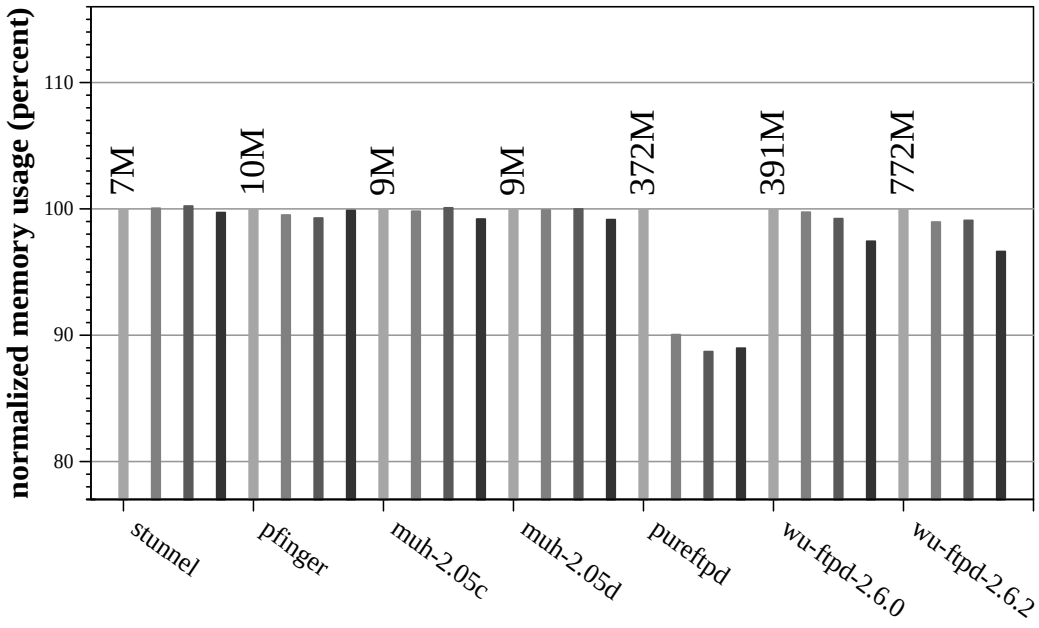


(b) Normalized BB-visits.

Figure 4.10: Performance results of $\mathcal{DU}$ and its variant, on CS pointer analysis (part 1).



(c) Efficiency.



(d) Normalized memory usage.

Figure 4.11: Performance results of $\mathcal{DU}$ and its variant, on CS pointer analysis (part 2).

Chapter 5

Relevance-Based Context Partitioning

5.1 Introduction

Pointer information is important for many interprocedural analyses, including program slicing and security analysis. For such analyses, both flow- and context-sensitivity are important [144, 42, 54]. However, because the cost of context-sensitive analysis grows exponentially with the size of the call graph [36], context-sensitive pointer analysis requires some way to limit analysis costs.

Grove et al. [47] describe a general framework in which each procedure's contexts are partitioned, and all contexts in the same partition are analyzed together. The analysis time thus grows with the number of partitions instead of number of contexts. Alias patterns [153] and types [1] have been used to partition contexts in pointer analysis and type analysis, respectively. Alias-based partitioning can suffer from decreased precision, while Agesen's Cartesian Product Algorithm, which has

used type-based partitioning [1], does not apply well to pointer analysis, because the large points-to sets would lead to an explosion in the number of partitions.

In this research, we aim to partition contexts in such a way that reduces the amount of unimportant computations. We will achieve this goal by extending Agesen's Cartesian Product Algorithm to pointer analysis. The key observation is that points-to information is not useful in partitioning contexts. We instead care about the precision of the client analysis that uses the pointer information. Our solution thus couples the pointer analysis with the client analysis and lazily partitions contexts based on the client analysis' dataflow values. For example, consider a client that tracks two possible flow values, T and U, and consider a procedure that takes one parameter. In our solution, contexts in which the procedure is passed a T value are placed in one partition, while those in which the procedure is passed a U flow value are placed in a separate partition. All contexts in the same partition are analyzed context-insensitively, as illustrated in Figure 5.1(c). With this scheme, irrelevant information can be merged within a partition, perhaps even propagating information across unrealizable paths, but relevant information, as defined by the client analysis, always merges identical values, so no precision is lost. We refer to our solution as Relevance-Based Context Partitioning (RBCP).

Relevance-Based Context Partitioning can be applied to many possible pointer analysis algorithms, as well as to other "service analyses," such as the computation of reaching definitions. To show the power of RBCP, we first apply it to a flow-sensitive context-sensitive pointer analysis (FSCS), and we show that the resulting analysis, Relevant-Based Context-Sensitive pointer analysis (RBCS), is dramatically more efficient. We also apply RBCP to the Client-Driven pointer analysis [54] to produce an algorithm that we refer to as RBCS-CD. The Client-Driven

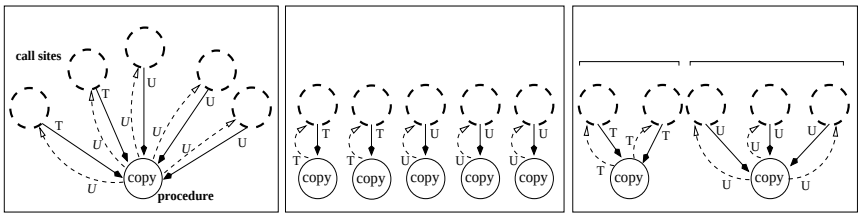


Figure 5.1: Partitioning call sites. Flow values are propagated along call edges (solid lines) and return edges (dashed edges). (a) Context-insensitive analysis merges flow values from all call sites (T⊓U=U) before they are propagated back to the callers. (b) Context-sensitive analysis prevents merging of flow values, but the cost depends directly on the number of contexts. (c) Relevance-Based Context Partitioning puts the contexts into two groups, one for each distinct flow value (regardless of number of contexts), and the flow values are not merged.

pointer analysis algorithm uses information from the client analysis to identify the subset of procedures that must be analyzed context-sensitively to ensure a precise client analysis, allowing all other procedures to be analyzed context-insensitively.

In the broader picture, we believe that the notion of Coupled Analyses, in which the results of a client analysis are used to improve the performance of some service analysis, is an important paradigm that deserves further study.

5.1.1 Contributions

This chapter makes the following contributions:

- We introduce RBCP, an extension Agesen's Cartesian Product Algorithm that can be applied to pointer analysis; the key idea is to use the client analysis' dataflow values to define the pointer analysis' context partitions. One key to avoiding precision loss is to lazily refine contexts, as we describe in Section 5.2.6.

- We evaluate two algorithms that implement this idea by using them to perform

five security analyses on a collection of 19 open-source servers and system utilities. The RBCP technique improves the scalability of the FSCS algorithm, enabling the largest 60% of the programs to complete. For those cases where both algorithms complete, RBCP is an average of 7× faster, and its memory usage is reduced by 4× on average.

The RBCP technique speeds up the analysis time of the Client-Driven pointer analysis by an average of 1.6×, excluding the one case where RBCS-CD completes but the baseline does not; memory usage is reduced by an average of 1.5×. We show that our solution is efficient because it requires an average of 1.6 context partitions per procedure.

- We argue that the notion of Coupled Analyses, in which the results of a client analysis are used to improve the performance of some service analysis, is an important paradigm that deserves further study.

The remainder of this chapter is organized as follows. Section 5.2 presents our solution, and Section 5.3 presents results. Section 5.4 discusses the more general framework of Coupled Analysis. Section 5.5 concludes this chapter. We review related work later in Section 7.3.

5.2 Our Solution

In this section, we will start with the objective and a high-level overview of our solution. Sections 5.2.2–5.2.7 will provide additional details to the major steps in the algorithm. Section 5.2.8 discusses precision issues, and Sections 5.2.9 describes an extension to the algorithm. Section 5.2.10 concludes this section with a brief description of our implementation.

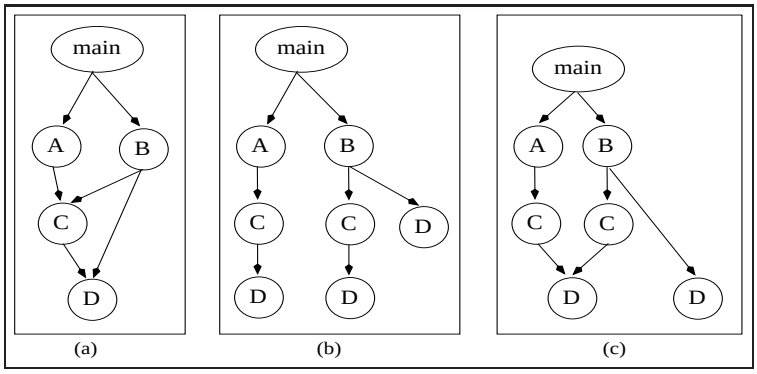


Figure 5.2: Call graph example. (a) An example call graph. (b) Its invocation graph. (c) Invocation graph used by Relevance-Based Context Partitioning.

We can describe the goal of our solution as follows. For a given callgraph (Figure 5.2(a)), the invocation graph [36] (Figure 5.2(b)) grows exponentially, because it represents all possible procedure invocations.[1] Thus, our goal is to merge nodes in the invocation graph, which corresponds to merging contexts, which might lead, for example, to the graph in Figure 5.2(c). Each node in the merged graph represents a set of contexts. We use a *contour* as an environment for the analysis of a procedure.[2] Section 5.2.1 explains a contour in more details. For each contour, the procedure is analyzed in a context-insensitive manner. Flow information may be merged within a contour, but merging is not possible between contexts belonging to different contours.

Our solution is an algorithmic framework that performs an iterative dataflow analysis on the invocation graph. Starting with just a single node for the root procedure, the call graph is built lazily, with nodes being logically merged as the analysis proceeds. During analysis on a procedure, the statements in the procedure are visited. At each statement, transfer functions are first applied for the pointer

[1] To simplify the presentation, we omit statement nodes in the graph, but these are necessary to distinguish different calls in the same procedure.

[2] Grove et al. [47] use a different definition for the same term.

analysis and then for the client analysis, which gives the client analysis access to the pointer information that it needs. Section 5.2.3 describes the major steps involved in analyzing a procedure.

If the statement is a procedure invocation, we either expand the invocation graph by adding a new node, i.e., a new contour, for the callee, in which case we recursively continue the analysis on the callee using the new context, or we reuse the previous node for the same callee. Sections 5.2.4–5.2.6 describes how we choose a contour. It is this reuse of the previous node that logically merges nodes in the invocation graph. When reusing a node, it is possible to reuse results of the node, as opposed to needing to reanalyze the procedure. The conditions for reusing results are described in Section 5.2.7, but the main point is that the procedure's context values—the flow values of its actual parameters and global variables evaluated at that context—must match that of the node's previous context. Handling of recursions is described in Section 5.2.3.

One detail is that aliasing of formal parameters can cause undesired merging of flow values. We correct this problem by *contour refinement,* which we will explain in Section 5.2.6.

In a perfect scenario, a client analysis would know *a priori* what parameter flow values were of interest. For example, the client may ignore a parameter that is never assigned more than one value, or whose value is never used. Unfortunately, we in general do not have such advance knowledge regarding the behavior of the procedure without at least a partial solution of the analysis itself. Instead, our solution conservatively uses all parameter flow values.[3]

[3]To simplify our description of the algorithm, we do not include global variables until Section 5.3.

5.2.1 What is in a Contour?

Abstractly, we consider a contour as an environment used in the analysis of a procedure. There can be multiple contours for each procedure. Semantically, the contours represent the disjoint subsets of contexts for that procedure. Operationally, each contour stores all intermediate and final results for the analysis of the procedure in those contexts. These results include flow values and def-use chains computed for all variables at all statements. If multiple contexts are associated with a contour, then the procedure is analyzed context-insensitively with respect to this set of contexts. This means that all flow values imported from all callers must be merged at the procedure entry.

5.2.2 Algorithm Overview

Figure 5.3 shows two routines responsible for analyzing a procedure and iterating through the call graph. The routine `analyzeProc` processes the contents of a procedure P, and invokes `analyzeCall` whenever it encounters call sites.[4] This second routine employs three routines that constitute the RBCP algorithm:

- `computePV`: Given a call site and the current contour of the caller P, computes the *Partitioning Vector* V using the flow values at the site.
- `chooseContour`: Using the vector V, use a hash table to obtain a contour, creating a new contour if necessary. The vector, therefore, acts as a key that maps flow values to contours. Calling contexts that yield the same V are considered similar contexts.

[4]These routines are added into the analysis framework responsible for analyzing procedures, which we outlined in Section 3.1.2. That is, we insert new codes into the generic algorithm in Figure 3.1 (page 35). The result shown in Figure 5.3 highlights newly inserted code.

- `refineContour`: The mapping is refined, if necessary, to remove any imprecision alluded to above steps.

The Partitioning Vector can be seen as a snapshot of the state of analysis at a call. The contents of the vector are carefully chosen so that different calling contexts with same computed vector are deemed as similar contexts, which are then grouped and analyzed collectively by the RBCP algorithm.

The next section explains the steps performed by `analyzeProc` and `analyzeCall`. Sections 5.2.4–5.2.7 will then explain in detail the three RBCP routines.

5.2.3 Analyzing a Procedure

The algorithm starts by calling the `analyzeProc` routine on the root procedure. The routine also takes a contour parameter *contour*. As usual, it uses a worklist to iteratively process the statements in the procedure. For each statement, a pointer analysis is first applied followed by a client analysis, so that the latter has access to pointer information.

The `analyzeCall` procedure is invoked to process procedure calls. It is responsible for deciding how a callee procedure is analyzed. It first appends the call site to the current caller contour *contour* to obtain *cxt*, which represents the current calling context. The new callee is then processed accordingly:

- If the callee is not in the current call stack, `analyzeCall` then prepares to analyze the callee; if it is not a recursive procedure, then the RBCP routines are invoked to obtain a contour. If the callee is predetermined to be a recursive procedure, then only one contour—its *default contour*—is used for all its contexts.

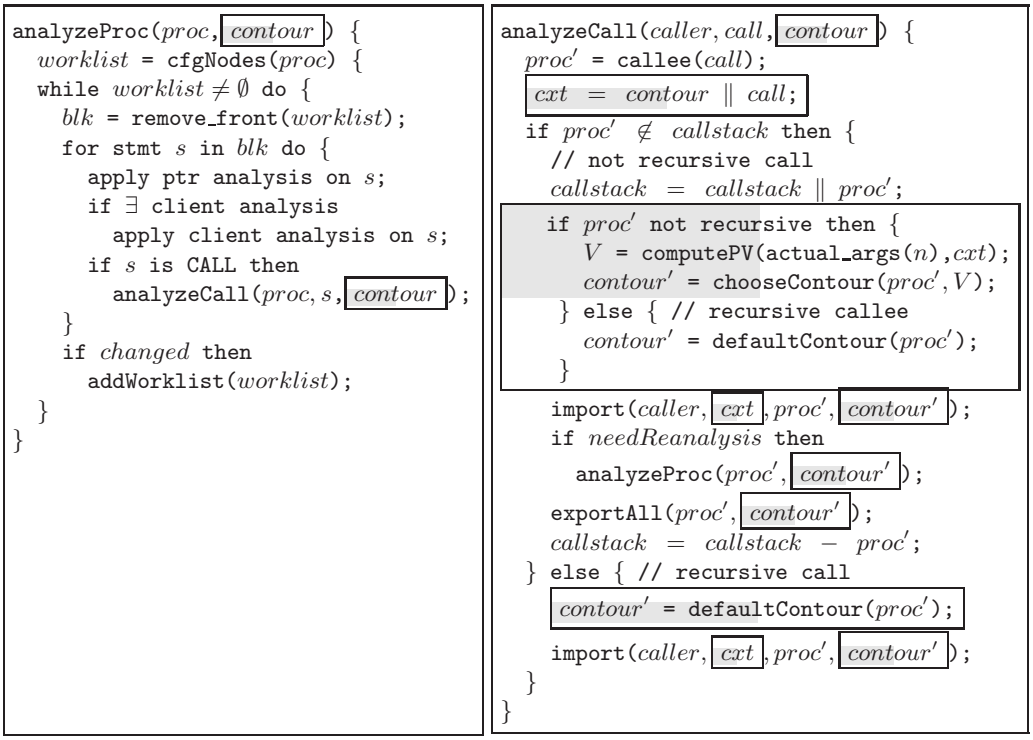


Figure 5.3: RBCP: Analysis on a procedure. Analysis starts by a call to `analyzeProc` with the root procedure and its default contour. For a non-recursive call, `analyzeCall` computes the Partitioning Vector V, uses it to pick a contour and, if necessary, refines the contour selection. Sections 5.2.4–5.2.6 and Figure 5.4 explain these steps in more detail.

Using the contour just chosen, `analyzeCall` then imports flow values from the caller context to callee contour. This involves processing the interprocedural Φ-functions by updating formal parameter and global variable values. If this merging updates any flow value, then `analyzeProc` is then called to analyze the callee. After analysis on the callee completes, flow values are then exported from the callee to *all* call sites associated with `contour`.

- If the callee is in the current call stack, `analyzeCall` does not analyze the callee. Instead, using the default contour, flow values are imported from the

current call stack to the callee. This step allows the callee to be analyzed later in a non-recursive context.

We next discuss in detail the steps on choosing and refining contours, before moving on to issues of soundness, efficiency, and implementation.

5.2.4 Computing the Partitioning Vector

The routine `analyzeCall` (in Figure 5.3) first calls `computePV` to obtain a Partitioning Vector, which is then used to obtain a contour via a call to `chooseContour`. The latter routine in turn also invokes a routine `refineContour`, which is responsible for contour refinement. The relations among these routines are shown in Figure 5.4. We now explain these steps in detail in this and the next two sections.

Given the actual parameters of a call to procedure *proc*, `computePV`, combines the actual parameters' flow values into a Partitioning Vector

$$V \quad = \quad \langle v_1, \ldots, v_n \rangle \tag{5.1}$$

where n is the number of parameters. There is more than one method to compute V, and our RBCS algorithm is sound as long as the same method is always applied to each callee. However, a more aggressive method can lead to a more precise solution. We describe one such method as follows.

Given a variable x, let $R^1(\{x\})$ be x's points-to set plus the field variables of x (if any). The transitively *reachable variables* from x, or $R^+(\{x\})$, are all the

```
computePV(args, cxt) {
  for i in 1..|args| do {
    S = resolvePtr(args[i], cxt); // resolve any pointer dereferences
    V[i] = ⊓ cfv(y) ∀ y ∈ R+(S, cxt);
  }
  return V;
}
chooseContour(proc, V) {
  c = table_proc[V];
  if ∄c then {
    c = table_proc[V] = newContour(proc);
    return c;
  }
  return refineContour(proc, c, V);
}
refineContour(proc, contour, V) {
  L = multimap_proc[V];
  do {
    formals = tentativeUpdateFormals(proc, contour);
    V' = computePV(formals, contour);
    if V == V' then return contour;
    pick another contour from L;
  } while ∃contour;
  contour = newContour(proc);
  add contour to end of L;
  return contour;
}
```

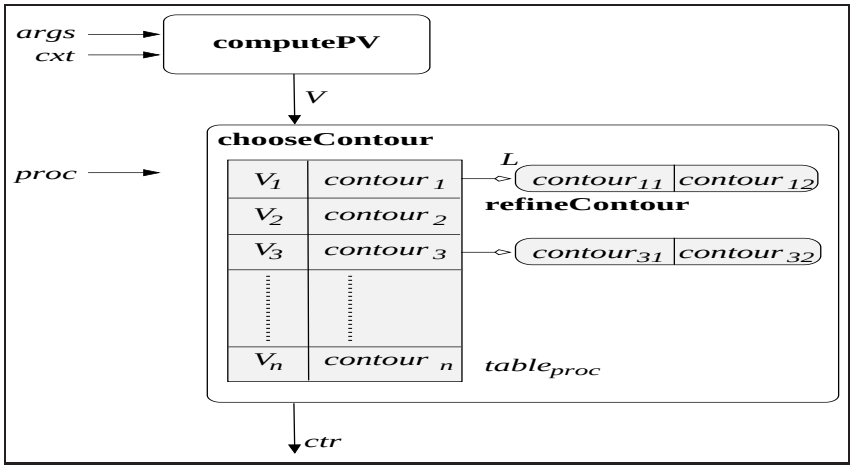


Figure 5.4: RBCS Analysis Algorithm.

variables after transitively following the points-to and fields-of operators:

$$\begin{aligned} R^1(X) &= \bigcup_{x \in X} \{y | x \text{ points to } y\} \cup \text{fieldsof}\{x\} \\ R^{i+1}(X) &= R^1(R^i(X)) \\ R^+(X) &= \bigcup_{i>0} R^i(X) \end{aligned} \tag{5.2}$$

Let cfv(y) denote the client flow value for variable y. We then define

$$v(x) = \sqcap \text{cfv}(y) \quad \forall\ y \in R^+(\{x\}) \tag{5.3}$$

i.e., the greatest lower bound value of the client flow values of all the reachable variables from x, with respect to the meet operator $\sqcap$. We call $v(x)$ the *reachable value* from x. Sometimes we will simply write v if it is clear what variable x we are referring to.

Using definitions (5.1)–(5.3), the routine `computePV` computes v_i, the ith-component of the Partition Vector, using the ith actual parameter, after resolving any pointer or field dereferences in the parameter. For example, if the actual parameter is `*p`, we use the pointer analysis to determine `p`'s points-to set S, and then compute v_i using $R^+(S)$. Note that the points-to set, the reachable variables $R^+(S)$, and reachable value cfv(y) are all computed using the current calling context *cxt* in the caller. The different v_i are combined to form the Partitioning Vector.

Figure 5.5 provides an example of computing the Partitioning Vector. Assume a flow value lattice given in Figure 5.5(a), and a procedure `foo` with two parameters. Figure 5.5(b) shows two calls to the procedure, and the points-to graphs at each call. At the first call, the first parameter points to two targets, whose client flow values are A and B. The reachable value is computed as $A \sqcap B = C$. The sec-

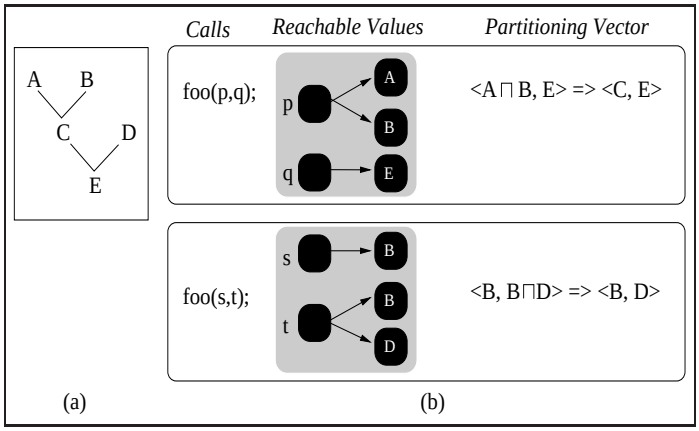


Figure 5.5: Example of computing the Partitioning Vector. (a) The lattice of flow values. (b) Two calls to a procedure `foo`. For each call, from the points-to graph we can derive the reachable variables and reachable values from each actual parameter. The components of Partitioning Vector are computed by merging the reachable values from each parameter. In the example, the two calls have identical Partitioning Vectors.

ond parameter points to a singleton whose value is E, and hence, the Partitioning Vector is $\langle C, E\rangle$. The Partitioning Vector for the other call is similarly computed. It turns out that the vectors for these two calls are different.

Additional components of the Partitioning Vector

The contents of a Partitioning Vector are the client flow values that flow from callers to callees, so that these values are merged in different ways depending on how the contexts are partitioned. Such merging directly influences the client analysis on the callees. One natural question is, are there any other flow values that can affect the client analysis and, therefore, should also be included in the vector?

It turns out that values of function pointers should be included also. If a procedure P has a function pointer as a parameter, which points to different targets under different contexts, then analyzing these contexts with a single contour

could lead to undesired consequences: if P contains a call via this function pointer, spurious contexts will be created for callees of this call, which leads to imprecise analysis.

The solution is to include values of function pointers in the Partitioning Vector of procedure P, if the pointers are used by P or any callees of P. The vector will contain additional components of the form "$x \rightarrow \{foo\}$". We find that procedures tend not to use many (if any) function pointers, and function pointers tend to have "stable" values in the program, i.e., they do not change often in the programs. Therefore, the addition to the Partitioning Vector is a relatively small overhead.[5]

5.2.5 Choosing a Contour

The Partitioning Vector V is used as a key to map contexts to contours: contexts that yield different vectors using `computePV` shall be mapped to different contours. This property is easily achieved by way of a hash table per procedure.

Once the key is computed, it is passed on to the routine `chooseContour`. If the key has not been seen before, a new contour is created, and the hash table is updated. Otherwise, an existing contour is chosen. Contour refinement may be performed on this existing contour, and we will explain this step in the next section.

Note that new contours are lazily created on demand, since not all keys may be needed. The number of contours required depends on n, the number of parameters, and f, the number of possible flows values per variable. In the worst case, as many as f^n keys are needed per procedure, but in practice, we find that often only a small number of keys are used. One reason is that f is often small. The

[5]Even in object-oriented languages, where function pointers disguised in abstract method calls are very common, we do not have to include these function pointers in the Partitioning Vector if we group contexts based on objects' concrete types.

other reason is that many client analyses often focus on a subset of "interesting" variables by assigning special flow values to these variables. For example, a client analysis that tracks file status (the File-Access problem described in Section 3.3.2) will assign a "file-open" value only to "`FILE`" variables in the program.

RBCP exploits these characteristics in many analyses and programs. If very few keys are needed for a procedure, and if there are many call contexts to the same procedure, then there is great potential for grouping the many contexts into a small number of partitions.

5.2.6 Contour Refinement

The previous section describes how we choose a contour based only on the actual parameters' flow values. The steps described, so far, turn out to be insufficient due to parameter aliasing: when a contour is chosen, the actual parameters from different contexts are aliased via updates on the formal parameters. This aliasing could lead to unintended merging of values.

Too see why unintended merging can occur, it is important to keep in mind that a contour is an environment for storing the results for each time that analysis is performed on a procedure. Those results include information such as points-to sets or other values computed for local variables in the procedure. Such information is shared if the contour is reused for different contexts.

We now illustrate how the unintended merging can occur by way of the example in Figure 5.6. This example assumes the client analysis has two distinct flow values T and U. There are two calls to procedure `F`, and in each case, the actual parameter has reachable value T. The Partitioning Vector $V = \langle \mathrm{T} \rangle$ is computed for both calling contexts, so that both contexts are mapped to a common contour. Note

that V must be computed *before* a contour is chosen or created.

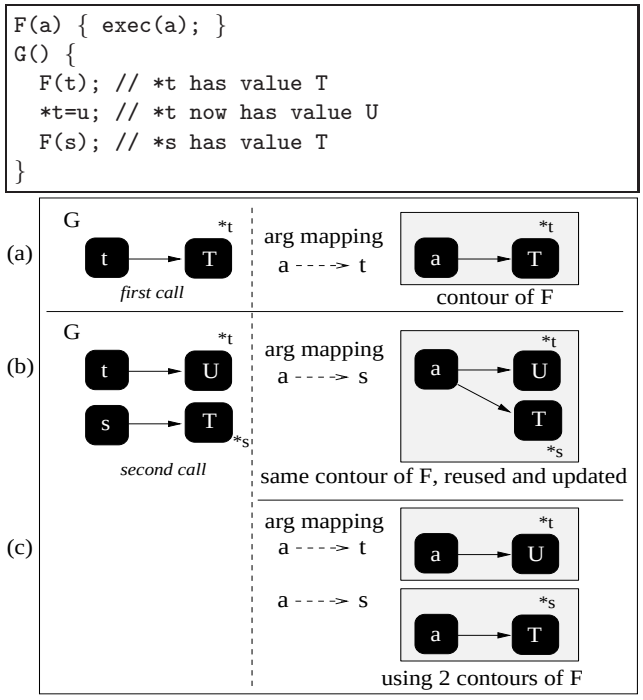


Figure 5.6: Example illustrating the need for contour refinement. (a) At the first call to `F`, the actual parameter has reachable value T, and one contour is used. `*t` turns U after the call. (b) At the second call, the new actual parameter still has value T, but if the same contour is reused, update on the formal parameter leads to undesired value merging. (c) If separate contours are used, no value merging occurs.

Now consider what happens when this contour is actually used to analyze `F`, which is analyzed twice. For the first call, Figure 5.6(a) shows that the formal parameter `a` is set to point to {`*t`}, so the reachable flow value is T. When this first analysis on `F` completes, and analysis is back at the caller, `*t` is assigned a new value U before the next call. If the same contour is reused at the second call, as shown in Figure 5.6(b), the formal parameter is updated to point to `*t` and `*s`, which now have two distinct values. The new reachable flow value involves merging of different values, leading to loss of precision.

The problem arises because it is insufficient to choose a contour *ctr* based only on available information at a call site. To fix the problem, we need to validate that reusing contour *ctr* does not lead to undesired merging (the client analysis is responsible for deciding what is undesired, such as the merging of T and U in the example). If such merging can occur, another contour must be used instead. This implies that each Partitioning Vector V must be mapped to multiple contours.

The refinement step, via the routine `refineContour` in Figure 5.3, uses a multi-way hash table to map a key to a list L of contours. The routine validates a contour by first tentatively updating the points-to sets of the formal parameters, without updating the contour. A Partitioning Vector V' is then computed using the updated formal parameters, which is then compared against the key V. If they do not match, another contour from the list L is tested. The routine returns the first validated contour from the list. If no contour can be validated, a new one is used.

Finishing the example, Figure 5.6(c) shows that our algorithm will use two contours for `F`: the first is created for the first call to `F`, with `a` pointing to `*t`, and a second contour is used by second call site with `a` pointing to `*s`.

The refinement step does not affect the soundness of the analysis: using more contours helps to avoid precision loss and hence leads to more precise solution. The client analysis, therefore, has a degree of freedom. In contrast, previous solutions such as Partial Transfer Functions (PTFs) [153, 101] did not discuss such options. Later in Section 5.3, we will present statistics on the number of contours.

5.2.7 Reusing Contours Context-Insensitively

An important property of context-insensitive analysis is that if all of a procedure's input flow values do not change at a call site, the procedure need not be analyzed

again. Specifically, each formal parameter value v_f (from a previous analysis) is updated with a value v_a obtained from the new call site, yielding a new value $v'_f = v_f \sqcap v_a$. If v'_f and v_f are the same, no change has occurred. The test is also repeated for global variables, and other reachable variables that are used in the procedure. Based on these tests, the flag *needReanalysis* in the routine `analyzeCall` in Figure 5.3 is set accordingly.

When no reanalysis is needed, the procedure's side effects are merely exported to the calling context. With Relevance-Based Context Partitioning, each procedure is analyzed context-insensitively for each contour. Therefore, we reap some of the performance benefit of context-insensitive analysis, as long as we can remember the procedure input values for each contour. Similarly, each contour must maintain its own set of side effects that need to be exported.

5.2.8 Precision and Accuracy

RBCS analysis provides an alternative level of context-sensitivity—two other levels are a context-sensitive analysis (CS) and a context-insensitive analysis (CI). CI is imprecise because it propagates flow values along interprocedural unrealizable paths. CS is precise because it avoids propagation along such paths. RBCS analysis prevents propagation along some of those unrealizable paths. Therefore, in the spectrum of precision, CS and CI are the two ends on the spectrum, while RBCS analysis is lies in between.

On the other hand, with respect to the client analysis, the propagations along unrealizable paths in a RBCS analysis does not lead to undesired merging of flow values: if a flow value is never propagated along any path in a CS analysis, then that value is also never propagated along any path in a RBCS analysis. Therefore,

RBCS achieves the same level of accuracy as a CS analysis.

5.2.9 Application to Client-Driven Pointer Analysis

Our discussion until now has ignored the details of the pointer analysis, which has simply been assumed to have its own set of flow functions. In preparation for describing the integration with the RBCP algorithm, we now describe the Client-Driven pointer analysis [54], which employs a two-pass analysis framework; in each pass, the client analysis and pointer analysis are performed concurrently. In the first pass, a fast and imprecise (flow-insensitive, context-insensitive) analysis is used to gather information that will guide the second pass. In particular, the first pass identifies assignments, including assignments to formal parameters, that lose precision due to the merging of client flow values. These sources of precision loss identify the procedures that will be analyzed context-sensitively in the second pass, as well as the variables that will be analyzed flow-sensitively in the second pass.

Relevance-Based Context Partitioning can be seamlessly applied to Client-Driven analysis. Specifically, let Π denote the set of procedures to be analyzed context-sensitively in the Client-Driven algorithm's second pass. When a procedure P is added to Π by the first pass of the Client-Driven algorithm, it is because some parameter x, or some of its reachable variables, will otherwise lose precision that is important to the client. Let criticalPara(P) be the set of these parameters x. Thus, to combine RBCP with the Client-Driven algorithm, we let the latter decide which parameters to include in the computation of the Partitioning Vector: it will only use the variables in the set criticalPara(P). Only the routine `computePV` (Figure 5.3) needs to be modified. In this way, when P is analyzed in the second pass, the contexts will be grouped in such a way that will cause any undesired merging of x's

flow values.

Relevance-Based Context-Sensitive analysis complements Client-Driven analysis, because the former groups similar contexts together, while the latter provides the minimum V required for each procedure. Without our RBCS algorithm, the Client-Driven algorithm has to separately analyze all contexts of the procedures in Π.

5.2.10 Implementation

Recall that Section 3.2.1 (page 43) explains how Broadway represents an input program by means of location trees. Particularly, a procedure location is used to represent a procedure, and multiple instances of these locations are used in a context-sensitive analysis: one instance per calling context. In each instance, the procedure is analyzed independently; all analysis results computed for a context, such as points-to sets, use-def chains, and client flow values, are stored using the associated procedure location and are, therefore, kept separate from results computed for another context. In a context-insensitive analysis, only instance of procedure location is used for each procedure.

In lieu of the above properties of a procedure location, the location tree is a natural option to implement a contour; RBCS simply uses a procedure location to represent a contour. Unlike in a context-sensitive analysis, RBCS uses a procedure location to represent more than one context, but unlike a context-insensitive analysis, there can be more than one procedure location per procedure. For each procedure location *procloc*, RBCS analyzes a procedure context-insensitively, using those contexts associated with *procloc*.

5.3 Evaluation

This section describes our experimental results, beginning with methodology.

5.3.1 Methodology

We again evaluate our ideas in the context of the Broadway compiler, which supports, among other modes, a context-sensitive and flow-sensitive (FSCS) interprocedural pointer analyses and the Client-Driven (CD) pointer analysis. We evaluate our Relevance-Based Context Partitioning by applying it to these two pointer analyses; we call these new algorithms RBCS and RBCS-CD, respectively.

RBCP cannot be applied to pointer analysis alone: a client analysis must be specified. As before, the client analysis is run concurrently with the pointer analysis. We will use the same five analysis problems described in Section 3.3.2 as the client analyses. Table 5.1 shows the number of possible flow values per variables in each of the client analysis problems.

client	distinct flow values
File-Access	2
FSV	2
Remote-FSV	3
Remote-Access	3
FTP-Behavior*	3×10

Table 5.1: Number of distinct flow values per variable in the client analyses. The FTP-Behavior problem is made up of two subproblems with disjoint sets of flow values.

We use the same benchmarks described earlier for our evaluations (Table 3.2, page 52). Note that RBCS-CD inherits a feature of the Client-Driven analysis: if no errors are found during the first pass, no second pass is needed, and RBCP is not invoked. Out of the 95 program-problem cases, there are 30 such one-pass-only cases.

5.3.2 Context-Sensitive Analysis: RBCS versus FSCS

Table 5.2–5.4 compares the RBCS and FSCS algorithms. The results in this table are grouped according to the clients, and the benchmarks are roughly sorted in decreasing order of analysis time, with the order preserved across the clients. The columns in gray show the speedup in analysis time, and ratios of memory usage by RBCS against FSCS: the higher the numbers in these columns, the better for RBCS. We see that our technique significantly reduces analysis time and memory consumption. Without our technique, FSCS analysis completes successfully for only seven of the 19 benchmarks (space is the limiting factor); with our technique, RBCS completes for 18 of the 19 benchmarks, while the largest benchmark `sendmail` completes on two of the five clients. The extreme cases are indicated by asterisks (`*`) in the table, which represent cases where RBCS completes but FSCS does not.

Among other cases, on average the speedup is 7× for all clients, and 4× less memory is consumed. Not surprisingly, larger benchmarks with large numbers of procedures tend to benefit most from our technique.

We now explain these tables in a bit more detail. The columns Π and **Cxts** are statistics showing, respectively, the number of procedures analyzed context-sensitively, and the total number of contexts for these procedures in Π.

For each client, the percentage improvement depends on the benchmark's characteristics, such as the length of its maximum acyclic call string and the number of call sites per procedure. For each benchmark, the improvement can vary among the clients, because the client dictates how the contexts are partitioned. The large performance gains can be attributed to the small number of contours required for each procedure.

Among all the benchmark-client pairs, on average each procedure requires

FSCS vs. RBCS		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**FSCS**	**RBCS**	×	**used**	**avg**	×
sendmail	File-Access	359	416	2571	-	434k	*	682	1.9	*
sqlite		333	386	2300	-	32k	*	505	1.5	*
nn		451	494	2441	-	5136	*	445	1.0	*
privoxy		221	223	1958	-	833.2	*	230	1.0	*
openssh-server		562	601	3896	-	776.0	*	701	1.2	*
cfengine		395	421	2260	-	144.2	*	466	1.2	*
openssh-client		412	438	3306	-	514.4	*	536	1.3	*
apache		295	313	976	-	627.7	*	416	1.4	*
make		101	167	372	-	467.7	*	100	1.0	*
BlackHole		70	71	1056	-	62.1	*	73	1.0	*
bind		193	210	729	-	43.9	*	223	1.2	*
fcron		99	100	513	-	9.7	*	109	1.1	*
pureftpd		109	116	536	1883	5.6	335.3	113	1.0	35.7
wu-ftpd-2.6.2		195	205	882	420.0	34.0	12.3	198	1.0	6.4
wu-ftpd-2.6.0		178	183	839	208.6	20.4	10.2	181	1.0	5.2
pfinger		39	47	118	17.4	3.6	4.8	41	1.1	3.0
muh-2.05d		83	84	335	7.1	2.2	3.2	95	1.1	2.1
muh-2.05c		83	84	333	7.0	2.2	3.2	95	1.1	2.1
stunnel		41	42	177	1.6	0.7	2.1	53	1.3	2.1
	Mean:						8.9		1.2	4.4
sendmail	FSV	359	416	2571	-	507k	*	594	1.7	*
sqlite		333	387	2302	-	125k	*	784	2.4	*
nn		451	494	2448	-	6767	*	630	1.4	*
privoxy		221	223	1958	-	1795	*	267	1.2	*
openssh-server		562	601	3896	-	994.4	*	658	1.2	*
cfengine		395	421	2260	-	490.8	*	608	1.5	*
openssh-client		413	438	3306	-	435.4	*	479	1.2	*
apache		295	313	972	-	497.3	*	404	1.4	*
make		101	167	372	-	555.7	*	141	1.4	*
BlackHole		70	71	1056	-	133.5	*	113	1.6	*
bind		193	210	729	-	81.9	*	271	1.4	*
fcron		99	100	513	-	26.1	*	128	1.3	*
pureftpd		109	116	536	2013	7.7	262.9	127	1.2	37.0
wu-ftpd-2.6.2		195	205	882	471.3	87.8	5.4	278	1.4	4.6
wu-ftpd-2.6.0		178	183	839	245.2	43.1	5.7	254	1.4	3.9
pfinger		39	47	118	19.2	2.8	6.9	45	1.2	3.2
muh-2.05d		83	84	335	7.4	2.3	3.2	96	1.2	2.1
muh-2.05c		83	84	333	7.3	2.3	3.1	97	1.2	2.1
stunnel		41	42	177	1.7	0.6	2.9	41	1.0	2.3
	Mean:						7.7		1.4	4.2

(continue in Table 5.3)

Table 5.2: Relevance-Based Context-Sensitive analysis (RBCS) vs. FSCS analysis, part 1. Π is the set of non-root non-recursive procedures. The columns marked × are ratios measuring analysis speedups and memory savings by RBCS: large numbers (> 1) means RBCS is better. The **used** column shows the total number of contours used; **avg** is the average number of contours used per procedure. Benchmarks that do not complete are indicated with a **-**. The ***'s** indicate cases where our technique enables an analysis to complete and FSCS cannot. The largest benchmark `sendmail` completes in two of the five clients. The means for all program-clients are shown at the end of Table 5.4.

(continued from Table 5.2)

FSCS vs. RBCS		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**FSCS**	**RBCS**	×	**used**	**avg**	×
sendmail	Remote-FSV	-	-	-	-	-	-	-	-	-
sqlite		333	387	2302	-	106k	*	787	2.4	*
nn		451	494	2448	-	8116	*	706	1.6	*
privoxy		221	223	1958	-	2413	*	300	1.4	*
openssh-server		561	601	3896	-	1133	*	880	1.6	*
cfengine		395	421	2260	-	1235	*	813	2.1	*
openssh-client		413	438	3306	-	655.4	*	601	1.5	*
apache		295	313	974	-	674.4	*	456	1.5	*
make		101	167	372	-	605.8	*	100	1.0	*
BlackHole		70	71	1056	-	329.0	*	148	2.1	*
bind		193	210	729	-	72.1	*	261	1.4	*
fcron		99	100	513	-	11.9	*	113	1.1	*
pureftpd		109	116	536	2019	7.7	260.7	135	1.2	35.5
wu-ftpd-2.6.2		195	205	882	506.2	105.0	4.8	299	1.5	4.4
wu-ftpd-2.6.0		178	183	839	247.7	37.2	6.7	250	1.4	4.1
pfinger		39	47	118	19.9	4.9	4.0	50	1.3	2.6
muh-2.05d		83	84	335	7.4	2.5	2.9	115	1.4	2.0
muh-2.05c		83	84	333	7.2	2.4	3.0	113	1.4	2.0
stunnel		41	42	177	1.7	0.9	1.9	61	1.5	2.0
	Mean:						6.6		1.5	3.9
sendmail	Remote-Access	-	-	-	-	-	-	-	-	-
sqlite		333	387	2302	-	105851.0	*	783	2.4	*
nn		451	494	2448	-	8163.4	*	707	1.6	*
privoxy		221	223	1958	-	2401.3	*	298	1.3	*
openssh-server		561	601	3896	-	1125.1	*	881	1.6	*
cfengine		395	421	2260	-	1233.2	*	812	2.1	*
openssh-client		413	438	3306	-	654.2	*	601	1.5	*
apache		295	313	974	-	672.2	*	456	1.5	*
make		101	167	372	-	605.7	*	100	1.0	*
BlackHole		70	71	1056	-	333.4	*	148	2.1	*
bind		193	210	729	-	72.2	*	261	1.4	*
fcron		99	100	513	-	11.7	*	113	1.1	*
pureftpd		109	116	536	2026	7.8	261.2	135	1.2	36.0
wu-ftpd-2.6.2		195	205	882	502.8	104.9	4.8	299	1.5	4.4
wu-ftpd-2.6.0		178	183	839	247.9	37.1	6.7	250	1.4	4.0
pfinger		39	47	118	19.5	4.9	4.0	50	1.3	2.6
muh-2.05d		83	84	335	7.3	2.5	2.9	115	1.4	2.0
muh-2.05c		83	84	333	7.2	2.4	3.0	113	1.4	2.0
stunnel		41	42	177	1.6	0.9	1.9	61	1.5	2.0
	Mean:						6.6		1.5	3.9

(continued in Table 5.4)

Table 5.3: Relevance-Based Context-Sensitive analysis vs. FSCS analysis, part 2.

(continued from Table 5.3)

FSCS vs. RBCS		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**FSCS**	**RBCS**	×	**used**	**avg**	×
sendmail	FTP-Behavior	-	-	-	-	-	-	-	-	-
sqlite		333	387	2302	-	240k	*	866	2.6	*
nn		451	494	2448	-	15k	*	880	2.0	*
privoxy		221	223	1958	-	4387	*	366	1.7	*
openssh-server		562	601	3896	-	2008	*	1005	1.8	*
cfengine		395	421	2260	-	2412	*	1111	2.8	*
openssh-client		413	438	3306	-	1346	*	901	2.2	*
apache		295	313	976	-	7197	*	810	2.7	*
make		101	167	372	-	907.9	*	151	1.5	*
BlackHole		70	71	1056	-	255.0	*	208	3.0	*
bind		193	210	729	-	174.6	*	433	2.2	*
fcron		99	100	513	-	31.9	*	163	1.6	*
pureftpd		109	116	536	2237	11.1	99.5	155	1.4	97.2
wu-ftpd-2.6.2		195	205	882	595.1	171.3	71.2	362	1.9	74.7
wu-ftpd-2.6.0		178	183	839	276.5	52.4	81.0	302	1.7	73.0
pfinger		39	47	118	21.9	7.8	64.4	60	1.5	52.7
muh-2.05d		83	84	335	7.7	2.6	66.2	135	1.6	48.9
muh-2.05c		83	84	333	7.7	2.6	66.4	135	1.6	48.9
stunnel		41	42	177	1.7	1.0	40.3	62	1.5	50.1
Mean:							5.5		2.0	3.6
All-Mean:							7.0		1.6	4.0

Table 5.4: Relevance-Based Context-Sensitive analysis vs. FSCS analysis, part 3.

1.6 contours. Out of all the contours, 8.2% are created due to contour refinement. For each benchmark-client pair, the percentage is between 0% (no refinement) and 38.5% (std-dev. 8.2%).

Global variables

Finally, note that because we do not include global variables in the Partitioning Vector, we could lose precision. Fortunately, among those seven smaller benchmarks where we can compare precision, no precision loss on the clients is detected. Nevertheless, we experiment with a simple extension, where we add a global variable into the Partitioning Vector V of all procedures if that variable is ever assigned a non-default flow value. When applied to the benchmarks, on average 6.6 global variables

are added to V (std.dev. 11.6, max 52). With this extension, we find that precision is improved in two out of the 95 program-client cases (both in the `cfengine` benchmark), but performance suffers significantly: the mean speedup (over FSCS) drops from 7.0 to 4.7.

5.3.3 Client-Driven Analysis: RBCS-CD versus CD

Table 5.5–5.7 shows similar results for comparison between RBCS-CD and the Client Driven analysis (CD). We leave blank those table entries, such as `nn` File-Access, where no procedure requires any context-sensitivity, i.e. analysis completes after one pass. The largest benchmark, `sendmail`, was not able to complete with CD in the FTP-Behavior client, but now completes with our new technique (the case with an asterisk).

Client-Driven analysis analyzes many fewer procedures context-sensitively than the FSCS analysis, giving our technique less room for improvement. Nevertheless, the improvement is generally good. `sendmail` sees only 1.1× speedup in three clients; we believe this is due to memory system effects (disk swapping).

There are some cases where the analysis slows down with our analysis. These cases typically require only a small amount of context-sensitivity in Client-Driven mode, so the overhead of computing Partitioning Vectors and maintaining the hash tables outweighs the benefit. For example, the overhead in the computation of reachable flow values from the parameters is quite large in cases such as `openssh-server` File-Access.

CD vs. RBCS-CD		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**CD**	**RBCS-CD**	×	**used**	**avg**	×
sendmail	File-Access	0								
sqlite		0								
nn		0								
privoxy		0								
cfengine		0								
openssh-server		10	601	415	105.1	138.2	0.8	31	3.1	1.1
openssh-client		3	441	26	21.7	21.8	1.0	10	3.3	1.0
apache		1	313	4	12.8	12.9	1.0	2	2.0	1.0
make		0								
BlackHole		0								
bind		0								
fcron		0								
pureftpd		0								
wu-ftpd-2.6.2		0								
wu-ftpd-2.6.0		0								
muh-2.05d		0								
muh-2.05c		0								
pfinger		0								
stunnel		0								
	Mean:						0.9		2.8	1.0
sendmail	FSV	63	416	1093	66k	58k	1.1	492	7.8	2.9
sqlite		0								
nn		22	494	687	3829	1748	2.2	100	4.5	1.2
privoxy		0								
cfengine		15	421	741	512.6	114.7	4.5	48	3.2	3.2
openssh-server		0								
openssh-client		0								
apache		3	313	16	10.3	10.3	1.0	7	2.3	1.0
make		0								
BlackHole		0								
bind		1	210	9	13.5	13.1	1.0	2	2.0	1.0
fcron		0								
pureftpd		0								
wu-ftpd-2.6.2		7	205	29	37.9	34.8	1.1	23	3.3	1.1
wu-ftpd-2.6.0		5	183	81	17.5	16.5	1.1	13	2.6	1.1
pfinger		0								
muh-2.05d		0								
muh-2.05c		0								
stunnel		0								
	Mean:						1.4		3.7	1.5

(continued in Table 5.6)

Table 5.5: Relevance-Based Context Partitioning for Client-Driven mode (RBCS-CD) vs. Client-Driven analysis (CD), part 1. Legends are the same as in Table 5.2. Blank entries indicate that there is no interesting comparison, because the Client-Driven analysis has determined the benchmark requires no context-sensitive analysis. The ***'s** indicate cases where our technique enables an analysis to complete. The means for all program-clients are shown at the end of Table 5.7.

(continued from Table 5.5)

CD vs. RBCS-CD		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**CD**	**RBCS-CD**	×	**used**	**avg**	×
sendmail	Remote-FSV	68	416	1129	71k	65k	1.1	608	8.9	2.6
sqlite		0								
nn		23	494	694	4043	1880	2.2	112	4.9	1.2
privoxy		0								
cfengine		19	421	806	825.7	275.9	3.06	71	3.7	2.4
openssh-server		0								
openssh-client		2	441	96	41.0	46.2	0.9	5	2.5	1.0
apache		3	313	16	14.8	15.1	1.0	8	2.7	1.0
make		0								
BlackHole		0								
bind		1	210	9	15.9	15.3	1.0	2	2.0	1.0
fcron		0								
pureftpd		0								
wu-ftpd-2.6.2		7	205	29	43.9	40.0	1.1	24	3.4	1.1
wu-ftpd-2.6.0		5	183	81	18.2	16.5	1.1	12	2.4	1.1
pfinger		0								
muh-2.05d		0								
muh-2.05c		1	84	2	1.4	1.4	1.0	1	1.0	1.0
stunnel		0								
	Mean:						1.3		3.5	1.3
sendmail	Remote-Access	68	416	1129	71k	65k	1.1	608	8.9	2.6
sqlite		0								
nn		19	494	326	1888	1619	1.2	80	4.2	1.0
privoxy		10	223	759	894.5	264.9	1.8	35	3.5	1.8
cfengine		14	421	269	709.3	231.2	1.7	52	3.7	1.7
openssh-server		14	601	264	352.1	149.2	1.7	67	4.8	1.7
openssh-client		18	441	318	120.9	78.4	1.4	58	3.2	1.4
apache		17	313	187	33.8	20.9	1.4	45	2.6	1.4
make		0								
BlackHole		2	71	65	72.5	70.2	0.9	7	3.5	0.9
bind		1	210	9	15.8	15.2	1.0	2	2.0	1.0
fcron		0								
pureftpd		4	116	29	3.5	3.5	1.0	12	3.0	1.0
wu-ftpd-2.6.2		8	205	39	45.2	41.2	1.1	27	3.4	1.1
wu-ftpd-2.6.0		4	183	24	18.6	17.9	1.0	7	1.8	1.0
pfinger		1	47	10	2.0	1.9	1.1	2	2.0	1.0
muh-2.05d		0								
muh-2.05c		0								
stunnel		0								
	Mean:						1.4		3.6	1.3

(continued in Table 5.7)

Table 5.6: Relevance-Based Context Sensitive with Client-Driven analysis vs. Client-Driven analysis, part 2.

(continued from Table 5.6)

CD vs. RBCS-CD		Procs			Time(sec)			Contours		Mem
Benchmark	**Client**	|Π|	total	**call sites**	**CD**	**RBCS-CD**	×	**used**	**avg**	×
sendmail	FTP-Behavior	78	416	1339	-	129k	*	758	9.7	*
sqlite		0								
nn		31	494	754	7082	3295	2.1	194	6.3	1.2
privoxy		11	223	764	2472	345.8	7.1	37	3.4	6.7
cfengine		35	421	1210	2144	564.4	3.8	200	5.7	3.3
openssh-server		0								
openssh-client		0								
apache		26	313	296	334.9	36.6	9.2	103	4.0	7.0
make		0								
BlackHole		5	71	111	209.7	114.3	1.8	18	3.6	1.1
bind		4	210	43	20.8	21.8	1.0	10	2.5	1.0
fcron		0								
pureftpd		9	116	262	85.2	6.2	13.8	49	5.4	3.1
wu-ftpd-2.6.2		21	205	418	102.2	70.0	1.5	134	6.4	2.4
wu-ftpd-2.6.0		18	183	410	61.1	26.0	2.4	181	10.1	2.9
pfinger		0								
muh-2.05d		8	84	149	2.3	2.0	1.1	40	5.0	1.2
muh-2.05c		8	84	148	2.2	1.9	1.1	36	4.5	1.2
stunnel		0								
Mean:							2.7		5.5	2.2
All-Mean:							1.6		4.1	1.5

Table 5.7: Relevance-Based Context Sensitive with Client-Driven analysis vs. Client-Driven analysis, part 3.

5.3.4 RBCS versus PTF

We perform a simple experiment to estimate the differences between our RBCS and a Partial Transfer Function (PTF) solution [153], which uses input alias patterns to partition contexts. For example, a procedure with two parameters has two possible input alias patterns: their points-to sets empty or non-empty intersections. We estimate the number of PTFs used by each procedure, by first running the FSCS pointer analysis (without client), then counting the number of contexts of each procedure with distinct input alias patterns.

For the benchmarks for which the FSCS pointer analysis can complete, we estimate that 1.3 PTFs are required per procedure, while RBCS uses on average 1.6

contours per procedure.

Since fewer PTFs are used when alias patterns are used, there is a potential for a PTF solution to lose precision. We explore this possibility with another experiment. By inspection, we found 28 cases (each case is a particular client applied to a particular benchmark) that contain procedures where only one PTF is used but where RBCS uses more than one contour. We then run a FSCS pointer and client analysis on these cases, but forcing these procedures to be analyzed context-insensitively; i.e., we are simulating using one PTF on these procedures. The results, summarized in Table 5.8, show that in seven of the 28 cases, the PTF approach leads to less precise results (the clients report more errors). Note that the result is an underestimation of the PTF's imprecision, because the experiment does not consider other procedures that require multiple PTFs and require more contours than PTFs.

5.4 Coupled Analyses

The key to Relevance-Based Context Partitioning is the use of the client analysis to guide the implementation of the pointer analysis. This general idea bears a strong resemblance to the Client-Driven (CD) pointer analysis. In this section, we first highlight the similarities and differences of these two algorithms. We then discuss the prospects for other Coupled Analyses as a new paradigm for performing difficult analyses.

The RBCP and CD algorithms are both Coupled Analyses in which information from the client analysis guides the behavior of a service analysis, in these cases, pointer analysis. The goals of the two algorithms are orthogonal: The CD algorithm reduces the number of procedures that must be analyzed context-sensitively but does not attempt to partition the contexts for these procedures; the RBCS algo-

Benchmark	Client	change in #reported errors
pfinger	Remote-Access	same
muh-2.05c	Remote-FSV	same
muh-2.05c	FTP-Behavior	same
muh-2.05d	FTP-Behavior	same
pureftpd	Remote-Access	18 → 19
pureftpd	FTP-Behavior	same
wu-ftpd-2.6.0	FSV	same
wu-ftpd-2.6.0	Remote-FSV	same
wu-ftpd-2.6.0	FTP-Behavior	4 → 5
wu-ftpd-2.6.2	FSV	same
wu-ftpd-2.6.2	Remote-FSV	same
wu-ftpd-2.6.2	Remote-Access	same
wu-ftpd-2.6.2	FTP-Behavior	same
apache	FSV	same
apache	Remote-FSV	same
apache	Remote-Access	same
apache	FTP-Behavior	same
BlackHole	Remote-Access	0 → 85
BlackHole	FTP-Behavior	5 → 19
openssh-client	File-Access	same
openssh-client	Remote-FSV	same
openssh-client	Remote-Access	same
privoxy	Remote-Access	same
openssh-server	File-Access	same
cfengine	Remote-FSV	5 → 9
cfengine	Remote-Access	58 → 91
cfengine	FTP-Behavior	5 → 10
nn	FSV	same

Table 5.8: Results of simulations of one PTF on selective programs: the third column shows the effect of the reduced precision on the number of reported errors. Accuracy of the client analyses are reduced in seven cases, because the numbers of reported errors have increased.

rithm attempts to partition contexts for context-sensitive analysis, but it does not attempt to reduce the number of procedures that are analyzed context-sensitively.

The two algorithms differ in one interesting aspect. The CD algorithm requires multiple passes, because it receives its client information at the end of each pass. Such multi-pass algorithms are likely to be common, because in many cases, the client information is most useful after the analysis has converged (i.e., reached a fix point). By contrast, the RBCP algorithm provides a steady supply of information to its pointer analysis, which simply improves its efficiency as the information that it is given converges.

In addition to these two coupled pointer analyses, we can imagine other coupled pointer analyses. For example, Nystrom, et al.'s inlining-based pointer analysis [106] could use a client to guide selective inlining. Moreover, dimensions of pointer analysis other than context-sensitivity could be guided by client information, including the choice between field-sensitive and field-insensitive analysis and the granularity of the heap model.

Beyond pointer analysis, we can imagine other examples of service analyses that could benefit from client information:

- A reaching definition analysis that acts as a service analysis for a constant propagation could partition contexts based on the reaching values of variables, namely, contexts with the same constant reaching value are grouped together.

- Conceptually, a chopping analysis computes the intersection of two service analyses, a forward slicing analysis, and a backward slicing analysis [119]. It may be possible to compute the intersection efficiently by passing information from one service analysis to another, as the boundaries of, say, the forward slice can limit the work of the backward slice.

We now list three necessary requirements for Coupled Analyses:

1. The service analysis must provide tangible intermediate results that are available and useful to the client. For example, the intermediate results of a constraint-based pointer analysis are constraints, which are not readily useful to the client.

2. The client must communicate its needs to the service analysis, including specific information, such as, "I do not want to merge these two flow values."

3. The service analysis needs some means to adjust its behavior in response to the client.

Prospects for the future. Looking to the future, many service analyses are non-separable and extremely difficult to scale. For example, after many years of study, extremely precise pointer analysis remains an elusive problem. Its importance, however, is only likely to grow, as programs become increasingly complex and as new problems appear, such as the need for tools that can help parallelize pointer-based codes for multi-core chips. Thus, as the prospects for scalable and precise standalone solutions diminish, it makes sense to explore ways in which additional information can assist these analyses. The paradigm of Coupled Analyses is one such approach, so we believe that its importance will only continue to grow..

5.5 Conclusion

As the desire for large robust software continues to increase, so does the demand for precise and scalable service analyses, such as pointer analysis and the computation of reaching definitions, which are useful for problems such as security analysis, program

slicing, and other analyses for program understanding. This chapter contributes one technique for improving the scalability of context-sensitive service analyses.

In particular, we have introduced the notion of Relevance-Based Context Partitioning, which partitions contexts for pointer analysis into equivalence classes based on the computed flow values of a client analysis. Our technique allows irrelevant information to be merged but ensures that relevant information is never merged. Consequently, RBCP reduces great amounts of unimportant computations, so that performance and memory utilization are improved without sacrificing precision.

To show the power of our technique, we have used it to improve the efficiency of a set of flow- and context-sensitive error-checking analyses. Our results show that Relevance-Based Context Partitioning greatly improves the scalability of these analyses. Without our technique, only the seven smallest of our 19 benchmarks could be successfully analyzed. With our technique, all except one benchmark could be successfully analyzed; the last benchmark `sendmail` could complete on two out of five clients. The average speedup among the seven smallest benchmarks is 7.0×. When combined with the Client-Driven pointer analysis, our technique allows us to analyze the 69K line `sendmail` program, and it reduces the analysis time of all larger benchmarks by an average of 1.6×.

Finally, we observe that Relevance-Based Context Partitioning is the second example where information from the client analysis can improve the performance of a service analysis, and we argue that this is a paradigm that is likely to become more important in the future.

Chapter 6

Reachability-Based Analysis

6.1 Introduction

In the previous two chapters, we present evidence of large amounts of unimportant computations in dataflow analyses. We are interested to know if unimportant computation also exists in other forms of program analyses. In this section, we explore that question in reachability-based analysis.

Under this paradigm, dataflow facts at statements are encoded as nodes of a graph, with edges representing how those facts flow or are transformed between statements. Solving the analysis problem reduces to determining what nodes are reachable from a set of predetermined initial "start" nodes.

For better precision, the analysis problem is transformed into a special kind of graph-reachability problem, instead of an ordinary transitive-closure reachability problem. The generalized reachability problem involves the constraint that only *interprocedurally realizable paths* can be considered in the analysis.[1] A realizable

[1]Sometimes the equivalent class of *context-free-language reachability problems*, or *CFL-reachability* problems [120], is used in the literature instead.

path mimics the call-return structure of a program: if a path contains a call edge (caller-to-callee) and a corresponding return edge (callee-to-caller), then the return site must match with the call site, i.e., back to the original caller. Without the constraint, propagation of dataflow information along unrealizable paths can lead to spurious, and therefore imprecise, analysis results.

The class of interprocedural, finite, distributive, subset dataflow problems (IFDS) is the most widely cited example when the dataflow problems are transformed into graph-reachability problems. The class of IFDS problems is important because it includes many widely used interprocedural problems, including locally separable problems [77]. The algorithm by Reps, Horwitz, and Sagiv [121] to solve IFDS problems has been widely cited [5, 11, 12, 13, 34, 35, 37, 39, 40, 41, 46, 50, 59, 76, 86, 92, 89, 95, 112, 113, 116, 127, 139] because it performs a precise analysis—it is both flow-sensitive and context-sensitive—in polynomial rather than exponential time. After encoding a dataflow problem as a graph, call an *exploded graph*, the problem can then be solved in $O(ED^3)$, where E is the number of edges in the graph, and D is the maximum number of local and global variables in any procedure.

Despite its theoretical elegance, the practical limits of the IFDS algorithms are not well studied. In particular, there are three potential limitations:

1. The cubic time complexity assumes a data structure that provides constant-time cost for certain common operations, such as membership tests. This constant-time cost can be achieved by using random-access $O(ND^2)$ arrays, but the space complexity of such arrays becomes infeasible as the program size grows. Our results show that often less than 1% of each such array is used, so it is difficult to justify the use of such a data structure. In fact, the algorithm per se cannot complete beyond our few smallest programs.

2. The presence of pointers introduces a large number of invisible variables that are accessed through pointers, so the value of D can grow with the program size. With a $O(ED^3)$ complexity, the analysis does not scale well to large programs.

3. Third, the presence of pointers complicates the creation of the exploded graph, and the impact of pointer analysis precision on this construction is not well studied.

6.1.1 Contributions

To address the three limitations described in previous section, we make two sets of changes to the algorithm presented by Reps et al. [121] The first set makes some refinement to the data structures used in the algorithm. The more space-efficient internal data structure helps to cope with the memory requirement. We present these changes in Section 6.2.3, after we summarize the original algorithm. Applying these changes enable the algorithm to complete our larger benchmarks. For this reason, we will refer to this refined algorithm as our *baseline* algorithm.

Section 6.2.3 also briefly explains the effects of pointer analysis precision on the IFDS analysis, and how we choose which pointer analysis to use in our experiments.

The second set, which is the bulk of our contributions, consists of a collection of new algorithms. They are derived from two new algorithms we devised. By combining them and by introducing new variants, we obtain a diverse set of algorithms that improve performance without sacrificing precision. We briefly summarize two new algorithms as follow:

1. We present the *Sparse IFDS* algorithm, which reduces the size of the exploded

graph by representing data dependences instead of control dependences. For our benchmark suite, we find that the exploded graphs produced by these sparse algorithms use only, on average, 11% of nodes used by the original algorithms. As a result, the Sparse IFDS algorithm is on average 2.6× faster than the baseline algorithm. We present the detailed algorithm in Section 6.3.

2. The second algorithm, *Variable-Pruning IFDS* algorithm, incorporates a lightweight slicing algorithm that determines the set of variables that do not contribute to answering any queries raised by a client analysis. These variables are excluded, or pruned away, during exploded graph construction. On average, the new graphs use 32% of nodes used by the baseline algorithm. As a result, the analysis is 3.3× faster than the baseline algorithm. In Section 6.4, we will present in detail the algorithm, and its differences from the Demand IFDS algorithm [121].

Both algorithms address the presence of unimportant work in IFDS analysis. The Sparse IFDS algorithm avoids creating nodes and edges where there are no effects on the associated flow values, while the Variable-Pruning IFDS algorithm is driven by avoiding nodes whose flow values are not used in any queries. The reduction in the graphs directly reduces the amount of unnecessary propagations during reachability analysis.

The two algorithms are orthogonal, and each can yield better performance over the other in our suite of benchmarks. We can also combine them to yield even better performance; on average, the combined algorithm's analysis time is 5.5× faster than the baseline algorithm, with no precision loss.

The performance results cited above are based on using a fixed FSCI pointer analysis prior to the IFDS analysis. Since the same pointer analysis is used, these

comparisons use IFDS analysis time only and exclude time spent in pointer analysis.

In order to achieve good performance in total pointer and IFDS analysis time, we also try using other pointer analyses. Specifically, we use a multiple-passes multiple-pointer analysis framework. The objective is to let a client analysis provide *feedback* allowing the Variable-Pruning IFDS analysis to prune away even more variables.

When we compare total pointer and IFDS analysis time, among the algorithms we tried, the best algorithm is the combined Sparse Feedback-based Variable-Pruning algorithm; on average, it is 2.6× faster than the baseline algorithm. All algorithms do not lose precision compared to the baseline.

An important question is, how does the IFDS analysis compare to a dataflow analysis? For this purpose, we compare our best IFDS algorithms with the Client-Driven analysis [54], one of the most scalable and precise dataflow analyses. Our experiment shows that our best IFDS algorithm is on average 1.8× *slower* than the Client-Driven analysis.

Finally, at the end of this chapter, we will present our attempts to apply cycle-elimination techniques to IFDS analysis. The motivation is based on the observation that strongly connected components in the exploded graph are redundancies, because if one node is reachable, all nodes in the same component are reachable. Collapsing all nodes in a component, and replacing them with a single representative node, can lead to more efficient reachability analysis. The main result of this exercise is that despite their apparent attractiveness, these techniques do not help IFDS analysis in practice. The reasons are (1) graph construction is a major component, so cycle detection after the graph is constructed is too late; (2) high overhead in detecting cycles; and (3) overshadowing by the Sparse IFDS algorithm, which already

avoids creating many cycles. But even when an FICI pointer analysis is used, so that the Sparse IFDS algorithm is not applicable, there is still no improvement over the baseline.

The rest of this chapter is organized as follows. Section 6.2 summarizes the original IFDS algorithm and, after we apply some modifications, a baseline algorithm. The next two sections explain the Sparse and Variable-Pruning IFDS algorithms in details, respectively. Section 6.4.4 discusses different analysis configurations that incorporate our new algorithms. Section 6.5 presents our experimental results. Section 6.6 summarizes our attempt to incorporate cycle elimination techniques. Finally, Section 6.7 concludes the chapter. We review related work later in Section 7.4.

6.2 IFDS Baseline Algorithm

Reps et al. [121] present the IFDS algorithm to solve the class of *IFDS problems*. In this chapter, we will present our new algorithms to solve the same class of problems. For convenience, we will call an execution of any of these algorithms to solve a given problem an *IFDS analysis*. This execution includes all initialization steps—including graph construction—and the reachability analysis on the graph.

Before we present our algorithms, we start with this section that briefly explains the class of IFDS problems, and highlights the essential features of the original IFDS algorithm by Reps et al. We also discuss some practical issues and our modifications. We will call the result of applying these modifications our IFDS baseline algorithm, or simply, *the* IFDS algorithm. We sometimes also denote it by *IFDS*.

6.2.1 IFDS Problems

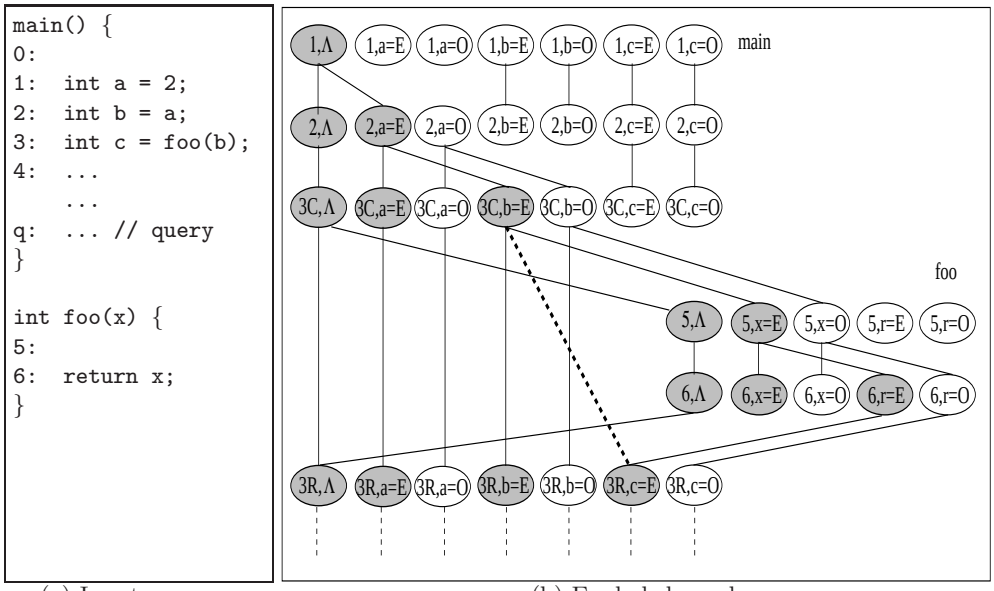
The objective of an IFDS analysis is to find precise and efficient solutions to the class of *interprocedural, finite, distributive, subset problems*, or *IFDS problems*. In this class of problems, the set of *dataflow facts*, or *dataflow values*, is a finite set, and the dataflow functions distribute over the meet operator (either union or intersection, depending on the problem). The class contains all locally-separable problems as well as many other non-separable problems. This last property, plus the existence of polynomial algorithms to solve them, make the IFDS algorithms attractive and popular. Unfortunately, there are still many important analysis problems, such as pointer analysis, that do not fall into this category.

6.2.2 The Original IFDS Algorithm

Overview

Rep et al.'s algorithm adopts the functional approach to precise interprocedural analyses as described by Sharir and Pnueli [136]. In the algorithm, a program is represented by a supergraph composed by a collection of flow graphs, one per procedure. There is also a finite data-set D_p per procedure, where each element $d \in D_p$ represents a dataflow value. For example, suppose the analysis problem is to determine integer parities, then one way is to let each d take the form "$x = Odd$" or "$x = Even$" for each variable x in the procedure. Therefore, the size of D_p depends on the number of variables and the fixed number of property values (Odd and $Even$) in analysis problem.

Using the flow graphs and D_p, the exploded graph is constructed as follows: for each node n in the flowgraph of procedure p, and for every element $d \in D_p$, there is a node $\bar{n} = \langle n, d \rangle$ in the exploded graph. The edges on $\bar{n}$ are created using the



(a) Input program (b) Exploded graph

Figure 6.1: Exploded graph constructed for an example program, for the dataflow problem that determines parity of integers. The elements of data-sets are of the forms $a = E$ or $a = O$. For example, the node $\langle 2, a = E\rangle$ means the fact "variable a is even just before statement 2." Each call site has two set of nodes: for the call at statement 3, the nodes labeled 3C represent the call to callee, while the nodes labeled 3R represent return from callee. The symbol r represents the return value of `foo`. Λ is a special symbol not in any data-set, so that the node $\langle 0, \Lambda\rangle$ is the "start" node for the reachability analysis. Reachable nodes from this start node are colored gray. The dash edge is a summary edge in `main`, which is created due to the path $\langle 5, x = E\rangle \rightarrow \langle 6, r = E\rangle$ in the callee `foo`.

edges on n as well as the transfer function on d at n. Figure 6.1 shows part of an exploded graph of an example program for the dataflow problem that determines the parity of integers.

The algorithm then proceeds to perform reachability analysis, by computing *path edges* on the exploded graph. Each path edge is in the form

$$\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle,$$

where s_p is the entry node for a procedure p, and n is a node in the same procedure. The presence of a path edge indicates that $\langle n, d_2 \rangle$ is reachable from $\langle s_p, d_1 \rangle$, which translates to the propagation of dataflow information from s_p to n.

If n is the exit node, then a *summary edge* is also created at the corresponding call-return pair of nodes in the callers of p:

$$\langle n_c, d_1 \rangle \rightarrow \langle n_r, d_2 \rangle,$$

where n_c, n_r are the call and return nodes in a caller. These summary edges are then used in the computation of path edges in the callers. Because n_c and n_r must belong to same caller, the summary edges allow propagation of flow information only along interprocedurally realizable paths, thereby making the IFDS analysis context-sensitive.

Since reachable nodes are computed for each statement, the IFDS analysis is also flow-sensitive. The set of reachable nodes at a statement captures the solution at the statement. For example, if the node $\langle q, c = E \rangle$ is reachable while $\langle q, c = O \rangle$ is not, then the variable c is even at statement q.

6.2.3 Practical Issues and Modifications

In this section, we discuss three practical issues pertaining to the IFDS analysis: the presence of pointers in programs, graph construction algorithms, and internal data structures.

Pointers

The data-set D_p is dependent on the dataflow problem on-hand, but it usually contains all the visible variables in the procedure p. In the presence of pointers,

we have to decide how to treat pointer dereferences, since they sometimes represent invisible variables and heap objects. One possibility is to use symbolic names. The advantage is D_p remains dependent only on the procedure alone. The disadvantage is, in order not to lose precision, the algorithm must now handle possible alias among input parameters. We could adopt a strategy similar to that used in the Partial Transfer Functions [153], where one solution is computed per input alias pattern.

The other possibility is to determine (via a pointer analysis) all the actual targets of each dereference, and include them in the data-set. This approach is dependent on the rest of the program, so that D_p may grow with the size of the program. While the algorithm is still $O(ED^3)$, the pathological worse case is now $O(EN^3)$.

Because the first approach can complicate the original IFDS algorithms in such ways that we are no longer evaluating the IFDS algorithms in their normal form, in this chapter we choose the second approach.

Incremental Graph Construction

The exploded graph is constructed using each procedure's flow graph and data-set D_p. The possible elements for D_p are the local and global variables and, in the presence of pointers, other invisible and heap variables. This set is too big for large programs. To build a minimal exploded graph, D_p should only include those variables actually used in the procedure. Our strategy is to start with a minimal initial D_p and allow it to grow incrementally.

The initial D_p consists of local variables and all inputs to the procedure. The inputs are the parameters as well as pointer-derived hidden and heap variables. To

acquire these inputs, we charge the pointer analysis the responsibility to collect the set of all inputs to a procedure that is actually used in the procedure.

The initial sets for all procedures are used to construct the exploded graph. For each procedure p and D_p, we invoke the routine `explode_proc` in Figure 6.2. The routine uses the entry node s_p to generate an initial set of exploded graph nodes, which are put in a worklist. From each node $\langle n, d \rangle$ taken from the worklist, the routine `explode` creates new nodes and target nodes by applying the transfer functions on n and d. New unexploded nodes are also put back on the worklist. At a call site, new flow values are exported to the callee's D set. At a call return site, new values are exported back to the caller. The data-set grows when a new value is created by a transfer function, or when it is imported from another procedure. This incremental graph construction algorithm also has the additional advantage that if a new value d is discovered at node n, we do not use d to explode statements before n.

Data Structure: Space and Time Tradeoffs

To enable unit-time tests for membership, the original algorithm uses arrays to store all the path and summary edges. This requires $O(ND^2)$ storage space per procedure, where N is the number of nodes. (It already takes into account all path edges have source nodes in the form $\langle s_p, d \rangle$, where s_p is a start node.) Unfortunately, because D can be large, this scheme causes the algorithm to quickly run out of memory. A study on a few selected small programs reveals that sometimes less than 1% of one such array is actually used. This is not surprising since not all pairs of nodes in the exploded graph are valid path edges.

To overcome this problem, we are forced to trade in time for space, by replac-

```
explodeProc(p, D) {
  s_p = start node of p;
  for d ∈ D do {
    s̄ = ⟨s_p, d⟩;
    if s̄ not marked as exploded
      add s̄ to WL_p; // worklist
  }
  while WL_p not empty {
    remove ⟨n, d⟩ from WL_p;
    new_nodes = explode(⟨n, d⟩);
    WL_p ∪= new_nodes;
    if n is CALL {
      q = callee(n);
      if export(n, q, d)
        explodeProc(s_q, {q});
    } elif n is EXIT {
      q = caller(n);  // q is RET node
      if export(n, q, d)
        WL_q ∪= explode(⟨q, d⟩);
    }
  }
}
```

```
explode(n̄) {
  mark n̄ as exploded;
  let n̄ = ⟨n, d⟩;
  for s ∈ successors(n) do {
    // s, n are in same procedure
    targets = apply_xfer(n, s, d);
    for d′ ∈ targets do {
      s̄ = ⟨s, d′⟩;
      add_edge(n̄, s̄);
      if s̄ not marked as exploded
        add s̄ to new_nodes;
    }
  }
  return new_nodes;
}
```

Figure 6.2: Constructing exploded graph.

ing the arrays with log-time sets. Specifically, we use the following multi-dimensional arrays of sets to represent path edges:

$$\begin{aligned} PathEdges &\rightarrow \text{array } 1 \ldots P \text{ of } DVec \\ DVec &\rightarrow \text{array } 1 \ldots N \text{ of } DPairs \\ DPairs &\rightarrow \text{set of } \langle d_1, d_2 \rangle \text{ pairs} \end{aligned}$$

where P is the number of procedures, and N is the number of flowgraph nodes per procedure. In the original algorithm, $DPairs$ is a two-dimensional array, so that the membership test on a path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is reduced to a boolean test on $PathEdge[p][n][d_1][d_2]$. After we replace $DPair$ with a set, each membership test and insertion operation has a $O(\log D^2)$ cost. Consequently, the overall analysis

complexity becomes $O(ED^3 \log D^2)$.

6.2.4 Effects of Pointers on IFDS Analysis

It is well known that the precision of a pointer analysis has significant impact on the subsequent analysis dependent on it. What is not necessarily clear is the *degree* of that impact on different analyses. As far as we know, there is no previous study of the impact on the IFDS analysis. The immediate questions we face now are: (1) how exactly does a pointer analysis affect an IFDS analysis? and (2) which pointer analysis should we use in order to properly evaluate the baseline IFDS algorithm and our new algorithms? This section attempts to answer these two questions.

A pointer analysis affects the IFDS analysis (or any other analysis for that matter) in competing ways, and there does not exist a best pointer analysis that fits all situations. Perhaps the most important and obvious factor is the scalability of the pointer analysis. If it cannot or takes too long to complete analysis on a given program (e.g., requires too much memory), then we cannot even perform the IFDS analysis.

On the other hand, a very scalable pointer analysis often produces larger points-to sets. Such imprecise results can cause the IFDS analysis to slow down. For example, in statement `s:x=*p`, the number of nodes and edges in the exploded graph created for that single statement is proportional to the size of points-to sets of the dereferenced pointer. Figure 6.3 shows how the graphs look when the points-to set has size 1 and 2. Note that nodes and edges for subsequent statements are also affected. Not only does the IFDS analysis possibly create a much larger graph for a small increase in points-to set, the precision of the IFDS analysis also suffers. Using the same example, the extraneous path to query at node $\langle n, z \rangle$ in Figure 6.3(b) can

cause the analysis to answer true to a query, when the answer should be false.

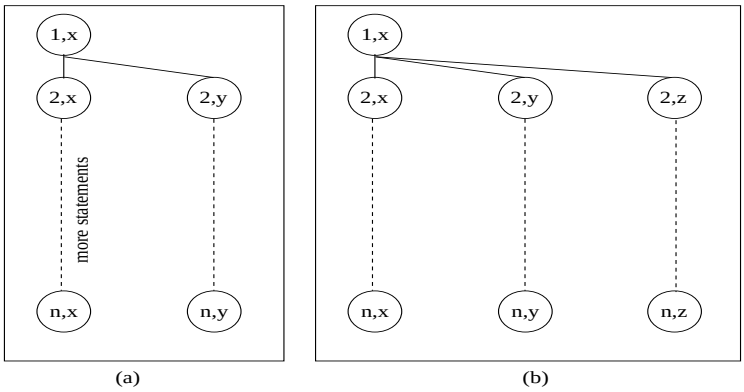


Figure 6.3: Effect of pointer analysis precision on IFDS analysis on a statement `1:x=*p`. If the points-to set for `p` is {`y`}, the graph in (a) is constructed; if instead the points-to set is {`y,z`}, the larger graph in (b) is constructed, with more nodes and edges from statement 2 downward.

Later, in Section 6.5.2, we will quantify the effects of pointer analysis on IFDS analysis. In that experiment, we compare three IFDS analyses that use pointer analyses with different precision modes: a flow-insensitive context-insensitive analysis (FICI), a flow-sensitive context-insensitive analysis (FSCI), and a flow-sensitive context-sensitive analysis (FSCS). The key results are (1) FSCS is too expensive and can complete analysis only in a small subset of programs; (2) among our benchmarks, FSCI does not cause IFDS analysis to lose precision compared to when FSCS is used; and (3) FICI is too imprecise, and both performance *and* precision of IFDS analysis suffers substantially. For these reasons, for the remainder of this chapter, unless otherwise stated, we will assume an FSCI pointer analysis is used prior to any IFDS analysis.

We are now ready to present our new Sparse and Variable-Pruning IFDS algorithms.

6.3 Sparse IFDS

In this section, we present our new Sparse IFDS algorithm, denoted by *Sparse-IFDS*. We first explain how the lack of sparsity in the original algorithm leads to inefficient analysis, before we present the Sparse IFDS algorithm. The results of the algorithm are presented later in Section 6.5.3.

6.3.1 Motivation

In the IFDS algorithms, a node n in the supergraph is exploded into many copies of $\bar{n} = \langle n, d \rangle$ in the exploded graph, where $d \in D$. The out-edges incident on $\bar{n}$ depends on the transfer function t on the node n. Considering only intraprocedural edges, the side effect of n is usually on a small set of variables. For this reason, for many nodes $\bar{n}$, all out-edges on the node are in the form:

$$\langle n, d \rangle \quad \rightarrow \quad \langle s, d \rangle \quad s \in \text{successor}(n), \tag{6.1}$$

i.e., they share the same d value. These edges represent a "no change" semantic of d at n. By induction, many nodes $\bar{n}$ also have only one type of in-edges in the form:

$$\langle p, d \rangle \quad \rightarrow \quad \langle n, d \rangle \quad p \in \text{predecessor}(n) \tag{6.2}$$

Because most statements have only one successor and one predecessor each, these edges together form many non-trivial paths where there is no branching along each path. Nodes along these paths are redundant and could be bypassed. The redundancies lead to inefficiency in space and time usage, both in building the graph as well as in the reachability analysis.

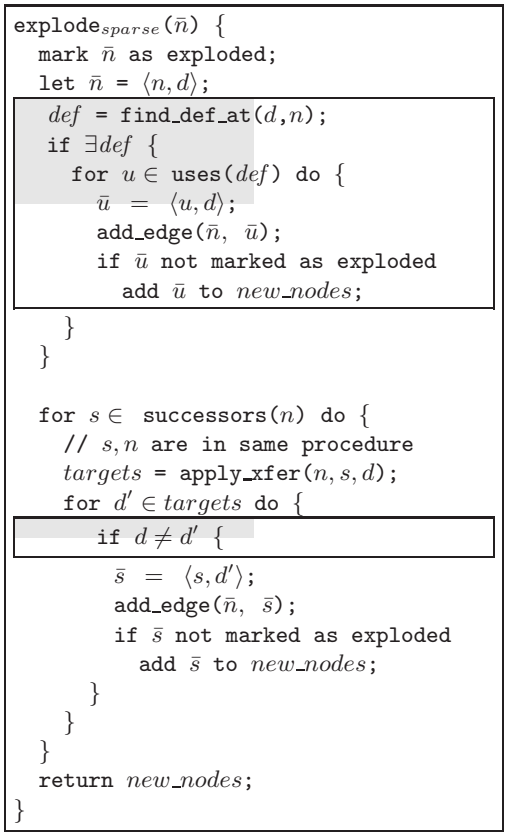


Figure 6.4: Constructing sparse exploded graph, modified from Figure 6.2, with new or modified statements highlighted with gray boxes. If a def for d exists at node n, new nodes and edges are constructed using the def-use chains.

6.3.2 Algorithm

Our Sparse IFDS algorithm solves this problem by observing that if a node $\bar{n}$ has any out-edge not in the form of (6.1), then d is used at n to modify another flow value $d' \neq d$. Similarly, if the node has any in-edge not in the form of (6.2), then d is modified at n. Therefore, if we get rid of the redundant nodes (and edges) described earlier, we are left with nodes $\bar{n}$ such that there is either a modification (def) or a use on the value of d at n. Therefore, for each d, we only need to create nodes at its def and use sites, and connect the nodes with edges corresponding to def-use chains.

We charge the responsibility of building the def-use chains to the pointer analysis, which was performed before the IFDS analysis begins. Specifically, the pointer analysis must be able to answer a query "what are the uses of a given def?" Therefore, the new Sparse IFDS algorithm requires a flow-sensitive pointer analysis. To handle interprocedural edges correctly, we require the following def/use chains to be created for d as well:

- at a call node c with a corresponding callee p whose entry node is s_p, if the variable for d is exported to (used in) the callee, then the variable must have a use at c and a def at s_p, and the edge $\langle c, d\rangle \rightarrow \langle s_p, d\rangle$ is added to the exploded graph.

- at a return node r with a corresponding callee p whose exit node is e_p, if the variable for d is exported to the caller, then the variable must have a use at e_p and a def at r, and the edge $\langle e_p, d\rangle \rightarrow \langle r, d\rangle$ is added to the exploded graph.

The Sparse IFDS algorithm, shown in Figure 6.4, replaces the routine `explode` in the original algorithm (Figure 6.2) with a new version $\texttt{explode}_{sparse}$. The first part of the new routine checks if there is a def for d at n; if yes, the def-use chains are used to create new nodes and edges. The second part of the routine is the same as the original version, except for an additional check to avoid adding redundant edges.

6.4 Variable-Pruning IFDS

The Sparse IFDS algorithm works by creating fewer nodes for each statement, by using only variables used or modified at a statement. Another way to further reduce the graph is to not use those variables that do not contribute to the analysis queries.

The key is an inexpensive way to determine which variables to exclude. We achieve this by incorporating a new technique call Variable-Pruning, and we call the new algorithm *Prune-IFDS*. In this section, we describe the algorithm, and explain how it differs from Sparse IFDS analysis and Demand IFDS analysis [121]. Finally, we also describe a variant to the algorithm, which we call Feedback-based Variable-Pruning analysis.

6.4.1 Algorithm

Suppose an analysis makes a query in the form "what is the value of x at statement $\mathtt{s}$?" The value of x depends on values of y's such that there are assignments in the program where x is modified and y is accessed on the right-hand side. We let $x \rightsquigarrow y$ to denote this dependence. The transitive closure of such dependences is, therefore, the set of variables that *contribute* to the query. All variables that do not contribute to any query can be safely dropped and not used in constructing the exploded graph.

We do not need to include pointer variables when we compute the dependences. This is because all pointer dereferences must be resolved before constructing the exploded graph. For example, each assignment in the form "$x = *p$" is logically replaced by a set of assignments "$x = y$" for each y in the points-to sets of p. These assignments are then used to construct the dependences, and the pointer variable p is not used.

Our algorithm involves constructing a variable dependence graph (VCG), where the nodes are variables and the edges are the dependences. The easiest way to compute a dependence graph is to assign the task to the pointer analysis: generate the dependence edges for each assignment in the program. This is shown as

```
AddDependence(assign, deps) {
  for var l in LHS(assign) do
    for var r in RHS(assign) do
      add (r ⇝ l) to deps;
}
```

```
PruneVariables(deps,queries) {
  Set incl = ∅;
  for q in queries do
    for var v in q do
      Set r = all_reachables(v, deps);
      add r to incl;
  return incl;
}
```

Figure 6.5: Variable-Pruning IFDS Algorithm. `AddDependence` is called at each assignment during a pointer analysis. After the dependence graph is completed, given a set queries, `PruneVariables` compute the set of variables that should be included in the subsequent analysis.

`AddDependence` in Figure 6.5. The added computational cost is small per statement, while the additional memory overhead is usually much less than the upper limit of $O(V^2)$, where V is the number of variables. When the dependence graph is complete, given a set of queries, `PruneVariables` can then compute the set of variables that should be included in the subsequent analysis.

Note that using the dependences to prune variables is effectively a lightweight version of a slicing algorithm [144]. The latter, in general, may prune away more variables, but such algorithms are also more expensive.

6.4.2 Difference from Sparse IFDS

The Sparse IFDS algorithm and the Variable-Pruning IFDS algorithm share the same goal of avoiding nodes that are not needed, because they do not carry relevant data flow. Their approaches are different and can be combined to yield even better results. This section explains their strengths and differences with an example.

Figure 6.6(a) shows a simple flowgraph with five nodes. There are three variables x, y, z, leading to an exploded graph with 15 nodes, shown in Figure 6.6(b). There are assignments between the variables, leading to edges $\langle 1, x\rangle \rightarrow \langle 2, y\rangle$ and $\langle 3, z\rangle \rightarrow \langle 4, x\rangle$. These are the only edges between different variables. The client

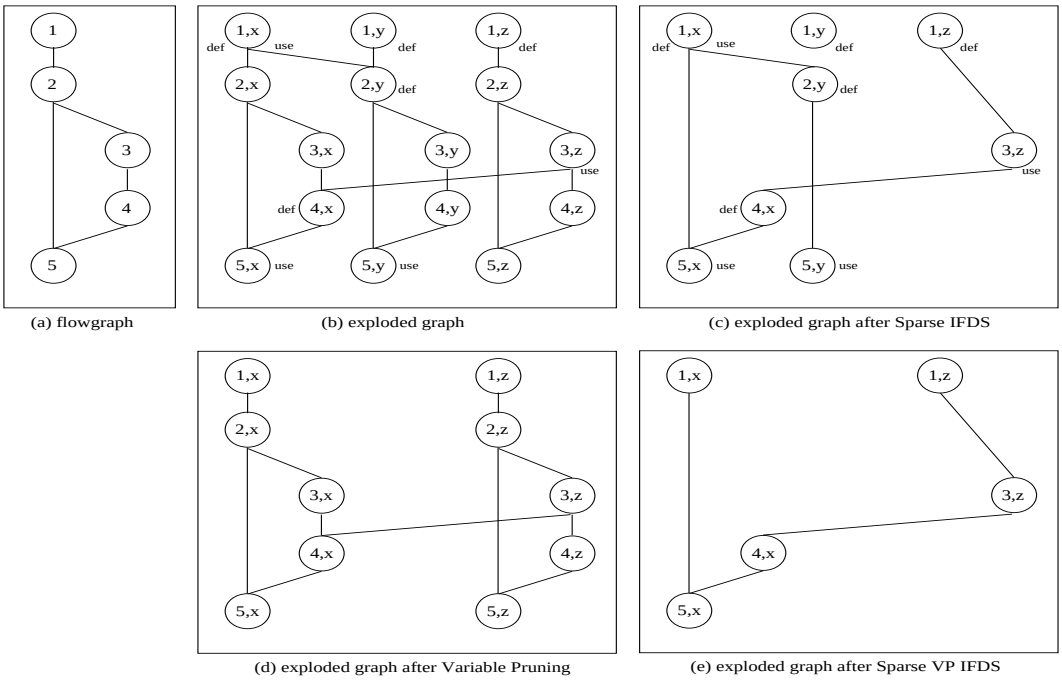


Figure 6.6: Difference between Sparse IFDS algorithm and Variable-Pruning IFDS algorithm. In (c), the graph is determined by the def-use chains. In (d), assuming there is a query on value x at node 5, only the variables $\{x, z\}$ are used to construct the graph. The combined algorithm yields the graph in (e)

analysis also makes a single query on value of x at node 5.

Figure 6.6(c) shows the graph constructed by the Sparse IFDS algorithm. For each variable, the algorithm only creates nodes at its def and use sites, and connects them with edges using def-use chains. These are global variables, and therefore, by default, each of them has a def at the procedure entry: i.e., statement 1 serves as the merge point for collecting flow values from different callers. The same statement can also be a use site, as in the case of $\langle 1, x\rangle$, where x is used in the assignment to y. Note that all three variables are still present, and that the client query does not play a role in the algorithm.

The Variable-Pruning IFDS algorithm, on the other hand, determines that in order to answer the client query, it needs to include the values of $\{x, z\}$. It can exclude $\{y\}$ because y does not contribute to answering the query. Armed with this information, it constructs a different exploded graph, as shown in Figure 6.6(d). The size of this graph depends on the set of queries and variable dependences. Although in this example the graph resulting from Variable-Pruning is larger, in general if there are many other y's that Variable-Pruning can prune away, and if these y's have more def-use chains, then Variable-Pruning can produce smaller graphs than Sparse IFDS.

Since the two algorithms work in different ways but with the same objective, they can be combined to yield better results. When applied to the example, the result, shown in Figure 6.6(e), has fewer nodes than earlier results.

6.4.3 Difference from Demand IFDS

Demand IFDS [121] is a variant of the IFDS algorithm that has an implicit built-in slicing mechanism. It is, therefore, interesting to compare its difference from the Variable-Pruning IFDS algorithm. This section briefly describes how the Demand IFDS algorithm works, and explains how it differs from the Variable-Pruning IFDS algorithm.

The Demand IFDS algorithm is, in a nutshell, an IFDS analysis but with a backward direction of analysis. It starts at the client queries, and performs graph construction and reachability analysis backward. That is, given a node $\langle n, d\rangle$, it determines its immediate predecessors:

$$\langle p, d'\rangle \rightarrow \langle n, d\rangle \tag{6.3}$$

where p is a predecessor in the flowgraph. The algorithm iteratively find new nodes and edges until no more nodes can be added. During this process, the algorithm checks if the graph contains any node $\langle s_{main}, d\rangle$ where d is any dataflow value known to be a fact at the start of the program. If yes, the algorithm stops, and the query is answered yes.

By the nature of computing predecessors using (6.3), the algorithm will compute nodes using only dataflow values contributing to a query. That is, its side effect is to prune away values that do not contribute to any query. The net effect, however, is different from that of the Variable-Pruning IFDS algorithm due to two key differences:

1. The Variable-Pruning IFDS algorithm (*Prune-IFDS*) ignores control dependences when computing variable dependences. That is, if x depends on y, which depends on z, the dependence graph implicitly includes the dependence $x \rightsquigarrow z$, even if the order of assignments excludes such dependence. On the other hand, the dependences used in the Demand IFDS are flow-sensitive. This difference can be fixed by using a more precise slicing algorithm in *Prune-IFDS*.

2. Even when the two algorithms use the same set of variables, their results can still be different. The fundamental reason is due to different direction in the graph construction.

We now provide more details on the second difference by using an example. Figure 6.7(a) shows a simple flowgraph with a branch at the end of 100 statements. It contains two assignments on variable z, one assignment on variable y, and no assignment on variable x. Suppose we wish to perform an Uninitialized-Variable analysis problem, with the following query: is variable z initialized at node 103? The

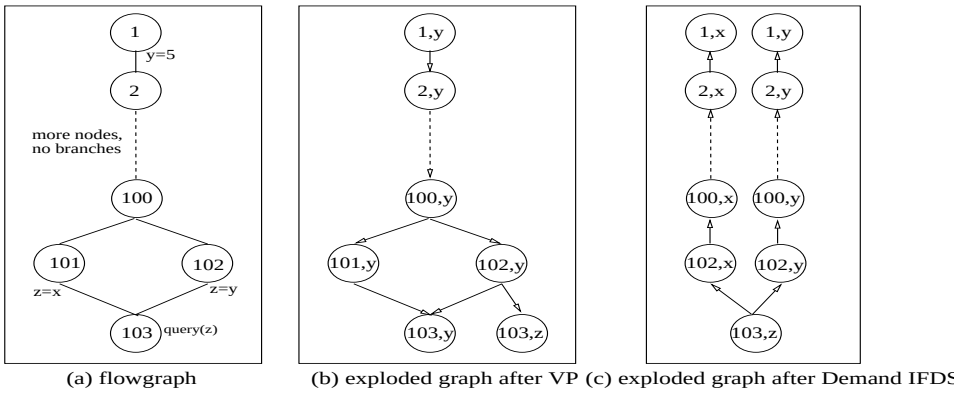


Figure 6.7: Difference between Demand IFDS algorithm and Variable-Pruning IFDS algorithm. This example assumes an Uninitialized-Variable analysis problem, and that there is a query on value z at node 103. The arrows indicates the direction in which the nodes are created.

analysis uses the following convention: a node $\langle n, v\rangle$ means variable v is definitely initialized at statement n.

Under the Variable-Pruning IFDS algorithm, the three variables are not pruned away. The corresponding exploded graph is shown in Figure 6.7(b). Starting from the assignment at statement 1, nodes are added in a forward direction until the last statement. The arrows show the direction nodes are added. In the end, because $\langle 103, z\rangle$ is reachable from the start, the answer to the query is yes.

On the other hand, the Demand IFDS builds the graph by starting from the query, and new nodes are added in a backward direction. In the example, due to the assignment $z = y$, the query "is z initialized at 103" is transformed to "is y initialized at 102?" This process repeats until no more node can be added, at which point we verify that $\langle 1, y\rangle$, a known fact, is reachable, and therefore, the answer to the query is yes.

Though they use the same set of variables, the graphs they create are differ-

ent, because they compute reachable nodes from different initial nodes in different directions. In this example, the Demand IFDS creates a larger graph, but in general, either algorithm could generate a bigger graph. This is not surprising since the nodes in the forward analysis represent all feasible states, but not all are relevant to the query; on the other hand, some nodes in the backward analysis can represent infeasible states (such as $\langle 1, x \rangle$).

6.4.4 Analysis Configurations and Feedback-Based Analysis

We now discuss different ways to incorporate the Sparse IFDS algorithm and the Variable-Pruning IFDS algorithm into a pointer and IFDS analysis configuration. The minimal setup is to run a pointer analysis first, followed by Variable-Pruning (VP), before performing the IFDS analysis (by default we use an FSCI pointer analysis). The variable dependence graph is built during pointer analysis. After pointer analysis, VP computes the set of variables reachable in the dependence graph, starting from those variables used in queries in the program pertaining to the analysis problem on hand. Variables not in the reachable set are then pruned away from the IFDS analysis. We call this algorithm *Prune-IFDS*.

Another algorithm, *Prune-Sparse-IFDS*, is derived by combining the Sparse IFDS algorithm with *Prune-IFDS*. These two, together with the baseline algorithm, are shown in the first three rows in Table 6.1.

In addition, we also use a different strategy to derive another set of new algorithms. These are the two *Feedback-based* algorithms in the last two rows of Table 6.1: *Feedback-Prune-IFDS* and *Feedback-Prune-Sparse-IFDS*. Each of them uses multiple pointer analyses, in pass 1 and pass 2. This setup is similar to the framework adopted by a Client-Driven analysis [54] but are different in objective. The

Algorithm	Analysis Passes			
	1	**1+**	**2**	**3**
Baseline *IFDS*	FSCI Ptrs		IFDS	
Prune-IFDS	FSCI Ptrs +VDG	VP	IFDS	
Prune-Sparse-IFDS	FSCI Ptrs +VDG	VP	Sparse IFDS	
Feedback-Prune-IFDS	FICI Ptrs+Client +VDG	VP+	FSCI Ptrs	IFDS
Feedback-Prune-Sparse-IFDS	FICI Ptrs+Client +VDG	VP+	FSCI Ptrs	Sparse IFDS

Table 6.1: Setups for applying Variable-Pruning (VP) to IFDS analysis. The algorithm in each row is broken down into two or three passes. VDG means that the pointer analysis also computes the variable dependence graph. We consider VP as an intermediate step between analysis passes, and for convenience we label it as pass 1+. VP+ is the same as VP except it use results from the Client analysis (in the first pass) to further prune away variables.

pruning step VP+ is also slightly different from VP. We will explain the differences in the next few paragraphs.

In each of these two algorithms, the first pass is an imprecise and fast FICI pointer and client analysis. The purpose of executing the client analysis in the first pass is to prune away more variables than *Prune-IFDS*. Specifically, the pruning step VP+ attempts to prune away more variables than VP by handling analysis queries in different ways:

- As explained earlier, at each query VP computes the set of reachable variables, without testing the result of the query. It cannot evaluate the query since the client analysis is not performed.
- VP+ invokes the client analysis to evaluate each query. Only when the result of the query is yes does VP+ compute the set of reachable variables.

One side effect of the new VP+ step is that if no query answers yes, then the analysis can terminate without any more analysis passes. In such situations, an IFDS analysis is not performed, and the overall analysis time becomes very fast.

If further passes are needed, pass 2 *Feedback-Prune-IFDS* invokes an FSCI pointer analysis without client analysis. The main purpose of this pass is that we want to use the results from the more precise FSCI pointer analysis in the IFDS analysis. This is important because, generally, the FICI pointer analysis in the first pass is too imprecise that it hurts *both* precision and performance of the IFDS analysis. Only with the FSCI pointer analysis can the *Feedback-Prune-IFDS* guarantee to compute output with the same precision as *Prune-IFDS*.

Note that we do not use the results of VP+ to exclude variables in the second pointer analysis. To handle pointer analysis correctly, the variable dependences computed in the first pass would have to also include dependences between pointers and their points-to targets. It is not immediately clear if such dependences are all necessary or are sufficient, and we leave the question to future research.

Finally, the last pass of *Prune-IFDS* can also be combined with the Sparse IFDS algorithm to yield the *Prune-Sparse-IFDS* algorithm.

The results of all algorithms are presented later in Section 6.5.4.

6.5 Evaluation

This section first explains our methodology, followed by results of many experiments. Section 6.5.2 evaluates the effects of pointer analysis on an IFDS analysis. The next section compares our Sparse IFDS algorithm to the baseline algorithm. Section 6.5.4 presents many results on the Variable-Pruning IFDS algorithm, including comparison against the baseline algorithm, against our Sparse IFDS algorithm, and against the Demand IFDS algorithm. Section 6.5.5 presents results of the Feedback-based Variable-Pruning IFDS algorithm. Section 6.5.6 presents a table comparing in detail all algorithms for all programs, using IFDS analysis time and total pointer and

IFDS analysis time. We pick the two best algorithms from this table. Finally, Section 6.5.7 compares the two best algorithms against a dataflow analysis.

6.5.1 Methodology

Our experiments consist of the five error detection analysis problems, described earlier in Section 3.3.2 (page 49). We use the same 19 programs in Table 3.2 (page 52). Together, there are 95 experiment cases. As we will see in this section, none of the IFDS algorithms can complete more than 90 of these cases. When an algorithm cannot complete a test case, the reason is that the system runs out of memory (2GB), which is either because the program is large, or because the analysis problem requires too many flow values. The latter happens most often with the FTP-Behavior problem.

For the set of experiments evaluating the effects of pointer analysis on IFDS analysis, we label each algorithm with a subscript for the type of pointer analysis. For example, $IFDS_{fsci}$ refers to the baseline *IFDS* algorithm running on top of an FSCI pointer analysis. Where the pointer analysis is fixed and understood, we will drop the subscript.

All IFDS algorithms are graph-based solutions that explicitly represent flow functions in the exploded graph. For this reason, when comparing performance, we measure the IFDS analysis time as the total time spent in constructing the graph as well as in reachability analysis.

Besides analysis time, we also compare memory usage and precision of the different algorithms where appropriate. We will measure precision by looking at the reports generated by the error detection problems: if two analyses report same set of errors, we say they share the same precision. All analyses are safe—if an error

is present in reality, the analysis always reports the error. Therefore, an analysis is more precise if it reports fewer errors.

The graphs presented in the rest of this section use a unique format and, therefore, require some explanations. Each graph, such as the one in Figure 6.8, compares the analysis time, memory usage, and precision of an algorithm B relative to algorithm A:

- The vertical and horizontal axes, respectively, are log-scale ratios for analysis time and memory usage.
- The origin is at (1,1), which indicates no difference in the time or space consumptions. For example, a datum point (0.5,0.3) in the graph means the normalized memory consumed by algorithm B relative to algorithm A is 0.5 and that algorithm B takes up 0.3 seconds for every second consumed by algorithm A. Therefore all data points in the lower-left quadrant favor algorithm B in both dimensions.
- If the algorithms yield different results in terms of precision, the cases are labeled with different symbols (such as a cross).
- The geometric mean time and memory is labeled with a box.
- Below each graph is a summary of the experiments, such as number of cases compared, any precision differences, and the means.

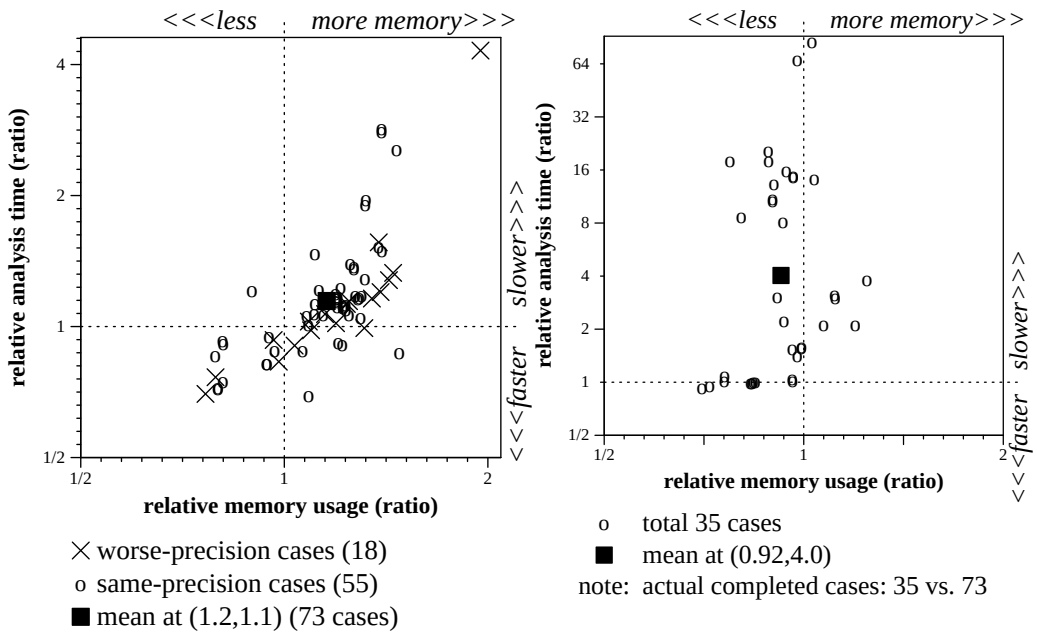
For example, the first graph in Figure 6.8 compares 73 cases of *Sparse-IFDS* against *IFDS*. The data are all in the third quadrant: *Sparse-IFDS* is always faster and consumes less memory. *IFDS* only completes 73 cases, while *Sparse-IFDS* completes 86 cases. The mean is at (0.16,0.39). There is no precision difference.

6.5.2 Effects of Pointers on IFDS Analysis

Earlier in Section 6.2.4, we discuss how pointer analysis affects an IFDS analysis. We have two goals in this section. First, we quantify the effect of pointer analysis on an IFDS analysis, using the following metrics: completion (does the analysis complete successfully?), precision, and performance in time and space. We will compare three algorithms $IFDS_X$, i.e., a pointer analysis X followed by the baseline IFDS algorithm, where X is either FICI, FSCI, or FSCS. The second goal is to justify the choice of $IFDS_{fsci}$ as our baseline algorithms. This baseline configuration will be used to compare performance of our new algorithms, which also use FSCI pointer analysis unless otherwise stated.

Figure 6.8 examines the differences between the three analyses. The first graph compares 70 cases of $IFDS_{fici}$ relative to $IFDS_{fsci}$ (25 cases did not complete in one or both modes). Many cases consume more time and more memory with $IFDS_{fici}$, i.e., there are more points in the upper right quadrant. This result is not surprising because an imprecise pointer analysis yields larger points-to sets, which leads to larger data-sets. In fact, $IFDS_{fsci}$ actually completed 71 cases, 1 more than $IFDS_{fici}$. The geometric mean is at (1.1,1.2), close to the origin. In terms of precision, $IFDS_{fici}$ produces 18 worse precision cases, again due to the imprecise pointer analysis.

Figure 6.8(b) compares $IFDS_{fscs}$ against $IFDS_{fsci}$. This time, $IFDS_{fscs}$ uses less memory, most likely due to smaller points-to sets. However, it consumes much more time. This is because the more precise pointer analysis is providing more information (particularly, more context information that is absent in a context-insensitive pointer analysis), so the IFDS algorithm takes longer to process this information. There is no difference in the precision. However, the context-sensitive



(a) $IFDS_{fici}$ relative to $IFDS_{fsci}$

(b) $IFDS_{fscs}$ relative to $IFDS_{fsci}$

	$IFDS_{fici}$	$IFDS_{fsci}$	$IFDS_{fscs}$
(A) completed cases	70	71	33
(B) same-precision cases	45	45	21
best-precision cases/(A-B)	5/25	21/26	12/12
exploded graph nodes*	14k–2.1m	13k–1.9m	12k–1.8m
exploded graph edges*	17k–2.4m	16k–2.2m	14k–2.0m
average data-set sizes*	8–763	7–751	7–747

*: using the 33 cases where $IFDS_{fscs}$ completes.

(c) Summary: comparing the three algorithms.

Figure 6.8: Effects of pointer analysis on IFDS algorithms. The first part of the table compares performance of IFDS algorithms using time, memory, and precision when different pointer analysis is used. The cost of pointer analysis is excluded from the comparisons. These results show that $IFDS_{fsci}$ is better than either $IFDS_{fici}$ or $IFDS_{fscs}$. The second part of the table compares the sizes of the exploded graphs created.

pointer analysis can complete only 35 cases, out of which $IFDS_{fscs}$ can complete 33 cases.

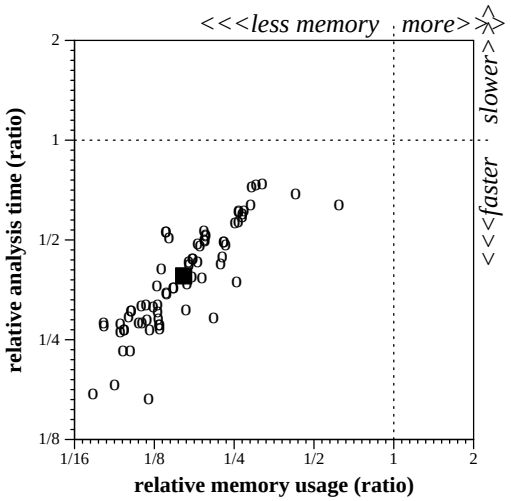
Given that $IFDS_{fscs}$ can complete far fewer cases, and that $IFDS_{fici}$ is the least precise, the natural choice of baseline is $IFDS_{fsci}$. For the remainder of the

chapter, we will use *IFDS* to denote this baseline.

6.5.3 Results of Sparse IFDS

This section evaluates the following question: How does the new Sparse IFDS algorithm compare to the baseline algorithm? The graph in Figure 6.9 compares the new algorithm relative to the baseline. FSCI pointer analysis is used in all cases, but the time spent in pointer analysis is excluded from comparisons. The graph indicates that *Sparse-IFDS* can complete all 73 cases that *IFDS* completes, plus 13 more. On average, *Sparse-IFDS* spends 0.39 seconds for every second *IFDS* consumes; or equivalently, *Sparse-IFDS* is 2.6× faster. The actual range is that *Sparse-IFDS* is 1.3–6.0× faster (see table). The average memory consumption by *Sparse-IFDS* is only 16% of that by baseline.

Our Sparse IFDS algorithm performs well, because it reduces the size of the exploded graphs. Table 6.2 quantifies these reductions by computing the fractions of number of nodes and edges generated by *Sparse-IFDS* relative to *IFDS*. Reported in the table under the column "*Sparse*", on average, the graphs created by *Sparse-IFDS* have only 11% as many nodes, and 13% as many edges as in *IFDS*. These lead to only 15% as many path-edges. All these numbers confirm there is indeed a large degree of sparsity in the original graphs, and *Sparse-IFDS* is able to exploit this sparsity to save time and space (the last few rows on "flow values" are not relevant to the Sparse IFDS algorithm. The remainder of the table is used in later sections).



(a) *Sparse-IFDS* relative to *IFDS*

Algorithm A:		*IFDS*	*Sparse-IFDS*
Completions:		73	86
cases completed by *IFDS* but not A:		-	0
cases completed by A but not *IFDS*:		-	13
speedup	Mean:	-	2.6×
	Min:	-	1.3×
	Max:	-	6.0×

(b) Summary: comparing the two algorithms.

Figure 6.9: Performance results of *Sparse-IFDS* relative to *IFDS*.

6.5.4 Results of Variable-Pruning IFDS

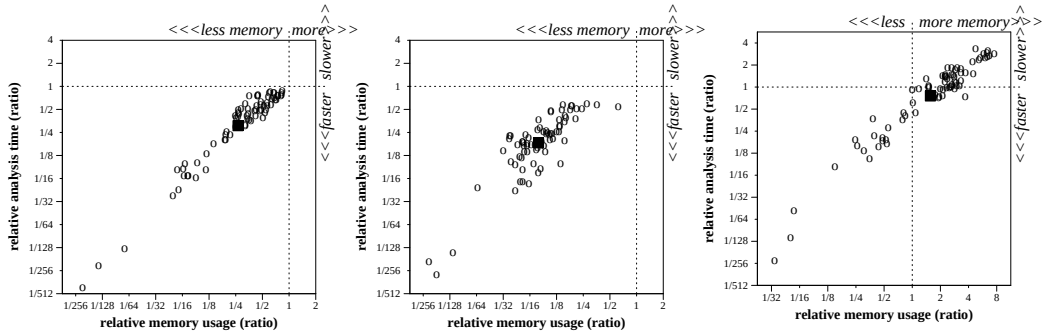
The Variable-Pruning IFDS algorithm can be combined with the Sparse IFDS algorithms, and we call the combined algorithm *Prune-Sparse-IFDS*. The first two graphs in Figure 6.10 compare *Prune-IFDS* and *Prune-Sparse-IFDS* against the baseline, respectively. In these graphs, all data points lie in the lower-left quadrants, indicating speedups and less memory usage in all cases. The mean speedup is 3.3× faster for *Prune-IFDS*, and 5.5× for *Prune-Sparse-IFDS*.

		baseline	*Sparse*	*Demand*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	cases	73	86	75	76	87	84(54)	87(57)
nodes	**Mean**	1.00	0.11	0.27	0.32	0.06	0.45	0.07
	Max	-	0.30	3.84	0.94	0.25	0.91	0.22
	Min	-	0.03	< 0.01	< 0.01	< 0.01	0.04	0.01
	Worse-cases	-	-	14	-	-	-	-
edges	**Mean**	1.00	0.13	0.28	0.33	0.08	0.46	0.09
	Max	-	0.35	3.89	0.94	0.26	0.92	0.26
	Min	-	0.03	0.01	0.01	0.01	0.04	0.01
	Worse-cases	-	-	14	-	-	-	-
path edges	**Mean**	1.00	0.15	< 0.01	0.28	0.08	0.44	0.08
	Max	-	0.38	465	0.91	0.37	0.91	0.30
	Min	-	0.04	< 0.01	< 0.01	< 0.01	0.01	< 0.01
	Worse-cases	-	-	24	-	-	-	-
flow values	**Mean**	1.00	-	0.55	0.09	0.08	0.14	0.14
	Max	-	-	1.56	0.64	0.64	0.61	0.61
	Min	-	-	0.16	< 0.01	< 0.01	0.03	0.03
	Worse-cases	-	-	6	-	-	-	-

Table 6.2: Effects of various new IFDS algorithms on exploded graphs: Normalized number of nodes and edges created by the different algorithms (with respect to the baseline algorithms). The table also compares the number of distinct flow values in the graphs. An FSCI pointer analysis in the first four algorithms. The drastic reduction in the graphs explains the improved performance by the sparse and pruning algorithms.

The results can again be explained by considering the reduced graphs. As reported in Table 6.2, column "*Prune*", the graphs created by *Prune-IFDS* have only approximately a third as many nodes and edges as the baseline algorithm. These lead to only 28% as many path-edges.

In addition, we also compare the number of distinct flow values used in the analyses. Specifically, suppose the analysis problem has two distinct lattice values $\{A, B\}$, then for each variable x, "$x = A$" and "$x = B$" are counted as two distinct flow values. These values are created during analysis on demand. Therefore, fewer



(a) *Prune-IFDS* (b) *Prune-Sparse-IFDS* (c) *Prune-IFDS* against *Sparse-IFDS*

Algorithm A:		*IFDS*	*Prune-IFDS*	*Prune-Sparse-IFDS*
Completions:		73	76	87
cases completed by *IFDS* but not A:		-	0	0
cases completed by A but not *IFDS*:		-	3	14
speedup	Mean:	-	3.3×	5.5×
	Min:	-	1.1×	1.6×
	Max:	-	422×	285×

(d) Summary: comparing the first two algorithms against baseline.

Figure 6.10: Results of Variable-Pruning IFDS against baseline *IFDS*.

flow values are used when variables are pruned away. For example, we can see in Table 6.2, under the row with heading "flow values," that *Prune-IFDS* uses only 9 flow values for every 100 flow values used in the baseline algorithm (0.09). Such reductions are the reason behind the smaller graphs.

Variable-Pruning versus Sparse IFDS

In order to provide a clearer picture of the different algorithms, we also directly compare *Prune-IFDS* against *Sparse-IFDS*, as shown in the graph in Figure 6.10(c). Notice the slight difference in the horizontal axis scales. From the graph, it is obvious that neither algorithm is always better than the other. While *Sparse-IFDS* can

complete more cases (86 versus 76), on average *Prune-IFDS* is actually faster: 0.76 normalized time relative to *Sparse-IFDS*, or approximately 1.3× faster. The reason is that while *Prune-IFDS* is more computationally efficient, its space inefficiency causes it to crash in more cases than *Sparse-IFDS*. This last reason is supported by evidence provided in Table 6.2, which compares graph size: the mean number of nodes and edges are both significantly higher when the graphs are constructed by *Prune-IFDS*.

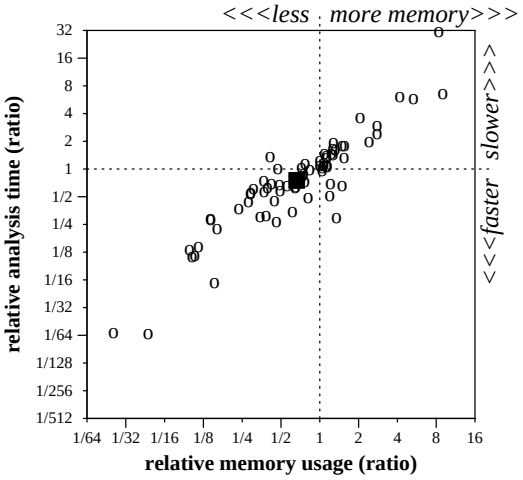
Variable-Pruning IFDS versus Demand IFDS

Since the Variable-Pruning IFDS algorithm has similarities with Demand IFDS analysis (they prune variables used in analysis), we have implemented the Demand IFDS algorithm, denoted as *Demand-IFDS*. Figure 6.11 compares *Prune-IFDS* relative to *Demand-IFDS*.

It is easy to tell from the graph that the *Prune-IFDS* is in many cases much faster than *Demand-IFDS*, and vice versa in other cases. The mean speedup of *Prune-IFDS* over *Demand-IFDS* is moderate at 1.3×, with a maximum at 59×.

While the algorithms *Prune-IFDS* and *Demand-IFDS* can complete almost the same number of cases, it is interesting to observe that each algorithm can complete five to six cases the other other algorithm cannot. This is not surprising given the different directions of analysis. For example, given a large program with no query, the forward analyses will take a long time, whilst the backward analyses will complete quickly. We leave as future work answer the interesting question regarding which analyses to pick in advance for any given program.

Table 6.2 (page 142) provides additional insight into the performance differences, by comparing graph sizes produced by the two algorithms relative to the



(a) *Prune-IFDS* against *Demand-IFDS*

Algorithm A:		*Demand-IFDS*	*Prune-IFDS*
Completions:		75	76
cases completed by *Demand-IFDS* but not A:		-	5
cases completed by A but not *Demand-IFDS*:		-	6
speedup	Mean:	-	1.3×
	Min:	-	$< 0.1\times$
	Max:	-	59×

(b) Summary: comparing the two algorithms.

Figure 6.11: Results of Variable-Pruning IFDS relative to Demand IFDS algorithm.

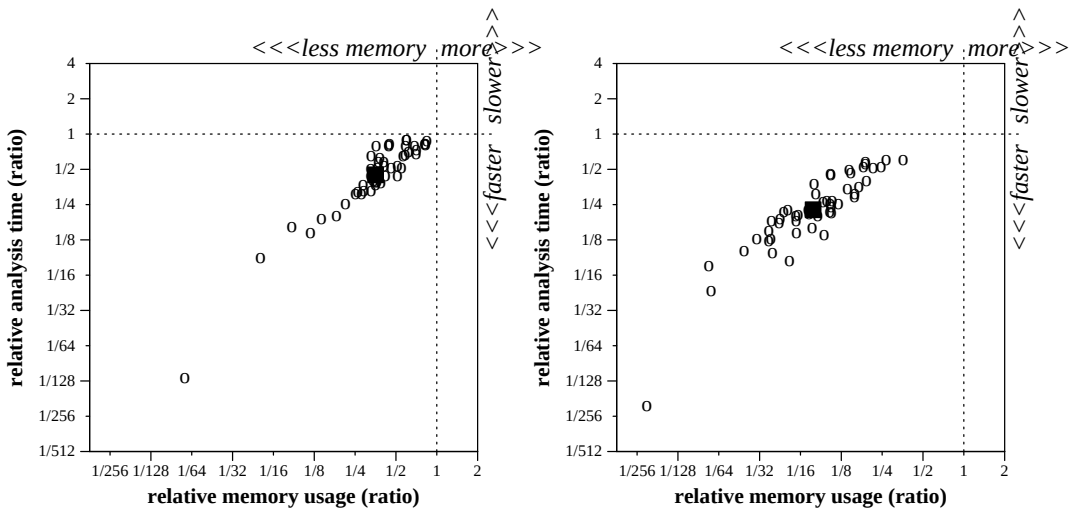
baseline algorithm. Notice that the Demand IFDS algorithm can sometimes produce larger graphs (larger maximum ratios for nodes and edges) relative to baseline, because it generates nodes in a backward direction. On the other hand, *Prune-IFDS* always produces smaller graphs with fewer variables. The mean number of nodes and edges are quite close to those of *Demand-IFDS*.

6.5.5 Results of Feedback-based Variable-Pruning IFDS

We now evaluate the Feedback-based versions of Variable-Pruning IFDS algorithms. Figure 6.12 compares the algorithm *Feedback-Prune-IFDS* and the variant *Feedback-Prune-Sparse-IFDS*, relative to the baseline algorithm. Both Feedback-based algorithms use multiple analysis passes, and in 30 program-problem cases, analysis completes successfully after the first pass, so no IFDS analysis is needed. For the remaining cases, *Feedback-Prune-IFDS* improves analysis time by 2.2× speedup on average, while *Feedback-Prune-Sparse-IFDS* achieves an average of 4.4× speedup.

At first glance, the results in Figure 6.12 (the two Feedback-based algorithms) appear to be worse than the results in Figure 6.10 (the two non-Feedback-based variants): the latter have better speedups. The speedups computed in these figures compare only IFDS analysis time, without taking into account those cases that the Feedback-based algorithms can complete successfully after one pass, without invoking IFDS analysis.

To account for this discrepancy, Figure 6.13 compares the algorithms by comparing the *total* pointer and IFDS analysis time. The first graph compares *Feedback-Prune-IFDS* relative to *Prune-IFDS*, while the second compares *Feedback-Prune-Sparse-IFDS* against *Prune-Sparse-IFDS*. In each graph, the data points seem to belong to two groups: those lying near the horizontal axis, and those dispersed below same axis. This division suggests that when an IFDS analysis is performed, the performance differences between the pairs of algorithms are moderate, but for those cases where Feedback-based algorithms do not perform an IFDS analysis, the differences are huge. The table below the graphs further supports this theory: when we compare IFDS analysis time only, the mean speedups are 1.0×; but when we compare total pointer and IFDS analysis time, the mean speedups increase to 1.6×



o total 50 cases
■ mean at (0.35,0.45)
note: actual completed cases: 84 vs. 73

(a) *Feedback-Prune-IFDS*

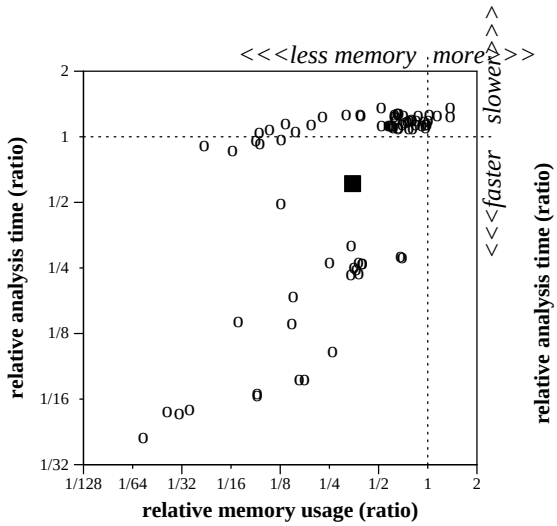
o total 50 cases
■ mean at (0.08,0.23)
note: actual completed cases: 87 vs. 73

(b) *Feedback-Prune-Sparse-IFDS*

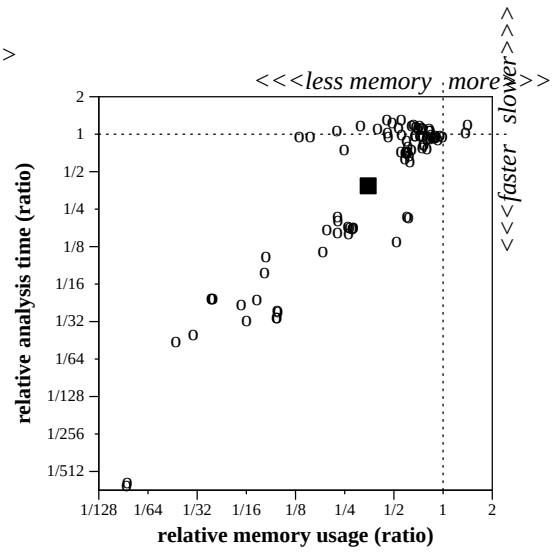
Algorithm A:		*IFDS*	*Feedback-Prune-IFDS*	*Feedback-Prune-Sparse-IFDS*
Completions:		73	84	87
No-IFDS (1-pass only) cases:		-	30	30
IFDS completion cases:		-	54	57
cases completed by *IFDS* but not A:		-	0	0
cases completed by A but not *IFDS*:		-	11	14
speedup	Mean:	-	2.2×	4.4×
	Min:	-	1.1×	1.6×
	Max:	-	116×	204×

(c) Summary: comparing the two algorithms against baseline.

Figure 6.12: Results of Feedback-based Variable-Pruning IFDS algorithm, relative to baseline algorithm.



(a) *Feedback-Prune-IFDS* against *Prune-IFDS*



(b) *Feedback-Prune-Sparse-IFDS* against *Prune-Sparse-IFDS*

Algorithm		A	B	A	B
		Prune-IFDS	*Feedback-Prune-IFDS*	*Prune-Sparse-IFDS*	*Feedback-Prune-Sparse-IFDS*
Completions:		76	84	87	87
No-IFDS (1-pass only) cases:		-	30	-	30
IFDS completion cases:		-	54	-	57
cases completed by A but not B:		-	0	-	0
cases completed by B but not A:		-	8	-	0
speedup: IFDS time only	Mean:	-	1.0×	-	1.0×
	Min:	-	0.8×	-	0.8×
	Max:	-	4.0×	-	3.1×
speedup: Ptrs+IFDS time	Mean:	-	1.6×	-	2.7×
	Min:	-	0.7×	-	0.8×
	Max:	-	23.8×	-	651×

(c) Summary: comparing the two Feedback-based algorithms against their non-Feedback-based counterparts.

Figure 6.13: Results of Feedback-based Variable-Pruning IFDS algorithm, relative to non-Feedback Variable-Pruning IFDS algorithm. The total pointer and IFDS analysis times are used in these comparisons.

and 2.7×, with maximum speedup up to 651× in the case of the *Feedback-Prune-Sparse-IFDS* relative to *Prune-Sparse-IFDS*.

Next, we again examine the algorithms by looking at the graphs they construct. Refer back to Table 6.2 (page 142), where we compare graph sizes normalized to those created by the baseline algorithm. The two algorithms, *Feedback-Prune-IFDS* and *Feedback-Prune-Sparse-IFDS*, also create smaller graphs like their non-Feedback-based counterparts (*Prune-IFDS* and *Prune-Sparse-IFDS*).

As we explained earlier, the Feedback-based algorithms attempt to prune away more variables than their non-Feedback-based counterparts. Unfortunately, that is not always the case. Specifically, there are 16 cases where *Feedback-Prune-IFDS* uses a larger set of variables than *Prune-IFDS*. The differences are due to one or two reasons:

1. *Feedback-Prune-IFDS* evaluates each query after the first pass, and only when the query answers yes are the variables involved in the queries included in an IFDS analysis later. Therefore, the algorithm can potentially prune away more variables than *Prune-IFDS*.

2. However, the first pass of *Feedback-Prune-IFDS* uses an FICI pointer analysis, which sometimes results in larger points-to sets than an FSCI pointer analysis. Therefore, the set of variables involved at a query is sometimes larger. When this happens, then (i) *Feedback-Prune-IFDS* may evaluate more queries to answer yes, and (ii) when both algorithms evaluate the same query to answer yes, *Feedback-Prune-IFDS* will include extra variables that *Prune-IFDS* would not include.

Neither factor seems to be dominant for all programs. We gather this result from Table 6.3, which compares the graphs computed by *Feedback-Prune-IFDS* relative

		Prune	*Feedback Prune*
	cases	76	84(54)
nodes	**Mean**	1.00	0.97
	Max	-	1.32
	Min	-	0.20
	Worse-cases	-	17
edges	**Mean**	1.00	0.97
	Max	-	1.32
	Min	-	0.20
	Worse-cases	-	19
path edges	**Mean**	1.00	0.99
	Max	-	1.42
	Min	-	0.20
	Worse-cases	-	24
flow values	**Mean**	1.00	0.96
	Max	-	1.47
	Min	-	0.19
	Worse-cases	-	17

Table 6.3: Effects of using feedbacks in Variable-Pruning IFDS algorithms. The table compares the graph sizes created by *Feedback-Prune-IFDS* relative to those generated by *Prune-IFDS*. The legends are the same as those used earlier in Table 6.2 (page 142). Among all cases, in 17 cases *Feedback-Prune-IFDS* uses more variables and generates larger graphs.

to *Prune-IFDS*. Whether we are comparing the number of nodes or edges, or the number of flow values used in the constructions, the means are all close to one.

6.5.6 Detail Comparisons

So far, we have seen pairwise comparisons of different combinations of Sparse, Variable-Pruning, and Feedback-based IFDS algorithms. Table 6.4 compares five such algorithms in more details by showing their speedup, relative to the baseline algorithm, for all 95 benchmark/problem cases. For each case, we compute the speedups for both (a) IFDS analysis time only and (b) total pointers and IFDS

analysis time (τ). The caption at the top of the table explains the legend and other important notes, while some important overall statistics are shown at the end of the table. The caption also summarizes key lessons we learned from the results. Two key results are (1) using only average IFDS analysis time, the best algorithm is *Prune-Sparse-IFDS*; and (2) using total pointers and IFDS analysis time, the best algorithm is *Feedback-Prune-Sparse-IFDS*.

Table 6.4: Comparing Sparse, Variable-Pruning, and Feedback IFDS algorithms. The table compares six algorithms in detail. The first algorithm is the baseline *IFDS* algorithm, while the second is the *Sparse-IFDS* algorithm. The last four algorithms use Variable-Pruning, and the last two are Feedback-based. All numbers are either time in seconds or are the speedup factors (e.g., 2× means twice as fast relative to baseline). †means system crash (out of memory). Each row labeled with τ shows the total pointer and IFDS analysis time.
For the Feedback-based algorithms, there are 30 program-client cases (♡) where there is no error after one-pass (no IFDS analysis needed). The asterisks (*) mark cases where *IFDS* cannot complete, but the new algorithms can. The best time for each row is highlighted: light gray when comparing IFDS analysis time only, and dark gray for total pointer and IFDS analysis time (τ).
The results in this table show that: (1) all the Feedback-based algorithms yield best performance for those ♡ cases; (2) if we compare IFDS analysis time only, *Sparse-IFDS* and *Prune-IFDS* both improve over baseline *IFDS*, with *Prune-IFDS* having a better overall mean, but each is better than the other in different subsets of cases; (3) with few exceptions, the difference between *Prune-IFDS* and *Feedback-Prune-IFDS*is small; and (4) the combination of all algorithms, i.e., *Feedback-Prune-Sparse-IFDS*, often leads to the best performance for both IFDS-time and total time; it also has the most number of completion cases (tie with *Prune-Sparse-IFDS*).

		baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
Program	**Client**						
stunnel	File-Access	0.5	1.4x	1.4x	1.7x	1.3x	1.6x
	τ	1.0	0.9x	1.1x	1.0x	0.9x	0.9x
	FSV	0.2	1.6x	1.7x	1.8x	♡	♡
	τ	0.8	0.9x	1.2x	0.9x	4.0x	4.0x
	Remote-FSV	0.6	2.0x	1.9x	2.8x	1.6x	2.4x

continued on next page

(Table 6.4) continued from previous page

Program	Client	baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	τ	1.1	1.1x	1.3x	1.2x	1.0x	1.0x
	Remote-Access	0.6	2.0x	4.9x	5.0x	♡	♡
	τ	1.1	1.1x	1.7x	1.3x	5.9x	5.9x
	FTP-Behavior	2.3	2.7x	1.7x	3.6x	♡	♡
	τ	2.8	1.8x	1.5x	2.0x	14.4x	14.4x
pfinger	File-Access	1.7	2.0x	2.0x	3.3x	1.9x	3.3x
	τ	4.4	0.9x	1.2x	1.0x	1.0x	1.1x
	FSV	1.4	2.3x	1.7x	3.1x	1.7x	3.2x
	τ	4.1	0.9x	1.1x	0.9x	0.9x	0.9x
	Remote-FSV	3.7	3.1x	2.2x	5.6x	2.2x	5.4x
	τ	6.4	1.2x	1.4x	1.4x	1.2x	1.4x
	Remote-Access	3.7	3.1x	3.2x	7.0x	3.1x	6.8x
	τ	6.4	1.2x	1.6x	1.4x	1.3x	1.4x
	FTP-Behavior	17.1	4.3x	1.4x	5.6x	♡	♡
	τ	19.8	2.5x	1.4x	2.8x	23.9x	23.9x
muh-2.05c	File-Access	3.4	2.0x	2.7x	4.7x	2.5x	4.5x
	τ	5.4	1.1x	1.7x	1.4x	1.4x	1.5x
	FSV	2.7	2.5x	5.5x	7.0x	5.2x	7.1x
	τ	4.7	1.1x	1.9x	1.4x	1.5x	1.4x
	Remote-FSV	4.7	2.8x	7.5x	12.0x	6.8x	11.8x
	τ	6.7	1.4x	2.5x	2.0x	2.0x	2.0x
	Remote-Access	4.8	2.8x	14.3x	17.0x	♡	♡
	τ	6.8	1.4x	2.9x	2.1x	11.5x	11.5x
	FTP-Behavior	19.7	3.4x	1.3x	3.9x	1.5x	4.7x
	τ	21.7	2.5x	1.3x	2.7x	1.3x	2.8x
muh-2.05d	File-Access	3.5	2.0x	2.7x	4.8x	2.5x	4.6x
	τ	5.5	1.1x	1.6x	1.5x	1.4x	1.5x
	FSV	2.9	2.7x	15.3x	13.0x	♡	♡
	τ	4.9	1.2x	2.2x	1.5x	8.3x	8.3x
	Remote-FSV	5.0	2.9x	26.2x	22.8x	♡	♡
	τ	7.1	1.5x	3.1x	2.2x	11.8x	11.8x
	Remote-Access	4.9	2.9x	14.4x	17.2x	♡	♡
	τ	7.0	1.5x	2.9x	2.1x	11.8x	11.8x
	FTP-Behavior	20.3	3.5x	1.3x	4.0x	1.4x	4.3x
	τ	22.4	2.5x	1.2x	2.8x	1.3x	2.6x
pureftpd	File-Access	5.5	2.3x	2.0x	3.8x	1.9x	3.9x
	τ	11.2	1.0x	1.3x	1.1x	1.1x	1.2x

continued on next page

(Table 6.4) continued from previous page

Program	Client	baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	FSV	4.4	2.4x	11.5x	9.9x	♡	♡
	τ	10.0	1.0x	1.6x	1.1x	6.1x	6.1x
	Remote-FSV	8.0	3.4x	21.6x	18.3x	♡	♡
	τ	14.8	1.4x	2.4x	1.7x	10.3x	10.3x
	Remote-Access	7.3	3.1x	3.1x	6.2x	2.6x	6.2x
	τ	13.5	1.3x	1.7x	1.4x	1.3x	1.4x
	FTP-Behavior	30.4	3.5x	1.5x	4.1x	6.0x	12.9x
	τ	36.1	2.1x	1.4x	2.3x	2.8x	3.1x
fcron	File-Access	3.8	1.7x	2.0x	2.6x	1.8x	2.7x
	τ	12.9	0.8x	1.1x	0.9x	1.0x	0.9x
	FSV	5.3	3.4x	12.0x	9.8x	♡	♡
	τ	17.6	1.2x	1.9x	1.3x	7.8x	7.8x
	Remote-FSV	6.3	3.2x	14.5x	11.6x	♡	♡
	τ	16.6	1.1x	1.8x	1.2x	5.5x	5.5x
	Remote-Access	5.1	2.6x	2.4x	3.7x	♡	♡
	τ	14.2	0.9x	1.3x	1.0x	4.7x	4.7x
	FTP-Behavior	26.5	3.7x	1.9x	4.7x	♡	♡
	τ	35.8	1.8x	1.6x	1.9x	11.0x	11.0x
apache	File-Access	9.3	1.3x	1.3x	1.6x	1.2x	1.6x
	τ	38.1	0.7x	1.1x	0.7x	0.9x	0.9x
	FSV	4.6	1.8x	1.3x	1.7x	1.2x	2.0x
	τ	33.4	0.6x	1.0x	0.6x	0.9x	0.8x
	Remote-FSV	11.9	1.9x	1.3x	2.1x	1.2x	2.2x
	τ	40.6	0.7x	1.1x	0.7x	0.9x	0.9x
	Remote-Access	11.8	1.9x	1.3x	2.0x	1.2x	2.1x
	τ	40.4	0.7x	1.1x	0.7x	0.9x	0.9x
	FTP-Behavior	39.2	2.4x	1.2x	2.6x	1.2x	2.6x
	τ	67.9	1.0x	1.1x	1.0x	1.0x	1.0x
make	File-Access	9.7	1.9x	1.6x	2.7x	1.5x	2.8x
	τ	264	0.5x	1.0x	0.5x	0.9x	0.8x
	FSV	307	6.0x	1.7x	6.5x	♡	♡
	τ	561	1.0x	1.3x	1.0x	31.1x	31.1x
	Remote-FSV	16.5	3.6x	1.9x	4.3x	♡	♡
	τ	271	0.5x	1.0x	0.5x	15.5x	15.5x
	Remote-Access	16.5	3.5x	2.2x	4.3x	♡	♡
	τ	269	0.5x	1.0x	0.5x	15.4x	15.4x
	FTP-Behavior	†	392 (*)	†	347 (*)	♡	♡

continued on next page

(Table 6.4) continued from previous page

Program	Client	baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	τ	†	904 (*)	†	852 (*)	21.3 (*)	21.3 (*)
BlackHole	File-Access	83.7	2.3x	11.9x	20.3x	11.1x	21.3x
	τ	141	1.0x	2.2x	1.3x	1.9x	1.8x
	FSV	205	1.9x	216x	143x	♡	♡
	τ	263	1.3x	4.5x	2.5x	32.0x	32.0x
	Remote-FSV	399	1.9x	422x	285x	♡	♡
	τ	457	1.4x	7.8x	4.4x	42.0x	42.0x
	Remote-Access	397	1.9x	2.5x	3.3x	2.2x	3.1x
	τ	455	1.4x	2.1x	2.1x	1.8x	1.8x
	FTP-Behavior	†	1413 (*)	†	1333 (*)	1611 (*)	942 (*)
	τ	†	1516 (*)	†	1434 (*)	1690 (*)	1102 (*)
wu-ftpd-2.6.0	File-Access	19.6	2.1x	2.0x	3.8x	1.8x	3.6x
	τ	41.3	1.0x	1.3x	1.1x	1.0x	1.1x
	FSV	31.7	3.1x	3.9x	5.4x	3.0x	4.5x
	τ	53.6	1.3x	1.7x	1.4x	1.3x	1.2x
	Remote-FSV	35.3	3.2x	4.1x	8.2x	3.1x	6.6x
	τ	57.3	1.3x	1.9x	1.5x	1.4x	1.3x
	Remote-Access	35.1	3.2x	2.6x	5.8x	2.5x	5.6x
	τ	57.0	1.3x	1.6x	1.5x	1.3x	1.2x
	FTP-Behavior	199	4.3x	1.8x	5.7x	2.3x	7.6x
	τ	221	2.8x	1.6x	3.3x	1.8x	2.9x
openssh-client	File-Access	24.4	1.6x	1.3x	1.9x	1.2x	1.9x
	τ	106	0.6x	1.0x	0.6x	0.9x	0.9x
	FSV	12.0	1.6x	1.7x	2.0x	1.5x	2.1x
	τ	95.3	0.5x	1.1x	0.6x	0.9x	0.6x
	Remote-FSV	23.3	1.7x	1.2x	1.7x	1.1x	1.7x
	τ	106	0.6x	1.0x	0.6x	0.9x	0.6x
	Remote-Access	23.5	1.7x	1.2x	1.7x	1.1x	1.7x
	τ	107	0.6x	1.0x	0.6x	0.9x	0.6x
	FTP-Behavior	123	2.2x	1.2x	2.5x	♡	♡
	τ	206	0.9x	1.1x	1.0x	20.1x	20.1x
privoxy	File-Access	849	2.6x	127x	193x	116x	204x
	τ	1002	1.7x	6.3x	3.8x	5.5x	5.1x
	FSV	†	12389 (*)	†	12452 (*)	♡	♡
	τ	†	12656 (*)	†	12718 (*)	19.5 (*)	19.5 (*)
	Remote-FSV	†	12647 (*)	†	12328 (*)	♡	♡
	τ	†	12915 (*)	†	12594 (*)	20.4 (*)	20.4 (*)

continued on next page

(Table 6.4) continued from previous page

Program	Client	baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	Remote-Access	†	13126 (*)	†	12545 (*)	†	12263 (*)
	τ	†	13394 (*)	†	12810 (*)	†	12638 (*)
	FTP-Behavior	†	†	†	†	†	?
bind	File-Access	13.9	2.3x	2.4x	4.1x	2.3x	4.0x
	τ	42.7	0.9x	1.2x	1.0x	1.0x	1.0x
	FSV	11.1	3.3x	2.2x	3.8x	1.7x	3.7x
	τ	40.3	0.9x	1.2x	0.9x	0.9x	0.8x
	Remote-FSV	17.1	3.7x	2.4x	5.2x	1.9x	4.7x
	τ	46.0	1.0x	1.3x	1.0x	1.0x	0.9x
	Remote-Access	17.2	3.7x	2.5x	5.2x	1.9x	4.8x
	τ	46.0	1.0x	1.3x	1.0x	1.0x	0.9x
	FTP-Behavior	85.7	5.8x	2.0x	6.7x	2.7x	9.7x
	τ	114	2.1x	1.5x	2.1x	1.6x	1.8x
wu-ftpd-2.6.2	File-Access	45.1	2.6x	4.1x	8.2x	3.9x	8.0x
	τ	80.5	1.1x	1.7x	1.4x	1.3x	1.4x
	FSV	61.6	3.4x	3.2x	4.5x	3.0x	4.3x
	τ	97.1	1.4x	1.8x	1.5x	1.4x	1.3x
	Remote-FSV	79.5	3.5x	3.3x	5.6x	3.0x	5.4x
	τ	115	1.5x	1.9x	1.7x	1.5x	1.5x
	Remote-Access	83.9	3.7x	2.4x	5.3x	2.2x	5.2x
	τ	120	1.6x	1.7x	1.7x	1.3x	1.4x
	FTP-Behavior	573	5.4x	2.1x	7.0x	2.5x	7.7x
	τ	609	3.8x	2.0x	4.5x	2.2x	3.8x
openssh-server	File-Access	29.7	1.4x	1.4x	1.9x	1.2x	1.9x
	τ	174	0.5x	1.0x	0.6x	0.9x	0.9x
	FSV	14.4	1.4x	9.7x	5.7x	♡	♡
	τ	161	0.5x	1.1x	0.5x	14.0x	14.0x
	Remote-FSV	44.0	1.6x	9.9x	9.4x	♡	♡
	τ	191	0.6x	1.3x	0.6x	16.2x	16.2x
	Remote-Access	44.0	1.6x	1.3x	1.9x	1.2x	1.9x
	τ	190	0.6x	1.1x	0.6x	0.9x	0.7x
	FTP-Behavior	165	2.0x	1.2x	2.4x	♡	♡
	τ	311	0.8x	1.1x	0.8x	19.5x	19.5x
cfengine	File-Access	132	2.3x	4.9x	10.3x	4.9x	10.0x
	τ	262	0.9x	1.7x	1.1x	1.5x	1.5x
	FSV	224	3.6x	1.1x	4.0x	1.1x	3.6x
	τ	355	1.2x	1.1x	1.3x	1.0x	1.0x

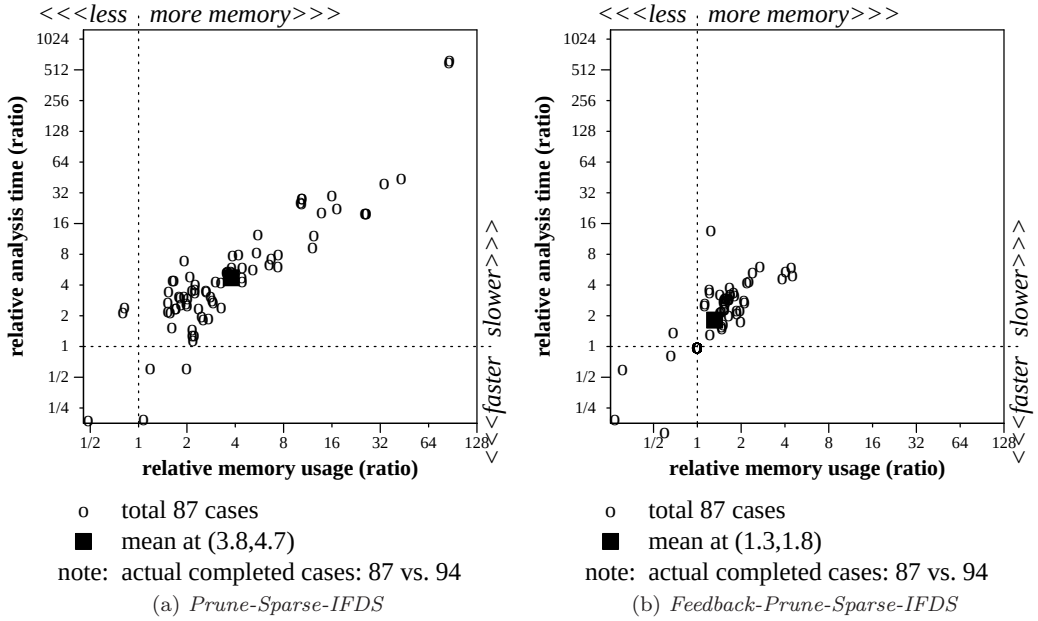
continued on next page

(Table 6.4) continued from previous page

Program	Client	baseline *IFDS*	*Sparse*	*Prune*	*Prune Sparse*	*Feedback Prune*	*Feedback Prune Sparse*
	Remote-FSV	449	3.6x	1.2x	4.1x	1.2x	3.8x
	τ	580	1.7x	1.1x	1.7x	1.1x	1.3x
	Remote-Access	456	3.7x	1.4x	5.1x	1.4x	4.9x
	τ	587	1.7x	1.3x	1.9x	1.2x	1.5x
	FTP-Behavior	†	867 (*)	†	790 (*)	†	549 (*)
	τ	†	1093 (*)	†	1015 (*)	†	984 (*)
sqlite	File-Access	†	106 (*)	9.0 (*)	5.8 (*)	10.0 (*)	4.8 (*)
	τ	†	1291 (*)	823 (*)	1172 (*)	922 (*)	960 (*)
	FSV	†	512 (*)	†	476 (*)	♡	♡
	τ	†	1690 (*)	†	1642 (*)	80.6 (*)	80.6 (*)
	Remote-FSV	†	498 (*)	†	461 (*)	♡	♡
	τ	†	1681 (*)	†	1629 (*)	79.9 (*)	79.9 (*)
	Remote-Access	†	489 (*)	†	465 (*)	♡	♡
	τ	†	1672 (*)	†	1634 (*)	80.1 (*)	80.1 (*)
	FTP-Behavior	†	3152 (*)	†	2867 (*)	♡	♡
	τ	†	4336 (*)	†	4029 (*)	88.4 (*)	88.4 (*)
nn	File-Access	†	65.5 (*)	25.0 (*)	15.3 (*)	27.3 (*)	14.9 (*)
	τ	†	1914 (*)	1154 (*)	1841 (*)	1276 (*)	1376 (*)
	FSV	†	39518 (*)	†	37394 (*)	†	39664 (*)
	τ	†	41386 (*)	†	39216 (*)	†	42544 (*)
	Remote-FSV	†	†	†	†	†	†
	Remote-Access	†	†	†	†	†	†
	FTP-Behavior	†	†	†	†	†	?
sendmail	File-Access	†	†	101 (*)	54.3 (*)	109 (*)	57.2 (*)
	τ	†	†	7896 (*)	14024 (*)	8921 (*)	10145 (*)
	FSV	†	†	†	†	†	†
	Remote-FSV	†	†	†	†	†	†
	Remote-Access	†	†	†	†	†	†
	FTP-Behavior	†	†	†	†	?	?
	Completions:	73	86	76	87	84	87
	(IFDS) cases / mean:		73 / 2.6x	73 / 3.3x	73 / 5.5x	50 / 2.2x	50 / 4.4x
	(IFDS-time) min / max:		1.3x / 6.0x	1.1x / 421.6x	1.6x / 285.1x	1.1x / 116.4x	1.6x / 203.9x
	(total) cases / mean:		73 / 1.1x	73 / 1.5x	73 / 1.3x	73 / 2.6x	73 / 2.6x
	(total time) min / max:		0.5x / 3.8x	1.0x / 7.8x	0.5x / 4.5x	0.9x / 42.0x	0.6x / 42.0x

6.5.7 Comparing Best IFDS Algorithm with Dataflow Analysis

Figure 6.14 compares the best IFDS algorithms against Client-Driven analysis, using total analysis time. The results show that Client-Driven still outperforms IFDS analysis.



(a) *Prune-Sparse-IFDS* (b) *Feedback-Prune-Sparse-IFDS*

Figure 6.14: The two best IFDS algorithms relative to Client-Driven analysis, using total analysis time.

6.6 Cycle Elimination

In this section, we discuss our attempt to incorporate cycle elimination techniques into IFDS analysis. Unfortunately, for reasons that we shall discuss, the techniques did not give us the speedups we were hoping. This section explains their apparent attractiveness, and discusses reasons for their shortfalls.

6.6.1 Motivation

The reachabilility analysis embedded in the IFDS analysis belongs to the class of *context-free-language reachability problems* (CFL-reachability problems) [120]. The best known algorithm has complexity $O(n^3)$, where n is the number of nodes. The key to scaling is to reduce the number of nodes. Both Sparse IFDS and Variable-Pruning are successful examples of applying this idea. Another idea that we investigate is to perform cycle elimination.

The idea is to detect strongly connected components (SCC) in the graph: every node in an SCC is reachable from every other node in the same SCC. Therefore, if we collapse all SCC's, we can reduce the time spent in reachability analysis.

The appeal of the technique is strengthened by their successful applications to other problem domains such as inclusion-based pointer analysis [38, 62, 111, 60], which are set constraint problems, which in turn are equivalent to CFL-reachability problems.

For the techniques to succeed, there must be a substantial number of cycle nodes. Table 6.5 presents the results of a feasibility study. The second column shows the number of loop nodes in the flow graphs. Each of the remaining five columns shows the number of SCC's and percentage of loop nodes in the explode graphs, for each of the five analysis problems. The data are obtained from graphs constructed by the baseline IFDS algorithm. We use the standard Tarjan's algorithm [143] to compute the SCC's after the IFDS analysis completes. The results in the table suggest there is a significant number of SCC nodes. Since reachability analysis has cubic complexity, if half of all nodes are in any SCC, then the analysis time can potentially be reduced to $\frac{1}{8}$ of the baseline.

Program	%FG nodes	File-Access SCC	File-Access %e-nodes	FSV SCC	FSV %e-nodes	Remote-FSV SCC	Remote-FSV %e-nodes	Remote-Access SCC	Remote-Access %e-nodes	FTP-Behavior SCC	FTP-Behavior %e-nodes
stunnel	18.9	72	27.2	40	32.9	148	31.1	148	31.1	792	32.2
pfinger	36.1	1207	51.2	1117	43.9	2921	45.5	2921	45.5	16937	44.5
muh-2.05c	28.6	654	30.6	384	33.7	941	32.1	941	32.1	5293	32.3
muh-2.05d	28.0	654	29.5	384	32.4	941	31.0	941	31.0	5302	31.2
pureftpd	36.2	1115	46.9	691	48.0	1720	42.7	1724	42.9	9976	40.4
fcron	36.5	1127	39.6	832	45.1	1549	41.7	1549	41.7	9814	39.7
apache	34.9	3017	30.3	1249	36.8	4148	33.7	4148	33.7	20733	32.3
make	51.4	4316	55.1	14310	57.2	13366	51.6	13366	51.6	?	
BlackHole	31.1	9766	37.5	12637	34.7	21489	33.5	21489	33.5	?	
openssh-client	17.5	4730	17.3	1406	17.0	4835	16.6	4835	16.6	32587	16.2
wu-ftpd-2.6.0	47.5	2370	63.9	3980	55.3	7115	57.7	7115	57.7	40418	58.2
wu-ftpd-2.6.2	47.2	5023	58.5	4883	54.0	9367	55.3	9367	55.3	53000	55.7
bind	39.5	2182	48.3	1367	46.9	3574	43.9	3574	43.9	20556	43.3
privoxy	25.7	22926	23.2	?		?		?		?	
openssh-server	21.3	5719	21.1	1552	23.1	6978	21.3	6978	21.3	39595	20.0
cfengine	34.2	16237	37.4	16668	37.2	32307	36.1	32175	36.1	?	
sqlite	40.6	6969	80.9	?		?		?		?	

Table 6.5: Percentages of loop nodes in flowgraphs exploded graphs. "?" indicates no data available.

6.6.2 No Speedup: Too Much, Too Late

In this and the next section, we report our experience when we apply cycle detection techniques to the IFDS algorithm and offer explanations why they fail to fulfill their promise for great speedups.

There are many cycle elimination techniques with different degrees of success. Tarjan's linear algorithm [143] requires the entire graph to be completely constructed. Online techniques [109, 90] detect cycles during graph construction, but they incur higher overhead. We tried applying these techniques to the IFDS analysis, but unfortunately, we could obtain only small speedups for a subset of programs.

There are two reasons to explain the results: the cycle detection overhead, and late detection. In order to explain these reasons, we need to better understand the behavior of the algorithms:

- The time spent in the IFDS algorithm can be broken into three components: initialize (build flowgraphs and transfer functions), graph construction, and reachability analysis. Figure 6.15 shows the breakdown for all the programs.
- All cycle detection techniques mentioned so far share the common characteristic that they perform detection after nodes and edges are created. They therefore add extra cost to graph construction, which already takes an average of 47% of total analysis time.
- On average, the reachability analysis takes up the smallest component, or 24% of total analysis time. Therefore, any improvement due to cycle elimination is already limited.

In other words, performing cycle detection after the graph is constructed may add too much overhead, and may be too late. For example, relative to baseline, Pearce and Kelly's technique [109] increases the time spent in graph construction to 63% (up from 47%), while time in reachability analysis reduces to 18% (down from 24%), giving us an overall *slowdown* of 11% among all programs; only four programs show any speedup.

6.6.3 Early Detection: Too Little, Not Enough?

The next question to ask is, can we do better than late detection? Can we avoid creating the strongly connected components in the graphs in the first place?

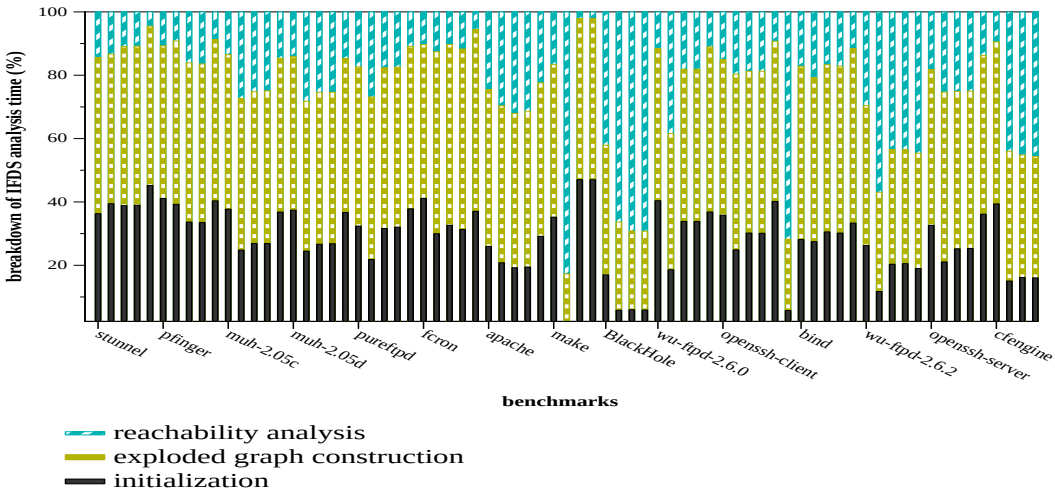


Figure 6.15: Breakdown of IFDS analysis time into initialization, graph construction, and reachability analysis. The averages for all programs are 29%, 47%, and 24%, respectively.

One way to do that is to use cycles in a flowgraph to drive the graph construction. For example, if there is no modification to a flow value d in a loop, then the nodes for d in the exploded graph will form an SCC. We therefore do not have to create all these nodes.

We experiment with an algorithm EC that adopts this strategy. The problem with EC is that it is limited to detecting cycles on nodes with the same flow value. Consequently, it is overshadowed by the Sparse IFDS algorithm, since the latter is already not creating cycles. Experiment results show that EC adds no benefit to Sparse IFDS. But since Sparse IFDS has to use a flow-sensitive pointer analysis, we also try applying EC to an IFDS analysis with FICI pointer analysis. Unfortunately, EC does not seem to affect performance at all.

6.7 Conclusion and Future Work

In this chapter, we have studied the performance of the IFDS algorithms in the presence of pointers. Our study has revealed that the original algorithm presented by Reps et al. is space inefficient, so it cannot complete on large programs with pointers where the datasets could be large. We modify the internal data structure in order to trade-off time for space.

The pointer analysis is critical to the IFDS algorithms: a flow-sensitive pointer analysis helps improve performance in all three dimensions—time, space, and precision—of the IFDS algorithm; a context-sensitive pointer analysis is even better except that it itself is too expensive—it only completes on a third of our benchmarks. A flow- and context-insensitive pointer analysis is worst: it leads to fewer completed cases, which are also less precise.

We have presented algorithms that reduce the graphs used in analysis. The difference between the original and reduced graphs, and the analysis cost on these graphs, represent unimportant computations. Our Sparse IFDS algorithm exploits the sparsity in the exploded graph by reducing the number of nodes and edges created in each statement. Our Variable-Pruning IFDS algorithm also reduces the graph, by reducing the set of variables—and hence, dataflow values—used in the graph construction. By reducing graph sizes, these algorithms are removing unimportant computations by (i) avoiding constructing nodes that carry redundant information, and by (ii) removing reachability analysis on these redundant nodes. Our results show that, on average, relative to the baseline algorithm's IFDS analysis time, the Sparse IFDS algorithm achieves 2.6× speedup, the Variable-Pruning IFDS algorithm achieves 3.3× speedup, while a combined algorithm of the two achieves 5.5× speedup, both relative to the baseline algorithm. When comparing total pointer and

IFDS analysis time, the new combined Sparse and Feedback-based Variable-Pruning IFDS algorithm is the best algorithm, which achieves 2.6× speedup.

Finally, our experiments also show that even with our improvement, our best IFDS algorithm is slower than the Client-Driven analysis, which is a very efficient and precise dataflow analysis algorithm.

Chapter 7

Related Work

We view our work in this thesis as part of our efforts to improve scalability of precise analyses by finding and removing unimportant computations in these analyses. There is no previous work that specifically studies, as a general problem, unimportant computations in static program analysis. In order to improve scalability, each previous work usually focuses on one dimension of precision. To place our work in context, we group the discussion in this section according to the major ideas in this dissertation.

7.1 Worklist

Flow-sensitive analyses often use a worklist; hence, the worklist algorithm plays a major role in the amount of repeated computations. Our solution to reduce unimportant computations in flow-sensitive analysis is through improving worklist algorithms. In this section, we will discuss existing worklist algorithms and other related algorithms.

Previous work on comparing worklist algorithms includes Atkinson and Gris-

wold's work [8], which shows that the performance difference between a round-robin algorithm and a worklist algorithm can be huge. They propose a hybrid algorithm that combines the benefits of the two. In separate work, Hind and Pioli [67] exploit loop structure by using a priority queue. Their algorithm still conservatively adds all reachable blocks to worklist. Their study on a worklist's impact on performance is one of the more recent. We find that Atkinson and Griswold's hybrid algorithm can sometimes be better and sometimes worse than Hind and Pioli's algorithm. All these studies do not measure the amount of repeated computations. To provide a basis for comparison with our new algorithm, we use as our baseline a version of the priority-queue approach that does not use the identity transfer function or IN/OUT sets. Our solution leads to a more efficient worklist, because we take data dependences into consideration. We find that exploiting def-use chains yields greater improvement than exploiting loop structure.

Other than worklist algorithms, the class of iterative algorithms also include round-robin and node listing [75, 2]. The round-robin approach is simple but is generally known to be inefficient, because it always revisits all blocks. A node listing is a statistically defined worklist order with repetition of nodes. This approach may be efficient for "bit-vector" separable dataflow problems, but it does not consider data dependences. Both the round-robin and node listing approaches are dense analyses in the sense that blocks are reanalyzed needlessly. Consequently, worklist algorithms are the only viable option for sparse analysis.

Besides iterative algorithms, the other family of dataflow algorithms are the elimination methods [132]. These methods, such as interval analysis, solve systems of equations. Two properties of these equations limit scalability of these methods. First, the use of IN/OUT sets in the equations inherently requires dense analysis.

Second, in the presence of pointers, the *GEN* and *KILL* sets in the equations may be unavailable or incomplete without complete pointer information in advance.

7.1.1 Other Approaches

For some classes of dataflow analysis problems, there exist techniques for efficient analysis. For example, demand algorithms also avoid unimportant computations: information irrelevant to answering queries are not computed. An example is the demand interprocedural dataflow analysis [70], which can produce precise results in polynomial time for interprocedural, finite, distributive, subset problems (IFDS). Unfortunately, this class excludes pointer analysis, and a separate pointer analysis phase may be required.

In the context of pointer analysis itself, previous work on flow-sensitive pointer analysis algorithms that makes use of worklists [131, 24] do not attempt to tune the worklist, so our worklist algorithm can be applied to such work to improve their performance. Other pointer analysis algorithms sometimes trade precision for scalability [66, 65, 62]. Our algorithm improves the efficiency of the worklist component that drives the analysis, without affecting the precision of the analysis.

Worklist algorithms have also been studied from other perspectives. For example, Cobleigh et al. [28] study the effects of worklist algorithms in model checking. They identify a few dimensions along which an algorithm can be varied. Their main result is that different algorithms perform best during different phases of analysis. We do not attempt to partition an analysis into phases. Similarly, we do not address the issue of partitioning the problem into subproblems [129], nor do we divide a large program into manageable modules [128, 125, 97]. We leave all these issues for future studies.

7.2 Sparse Analysis

The idea of exploiting sparsity to speedup analysis can be traced back to Reif and Lewis [117, 118]. Wegman and Zadeck use the idea by proposing using def-use chains to enable sparse constant propagation [147]. Their technique inspires our new worklist algorithm. We improve upon their approach by handling pointers, and we address the need to discover def-use chains on the fly as the analysis progresses. Our new approach poses the challenge that some information is not available when it is needed. Wegman and Zadeck's technique also help us develop our Sparse IFDS algorithm, in which we use def-use chains to create nodes in the graphs.

Another possible method of exploiting sparsity is to use a sparse evaluation graph (SEG) or its variants [25, 114], which are refinements of CFGs that eliminate irrelevant statements. Hind and Pioli report improvement with pointer analysis when an SEG is used [67], but because their use of *IN/OUT* sets does not fully exploit sparsity, it is unclear how much our worklist or our Sparse IFDS algorithm can benefit from an SEG, and we leave this study to future research.

7.3 Context-Sensitive Analysis

This section discusses work related to Relevance-Based Context Partitioning and Coupled Analyses.

7.3.1 Partitioning Contexts

In general, context-sensitive analysis improves the precision of analysis results [156, 83], but they can be very expensive. Previous solutions have recognized that exploiting similarities among contexts can improve performance. These previous work

did not explicitly discuss the roles of unimportant computation, and they did not consider the role of client analysis. For example, Grove et al. present a general framework for partitioning contexts [47]. Within this framework, the simplest approach to partitioning contexts is to limit context-sensitivity to the top k procedures on the call stack, but this solution loses precision and identifies partitions arbitrarily. Another example is Agesen's Cartesian Product Algorithm [1], which partitions procedure contexts based on parameters' concrete types at call sites, so as to improve type inference in programs with parametric polymorphism. Our Relevance-Based Context Partitioning is built on this algorithm, but we instead aims to improve pointers analysis in the presence of a client dataflow analysis.

Wilson and Lam [153] exploit the fact that many contexts of a procedure share the same input/output relationships. They use Partial Transfer Functions to group similar contexts, so that analysis results for one context may be reused for another, thus avoiding repeating some computations. Their PTF is one design choice from a space of PTFs, as described by Murphy and Lam [101]. The PTFs in this space differ in tradeoffs in complexity and precision. Their description of the domain space, however, does not consider the role of a client analysis. Therefore any choice is hardly justifiable based on performance needs *and* precision requirements by client analysis.

Our RBCP is conceptually similar to PTFs, but instead of partitioning contexts based on alias patterns, we partition based on the client's flow values, and instead of fixing a partitioning criterion at design time, we adaptively partition the contexts according to the client's needs. Our technique, therefore, is able to compute important flow values precisely, but computes unimportant flow values imprecisely. Our algorithm is applicable to a wide variety of dataflow analyses. It is not directly

applicable to pointer analysis in isolation: the pointer analysis has to be analyzed concurrently with a client analysis.

7.3.2 Other Context-Sensitive Analyses

More recently, BDD's have been used to cope with the high cost of context sensitive pointer analysis [150, 157]. The technique works well by exploiting similarities in the flow information, so that flow values from all contexts of a procedure can be represented compactly and completely. Consequently, analysis on a procedure is computed just once, aside from convergence on the call graph. For C programs, flow-insensitive, context-sensitive BDD-based analysis has been used to analyze programs on the order of 30,000 lines [9, 157]. The use of BDD's to perform flow-sensitive analysis remains an open problem. Besides, the algorithms sometimes require hand-tuning of the BDD data structure.

Client-Driven pointer analysis [54] provides selective flow-sensitivity and context-sensitivity as determined by the needs of the client analysis. Because only important flow values are computed precisely, unimportant computations are avoided. Relevance-Based Context-Sensitivity is complementary to Client-Driven analysis. The latter shrinks the set of procedures that are analyzed with context-sensitivity, while the former optimizes the context-sensitive analysis by reducing the number of contexts that must be considered. Client-Driven analysis interacts well with Relevance-Based Context-Sensitivity, improving both performance and precision.

7.3.3 Context-Loss Problem

Nystrom et al. [106] argue that there are two forms of context-sensitivity. The bottom-up context-sensitivity, which receives the attention of most previous work, focuses on eliminating propagations along unrealizable paths. The top-down context-sensitivity aims at removing the context-loss problem, which is imprecision in the callee due to the merging of flow values from different calls. Nystrom et al. present an efficient solution that only works for flow-insensitive pointer analysis. Our Relevance-Based Context Partitioning algorithm is both bottom-up and top-down context-sensitive for client analysis with pointer analysis, and our solution has no restriction on flow sensitivity.

Many other approaches construct procedure summaries [80, 36, 153, 21] and apply them to call sites. These approaches provide context-sensitive analysis, since they prevent propagation along unrealizable paths. They typically use symbolic names to represent invisible variables (e.g., `*p` where `p` is a formal parameter), so there are extra steps of mapping and unmapping variables at call sites. If only one summary is used per procedure, summary-based algorithms may suffer from the context-loss problem. Some recovery is possible by qualifying the summaries with constraints (such as assumed alias sets [80, 21]) or by using cloning-based techniques, including PTFs.

7.3.4 Coupled Analysis

Previous work [27, 19] has discussed the benefits of combining analyses and has described frameworks for integrating analyses. We instead focus on exploiting the relation between client and service analyses to improve the performance of the service analysis. The Client-Driven pointer analysis [54] was the first to exploit this

type of coupling, using the client to shrink the set of procedures analyzed in a context-sensitive manner. Relevance-Based Context Partitioning instead optimizes the context-sensitive analysis in a different way: with the help from the client analysis, RBCP partitions procedure contexts so each procedure is analyzed fewer times.

7.4 Reachability-Based Analysis

This section compares IFDS analysis to other types of analysis and describes how our new IFDS algorithms compare with previous work. Note that, in Section 7.2, we discussed how previous work on sparse analysis inspired our Sparse IFDS algorithm.

7.4.1 Relation to Other Analyses

The IFDS algorithm is based on an earlier framework defined by Sharir and Pnueli [136], who describe two general approaches to interprocedural analysis: the functional approach, which the IFDS algorithm belongs to, and the call string approach.

Solving dataflow problems by reducing them to reachability problems has been studied by Kou [78] and Cooper and Kennedy [30, 29]. All of these focus on intraprocedural and flow-insensitive problems. Reps et al.'s IFDS solution is more precise, because it is interprocedural and flow-sensitive.

The IFDS framework is useful for many distributive problems, including locally-separable problems such as reaching definitions and liveness analysis. Reps [120] discussed how program analysis can be solved by transforming them to graph-reachability problems. In particular, the general context-free-language (CFL) reachability problems are used in context-sensitive analysis because their solutions exclude unrealizable paths in program executions. He used as examples program

slicing, a version of shape analysis, and a flow-insensitive points-to analysis. In this dissertation, we focus on general IFDS problems.

Melski and Reps [94] showed that certain reachability problems are equivalent to certain set constraint problems. While this result has significant implications from both conceptual and implementation standpoints, we leave as future work the study of our ideas to set constraint problems.

The IDE framework is a strict generalization of IFDS framework. In the IFDS framework, the set of dataflow facts D_p is finite, while in the IDE framework, the lattice for D_p has finite height. While IFDS algorithms are used to solve realizable-path *reachability* problems, IDE algorithms are used to solve realizable-path *summary* problems. All IFDS problems can be encoded as IDE problems, but not vice versa. The copy-constant and linear-constant propagation problems are example problems solvable with the IDE framework. Sagiv et al. [133] report that running IFDS problems using IDE framework requires less memory and, therefore, can analyze larger programs. There is no other empirical evaluation of these algorithms that compare with other types of program analysis, such as dataflow analysis.

7.4.2 Variable Pruning

Weiser [148] first introduces the concept of *program slice* as a reduced, executable program S from a program P by removing statements, such that S replicates part of the behavior of P. Tip [144] provides a comprehensive survey and comparison of slicing techniques. Generally, there are two approaches to the problem: using dataflow equations [148] or using Program Dependence Graph (PDG) or other variants [148, 107, 69]. For example, a basic PDG-based algorithm can take up to

$O(V^3)$, where V is the number of PDG nodes. Adding features to this basic algorithm adds complexity to the new algorithms. For instance, to handle procedures, both context-insensitive [149] and -sensitive variants are presented [69, 71]. Different improvements are suggested in order to handle unstructured control flow [10, 23] and pointers [107, 68] correctly. There is also an algorithm to handle distributed programs [22]. On the other hand, there are also many algorithms that perform dynamic slicing [144, 96, 155].

The Variable Pruning algorithm does not utilize any of the flow equations or PDG variants used by the above slicing algorithms. Instead, we simply augment a pointer analysis with a low-cost operation—gather dependences—at each statement. Variable Pruning is very much a quick and deflated program slicing algorithm, especially given its flow-insensitive nature. That is, it does not trace the data and control dependences of statements. Therefore, it does not observe dominance relations between assignments, and it cannot perform statement pruning. When used to construct an exploded graph, the resulting graph can, therefore, be larger than if another more precise slicing algorithm is used.

While a more precise slicing algorithm can replace Variable Pruning, such algorithm comes with a higher cost. Even though the slicing algorithm may eliminate more nodes, these extra nodes do not contribute to answering the queries in the analysis; that is, the precision of IFDS analysis is not affected. Besides being simple and fast, Variable Pruning has the additional benefit that the dependence graph for a program can be recycled for another analysis problem: only the queries are different.

7.5 Cycle Elimination

Andersen first described the inclusion-based or constraint-based alias analysis in terms of type theory [6]. Faehndrich et al. [38] later reformulated the problem as computing the dynamic transitive closure of constraint graphs. Since then, techniques for cycle detection prove to be crucial for scalability of the alias analysis [60, 63, 110, 111]. On the other hand, Rountev et al. [126] introduce Offline Variable Substitution whose aim is to find and collapse pointer-equivalent variables. All these previous work reduce unimportant computations by detecting and removing equivalent information, effectively shrinking the problem size and dramatically reducing analysis time and memory usage. We tried to apply similar cycle elimination techniques to reduce the graphs used in reachability analysis. Unfortunately, we did not obtain significant improvement due to high overhead and limited opportunity for optimization.

Chapter 8

Conclusion

Developing precise and scalable program analysis continues to be challenging and demanding. Many previous work trades off between scalability and precision, which may result in unacceptable output qualities. We observe that some algorithms are precise but inefficient: they perform many unimportant computations, either because the same computations are repeated many times or because a lot of unimportant or irrelevant information is computed precisely. By identifying and removing these computations, we improve analysis time of analyses without sacrificing precision.

8.1 Contributions

In this dissertation, we have presented new algorithms for improving scalability of interprocedural program analysis. We identify and eliminate the source of repeated computations in flow-sensitive dataflow analysis by implementing a new worklist management algorithm. Our Relevance-Based Context Partitioning algorithm improves context-sensitive dataflow analysis by grouping similar contexts in such a way

that procedures are analyzed fewer times, and the analysis does not lose precision on important information. We also present new algorithms for reachability-based analysis. Our *Sparse IFDS* and *Variable-Pruning IFDS algorithms* and their variants work by reducing the graph used in an IFDS analysis. All our new algorithms produce significant improvement over existing to-date algorithms without sacrificing output qualities; in many cases, our new algorithm can complete when existing algorithms cannot, because they consume too much memory.

8.2 Future Work

Our work represents case studies of eliminating unimportant computations in precise program analysis. Besides flow- and context-sensitive analysis, there exist similar optimization opportunities in other precision dimensions. For example, not all paths are "interesting" in a path-sensitive analysis. In a field-sensitive analysis, certain fields may be merged or removed from the analysis when those fields do not carry important flow information relevant to a client analysis. Additional examples can be defined with object-oriented programs [130].

We observe that Relevance-Based Context Partitioning is an example of the broader notion of Coupled Analyses, where information from the client analysis can improve the performance of a service analysis. We believe this important paradigm is likely to become more important, and it deserves further study.

Bibliography

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECCOP'95 9th European Conference on Object-Oriented Programming, LNCS 952*, pages 2–26, 1995.

[2] A. V. Aho and J. D. Ullman. Node listings for reducible flow graphs. In *Proceedings of Seventh Annual ACM Symposium on Theory of Computing*, pages 177–185, 1975.

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[4] Stephen Alstrup, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[5] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Analysis of recursive state machines. In *Proc. 13th International Conference on Computer Aided Verification (CAV'01), LNCS 2102*, pages 207–220, 2001.

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[7] Paul Anderson, David Binkley, Genevieve Rosay, and Tim Teitelbaum. Flow insensitive points-to sets. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 79–89, November 2001.

[8] Darren C. Atkinson and William G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 52–61, November 2001.

[9] Dzintars Avots, Michael Dalton, V. Benjabin, Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, May 2005.

[10] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. In *In Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, Lecture Notes in Computer Science 749*, pages 206–222. Springer-Verlag, New York, May 1993.

[11] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate asbtraction of C programs. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 203–213, June 2001.

[12] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. of ICALP 2001, 28th Int. Colloq. on Automata, Languages, and Programming, Lecture Notes in Computer Science, Vol. 2076*, pages 652–666, July 2001.

[13] Jennifer Bevan and Jr E. James Whitehead. Identification of software instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 134–144, November 2003.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, pages 85–108. Springer-Verlag, 2002.

[15] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 196–207, June 2003.

[16] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[17] Paul R. Carini and Michael Hind. Flow-sensitive interprocedural constant propagation. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 23–31, 1995.

[18] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 115–125, January 2003.

[19] Craig Chambers, Jeffrey Dean, and David Grove. Framework for intra- and interprocedural dataflow analysis. Technical Report 96-11-02, University of Washinton, November 1996.

[20] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[21] Ramkrishna Chatterjee, Barbara G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 26th ACM Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, January 1999.

[22] Jingde Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Automated and Algorithmic Debugging*, pages 223–240, 1993.

[23] J. D. Choi and J. Ferrante. Static sicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[24] 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 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[25] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[26] Stephen Chong and Radu Rugina. Static analysis of accessed regions in recursive data structures. In Radhia Cousot, editor, *10th Annual International Static Analysis Symposium (SAS'03), volume 2694 of Lecture Notes on Computer Science*, June 2003.

[27] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[28] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. The right algorithm at the right time: comparing data flow analysis algorithms for finite state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 37–46, May 2001.

[29] K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–59, January 1988.

[30] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[31] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, September 1999.

[32] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[33] Manuvir Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 35–46, June 2000.

[34] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *8th Annual International Static Analysis Symposium (SAS'01), volume 2126 of Lecture Notes on Computer Science*, pages 260–279, 2001.

[35] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, October 2004.

[36] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[37] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlock. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[38] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[39] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new

fixpoint algorithm for distributive constraint systems. In *ESOP'98, 7th European Symposium on Programming, LNCS 1381*, pages 90–104, 1998.

[40] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In Radhia Cousot, editor, *10th Annual International Static Analysis Symposium (SAS'03), volume 2694 of Lecture Notes on Computer Science*, pages 439–462, June 2003.

[41] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 133–144, July 2006.

[42] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[43] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *ACM International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[44] 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 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[45] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Conference record of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338–350, January 2005.

[46] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, May 2004.

[47] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM SIGPLAN Conference On Object-Oriented Programming, Systems, Languages and Applications*, pages 108–124, 1997.

[48] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *Conference record of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 342–352, January 2004.

[49] Sumit Gulwani and George C. Necula. Precise interprocedural analysis using random interpretation. In *Conference record of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 324–337, January 2005.

[50] S.K.S. Gupta and S. Krishnamurthy. An interprocedural framework for determining efficient data redistributions in distributed memory machines. In *Sixth Symposium on the Frontiers of Massively Parallel Computing*, pages 233–240, Oct 1996.

[51] Samuel Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, pages 227–243. Springer-Verlag, 2002.

[52] Samuel Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, The University of Texas at Austin, May 2003.

[53] Samuel Z. Guyer, Emery D. Berger, and Calvin Lin. Detecting errors with configurable whole-program dataflow analysis. Technical Report TR-02-04, University of Texas at Austin, 2002.

[54] Samuel Z. Guyer and Calvin Lin. Client driven pointer analysis. In Radhia Cousot, editor, *10th Annual International Static Analysis Symposium (SAS'03), volume 2694 of Lecture Notes on Computer Science*, pages 214–236, June 2003.

[55] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *2nd Conference on Domain Specific Languages*, pages 39–53, October 1999.

[56] Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In R. F. Boisvert and P. T. P. Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, October 2000.

[57] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Conference record of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 310–323, January 2005.

[58] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[59] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of*

the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pages 69–82, June 2002.

[60] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[61] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, January 1995.

[62] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[63] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of C code in a second. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[64] Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *ACM SIGPLAN'92 Proceedings of the 1992 PLDI*, pages 249–260, 1992.

[65] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE 2001 Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[66] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In

ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000), pages 113–123, August 2000.

[67] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analysis. In *5th Annual International Static Analysis Symposium (SAS'98), volume 1503 of Lecture Notes on Computer Science*, pages 57–81, September 1998.

[68] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings SIGPLAN'89 Symposium on Compiler Construction*, pages 28–40, June 1989. Published as SIGPLAN Notices Vol. 24(7).

[69] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[70] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT'95 3rd Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[71] J. Hwang, M. Du, and C. Chou. Finding program slices for recursive procedures. In *Proceedings of the 12th Annual International Compiler Software and Applications Conference*, 1988.

[72] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. In *11th Annual International Static Analysis Symposium (SAS'04), volume 3148 of Lecture Notes on Computer Science*, Aug 2004.

[73] Neil D. Jones and Steven S. Muchnick. Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, pages 380–393. Prentice-Hall, 1981.

[74] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[75] K. W. Kennedy. Node listings applied to data flow analysis. In *Proceedings of the 2th ACM Symposium on Principles of Programming Languages*, pages 10–21, 1975.

[76] Jens Knoop, Oliver Rthing, and Bernhard Steffen. Towards a tool kit for the automatic generation of interprocedural data flow analyses. *Journal of Programming Languages*, 4(4):211–246, December 1996.

[77] Jens Knoop and Bernhard Steffen. Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework. Technical report, Institut fr Informatik und Praktische Mathematik, Christian-Albrechts-Universitt Kiel, Germany, Bericht Nr. 9309, 1993.

[78] L. T. Kou. On live-dead analysis for global data flow problems. *Journal of the ACM*, 24(3):473–483, July 1977.

[79] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[80] William Landi and Barbara G. Ryder. A safe approximate algorithm for

interprocedural aliasing. In *ACM SIGPLAN'92 Proceedings of the 1992 PLDI*, pages 235–248, 1992.

[81] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 24–31, June 1988.

[82] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[83] Ondrej Lhotak and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? Technical Report 2005-2, McGill University, 2005.

[84] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 199–215, 1999.

[85] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *8th Annual International Static Analysis Symposium (SAS'01), volume 2126 of Lecture Notes on Computer Science*, pages 279–298, July 2001.

[86] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 219–230, June 2004.

[87] Edward S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[88] Roman Manevich, Mooly Sagiv, G. Ramalingam, and John Field. Partially disjunctive heap abstraction. In *11th Annual International Static Analysis Symposium (SAS'04), volume 3148 of Lecture Notes on Computer Science*, August 2004.

[89] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 63–72, October 2004.

[90] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59(1):53–58, 1996.

[91] F. Masdupuy. *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. PhD thesis, Ecole Polytechnique, 1993.

[92] Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *In ECOOP 18th European Conference on Object-Oriented Programming*, pages 440–464, June 16-18 2004.

[93] Markus Müeller-Olm, Helmut Seidl, and Bernhard Steffen. Interprocedural analysis (almost) for free. Technical Report 790, Fachbereich Informatik, Universitt Dortmund, July 2004.

[94] David Melski and Thomas Reps. Interconvertibility of set constraints and

context-free language reachability. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, June 1997.

[95] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for pointsto and side-effect analyses for java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.

[96] Markus Mock, Darren C. Atkinson, Crag Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 71–80, 2002.

[97] Sungdo Moon, Xinliang Li, Robert Hundt, Dhruva Chakrabarti, Luis Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY — a framework for scalable cross-module IPO. In *2004 International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, pages 65–74, March 2004.

[98] Etienne Morel and Claude Renvoise. Interprocedural elimination of partial redundancies. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 6, pages 160–188. Prentice-Hall, 1981.

[99] Etienne Morel and Claude Renvoise. Global optimizations by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1981.

[100] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[101] Brian R. Murphy and Monica S. Lam. Program analysis with partial transfer functions. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 94–103, January 2000.

[102] Robert Muth and Saumya Debray. On the complexity of flow-sensitive dataflow analyses. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, 2000.

[103] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[104] Tim Newsham. Format String Attacks. www.lava.net/ newsham/format-string-attacks.pdf, September 2000.

[105] E. M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of 2004 SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 43–48, June 2004.

[106] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *11th Annual International Static Analysis Symposium (SAS'04), volume 3148 of Lecture Notes on Computer Science*, August 2004.

[107] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, pages 177–184, 1984.

[108] Jens Palsberg and Andrew A. Chien. Object-oriented type inference. In *OOPSLA'91: Conference Proceedings on Object-Oriented Programming, Systems, Languages, and Applications*, pages 146–161, November 1991.

[109] David. J. Pearce and Paul. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the Workshop on Efficient and Experimental Algorithms*, volume 3059 of *Lecture Notes in Computer Science*, pages 383–398, 2004.

[110] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of 2004 SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 37–42, June 2004.

[111] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 3–12, September 2003.

[112] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Conference record of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–255, January 2004.

[113] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.

[114] G. Ramalingam. On sparse evaluation representations. Research Report RC 21245(94831), IBM Research, July 1998.

[115] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, January 1999.

[116] G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 83–94, June 2002.

[117] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 104–118, 1977.

[118] John H Reif and Harry R Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences*, 32(3):280–313, June 1986.

[119] T. Reps and G. Rosay. Precise interprocedural chopping. In *ACM SIGSOFT'95 3rd Symposium on the Foundations of Software Engineering*, pages 41–52, October 1995. also in ACM SIGSOFT Software Engineering Notes 20(4).

[120] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, November/December 1998.

[121] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[122] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Proceedings of the 10th International Conference on Compiler Construction, number 2027 in Lecture Notes in Computer Science*, pages 133–149, April 2001.

[123] Noam Rinetzky, Jrg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure-local heaps and its abstractions. In *Conference record of the 32th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 296–309, January 2005.

[124] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th Annual International Static Analysis Symposium (SAS'05), volume 3672 of Lecture Notes on Computer Science*, September 2005.

[125] Atanas Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, August 2002. Technical Report DCS-TR-501.

[126] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 47–56, June 2000.

[127] Atanas Rountev and Beth Harkness Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering*, pages 254–263, May 2005.

[128] Atanas Rountev, Barbara G. Ryder, and William A. Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th European Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–253, September 1999.

[129] Erik Ruf. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, January 1997.

[130] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction, number 2622 in Lecture Notes in Computer Science*, pages 126–137, May 2003.

[131] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[132] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys (CSUR)*, 18(3):277–316, September 1986.

[133] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[134] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, January 1996.

[135] Zhong Shao. Flexible representation analysis. In *1997 ACM International Conference on Functional Programming*, pages 85–98, June 1997.

[136] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis.

In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[137] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[138] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University School of Computer Science, May 1991.

[139] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Conference record of the 31th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 26–38, January 2004.

[140] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[141] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 21–31, March 1998.

[142] Strom and Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Engineering*, 12(1), February 1986.

[143] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[144] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3, 1995.

[145] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *First Euromicro Conference on Software Maintenance and Reengineering, EUROMICRO 97*, pages 60–67, March 1997.

[146] Cheng Wang and Zhiyuan Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 119–130, June 2004.

[147] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[148] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[149] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1985.

[150] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[151] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *Pro-*

ceedings of the 9th International Conference on Compiler Construction, number 1781 in Lecture Notes in Computer Science, March 2000.

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

[153] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C program. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[154] C. L. Wong. *Thread Escape Analysis for a Memory Consistency Model Aware Compiler*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[155] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 94–106, June 2004.

[156] Bixia Zheng and Pen chung Yew. A hierarchical approach to context-sensitive interprocedural alias analysis. Technical Report 99-018, University of Minnesota, Computer Science and Engineering, April 1999.

[157] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157, June 2004.

Vita

Teck Bok Tok was born in Singapore on January 10, 1973, the son of Hock Bee Tok and Hoo Leong Ong. After graduating from high school, he served in the military for 30 months to fulfill the mandatory National Service. He then attended the National University of Singapore in July 1994, where he earned a Bachelor of Science degree with an Honors First Class designation in 1998, followed by a Master of Science degree in 1999. During this period, he was twice listed on the Dean's List, and has worked as a part-time teaching assistant for two years. He also briefly attended the University of Western Australia in February 1996 for one semester, as an exchange program student. In September 1999, he entered the Graduate School at The University of Texas at Austin to pursue a Ph.D. in Computer Sciences.

Permanent Address: Blk 434 Hougang Avenue 8
#11-912, Singapore 530434
Republic of Singapore

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.

Index

$\mathcal{DU}$, 57, 64

$\mathcal{DU}_{loop}$, 66

`memoryBlock`, 36

accuracy, 40

analysis configurations, 132

annotation language, 40

BB-change, 67

BB-visit, 67

benchmarks, 50

bundle, 55, 62

CFL reachability problems, 109

client analysis, 40

computation granularity, 16

context-insensitive heap model, 37

context-insensitive memory model, 37

context-loss problem, 23

context-sensitivity, 35, 169

contour, 79, 81

contour refinement, 80

coupled analysis, 104

def-use chain, 37

default contour, 82

Demand IFDS

- algorithm, 129
- results, 143

dominator tree, 44

expanded dominator trees, 45

exploded graph, 110

Feedback-based algorithms, 132

field-sensitivity, 27

flow-sensitivity, 35

heap model

- context-insensitive, 37

heap variables, 37

IFDS

- algorithm, 114
- analysis, 114
- baseline algorithm, 111, 114

problems, 110, 114, 115
independent-attribute analysis, 21

lattice, 15
levels of context-sensitivity, 35
levels of flow-sensitivity, 36

memory model, 36
context-insensitive, 37
default, 37

nearest reaching definition, 44

Partitioning Vector, 81
path edges, 116
pointer analysis, 39
FICI, FICS, FSCI, FSCS, 39
precision, 15, 16, 35, 40
precision dimensions, 2
PTF, 91, 103

queries, 40

RBCP, Relevance-Based Context Partitioning, 76
RBCS, Relevance-Based Context-Sensitive pointer analysis, 76
reachable value, 86
reachable variables, 84
realizable paths, 23, 109
relational-attribute analysis, 21
root procedure, 33

sound analysis, 17, 36, 39, 41, 48
Sparse IFDS
algorithm, 123
results, 139, 142
summary edge, 117

typestate, 48

use-def chain, 37

Variable-Pruning IFDS
algorithm, 125
results, 140, 142, 143

worklist efficiency, 67