SCALING CONTEXT-SENSITIVE POINTS-TO ANALYSIS

A Thesis

Submitted For the Degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Rupesh Nasre.

under the supervision of

Prof. R. Govindarajan

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

FEBRUARY 2012

TO

the beautiful moments
amidst the mad rush.

Acknowledgements

My PhD life has been supported by so many that acknowledgments turn out to be longer than the thesis abstract.

A research advisor is a person who always talks good about you despite all your tantrums. I learnt (and still learning) a great deal from Guruji – how to nurture a thought from idea to a presentable work, how to present a work well, and how to make your student go through a night-out for even a rebuttal. His ability to manage students amidst his administrative work is something which puzzles so many people around. He allowed me to work independently on problems that interested me and showed full trust in my limited ability. I thank him with deep gratitude.

The first paper in PhD life is always special. Therefore, a lot of credit is attributed to my co-authors Dr. Kaushik Rajan (now at Microsoft Research) and Prof. Uday Khedker (IIT Bombay). Kaushik was the one who saw the connect between points-to analysis and bloom filters. He also drove most of the implementation of that work – and still allowed me to lead the author list. My sincere thanks to him and the other members of HPC lab (Abhishek Udupa, Aditya Thakur, Ashwin Prasad, Girish B. C., Mrugesh Gajjar, R. Manikantan, and Sreepathi Pai). Uday Sir helped in clarifying our doubts about context-sensitivity and he being involved helped us relax on several aspects. Towards the later part of this work, he also introduced me to several faculty members for further career opportunities. I sincerely thank him.

While the work with Arnab De and Prof. Deepak D'Souza is not part of this thesis, it helped me learn about analyzing multithreaded programs. Discussions with Arnab have always been quite rewarding. In fact, it is due to him that I got my second paper; he introduced me to Newtonian Program Analysis on our way to coffee which eventually led me to develop a points-to analysis as a system of linear equations. My sincere thanks to him. Deepak Sir has always

been a source of simplicity. His comments as a reviewer of my Perspective Seminar and as an internal expert in my PhD colloquium were very useful. He also helped in my further career opportunities. I sincerely thank him.

I thank Dr. Subhajit Roy (now at IIT Kanpur) for some useful discussions on flow-sensitivity and context-sensitivity. He also gave very useful comments as a student reader for my Perspective Seminar.

I thank my comprehensive examination committee members: Prof. K. V. Raghavan, Prof. S. K. Nandi (SERC, IISc), Prof. Sunil Chandran, and Prof. Y. N. Srikant for not delaying my PhD by allowing me to clear the examination in the first attempt. I also thank the PhD interview panel: Prof. Hansdah, Prof. Priti Shankar, Prof. Shalabh Bhatnagar, Prof. T. Kavitha (now at TIFR), and Prof. Vijay Natarajan for selecting me. Thanks also to Prof. Abhik Roychoudhury, Prof. T. Kavitha, Prof. R. Govindarajan, and Prof. Y. N. Srikant for clearing me through their courses. Abhik also helped in my further career opportunities. Thanks a lot.

My interns Sandeep Putta (IIT Bombay) and Ankita Raman (NIT Nagpur) helped me learn several aspects of points-to analysis. Thanks guys.

Thanks to the reviewers of my accepted and rejected papers for spending their time in reading my work and giving me useful comments. Special thanks to the anonymous reviewer who suggested a comparison of solving points-to constraints using a linear solver against using a non-recursive constraint solver, in Section 6.7.3. Thanks to the following people/agencies for helping my conference travel: Prof. M. Narasimha Murty, IISc GARP funding, Microsoft Research travel grant, ACM SIGPLAN travel grant for CGO 2011, ACM SIGPLAN PAC travel grant and Ashwin Prasad (SERC, IISc).

Both, the former chairman Prof. M. Narasimha Murty and the current chairman Prof. Y. Narahari have been very supportive all throughout my PhD life. I happen to be on talking terms with almost all the faculty members in the department and their concern towards me, my work and family has made my stay in the department warm. The support staff in CSA office (Lalitha Madam, Suguna Madam, Meenakshi, Manju, Gopi, Babu and others), the technical staff (Jagadish Sir, BKP Sir, Megavannan Sir, Shankar Sir, and Ashalata Madam), and the security guards (Acharyaji, Gowdaji and Vaidyaji) have made me feel homely in the department. Every morning I receive a smiling greeting from them. My letters are delivered directly to me

and any of my requests are treated specially. I owe all of them my deep gratitude.

Thanks to the coffee breaks with Abhijit Khopkar, Arnab De, K. Vasanta Lakshmi, and Subramanya Bharadwaj for keeping me abreast with the surrounding developments in CSA, IISc, India and the world in general. Without those breaks, I would have still been living in 2007.

Thanks to Prof. Ambedkar Dukkipati, CCMD and the Director for their help in getting a married students' apartment inside campus. Our days in Ramanujan married students' apartments are going to be the most frequently remembered. Thanks also to Dr. Amal Medhi, Mitali Medhi, K. Vasanta Lakshmi, Arnab De, Payal De, Abhijit Khopkar, and Anamika, for several enjoyable evenings and for making each day of our stay rewarding. Special thanks to the milk and cylinder lorries to A-mess for some entertaining moments.

Thanks to the CSA servers *euler, neumann* and *pingu* and the student admins for never making me use the data backups. Thanks to the IISc network and CSA/SERC network admins for not creating any trouble at the last hour of my paper submissions. Thanks to Gmail for being an alternative source of sending emails whenever CSA webmail was down.

Thanks to Voices, the IISc newsletter for a wonderful time and for some life-long contacts in the form of Shyam and Sudhira.

Thanks to my friends Abhay, Meghana, Prachi, Prashant, and Rahul, for just being there.

Thanks to my parents in and outside law for their support despite the frequent exclamations in the family like "He still doesn't earn!".

No thanks to our son Chinmay due to whom *the mad rush* began. But it is he who made me learn that research is easier than parenting. His innocence and pranks made us leave our negative thoughts outside home to enjoy *now and here*. My blessings to him.

It appears customary to thank the spouse in the end and I didn't want to break the norm. Meghana has been part of everything I have been doing and not doing. This thesis is as much hers as is mine.

Publications based on this Thesis

1. R. Nasre and R. Govindarajan. Prioritizing Constraint Evaluation for Efficient Points-to Analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, 2011, Chamonix, France, 2011. ACM.

2. R. Nasre and R. Govindarajan. Points-to Analysis as a System of Linear Equations. In *Proceedings of the 17th International Conference on Static Analysis*, SAS '10, pages 422–438, Perpignan, France, 2010. Springer-Verlag.

3. R. Nasre, K. Rajan, R. Govindarajan, and U. P. Khedker. Scalable Context-Sensitive Points-To Analysis Using Multi-Dimensional Bloom Filters, In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 47–62, Seoul, Korea, 2009. Springer-Verlag.

Abstract

Pointer analysis is one of the key static analyses during compilation. The efficiency of several compiler optimizations and transformations depends directly on the scalability and precision of the underlying pointer analysis. Recent advances still lack an efficient and scalable context-sensitive inclusion-based pointer analysis. In this work, we propose four novel techniques to improve the scalability of context-sensitive points-to analysis for C/C++ programs.

First, we develop an efficient way of storing the approximate points-to information using a multi-dimensional bloom filter (multibloom). By making use of fast hash functions and exploiting the spatial locality of the points-to information, our multibloom-based points-to analysis offers significant savings in both analysis time and memory requirement. Since the representation never *resets* any bit in the multibloom, no points-to information is ever lost; and the analysis is sound, though approximate. This allows a client to trade off a minimal amount of precision but gain huge savings (two orders less) in memory requirement. By making use of multiple random and independent hash functions, the algorithm also achieves high precision and runs, on an average, 2× faster than Andersen's points-to analysis. Using Mod/Ref analysis as a client, we illustrate that the precision is above 98% of that of Andersen's analysis.

Second, we devise a sound randomized algorithm that processes a group of constraints in a less precise but efficient manner and the remaining constraints in a more precise manner. By randomly choosing different groups of constraints across different runs, the analysis results in different points-to information, each of which is guaranteed to be sound. By *joining* the results of a few runs, the analysis obtains an approximation that is very close to the one obtained by the more precise analysis and still proves efficient in terms of the analysis time. We instantiate our technique to develop a randomized context-sensitive points-to analysis. By varying the level of randomization, a client of points-to analysis can trade off minimal precision (less than 5%)

for large gain in efficiency (over 50% reduction in analysis time). We also develop an adaptive version of the randomized algorithm that carefully varies the randomization across different runs to achieve maximum benefit in terms of analysis time and precision without pre-setting the randomization percentage and the number of runs.

Third, we transform the points-to analysis problem into finding a solution to a system of linear equations. By making novel use of prime factorization, we illustrate how to transform complex points-to constraints into a set of linear equations and transform the solution back as a points-to solution. We prove that our algorithm is sound and show that our technique is 1.8× faster than Andersen's analysis for large benchmarks.

Finally, we observe that the order in which points-to constraints are processed plays a vital role in the algorithm efficiency. We prove that finding an optimal ordering to compute the fixpoint solution is NP-Hard. We then propose a greedy heuristic based on the amount of points-to information computed by a constraint to prioritize the constraints. This results in a dynamic ordering of the constraint evaluation which, in turn, results in skewed evaluation of constraints where each constraint is evaluated repeatedly and different number of times in a single iteration. Our prioritized analysis achieves, on an average, an improvement of 33% over Andersen's points-to analysis.

We illustrate that our algorithms help in scaling the state-of-the-art pointer analyses. We also believe that the techniques developed would be useful for other program analyses and transformations.

Contents

Acknowledgements **i**

Publications based on this Thesis **v**

Abstract **vii**

1 Introduction **1**
1.1 What is Pointer Analysis? . . . 2
1.2 Clients of Pointer Analysis . . . 3
1.3 Issues with Scalable Pointer Analysis . . . 3
1.4 Our Contributions . . . 4
1.5 Organization of this Thesis . . . 7

2 Background **9**
2.1 Definitions and Nomenclature . . . 9
2.2 Analysis Dimensions . . . 14
2.2.1 Flow-sensitivity . . . 14
2.2.2 Field-sensitivity . . . 15
2.2.3 Context-sensitivity . . . 16
2.3 Computability and Complexity . . . 18
2.4 Two Key Points-to Analysis Methods . . . 19
2.4.1 Normalized Input Format . . . 20
2.4.2 Andersen's Inclusion-based Analysis . . . 21
2.4.3 Steensgaard's Unification-based Analysis . . . 21
2.5 Inclusion-based Points-to Analysis as a Graph Problem . . . 22
2.6 Chapter Summary . . . 25

3 A Survey of Pointer Analysis Methods **27**
3.1 Introduction . . . 27
3.2 Surveys . . . 29
3.3 Complexity Results . . . 31
3.4 Use of Novel Data Structures . . . 32
3.5 Optimizations and Techniques . . . 33
3.6 Exact Methods . . . 35
3.7 Methods Achieving Explicit Trade-offs . . . 37
3.8 Client-driven and Demand-driven Methods . . . 39
3.9 Incremental and Probabilistic Methods . . . 41

3.10 Analysis of Parallel Programs and Parallel Analyses . . . 42
3.11 Application of Points-to Analysis . . . 43
3.12 Evaluations and Quantifications . . . 45
3.13 Points-to Analysis for Other Languages . . . 47
3.14 Chapter Summary . . . 48

4 Points-to Analysis using Bloom Filter 49
4.1 Introduction . . . 49
4.2 Bloom Filter . . . 51
4.2.1 Issues with a Naive Bloom Filter . . . 53
4.3 Multi-dimensional Bloom Filter . . . 53
4.3.1 Handling Copy Constraint . . . 54
4.3.2 Handling Load and Store Constraints . . . 56
4.3.3 Extracting Information from Multibloom . . . 59
4.4 Context-sensitive Analysis . . . 63
4.5 Experimental Evaluation . . . 65
4.5.1 Performance of Exact Analysis: Baseline . . . 67
4.5.2 Performance of Multibloom: Overall Effect . . . 69
4.5.3 Effect of Parameter C . . . 70
4.5.4 Effect of Parameter D . . . 70
4.5.5 Effect of Parameter B . . . 71
4.5.6 Effect of Parameter S . . . 71
4.5.7 Effect of Selected Configurations . . . 74
4.6 Comparison with Other Analyses . . . 77
4.7 Mod/Ref Analysis as a Client . . . 78
4.8 Related Work . . . 79
4.8.1 Methods using Novel Data Structures . . . 80
4.8.2 Use of Bloom Filters in Other Applications . . . 81
4.9 Chapter Summary . . . 82

5 Sound Randomized Points-to Analysis 83
5.1 Introduction . . . 83
5.2 Unification versus Inclusion . . . 84
5.2.1 Overview of the Approach . . . 84
5.2.2 Selection, Summarization and Composition . . . 86
5.2.3 Implementation Challenges . . . 87
5.2.4 The Algorithm . . . 88
5.3 Randomized Context-sensitivity . . . 89
5.4 Soundness . . . 92
5.4.1 Remark . . . 96
5.5 Experimental Evaluation . . . 96
5.5.1 Overall Effect of Representative Configurations . . . 97
5.5.2 Effect of Selection Probability . . . 99
5.5.3 Effect of Number of Runs . . . 101
5.5.4 Benchmarks with High Precision Loss . . . 101
5.5.5 Comparison with k-Context-sensitivity . . . 102
5.5.6 Effect of Mixed Randomization . . . 103

5.5.7 Adaptive Analysis . . . 104
5.6 Other Analysis Dimensions . . . 106
5.7 Related Work . . . 106
5.8 Chapter Summary . . . 107

6 Points-to Analysis as a System of Linear Equations 109
6.1 Introduction . . . 109
6.2 Naive Approach . . . 110
6.2.1 Issues . . . 111
6.3 Prime-Factorization Approach . . . 114
6.3.1 Step 1: Preprocessing . . . 114
6.3.2 Step 2: Solving the System . . . 117
6.3.3 Step 3: Post-processing . . . 117
6.3.4 Step 4: Evaluating Special Constraints . . . 118
6.3.5 Subsequent Iterations . . . 119
6.4 The Algorithm . . . 121
6.4.1 Solution Properties . . . 125
6.4.2 Implementation Issues . . . 125
6.5 Soundness and Precision . . . 125
6.6 Client Analysis . . . 130
6.7 Experimental Evaluation . . . 131
6.7.1 Analysis Time . . . 132
6.7.2 Memory . . . 134
6.7.3 Comparison with Bitwise Operations . . . 135
6.8 Related Work . . . 137
6.9 Chapter Summary . . . 138

7 Prioritizing Constraint Evaluation for Efficient Points-to Analysis 141
7.1 Introduction . . . 141
7.2 Optimal Constraint Ordering . . . 143
7.3 Prioritized Computation of Constraints . . . 146
7.3.1 Example . . . 146
7.4 Prioritization Framework . . . 150
7.4.1 Priority Schemes . . . 151
7.5 The Algorithm . . . 152
7.6 Experimental Evaluation . . . 155
7.6.1 Analysis Time . . . 155
7.6.2 Memory . . . 157
7.6.3 Overall Effect . . . 158
7.6.4 Effect of Bucketization . . . 159
7.6.5 Effect of Skewed Evaluation . . . 159
7.6.6 Comparison with Priority Queue . . . 161
7.7 Related Work . . . 163
7.8 Chapter Summary . . . 163

8 Conclusions and Future Work **165**
8.1 Summary 165
8.2 Future Work 167

References **169**

List of Tables

4.1 Benchmark characteristics 66
4.2 Performance of *exact* analysis. 68
4.3 Sensitivity to parameter C 71
4.4 Sensitivity to parameter D 72
4.5 Sensitivity to parameter B 73
4.6 Sensitivity to parameter S 75
4.7 Effect of select configurations on performance. 76
4.8 Time(seconds) and memory(MB) required for context-sensitive analysis 77

6.1 Time required in seconds for context-insensitive analysis 133
6.2 Time(seconds) and memory(MB) required for context-sensitive analysis 134
6.3 Memory required in MB for context-insensitive analysis 135
6.4 Time and memory comparison with bitwise operations 136

7.1 Analysis time (seconds) 156
7.2 Memory requirement (MB) 157
7.3 Comparison with Priority Queue: Analysis Time (seconds) 162

List of Figures

1.1 Placement of our contributions in pointer analysis algorithm 5

2.1 Constraint graph for Example 2.4 23
2.2 Constraint graph for Example 2.6 24

4.1 Example program to illustrate points-to analysis using bloom filters 55
4.2 Example program using two hash functions 56
4.3 Example program to illustrate handling complex statements 59
4.4 Example program to illustrate multiple analysis iterations 61
4.5 Placement of our analysis in LLVM compilation 65
4.6 Overall effect of various configurations. 69
4.7 Mod/Ref client analysis. 79

5.1 Unification versus Inclusion 85
5.2 Context-sensitivity (a) Original invocation graph (b) Modified invocation graph with function `f` summarized 91
5.3 Example to illustrate randomized context-sensitivity 91
5.4 Effect of two representative configurations 97
5.5 Overall effect across various configurations 98
5.6 Effect of selection probability ρ 99
5.7 Effect of number of runs N 100
5.8 Effect of some configurations over programs with high precision loss 101
5.9 Randomized versus k-context-sensitive analysis 102
5.10 Effect of mixed randomization 104
5.11 Adaptive analysis 105

6.1 Example to illustrate points-to analysis as a system of linear equations 110
6.2 Lattice over the compositions of primes guaranteeing five levels of dereferencing 117

7.1 (a) Input constraints and fixed constraint ordering for Deep Propagation (b) Constraint graphs for Deep Propagation 147
7.2 (a) Input constraints (b) Constraint graphs for Prioritized Deep Propagation . 148
7.3 Effect of prioritization for *vortex, art, vpr* and *gap* respectively 160
7.4 Effect of bucketization 161
7.5 Effect of skewed evaluation 161

Chapter 1

Introduction

Industrial code-bases are getting bigger. Code-bases with billions of lines of source code are no longer uncommon. Compilation of such huge programs typically takes several hours. The scalability of compilation is, therefore, a minimum requirement for current compiler frameworks like *gcc* [40] and *LLVM* [81]. Compilation of a program involves several static analyses which analyze and optimize the program. A static analysis of a program automatically analyzes the behavior of the program without actually running the program [131]. A static analysis can help one achieve faster runtime execution, more secure code, less buggy programs, invariant guarantees and a better program understanding, among other benefits. Since static analysis is an inseparable component of a compiler, compiler writers and analysis designers must strive hard to make their analyses as efficient as possible.

Pointer analysis is a static analysis to find out possible memory locations pointed to by various pointer variables in a program. It takes a program as input and computes, so called, points-to information about the program. This points-to information is used by other static analyses, called as clients, to optimize the program. The effectiveness of a client depends heavily on the points-to information computed by the underlying pointer analysis. For instance, it has been shown that if the pointer analysis is more precise, i.e., it adds less number of approximations, its client's analysis time can be significantly reduced [57]. With the enormous growth of the code-bases and the pressing need for efficient compilation, the scalability requirement of pointer analysis is unquestionable.

The points-to information computed by a pointer analysis depends upon how various program elements are modeled. Modeling these program elements in various ways results in different points-to information with varying analysis time and memory requirements. These program elements are often termed as analysis dimensions [57]. Some of these analysis dimensions relate to the control flow in the program (flow-sensitivity), to the calling context of functions (context-sensitivity), to the modeling of aggregates (field-sensitivity), to the modeling of the heap, to whole program versus local compilation, to the representation of points-to information, etc. More precise the modeling, more precise is the pointer analysis and, in effect, better is the points-to information computed for a client. For instance, a context-sensitive analysis considers the calling context of a function while analyzing the function. A context-insensitive analysis, on the other hand, ignores the calling context. Making a pointer analysis context-sensitive makes it more precise compared to its context-insensitive counterpart. However, on the downside, it also makes the analysis inefficient in terms of analysis time and memory requirement. Similarly, a flow-sensitive pointer analysis is relatively more precise but less efficient than a flow-insensitive pointer analysis. Achieving performance across all the three aspects, namely, analysis time, memory requirement and precision has been posing severe challenges to the pointer analysis research community. In this work, we deal with the problem of scaling context-sensitive pointer analysis and propose several novel ways to improve its performance. We mainly focus on C/C++ kinds of programs, but the techniques are extensible to other general purpose imperative languages like Java.

1.1 What is Pointer Analysis?

Pointer analysis or alias analysis is a mechanism to statically determine whether two pointers may point to the same memory location at run-time. Given a program `P` which uses pointers, a pointer analysis processes `P` to extract pointer specific information and computes an internal representation of the aliasing information present in `P`, called as points-to information. A client to the pointer analysis then queries this information to obtain answers to its specific queries regarding the alias relationship between pointers.

Several points-to analysis algorithms exist in literature (see a survey in Chapter 3). However, two of the most widely used algorithms are due to Andersen [3] and Steensgaard [123].

Andersen's analysis is based on inclusion of points-to sets and is more precise but less efficient than Steensgaard's analysis, which is based on unification.

1.2 Clients of Pointer Analysis

Pointer analysis is not an optimization in itself, i.e., once pointer analysis is run, it does not make the program run faster or become more secure. A program transformation needs to query alias information from pointer analysis to create an optimized version of the program. Further, there could be other analyses that do not alter the code but make use of pointer analysis to compute better dataflow information (to be used by other transformations). All such transformations and analyses are called as the clients of pointer analysis. Examples of transformation clients include automated bug correction, parallelization, common subexpression elimination. Examples of analysis clients include slicing, shape analysis and identification of security vulnerabilities.

1.3 Issues with Scalable Pointer Analysis

The effectiveness of a client optimization depends heavily on the underlying pointer analysis [57]. For instance, if the pointer analysis is very *precise*, a client's execution time gets sharply reduced. One of the key ways to make pointer analysis more precise is to make it context-sensitive, which takes into account the calling context of a function while analyzing a program statement. However, context-sensitivity has the potential of exponentially blowing up the space and time requirements of the analysis. For instance, the context-sensitive version of Andersen's analysis [3], one of the most widely used points-to analyses, without any optimizations runs out of memory on SPEC 2000 benchmark *176.gcc* on an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM. There have been several attempts towards achieving a scalable context-sensitive implementation of pointer analysis and those have been only partially successful. For instance, the use of Binary Decision Diagrams (BDD) has improved the memory requirement of pointer analysis. However, the execution times of the analyses using BDD may prove limiting in certain scenarios. As an example, Whaley and Lam [129] proposed a cloning-based context-sensitive pointer analysis using BDD which successfully analyses 687 KB of Java bytecode but requires around 20 minutes to complete. In another method, Lattner et al. [75]

proposed another cloning-based context-sensitive pointer analysis (without using BDD) which analyses a 200 kilo lines of C source code in 3 seconds, and claims significant performance benefits, but offers a precision only equal to that of a *context-insensitive* analysis in many cases due to unification of contexts (in some cases, it is more precise than context-insensitive inclusion-based analysis). In one of the recent works, Kahlon [66] proposed a bootstrapping technique for a scalable context-sensitive pointer analysis which has been shown to scale only upto 128 kilo lines of C code requiring around 300 seconds.

From these examples we see that a context-sensitive analysis which offers benefits in all the three dimensions – precision, analysis time and memory requirement – is still missing. We propose to offer solutions to fill up this void in this thesis.

1.4 Our Contributions

A pointer analysis may be pictorially viewed as shown in Figure 1.1. It has the following key elements.

- An input program `P` as a sequence of statements. The statements are converted into a suitable internal format called as constraints `C`.
- An algorithm `A` that iterates over the constraints `C` to compute points-to information.
- A storage `S` that stores the computed points-to information to be used by clients.

Our key contributions towards scaling a context-sensitive pointer analysis can now be depicted using Figure 1.1. The numbers in the figure correspond to the appropriate item below. The first two contributions are approximate, i.e., they offer a trade-off between precision and scalability. The other two contributions are *exact*, i.e., they offer a precision equal to that of an equivalent inclusion-based analysis.

1. To reduce the memory requirement of a context-sensitive pointer analysis, we propose to store the points-to information using hashing in a *bloom filter*. A bloom filter is a probabilistic data structure that allows us to store only a few bits per points-to fact. This reduces the analysis memory requirement to an order of magnitude less than the base analysis which uses sparse bitmaps. Without affecting the analysis soundness, our

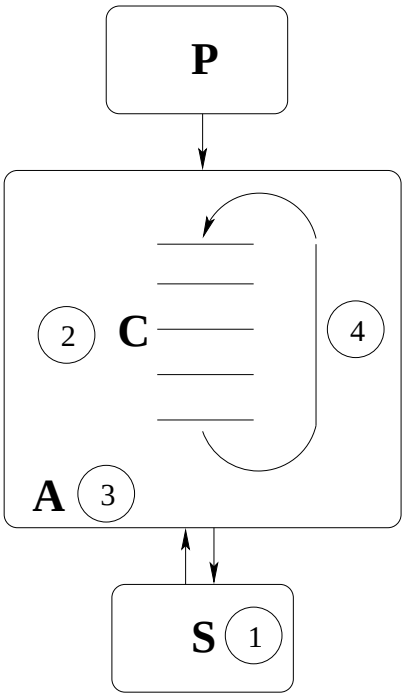


Figure 1.1: Placement of our contributions in pointer analysis algorithm

analysis also achieves significant improvements in the analysis time. We extend basic bloom filter to a multi-dimensional bloom filter to support pointer analysis operations. We also give probabilistic guarantees on the precision loss due to hashing the points-to information, and empirically show that our analysis achieves almost full precision in practice. Specifically, our analysis is able to achieve 75% reduction in memory requirement and 40% improvement in analysis time with less than 2% precision loss compared to Andersen's analysis [3]. A salient feature of our approach is that by controlling the size of the bloom filter, a client also gets an opportunity to trade off the right amount of precision for scalability.

2. Typically, all the points-to constraints `C` in Figure 1.1 are processed in the same manner. We propose a randomized, yet sound, pointer analysis technique that partitions `C` into two groups and processes each group in a different manner. Specifically, the idea is to randomly choose a set of constraints to be processed in a less precise manner (say context-insensitively) and the remaining in a more precise manner (say context-sensitively). Then, by merging the points-to information obtained from the two analyses (in an iterative manner), we get a points-to solution which lies in between the two precision limits. We prove

that the proposed randomized approach is sound, i.e., all the realizable points-to facts are computed by our randomized analysis. Further, we demonstrate that randomization works very well in practice and there exist several configurations using which one can achieve over 50% reduction in analysis time with less than 5% precision loss for a context-sensitive analysis. The technique is quite general in nature and can be applied to other analysis dimensions.

3. The algorithm `A` in Figure 1.1 may be replaced by another algorithm `A'` if the problem of pointer analysis can be reduced to the problem solved using `A'`. We propose to polynomially reduce the iterative pointer analysis problem to solving a set of linear equations. This would enable us to use a standard linear solver to solve the pointer analysis problem and to make use of various enhancements proposed in literature for solving a set of linear equations. However, this problem reduction poses several issues due to address-taken variables, pointer dereferencing and because a single pointer may point to multiple variables in the points-to solution. Further, two of the biggest challenges are to maintain *idempotency* of points-to analysis (adding a points-to fact to the solution multiple times should be equivalent to adding it once) while working with numbers and to keep the equations linear in each iteration. We solve all these issues using novel mechanisms based on prime factorization and show that, in practice, our approach is well suited especially for large programs. Specifically, our method is 43% faster than an optimized Andersen's analysis [3].

4. We observe that the order in which points-to constraints `C` are evaluated affects how quickly the points-to information is computed. In this direction, we first prove that finding an optimal constraint ordering is NP-Complete. We then propose a prioritization framework wherein different constraint orderings could be applied. The framework is quite general and we apply it to Andersen's analysis [3], Deep Propagation [100], and BDD-based Lazy Cycle Detection [49]. We also instantiate the framework with a greedy heuristic based on the amount of points-to information newly computed by a constraint in each iteration and show 16–44% improvement in analysis time. Further, we show that our technique gives significant memory improvements over the base analyses that use difference propagation wherein only the changed points-to information is propagated.

1.5 Organization of this Thesis

This thesis is organized as follows. We describe the necessary definitions and terms to understand the rest of the thesis in Chapter 2. Specifically, we explain with examples various analysis dimensions like flow-sensitivity, context-sensitivity and field-sensitivity which affect the performance of a pointer analysis. We also mention the results on computability and complexity of various forms of pointer analysis which have triggered three decades of literature on this important analysis.

In Chapter 3 we survey various pointer analysis methods. Due to the volume of the work done in this area, we divide the survey in 12 categories and discuss the work in each of these categories.

In Chapter 4 we explain how bloom filters can be used to improve storage requirements of pointer analysis. After introducing a bloom filter, we discuss why a naive bloom filter is unsuitable for pointer analysis. The discussion also forms the basis for extending a bloom filter to a *multibloom*. We illustrate how various pointer analysis operations can be modeled using a multibloom. Using extensive experimentation we carefully study the effect of various configuration parameters on the analysis performance. Since pointer analysis using multibloom affects precision, we also study its effect on the precision of Mod/Ref analysis as a client.

In Chapter 5 we present our randomized pointer analysis algorithm. We first explain our approach for two types of analyses: unification and inclusion, and introduce the building blocks of our approach, namely, selection, summarization and composition. We then apply our randomization technique to context-sensitivity and describe, using an example, how the selection, summarization and composition can be implemented. We show that our randomization approach is sound. Finally, we evaluate our randomization scheme with a detailed experimentation.

In Chapter 6 we translate the points-to analysis problem into that of solving a system of linear equations. We show the non-triviality of this problem-reduction by first describing a simple transformation and showing why it proves insufficient to cover all the aspects of an iterative pointer analysis algorithm. We then propose our novel approach based on prime-factorization of numbers and illustrate with an example how a complete pointer analysis algorithm can be modeled with it. After presenting the formal context-sensitive algorithm, we prove that it is sound and as precise as other context-sensitive inclusion-based pointer analyses. We then qualitatively explain how various clients can get benefited by the representation of points-to

sets as numbers. Finally, we study how our novel approach performs in practice by comparing it against state-of-the-art methods.

In Chapter 7 we propose the notion of a constraint priority to efficiently compute the points-to solution. After motivating the need for a better constraint ordering to achieve the solution faster, we show that finding an optimal ordering is NP-Complete. We then propose a greedy heuristic based on the amount of points-to information newly added by a constraint to decide the priority of the constraint. We then present the complete prioritization framework wherein different priority schemes can be plugged in. We instantiate the framework with the proposed greedy heuristic and explain our prioritized pointer analysis algorithm. We also show the effectiveness of our approach with an extensive experimental evaluation.

We summarize our work in Chapter 8 and also explain a few key directions in which this work can be extended.

Chapter 2

Background

In this chapter we present key concepts that are necessary to understand the forthcoming chapters. We first define important terms and then present various dimensions of points-to analysis that affect the precision of a pointer analysis algorithm. In Section 2.3, we present the computability and complexity results on various forms of pointer analysis. In Section 2.4, we present two important pointer analysis algorithms that are used extensively in the rest of the thesis, namely, Andersen's inclusion-based analysis [3] and Steensgaard's unification-based analysis [123]. Finally, in Section 2.5, we present points-to analysis as a graph problem [34] which will help readers understand the structure of points-to information computation better.

2.1 Definitions and Nomenclature

A pointer in a general purpose language like C/C++ is declared as `T *ptr` where `ptr` is the name of the pointer variable and `T` is the type of the variable it points to. Thus, a pointer declared as `int *ptr` can point to a variable of type integer.

Definition 2.1 (**Pointer**). *A pointer is a program variable whose r-value is either 0 (null) or the address of another variable.*

Definition 2.2 (**Pointee**). *A pointee is a program variable or a (heap) location whose address is the r-value of a pointer.*

A pointer is said to *point to* a pointee. Typically, a pointer-pointee relationship is established by an address-of assignment or by a memory allocation statement as below.

1: `ptr = &var;`

2: `ptr = (int *)malloc(10);`

Statement 1 is called an *address-of assignment* and `var` is called an *address-taken* variable. Here, `ptr` is a pointer and `var` is a pointee. Same variable may act as both a pointer and a pointee.

Unlike in Java, stack variables can act as pointees in C. Since C is a weakly-typed language, it is possible that a pointer declared to be pointing to a pointee of one type may point to another of some other type. Therefore, several pointer analysis algorithms ignore types altogether. In this thesis, we follow this tradition and assume that all the variables are potential pointers.

An expression `*ptr` is said to *dereference* pointer `ptr` to obtain the ℓ-value of its pointee. If `ptr` is null, i.e., it points to nothing, then its dereference is considered as invalid, which typically results into a runtime fault.

Definition 2.3 (**Aliases**). *Two pointers are aliases if they point to the same pointee.*

The relationship between two aliases is called an aliasing relationship, alias relationship or, simply, aliasing. In general, aliasing is defined as two expressions that evaluate to the same memory location. In C, aliasing may occur due to pointers, array indexing and `union` variables. The following assignment statement, as an example, makes `ptr1` and `ptr2` aliases of each other: `ptr2 = ptr1`.

The definition above is given for two pointers, but it can be readily extended for more than two pointers. Aliasing is a reflexive relation, i.e., `p` aliases with `p`. It is also a symmetric relation, i.e., `p` aliases with `q` implies `q` aliases with `p`.

If it is guaranteed that two pointers `p` and `q` always (i.e., in all the program executions) alias with each other at a program point, then they are said to be *must-aliases* of each other. If two pointers `p` and `q` alias with each other at some program point in some program execution, then they are said to be *may-aliases*. Identification of must-aliases helps in reducing the number of variables tracked during the analysis and hence helps in an efficient alias analysis. Henceforth, unless otherwise mentioned, we mean may-aliasing by the term aliasing.

Definition 2.4 (**Pointer Analysis, Alias Analysis**). *Pointer analysis or alias analysis is the mechanism of statically finding the aliasing relationship between pointers in a program.*

A pointer analysis may be *exhaustive* and compute the aliasing relationship for all possible

pointer pairs. On the other hand, a pointer analysis may be *demand-driven* which computes the aliasing relationship only for a given set of pointer pairs, computing more and more aliasing information as further queries are answered.

Note that an exhaustive pointer analysis could be very costly in terms of memory requirement [31]. Computing and maintaining the alias relationship for *all* the pointer pairs in a program of considerable size would require a huge amount of space. Further, not all clients are interested in the alias relationships for all the pointer pairs. Therefore, an alternative representation for storing the alias information is proposed. In this representation, each pointer is associated with the set of pointees it is pointing to. The alias relationship between two pointers is computed by iterating through the two sets of pointees associated with the two pointers and checking if they have a common pointee.

Definition 2.5 (**Points-to Fact, Points-to Pair**). *A points-to fact or a points-to pair is a pointer-pointee relationship between two (not necessarily distinct) program variables or between a program variable and a memory location.*

Throughout, we denote a points-to fact using a right-arrow, e.g., `ptr` $\rightarrow$ {`var`}, representing that pointer `ptr` points to the pointee `var`. Depending upon the analysis, a points-to fact is often specialized with additional control-flow information. For instance, in case of a flow-sensitive analysis (defined in the next section), a points-to fact is defined *at a program point*. The set of all points-to facts for a pointer constitutes its *points-to set* or *dereference set*. In this thesis, we represent a points-to set using a right-arrow to a pair of curly-braces, e.g., `ptr` $\rightarrow$ {`var1, var2`}. When the points-to set is a singleton, we often omit the braces. Thus, the points-to fact `ptr` $\rightarrow$ {`var`} is represented as `ptr` $\rightarrow$ `var`. The set of points-to sets for all the program pointers constitutes the program's *points-to information*.

Consider a program with the following statements: `ptr1 = &var; ptr2 = ptr1`. The points-to facts derived from this program are `ptr1` $\rightarrow$ `var`, `ptr2` $\rightarrow$ `var`. Note that aliasing relationship between `ptr1` and `ptr2` can be computed by checking for a common pointee (in this case, `var`) in the points-to sets of the two pointers.

Definition 2.6 (**Points-to Analysis**). *Points-to analysis is a mechanism of statically computing the points-to sets of pointers in a program.*

Typically, there are three aspects to measuring the performance of a points-to analysis:

analysis time, memory requirement and analysis precision. Analysis time is measured in terms of the analysis time complexity (e.g., $\mathtt{O(n^3)}$) and in terms of the absolute amount of wall-clock time taken by the analysis to compute the points-to information (e.g., 3.5 second). Memory requirement of an analysis is the total amount of memory in bytes used to store the initial points-to constraints, the final points-to information and the intermediate data structures used to compute the points-to information (e.g., 36 MB). We explain the meaning of analysis precision and how to measure it below.

Precision. The precision of a pointer analysis is a metric to compute the amount of conservativeness in the analysis. Similar to any compiler analysis, as a points-to analysis needs to preserve soundness (safety), it tends to be conservative. Thus, a static points-to analysis is conservative compared to its dynamic counterpart which analyzes a program while the program is executing. Further, different static pointer analysis methods may produce different points-to information. Therefore, in order to compare different points-to analyses, apart from their analysis times and memory requirements, their relative precisions are also computed. An analysis $\mathcal{A}_1$ is strictly more conservative than $\mathcal{A}_2$ if points-to information for $\mathcal{A}_1 \supset$ points-to information for $\mathcal{A}_2$. For instance, if the points-to information computed by $\mathcal{A}_1$ is `ptr1` $\rightarrow$ {`var1, var2`}; `ptr2` $\rightarrow$ `var1` and that by $\mathcal{A}_2$ is `ptr1` $\rightarrow$ `var1`; `ptr2` $\rightarrow$ `var1`, then $\mathcal{A}_1$ is less precise than $\mathcal{A}_2$.

It should be noted, however, that such an ordering may not exist between all the pairs of the points-to analyses, as the points-to information computed by one analysis may not be a superset of that of another. Although there have been a few attempts to define precision in the context of pointer analysis, unfortunately, there is no single definition used in literature. One customary way to represent precision is by defining an average dereference set size (DSS) of a pointer. It is defined as below (e.g., see [24]).

$$\text{average DSS} = \frac{\sum_{\text{p}} |\text{points-to set(p)}|}{\#\text{p}}$$

where |points-to set(p)| refers to the number of pointees `p` may be pointing to and #p is the number of pointers.

Thus, average DSS is the ratio of sum of the number of pointees of all the pointers to the number of pointers. Lower the average DSS, better is the analysis precision. For instance, if the points-to information computed by $\mathcal{A}_1$ is `ptr1` $\rightarrow$ {`var1, var2`}, `ptr2` $\rightarrow$ `var1` and that

by $\mathcal{A}_2$ is `ptr1` $\rightarrow$ {`var3`}, `ptr2` $\rightarrow$ `var1`, then the two points-to sets are non-comparable. In this case we could compute their average DSS to compare their precisions.

Average DSS($\mathcal{A}_1$) = (|{`var1, var2`}| + |{`var1`}|) / 2 = (2 + 1) / 2 = 1.5.

Average DSS($\mathcal{A}_2$) = (|{`var3`}| + |{`var1`}|) / 2 = (1 + 1) / 2 = 1.

Since average DSS($\mathcal{A}_1$) > average DSS($\mathcal{A}_2$), we say that $\mathcal{A}_1$ is less precise than $\mathcal{A}_2$.

Unfortunately, average DSS for an analysis may differ for the same input program depending upon the intermediate code representation (IR). Due to various IR formats used by researchers, it becomes difficult to compare precision across various pointer analyses [57].

Definition 2.7 (**Realizable Points-to Fact**). *A points-to fact is realizable if it occurs in at least one execution of the program.*

For instance, a points-to fact `ptr` $\rightarrow$ `var` is realizable if `ptr` points to `var` in some execution of the program.

Definition 2.8 (**Soundness**). *A (static) points-to analysis is sound if every realizable points-to fact is computed by the analysis.*

Thus, a points-to analysis is sound if it computes a superset of the realizable points-to facts. Note that the trivial solution where every pointer points-to all the variables is a sound approximation to the actual (realizable) points-to set, but is also imprecise.

Definition 2.9 (**Client**). *A client to points-to analysis is a program analysis, a program transformation or an application that queries the points-to or alias information of the points-to analysis.*

Examples of clients are various dataflow optimizations like copy propagation, common subexpression elimination and analyses like live variable analysis, program slicing, Mod/Ref analysis [57]. Interestingly, a client to points-to analysis $\mathcal{A}_1$ could be another points-to analysis $\mathcal{A}_2$ which could make use of the information from $\mathcal{A}_1$ to improve overall precision (e.g., see [66]).

Definition 2.10 (**Scalar, Non-Scalar**). *A scalar variable is a variable of a basic type. A non-scalar variable is a variable of an aggregate type.*

Examples of scalars are `int, int *, char`, while example of non-scalars are arrays and structures.

Dynamic memory allocation is a language support that allows for allocation of memory while a program is running. In C, dynamic memory allocation is supported using functions `malloc, calloc`, etc. and a pointer can point to a variable present either on the stack or in the heap, apart from global variables. However, in Java, a variable allocated on the stack cannot be address-taken. Therefore, the only pointees possible in Java are the heap allocated variables [130]. In the absence of dynamic memory allocation, a compiler can statically count the total memory requirement of a program. In case of dynamic memory allocation, the amount of total memory cannot, in general, be determined statically.

2.2 Analysis Dimensions

The precision of a program analysis depends upon how it models an input program's data and control-flow. The following is a list of key aspects using which this modeling is done; they are called as analysis dimensions [113]. Although these dimensions are applicable to other program analyses and transformations, we explain these in the context of pointer analysis.

2.2.1 Flow-sensitivity

An analysis is flow-sensitive if it takes into account the control-flow in the program while computing its solution [57]. A **control-flow** defines the ordering of various statements in a program. A flow-sensitive analysis computes dataflow information at the boundaries of the basic-blocks. A flow-insensitive analysis ignores the control-flow specified in the program and assumes that any basic-block can be reached from any other basic block (including self). Thus, a flow-insensitive analysis assumes that the control-flow graph (CFG) is *complete*. One may expect that a complete CFG would be more expensive to process than an incomplete CFG since the former has more edges and, in turn, requires more information to be propagated. This may lead to a conclusion that flow-insensitive analysis is more expensive than a flow-sensitive analysis. However, a complete CFG can be represented by a single basic-block with a self-loop containing all the statements from all the basic-blocks in the original CFG. This simple representation captures all the control-flows present in the original complete CFG and enables an efficient information processing. Thus, a flow-insensitive analysis processes all the program statements repeatedly in a sequential order. This makes a flow-insensitive analysis

more efficient than its flow-sensitive counterpart.

Example 2.1. Consider the following program fragment.

```
S1:  a = &x;
S2:  a = &y;
```

The labels `S1` and `S2` denote program points. A flow-sensitive points-to analysis of the above example program would yield the points-to information: `a` → `x` at `S1`, `a` → `y` at `S2`. On the other hand, a flow-insensitive points-to analysis would compute the same points-to information at all the program points ignoring the control-flow: `a` → {`x, y`}.

2.2.2 Field-sensitivity

Field-sensitivity deals with how *aggregates* are modeled by an analysis. An aggregate is a container for multiple fields. Examples of aggregates in C are `struct` variables and `union` variables.

A *field-sensitive* analysis treats each field of each aggregate instance separately [98]. *A field-insensitive* analysis assumes that an access to a field is that to its enclosing aggregate and, in turn, does not model fields at all. A *field-based* analysis has a limited field-sensitivity: it models each field of an aggregate type separately, and does not distinguish between different field-instances for different aggregate variables of the same type [98].

Example 2.2. Consider the following program fragment where variables `a` and `b` are the aggregates of the same type `T`.

```
a.f1 = &x;
b.f1 = &y;
a.f2 = &z;
```

A field-sensitive analysis computes: `a.f1` → `x`, `b.f1` → `y`, `a.f2` → `z`.

A field-insensitive analysis computes: `a` → {`x, z`}, `b` → `y`.

A field-based analysis computes: `T.f1` → {`x, y`}, `T.f2` → `z`.

A naive implementation of field-based analysis, under certain circumstances, may produce unsound points-to information [97].

2.2.3 Context-sensitivity

In order to define context-sensitivity, we need to first define an intra-procedural analysis and an inter-procedural analysis. An analysis is **intra-procedural** if it processes each function in isolation making conservative assumptions about the external environment (callers and callees).

Example 2.3. Consider the following program fragment.

```
main() {
  S1:  ptr2 = &z;
  S2:  fun(&x);
  S3:  fun(&y);
}
fun(int *ptr1) {
}
```

An intra-procedural analysis of the above example processes functions `main` and `fun` in isolation. In other words, it computes the points-to solution for the statements in each function by conservatively approximating the effect of external functions. Thus, for each pointer that may receive points-to information from outside a function through function arguments, it assumes that the points-to information contains all the address-taken variables in the program (namely, `x, y` and `z`). Therefore, an intra-procedural analysis computes the following points-to information for pointer `ptr1`: `ptr1` $\rightarrow$ {`x, y, z`}.

An intra-procedural analysis, as evident from the description above, is imprecise. However, note that all the pointers that *escape* a function, i.e., those pointers that are accessed outside a function, have the same points-to information. Therefore, such pointers can be merged, reducing the total number of variables tracked. Therefore, an intra-procedural analysis usually scales well with the program size.

In contrast, an **inter-procedural** analysis allows a more precise flow of information from one function to another. Thus, a function is not processed totally in isolation, but has information about all of its callers and callees. A customary way of representing the caller-callee relationship is using a **call-graph (CG)**. Each node in a CG is a function and a call from function `foo` to function `bar` is denoted by a directed edge from `foo` to `bar`. A cycle in a CG denotes recursion.

An inter-procedural analysis restricts the points-to information coming into a callee to be only from its callers. Thus, for the above example, an inter-procedural analysis would deduce the points-to information for `ptr1` as `ptr1` $\rightarrow$ {`x, y`}. Note that the analysis merged the points-to information coming from the two calls to `fun` and did not include other address-taken variable (namely, `z`) in the points-to set of `ptr1`.

Compared to an intra-procedural analysis, an inter-procedural analysis keeps track of the points-to sets separately for each escaping pointer. Therefore, its storage and processing requirements are more than the intra-procedural analysis.

A context-sensitive analysis is an inter-procedural analysis which distinguishes between various calls to the same function. In other words, a context-sensitive analysis keeps track of the calling context of a function `foo` while analyzing `foo`. A calling context of a function `foo` is the calling context of its caller appended with the caller's callsite that calls `foo`. More concretely, a calling context is a sequence of functions whose return address would be on stack when a function is executing at run-time. The calling context of `main` is empty. Typically, a complete context starting from `main` is unnecessary and analyses restrict the context length to a small value [25, 68]. Note that the analysis of a recursive program, which essentially has infinitely long calling context, can also be approximated by restricting its length to a fixed value.

For the above example program, a context-sensitive analysis would distinguish between the two calls to function `fun` from function `main` and compute a different points-to information for pointer `ptr1` across each context: `ptr1` $\rightarrow$ `x` along `S2`, `ptr1` $\rightarrow$ `y` along `S3`.

Even if the calling context is restricted to a fixed value `k`, the number of distinct contexts in a program varies exponentially with `k` and it may result in not only a large storage requirement but also a large analysis time. Therefore, context-sensitive points-to analysis has been at the forefront of research since several years [70, 31, 132, 18, 10, 13, 17, 129, 75, 66]. In this thesis, we focus on efficient context-sensitive points-to analysis for large programs.

We deal with context-sensitive, flow-insensitive and field-insensitive points-to analysis in this thesis.

2.3 Computability and Complexity

In this section we mention results on the algorithmic complexity of various forms of points-to analysis. The results not only provide a motivation for the significant amount of research done in this area, but also helps a reader put various analyses in perspective.

A problem is **solvable in polynomial-time** if there exists an algorithm whose running time is at most a polynomial expression in the input-size. Mathematically, the running time $\mathtt{T(n)} = \mathtt{O(n^k)}$ for some constant `k`. A problem is a **decision problem** if it has a *yes*/*no* answer. A decision problem is in the complexity class **NP** if it can be solved in polynomial time by a non-deterministic turing machine. A problem is **NP-Hard** if it is at least as hard as the hardest problems in NP. In other words, a problem is NP-Hard if any problem in NP can be reduced to it in polynomial-time. A decision problem is **NP-Complete** if it is both in NP and NP-Hard. A decision problem is in the complexity class **PSPACE** if it can be solved by a turing machine using a polynomial amount of space. A decision problem is **PSPACE-Complete** if it is in class PSPACE and if every problem in PSPACE can be polynomially reduced to it.

We now state the important results for various kinds of points-to analyses. The following theorem states that in the presence of dynamic memory allocation a precise flow-sensitive points-to analysis is undecidable. For a flow-insensitive analysis, the problem is still open.

Theorem 2.11 ([70, 103, 12])**.** *Single-procedural flow-sensitive points-to analysis is undecidable with dynamic memory allocation.*

The above theorem holds even when there are only scalar variables in the program.

When dynamic memory allocation is not allowed, the configuration space of pointers becomes finite and the problem becomes decidable. However, it is still not computable in polynomial time.

Theorem 2.12 ([71, 93, 12])**.** *When dynamic memory allocation is disallowed, single-procedural flow-sensitive points-to analysis is PSPACE-Complete.*

The above theorem holds even when there are only scalar variables with well-defined types in the program and only two levels of dereferencing are allowed (e.g., `int`, `int *` and `int **`, but not `int ***`).

For the simpler variant when a program contains only a single level of dereferencing, the problem is solvable in polynomial time [71].

Theorem 2.13 ([71]). *When dynamic memory allocation is disallowed, single-procedural flow-sensitive points-to analysis is solvable in polynomial-time when all variables are scalars with well-defined types and only a single level of dereferencing is allowed.*

Landi [71] gives a polynomial time algorithm for the case above.

We now state two important theorems for flow-insensitive analyses.

Theorem 2.14 ([59]). *When dynamic memory allocation is disallowed, flow-insensitive points-to analysis is NP-Hard for arbitrary number of dereferences.*

Theorem 2.15 ([12]). *When dynamic memory allocation is disallowed, flow-insensitive points-to analysis is solvable in polynomial-time for arbitrary number of dereferences when the variables are scalars with well-defined types.*

Thus, by restricting the flow-insensitive points-to analysis to strongly-typed languages without aggregates, it becomes theoretically easier to compute the points-to information. Chakaravarthy [12] gives a polynomial-time algorithm for the above case. This also suggests that a flow-sensitive analysis is more expensive to compute as compared to a flow-insensitive analysis.

These complexity results suggest that in a general setting where a program can have weak-typing, multiple functions, aggregates and dynamic memory allocation, one requires an approximate analysis to compute an approximation to the precise points-to solution. Thus, such an approximate analysis would conservatively compute more points-to information compared to a precise analysis.

2.4 Two Key Points-to Analysis Methods

In this section we present two important pointer analysis algorithms which compute approximate points-to information. One of them, well-known as Andersen's analysis [3], is based on the notion of set-inclusion. It gives a relatively higher precision but has a higher running time. The other algorithm, well-known as Steensgaard's analysis [123], is based on the notion of unification. It runs in almost linear time but has relatively lower precision. Several points-to analysis algorithms in literature are variants of one of these two algorithms. Therefore, these algorithms merit an explicit introduction. Both the algorithms are flow-insensitive.

We first explain the representation of the input program that these analyses work on which reduces the complexity of implementation of these algorithms and also helps us in explaining these better.

2.4.1 Normalized Input Format

Since the two algorithms are flow-insensitive, the input is simply a set of statements that can modify the points-to information. Each such statement is normalized into one of the formats (by creating a temporary variable, if required) as shown in the table below. Thus, there are four kinds of statements: address-of, copy, load and store. Load and store statements are collectively called *complex* statements.

address-of	`p` = &`q`	points-to-set(`p`) $\supseteq$ {`q`}
copy	`p = q`	points-to-set(`p`) $\supseteq$ points-to-set(`q`)
load	`p = *q`	$\forall$`r` $\in$ points-to-set(`q`), points-to-set(`p`) $\supseteq$ points-to-set(`r`)
store	`*p = q`	$\forall$`r` $\in$ points-to-set(`p`), points-to-set(`r`) $\supseteq$ points-to-set(`q`)

We assume that a heap allocation instruction is converted to an intermediate form similar to the address-of statement using a temporary variable.

In the forthcoming chapters, we use the term *restricted number of pointer depth* in various forms. Since its meaning differs depending upon the context, for better understanding, we clarify it apriori here.

- For a flow-insensitive analysis, it is the number of indirections in a single statement – which we assume to be one. We transform a statement with multiple indirections into multiple statements, by generating temporaries, as discussed at the start of this section.

- In Chapter 4, the restriction is on the types we model precisely. Thus, we model `int *p` and `int **p` precisely and model `int ***p` and higher level pointer types imprecisely, by assuming that they may point to any address-taken variable. This is briefly mentioned in Section 4.3.2.

- In Chapter 6, the restriction is on the number of times an indirection can be applied to a value. Our prime-number lattice works for a fixed number of indirections. Although,

this is fixed for an analysis run, it can be set to a high value by a user. This is mentioned in Section 6.3.

In the next two subsections we present the two algorithms.

2.4.2 Andersen's Inclusion-based Analysis

Andersen's inclusion-based analysis (also called *constraint-based* analysis or *subset-based* analysis) [3] works by converting each of the four kinds of statements into a set-constraint as shown in Column 3 of the table above. Thus, a statement of the form `LHS = RHS` is handled by adding a set-constraint `LHS` $\supseteq$ `RHS`, i.e., the points-to set of LHS is a superset of that of RHS. By obtaining such a set of `n` constraints, its solution can then be obtained using a constraint solver in $O(n^3)$ time [98]. The constraint solver iteratively processes the set of constraints and updates the points-to sets per Column 3 of the above table until a fixed-point. The fixed-point solution consists of the points-to information for each pointer.

Example 2.4. Consider the following program.

```
a = &x; b = &y; p = &a; b = *p;
```

The points-to information computed for the above example using Andersen's analysis is

$$\texttt{a} \rightarrow \{\texttt{x}\},\ \texttt{b} \rightarrow \{\texttt{x,y}\},\ \texttt{p} \rightarrow \{\texttt{a}\}.$$

2.4.3 Steensgaard's Unification-based Analysis

Steensgaard's unification-based analysis [123] employs the notion of bidirectional similarity to merge the points-to sets of `LHS` and `RHS` for a statement `LHS = RHS`. An invariant of unification-based analysis is that every pointer has a single points-to edge. Thus, if a pointer points-to multiple pointees, all those pointees are merged into the same set. Note that Andersen's analysis maintains multiple points-to edges per pointer and, therefore, it is possible in case of Andersen's analysis that two pointers may have a common pointee but different points-to sets, which is not possible in case of Steensgaard's analysis.

An alternative way of looking at unification is from the perspective of the aliasing relation. Recall from Definition 2.3 that aliasing is a reflexive and symmetric relation. However, it is

not transitive, i.e., if pointers `p` and `q` are aliases and pointers `q` and `r` are aliases, then `p` and `r` need not always alias with each other. For instance, if `p` $\rightarrow$ `{a}, q` $\rightarrow$ `{a,b}, r` $\rightarrow$ `{b}`, then `p` aliases with `q` and `q` aliases with `r`, but `p` does not alias with `r`. This property is maintained by Andersen's analysis (using the multiple points-to edges per pointer). However, due to a single points-to edge, Steensgaard's analysis conservatively adds transitivity to the aliases represented. Since the relation represented is reflexive, symmetric and transitive, it becomes an equivalence relation forming partitions of alias-sets. Thus, each pointer in an alias-set aliases with all the pointers in the set and does not alias with any pointer outside the alias-set. Due to this key property, alias-sets can be very efficiently implemented using a union-find data structure [21]. In fact, Steensgaard's analysis runs in time $O(m\alpha(m))$ where `m` is the number of constraints and α`(m)` is the inverse Ackermann's function [21], which, for all practical purposes, is a small constant (below 5). Thus, Steensgaard's analysis scales almost linearly with the program size. However, due to transitive aliasing, it has significantly less precision compared to Andersen's analysis.

Example 2.5. For the example program in the previous subsection, the points-to solution obtained using unification is

$$\texttt{a} \rightarrow \{\texttt{x,y}\},\ \texttt{b} \rightarrow \{\texttt{x,y}\},\ \texttt{p} \rightarrow \{\texttt{a}\}.$$

Note that for the example program, in comparison to Andersen's analysis, Steensgaard's analysis has an additional points-to fact `a` $\rightarrow$ `y`. In general, unification computes the points-to information $\mathcal{U}$ which is a superset of the points-to information $\mathcal{I}$ computed using inclusion, i.e., $\mathcal{U} \supseteq \mathcal{I}$.

2.5 Inclusion-based Points-to Analysis as a Graph Problem

As originally proposed by Andersen [3], an inclusion-based analysis iteratively computes the points-to solution by going over the points-to constraints. However, iteratively going through all the points-to constraints is not the most efficient way to compute the fixed-point of an inclusion-based analysis. Fahndrich et al. [34] proposed solving a set of points-to constraints using the notion of a constraint graph. The constraint graph gives a graphical representation

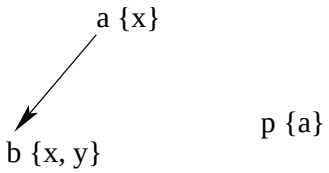


Figure 2.1: Constraint graph for Example 2.4

to the points-to analysis problem and enables exploiting its structural properties to optimize the analysis.

A **constraint graph** `G` is a directed graph where each node represents a pointer and each directed edge `u` $\rightarrow$ `v` represents the inclusion (or subset) relationship points-to set(`u`) $\subseteq$ points-to set(`v`). Each node also maintains its points-to set which gets updated as points-to information flows along its incoming edges and which gets propagated along the outgoing edges to other nodes in `G`. The graph is not static and new edges get added to it as complex constraints (i.e., load and store constraints) are evaluated. Formation of new edges also enables more points-to information propagation in `G`. When no more edges can be added and no more points-to information can be propagated, a fixed-point of the points-to information is computed at the nodes in `G`. The (final) constraint graph for Example 2.4 is shown in Figure 2.1. It has three nodes corresponding to each pointer `a, b, p`. Each node is associated with its points-to information, which is shown in curly-braces in the figure. A directed edge from `a` to `b` indicates the flow of points-to information from `a` to `b`. The edge is formed due to the processing of the load constraint `b = *p`.

Points-to analysis using a constraint graph is presented in Algorithm 1. Various nodes in the constraint graph `G` are first initialized with the points-to information computed using address-of constraints (Line 1). Initial set of subset edges are added to `G` using copy constraints (Line 2). The points-to information at the nodes is then propagated along the edges until saturation (Line 4). Next, the complex constraints are evaluated to add more edges in `G` (Line 5). The points-to information propagation and addition of new edges is then repeatedly done until a fixed-point is reached (`repeat-until` loop at Line 3).

Note that a naive analysis using a constraint graph does not give any additional benefits over the iterative Andersen's analysis. To make the analysis using constraint graph efficient, the structure of the constraint graph needs to be exploited. This is done using two main techniques: by dynamically collapsing cycles in `G` and by propagating points-to information in

Algorithm 1 Points-to Analysis using Constraint Graph.

Require: set `C` of points-to constraints
1: Process *address-of* constraints
2: Add edges to constraint graph `G` using *copy* constraints
3: **repeat**
4: Propagate points-to information in `G`
5: Add edges to `G` using *load* and *store* constraints
6: **until** fixed point

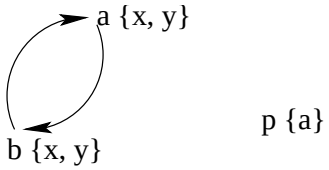


Figure 2.2: Constraint graph for Example 2.6

topological order [34].

It is established that online cycle detection and elimination is a key requirement for scaling points-to analysis. An interesting aspect of a cycle or a strongly connected component in a constraint graph is that all the pointers in the cycle eventually have the same points-to sets. This property is exploited to collapse the cycles and reduce the number of variables tracked during the analysis.

Example 2.6. Consider the following set of points-to constraints obtained by adding the constraint `a = b` to the constraints in Example 2.4.

```
a = &x; b = &y; p = &a; b = *p; a = b;
```

The final constraint graph generated for the above set of constraints is shown in Figure 2.2. Note that there is a cycle formed by the nodes `a` and `b`. The points-to information computed for the above example using Andersen's analysis is

$$\texttt{a} \rightarrow \{\texttt{x,y}\},\ \texttt{b} \rightarrow \{\texttt{x,y}\},\ \texttt{p} \rightarrow \{\texttt{a}\}.$$

We see that both `a` and `b` have the same points-to information. In general, all the pointers in a strongly connected component eventually have the same points-to information. Thus, we can represent all the pointers in a cycle using a single representative pointer to reduce the number of variables tracked during the analysis and to reduce the amount of points-to propagation in the constraint graph.

Note that cycle detection and elimination is performed dynamically (online) since edges are added to `G` dynamically. An important configuration criteria is how often to check for cycles. Evaluation of techniques which detect cycles at various frequencies is studied by Hardekopf and Lin [49]. Collapsing cycles converts the constraint graph into a directed acyclic graph (DAG). This enables a topological ordering of the nodes enabling efficient points-to information propagation.

2.6 Chapter Summary

In this chapter we studied the background necessary to understand the rest of the thesis. We defined points-to analysis and learnt about its precision and soundness. We next discussed various analysis dimensions that affect the analysis precision. We then presented important complexity results on various forms of pointer analysis. Next, we described two approximation algorithms, namely, Andersen's inclusion-based analysis and Steensgaard's unification-based analysis, which give precision-time trade-off. Finally, we presented points-to analysis as a graph problem and discussed how to exploit the structure of a constraint graph to efficiently compute the points-to solution.

Chapter 3

A Survey of Pointer Analysis Methods

3.1 Introduction

We present a survey of pointer analysis methods in this chapter. Summarizing three decades of research work on pointer analysis in one chapter is a difficult task. Therefore, instead of trying to cover the work exhaustively, we attempt to discuss the most interesting and important works on pointer analysis.

We divide the survey in 12 categories. The categories are not completely mutually exclusive. However, we try to classify the related work into these categories based on the focus of the work.

1. **Surveys** (Section 3.2): In this category, we discuss the surveys on pointer analysis. Typically, a survey contains a broader perspective than individual algorithms and although it may not add a new innovation to the area, it helps in understanding the higher level picture of the area in the form of trends, pitfalls and lessons.

2. **Complexity Results** (Section 3.3): In this category, we discuss the work which focuses on computational complexity of pointer analysis algorithms. Such a work identifies the complexity classes for variants of the algorithm and proposes polynomial solutions to the possible variants.

3. **Use of Novel Data Structures** (Section 3.4): In this category, we discuss the work

which proposes the use of novel data structures to store points-to information which helps in scaling pointer analysis, e.g., binary decision diagrams. We also discuss the work that compares the same pointer analysis algorithm using different data structures.

4. **Optimizations and Techniques** (Section 3.5): In this category, we discuss the work which proposes techniques to speedup points-to analyses. These techniques are generic enough to be applicable to multiple points-to analysis algorithms.

5. **Exact Methods** (Section 3.6): In this category, we discuss various novel algorithms for pointer analysis which use an inclusion-based approach.

6. **Methods Achieving Explicit Trade-offs** (Section 3.7): In this category, we discuss the work which offers an explicit control over the trade-off between precision and scalability.

7. **Client-driven and Demand-driven Methods** (Section 3.8): Exhaustive algorithms compute all possible points-to facts from a given set of constraints. In this category, we discuss their variants, namely, client-driven analysis which prioritizes various processing elements (functions, constraints, etc.) to suit the needs of a client, and demand-driven analysis which cumulatively processes only the required program fragments to answer a specific query.

8. **Incremental and Probabilistic Methods** (Section 3.9): In this category, we discuss incremental analysis and probabilistic analysis. An incremental analysis allows for dynamic addition or removal of constraints from the original set of constraints and computes points-to information only for the affected set of constraints. A probabilistic analysis assigns probabilities to the computed points-to facts based on the likelihood of the fact being realized at run-time.

9. **Analysis of Parallel Programs and Parallel Analyses** (Section 3.10): In this category, we discuss the work that deals with (sequential) pointer analysis of multi-threaded programs and also the parallel versions of pointer analyses.

10. **Applications of Points-to Analysis** (Section 3.11): Pointer analysis has been extensively used for various other analyses and transformations. In this category, we briefly discuss some of the applications of pointer analysis, e.g., slicing and shape analysis.

11. **Evaluations and Quantifications** (Section 3.12): In this category, we focus on the work which are experimental in nature and deduce some interesting characteristics of benchmarks or of the underlying analysis, which help us understand the analysis behavior.

12. **Points-to Analysis for Other Languages** (Section 3.13): We discussed our work in the context of C programs. However, other languages pose different challenges and provide different opportunities to pointer analysis. For instance, Java is type-safe which enables us to design a polynomial time points-to analysis for the language, but it also poses challenges due to dynamic class loading and reflection. In this category, we discuss the work on pointer analysis that deals with different programming languages.

3.2 Surveys

In this section, we discuss various surveys on pointer analysis. The most cited survey is by Hind and Pioli [57]. It identifies several dimensions which affect the trade-off between precision and scalability, and along which the existing work can be categorized. Some of the dimensions mentioned are flow-sensitivity, context-sensitivity, heap modeling which deals with how allocation sites are modeled, etc. The survey also discusses various issues in the terminology, metrics of precision evaluation and reproducibility of results. Finally, it discusses several directions for future research (in the year 2000): scalability, improving precision without affecting scalability, client-driven analysis, extending pointer analysis along the dimensions of path-sensitivity and context-sensitivity, modeling heap to improve precision without adversely affecting scalability, better modeling of aggregates, demand-driven and incremental analyses, handling features of object oriented languages, analyzing incomplete programs, providing annotations for improving precision, and pursuing and uncovering engineering insights for a scalable analysis. While mentioning the issues and the open problems, the survey also provides anecdotes from experts in the field.

Ryder [113] classifies various approaches to reference analysis for object oriented languages based on analysis dimensions. Some of the dimensions are also applicable to procedural languages. The dimensions are flow-sensitivity, context-sensitivity, program representation, object representation, field-sensitivity, reference representation and directionality of information flow

in a constraint. For each of these dimensions, she discusses the related work. She also discusses various open issues in the context of object oriented languages, namely, reflection, native methods, exceptions, dynamic class loading, and incomplete programs.

A few instances of informal classification of pointer-analysis algorithms exist. One of the most informative surveys on pointer analysis in the recent past is by Lhotak [76]. The survey deals with two aspects: various abstractions in the analysis which affect the precision and efficiency and algorithmic aspects which affect the analysis efficiency. The former discusses various abstractions like type filtering, and field-sensitivity, flow-sensitivity, context-sensitivity. The latter deals with points-to information propagation and implementation of sets to represent the points-to constraints and the points-to information.

Raman [104] surveys three methods: unification-based analysis (using a type system) [123], pointer analysis using BDDs [6] and an application of pointer analysis in bug detection [80]. He discusses various precision-scalability trade-offs involved in choosing an appropriate method for analyzing a program with pointers.

Wu [134] presents a survey of alias analysis. He discusses various kinds of flow graphs used in literature, namely, inter-procedural control flow graph [70, 74], invocation graph [32, 132], procedural call graph [18, 10] and context-sensitive procedural call graph [13, 17]. He also discusses about abstract data representations used to model variables inaccessible to a procedure [70, 32], global variables [132] and access paths [70, 17]. He further discusses various approaches used to model recursive data structures, namely, 1-level [31, 132], `k`-limiting [70, 17], beyond `k`-limiting using symbolic access paths [27], and using shape analysis [41]. Wu also categorizes various points-to analysis algorithms based on flow-sensitivity and context-sensitivity and also based on their alias representation, flow graph, target language, benchmarks used and modularity. In a later part of the survey, Wu discusses related work on assembly level alias analysis.

Rayside [105] provides a five-page summary of various important aspects of pointer analysis. Derived from Whaley's talk slides [127], Rayside classifies several analyses based on flow-sensitivity and context-sensitivity. It also mentions incremental analysis, handling incomplete programs and demand-driven analysis as new research challenges.

3.3 Complexity Results

In this section, we discuss various results on the computability of pointer analysis. Landi [70] and Ramalingam [103] proved that in the presence of non-scalar variables and dynamic memory allocation, flow-sensitive alias analysis is undecidable. Chakaravarthy [12] proved the undecidability of flow-sensitive aliasing even when a program contains only scalar variables when dynamic memory allocation is allowed. Landi [71] and Muth and Debray [93] proved that flow-sensitive points-to analysis is PSPACE-Complete even when all the variables are scalars with well defined types and only two levels of dereferencing (only `p, *p` and `**p`) are allowed. For the case when only a single level of dereferencing is allowed, Landi [71] gave a polynomial-time algorithm.

We now discuss some results related to flow-insensitive points-to analysis. Horwitz [59] proved that in the absence of dynamic memory allocation, flow-insensitive points-to analysis is NP-Hard when all the variables are scalars and arbitrary number of dereferencing is allowed. When the problem is restricted to well-defined types, the flow-insensitive analysis can be solved in polynomial time, even when arbitrary number of dereferencing is allowed [12]. This positive result is important for type-safe languages like Java and proposes a non-trivial points-to analysis variant which is solvable in polynomial time. It also theoretically proves that flow-insensitive analysis is easier than its flow-sensitive counterpart, since the flow-sensitive version is PSPACE-Complete.

In the context of object-oriented languages, Chatterjee et al. [14] discuss the complexity of points-to analysis in the presence of exceptions. They present a polynomial-time algorithm for points-to analysis in the presence of exceptions for program without threads. They also prove that an interprocedural points-to analysis with single-level types and exceptions with sub-typing, but without dynamic dispatch, is PSPACE-Hard. They also prove that the intraprocedural version of the above problem is PSPACE-Complete. They also improve the worst-case time complexity of points-to analysis in the absence of exceptions from $O(n^7)$ [72, 106] to $O(n^5)$.

Several open problems related to flow-insensitive analysis exist. It is unknown whether flow-insensitive points-to analysis in the absence of dynamic memory allocation with arbitrary number of dereferencing is in NP. Also, it is unknown whether, for a bounded number of dereferences, the problem can be solved in polynomial time. Further, it is not known whether the problem remains decidable when dynamic memory allocation is allowed.

3.4 Use of Novel Data Structures

In this section, we discuss the work that proposed innovative data structures for representing the points-to (or alias) information. Earlier approaches stored alias pairs explicitly [70]. Since this representation is storage-intensive, compact representation is proposed which stores only a few basic alias pairs explicitly and new alias pairs are derived based on dereference, transitivity and commutativity [18]. Later, a more crisp representation in the form of points-to pairs has been devised [31] which significantly reduces the storage requirement.

Heintze and Tardieu [54] propose the use of sparse bitmaps for storing points-to information. Since for most programs, the points-to information is actually sparse — i.e., only a few pointers have a large number of pointees while most others have very few pointees — and accessing a sparse bitmap is very efficient, sparse bitmap is a desirable data structure for storing points-to information. It is used in GCC 4.1 [49]. However, bitmaps cannot take advantage of the commonality across various points-to sets. Therefore, for a context-sensitive analysis, the use of bitmaps requires a large amount of memory.

Zhu [141] was the first one to observe that the vast amount of points-to information can be encoded in a space-efficient manner using binary decision diagrams (BDD) [8]. Until then, BDDs were used in symbolic model checking [9] and to represent large sets and maps [84]. Due to the storage efficiency, BDDs were quickly adapted for solving points-to analysis algorithms. Berndl et al. [6], Whaley and Lam [129] and Zhu and Calman [142] propose variants of points-to analysis algorithms using BDDs for Java.

Hardekopf and Lin [49] compared the performance of BDD-based and bitmap-based points-to analyses. They found that a BDD-based implementation is, on an average, 2× slower than a sparse bitmap-based implementation, but uses 5.5× less memory.

We propose to store points-to information in a bloom filter (Chapter 4). Bloom filters offer the best-of-both-the-worlds: its access time is as low as that for bitmaps and its memory requirement is even below that of BDDs. Although it incurs a minimal amount of precision loss, a bloom filter based analysis is likely to scale well with program size both in terms of memory and analysis time, as demonstrated in Chapter 4.

3.5 Optimizations and Techniques

In this section, we discuss the work that exploits properties of the problem to improve the efficiency of points-to analysis. Since these techniques exploit the problem structure, they are typically applicable to a wide variety of points-to analysis algorithms. While several techniques and engineering artifacts are proposed in almost every algorithm, we restrict this discussion to broader techniques which stand out on their own.

One of the most important optimizations in scaling points-to analysis is online cycle elimination [34]. This optimization is an outcome of the formulation of points-to analysis as a graph problem. The points-to constraints are represented using a constraint graph. Since the edges are dynamically added to the graph, cycles may get introduced during the analysis. Due to the abundance of large cycles, a fixed-point computation of points-to information along the cycles requires considerable number of propagations. The online cycle elimination technique exploits the fact that all the pointers involved in the cycle have the same points-to information at the fixed-point. This allows the technique to collapse the cyclic component into a single representative node which enables tracking fewer pointers than before.

Choi and Choe [19] propose cycle elimination for invocation-graph based context-sensitive points-to analysis. Their method first models sets of contexts as annotations and eliminates cycles only when the annotations appear in all the contexts.

Since cycles are formed dynamically, the cycle detection needs to be performed repeatedly. It is important to choose a good frequency of cycle detection, since checking for cycles too often can be costly and may outweigh its benefits; whereas checking for cycles very infrequently may reduce the benefits obtained using cycle detection. Hardekopf and Lin [49] propose Lazy Cycle Detection which checks for cycles when there is a good chance of finding one. This is done based on the heuristic that when an edge gets formed between two nodes of a constraint graph and the two nodes have the same points-to information, the two nodes may be in a strongly connected component. Lazy Cycle Detection significantly reduces the overhead of online cycle detection [49].

Hardekopf and Lin [49] also propose a Hybrid Cycle Detection which combines an offline analysis with an online cycle detection to improve the running time of the online analysis. The authors apply Hybrid Cycle Detection to three state-of-the-art solvers and illustrate significant benefits in the analysis performance.

To optimize the set of points-to constraints even prior to running the analysis, Rountev and Chandra propose offline variable substitution [108]. The technique identifies pointer-equivalent variables (i.e., pointers with the same points-to sets at the fixed-point) from the points-to constraints by building a subset graph without running the analysis. It has been established that offline variable substitution can reduce the number of constraints by a large amount. For instance, Hardekopf and Lin found that the technique reduced the number of constraints by 60 – 77% [49].

Hardekopf and Lin [50] improve upon offline variable substitution to find more pointer-equivalent variables. They also propose location equivalence to reduce the number of variables tracked during an analysis. Due to these offline optimizations, the authors found that sparse bitmaps actually require lesser memory than BDDs [50].

All the above optimizations reduce analysis time without affecting precision. Approximate pointer analysis [94] identifies pointers with *mostly similar* points-to sets as pointer equivalent and identifies pointees with *mostly similar* pointed-by sets as location equivalent. By suitably altering the similarity threshold, a client can obtain varying levels of analysis precision.

Several optimizations address efficient propagation of points-to information in the constraint graph. Pearce et al. [99] propose difference propagation to propagate only the difference in the points-to information across nodes. Wave and Deep Propagation [100] propagate points-to information in breadth-first and depth-first manner respectively for efficient analysis. Kanamori and Weise [67] propose several heuristics for choosing a node-ordering for points-to information propagation, e.g., Greatest Input Rise, Greatest Output Rise and Least Recently Fired. Pearce et al. [99] find that the Least Recently Fired strategy works very well in practice over other heuristics especially for large programs. Our work on prioritizing constraint evaluation (Chapter 7) is related, but deals with constraint evaluation ordering rather than points-to information propagation. It can be easily combined with any propagation related optimization for enjoying joint benefits.

Kahlon [66] proposes bootstrapping an analysis with the result of a prior analysis to improve scalability. He illustrates his technique by running a fast, imprecise analysis like Steensgaard's [123] to find disjoint alias sets and then running a slow but more precise analysis like Andersen's [3] on each of these alias sets to regain precision. The technique has been shown to scale well for moderately sized programs [66].

3.6 Exact Methods

In this section, we discuss the work which uses an inclusion-based approach for its analysis.

Inclusion-based points-to analysis is introduced by Andersen [3]. It is a flow-insensitive and context-insensitive points-to analysis that approximates the realizable points-to information based on inclusion of points-to sets. Approximations are inevitable since precise points-to analysis is NP-Hard [12]. For a pointer assignment $\mathtt{p}_{\mathtt{expr}} = \mathtt{q}_{\mathtt{expr}}$, it adds the points-to information of $\mathtt{q}_{\mathtt{expr}}$ into that of $\mathtt{p}_{\mathtt{expr}}$, achieving $\mathtt{pointsto}(\mathtt{p}_{\mathtt{expr}}) \supseteq \mathtt{pointsto}(\mathtt{q}_{\mathtt{expr}})$, which is essentially a set-inclusion constraint. For various pointer manipulating statements, Andersen's analysis adds such set-inclusion constraints and finally solves those constraints to achieve a fixed-point which represents a sound approximation to the points-to information.

Hind et al. [55] propose an approximation algorithm for interprocedural alias analysis. They also propose a technique for function pointer analysis that constructs a program call graph during alias analysis. They find that a flow-insensitive analysis with *kill* information does not improve precision over a flow-insensitive analysis without the *kill* information.

Liang and Harrold [78] find that Andersen's analysis [3] and Landi and Ryder's analysis [73] may not scale to large programs. Therefore, they propose a flow-insensitive, context-sensitive points-to analysis. Salient features of their analysis are that, similar to Steensgaard's analysis [123], it processes each pointer-related statement only once, computes a separate points-to graph for each procedure, and is modular. They demonstrate that their algorithm is almost as precise as Andersen's analysis, and its running time is within six times that of Steensgaard's analysis.

Yong et al. [135] propose a points-to analysis to handle structures and type-casting. They observe that supporting field-sensitivity can significantly improve the analysis precision. They also illustrate that making conservative assumptions when casting is involved usually does not cost much in terms of analysis time, space or precision.

Cheng and Mei [17] propose a modular interprocedural pointer analysis based on access-paths for pointers. They illustrate that access-paths can reduce the overhead of representing context-sensitive transfer functions.

Martena and Pietro [85] apply model checker SPIN to compute precise alias analysis information. They show that in the case of intra-procedural alias analysis, a model checking tool can enhance precision as well as efficiency.

Pearce et al. [98, 97] propose a field-sensitive points-to analysis for modeling aggregates and function pointers. They find that a field-sensitive analysis is more expensive to compute, but yields significantly better precision over a field-insensitive analysis.

Lattner et al. [75] propose a heap-cloning based context-sensitive points-to analysis. For achieving a scalable implementation, they illustrate several algorithmic and engineering design choices such as, using a flow-insensitive and unification-based analysis, and sacrificing context-sensitivity within strongly connected components.

Sotin and Jeannet [120] address the problem of interprocedural analysis in the presence of pointers to the stack. They use abstract interpretation to define local semantics for their language and then apply relational interprocedural analysis to the local semantics to generate a forward semantics manipulating sets of activation records. Finally, they apply their interprocedural analysis for verifying relational properties on program variables.

Rountev et al. [109] propose points-to analysis for Java using annotated constraints. They use the annotated inclusion constraints to precisely and efficiently model the semantics of virtual calls and the flow of values via object fields.

Fahndrich et al. [35] propose a context-sensitive flow-analysis using instantiation constraints. They apply their analysis to develop a points-to analysis algorithm. They show that flow information can be computed efficiently while considering only the paths with well-defined call-return sequences, even for higher-order programs.

Foster et al. [38] propose the use of annotated type-qualifier *restrict* to specify that certain pointers are not aliased to other pointers within a lexical scope. Aiken et al. [1] extend it to support another annotation called *confine* for restricting expressions, rather than single variables. They also give algorithms to infer restricted variables and confined expressions. They find that the use of annotation can significantly improve the analysis precision and can help in finding several real-world bugs.

Heintze and Tardieu [54] propose a database-centric analysis architecture called compile-link-analyze (CLA) and an algorithm for computing dynamic transitive closure to develop a very fast points-to analysis. Their system is able to analyze about a million lines of unprocessed C code in less than a second without using more than 10 MB of memory.

Whaley and Lam [128] use Heintze and Tardieu's points-to analysis [54] and Cheng and Mei's access paths [17] to develop an efficient reference analysis for Java. They show the

effectiveness of their field-sensitive and intra-procedural flow-sensitive method by computing precise static call-graphs for very large Java programs.

Hardekopf and Lin [50] propose a semi-sparse flow-sensitive analysis for efficient handling of strong updates. They convert non-address-taken or top-level variables to Static Single Assignment (SSA) form to improve analysis efficiency.

Yu et al. [136] propose a field-sensitive, flow-sensitive and context-sensitive pointer analysis by analyzing pointers according to their levels. Their analysis is fully sparse flow-sensitive which is a generalization of Hardekopf and Lin's semi-sparse flow-sensitivity [50].

Our work on solving points-to analysis as a set of linear equation is an exact (inclusion-based) analysis (Chapter 6).

3.7 Methods Achieving Explicit Trade-offs

In this section, we discuss those works which offer an explicit control over the trade-off between precision and scalability (in terms of analysis time and/or memory requirement).

Ryder [113] discusses several analysis dimensions that affect precision, including flow-sensitivity, context-sensitivity, field-sensitivity, program representation, directionality of information flow. Various choices for implementing these dimensions lead to different trade-offs between precision and scalability. For instance, a flow-sensitive, context-sensitive analysis is more precise and requires more time than its flow-insensitive, context-insensitive counterpart.

In the context of object oriented languages, Milanova et al. [88] propose parameterized *object sensitivity*, which analyzes a method separately for each object name on which that method is invoked. Their parameterization framework offers an explicit control over the trade-off between precision and analysis time by changing the parameters.

Buss et al. [11] propose an analysis space for pointer analysis based on ordering of program statements, modeling of conditionals and handling of strong updates. By choosing various values for these three configurable parameters, they show that one can design an analysis with the desired precision and scale.

In Steensgaard's analysis [123], every pointer has a single outgoing points-to edge for all its pointees, whereas in case of Andersen's analysis [3], a pointer has one outgoing points-to edge for each of its pointees. By choosing `k` outgoing edges between these two extremes, Shapiro

and Horwitz [117] propose an algorithm which can be tuned so that its worst-case time and space requirements and its precision range from those of Steensgaard's analysis to those of Andersen's analysis.

Hasti and Horwitz [52] propose an iterative algorithm to make the results of flow-insensitive analysis more and more precise using Static Single Assignment (SSA) form. Depending upon a client requirement, the algorithm can iterate only for a limited number of times to achieve the desired trade-off between analysis time and precision, still guaranteeing a sound result.

The concept of offline variable substitution [108] is widely known for maintaining the precision of the original analysis. However, as suggested by the authors in their paper, it can also be used to offer a trade-off between scalability and precision. This can be done by identifying a set of pointer variables and substituting them with a representative. If all the variables in the set are pointer equivalent, there is no loss in analysis precision. However, by adding pointers to this set which are not pointer equivalent, the number of variables tracked during the analysis can be reduced resulting in a more efficient analysis. This *imprecise* substitution results in some loss of precision. A client, depending upon its requirement, can select an appropriate precision-scalability trade-off.

Approximate pointer analysis [94] identifies pointers with *mostly similar* points-to sets as pointer equivalent and identifies pointees with *mostly similar* pointed-by sets as location equivalent. A client may choose an appropriate similarity threshold to achieve a desired trade-off between analysis precision and scalability.

The work on program decomposition [138] helps an analysis choose different parts of a program to be analyzed with varying levels of precision. Zhang et al. [138] present a program decomposition technique that partitions program statements to allow separate pointer analyses to be used on independent parts of the program. This decomposition enables exploration of trade-off between algorithm efficiency and precision.

Various configuration parameters in our work on points-to analysis using bloom filters (Chapter 4) offer a client an explicit control over the precision-scalability trade-off. By choosing different values for the number of bits in each bucket `B`, the number of context bits `C`, the number of hash functions `D`, etc., the analysis time, memory and precision can be tuned as per the requirement. We also show that with minimal precision loss, which can be probabilistically bounded, our bloom filter based points-to analysis achieves significant reductions in analysis

time and memory requirement.

Selection probability or the degree of randomization in our randomized points-to analysis (Chapter 5) also allows for explicitly controlling the scalability-precision trade-off. Further, a client can choose the number of randomized runs as an additional control parameter. Depending upon the values of these configuration parameters, our randomized analysis is able to achieve a significant reduction in analysis time at the cost of a small amount of precision.

Several client-driven analyses adjust their analysis time and precision based on the client needs. We review the work on client-driven analysis in the next section.

3.8 Client-driven and Demand-driven Methods

In this section, we discuss client-driven and demand-driven points-to analyses. A client-driven analysis adjusts its cost and the achieved precision according to the needs of the client analyses (or programs). Thus, if a client $\texttt{C}_1$ requires a high precision, a client-driven points-to analysis can tune its configurable parameters or its algorithm to extract precise points-to information from the program. In contrast, if the goal of another client $\texttt{C}_2$ is a scalable analysis, the same client-driven points-to analysis can configure itself to extract sound points-to information with as little analysis time as possible, at the cost of some precision.

Guyer and Lin [45, 46] propose client-driven pointer analysis for C programs. Their analysis has two passes. The first pass is a fast, low-precision points-to analysis to discover the precision demands of various parts of the program. The second pass uses this information along with the feedback from the client to run a customized precision policy on different parts of the program. Their analysis treats data objects in a flow-sensitive or flow-insensitive manner and procedures in a context-sensitive or context-insensitive manner depending upon the precision policy.

Shapiro and Horwitz's algorithm [117] is an instance of client driven pointer analysis. They propose a points-to analysis whose worst-case time and space requirements and its precision range from those of Steensgaard's analysis [123] to those of Andersen's analysis [3]. This is based on the requirement of a client to choose an appropriate, `k` number of outgoing points-to edges for a pointer. In Steensgaard's analysis [123], every pointer has a single outgoing points-to edge for all its pointees, whereas in case of Andersen's analysis [3], a pointer has one outgoing points-to edge for each of its pointees. By choosing `k` outgoing edges between these

two extremes, a client can use their algorithm suitable to its needs.

A client-driven points-to analysis, like a regular points-to analysis, is exhaustive; i.e., it computes the points-to sets for all the pointers in a program. However, often a client is interested only in a subset of the points-to information. A demand-driven points-to analysis computes only the required amount of points-to information to answer *a* particular query of the client. As more queries are processed, a demand-driven analysis computes more and more information on-the-fly. We discuss the work on demand-driven points-to analysis next.

Heintze and Tardieu [53] introduce demand-driven context-insensitive and flow-insensitive points-to analysis. They use deductive reachability formulation to propose a provably optimal demand-driven analysis for C. They observe that the performance of their demand-driven analysis depends heavily on the amount of points-to information that needs to be computed to answer an alias query. Thus, if a query requires only a small amount of points-to information to be computed, then the analysis is very fast. However, if a query requires a large amount of points-to information, then the analysis can be slower than an exhaustive analysis.

Sridharan et al. [122] propose demand-driven points-to analysis for Java. They formulate Andersen's analysis [3] as a context-free language (CFL) reachability problem [106] and show that Andersen's analysis for Java is a balanced-parentheses problem. By exploiting this balanced parentheses structure, they obtain an asymptotically faster analysis. Their analysis allows a client to set a time-budget for answering a query, terminating the query once the time-budget is exceeded. They show that their algorithm yields much higher precision than previous techniques within small time-budgets.

Sridharan and Bodik [121] build upon their previous work [122] to propose a demand-driven, client-driven refinement-based context-sensitive points-to analysis for Java. Their technique simultaneously refines handling of method calls and heap accesses allowing the analysis to precisely analyze important code, skipping irrelevant code. One of the major contributions of their work is to develop an inclusion-based context-sensitive points-to analysis that has context-sensitive call-graph and context-sensitive heap abstraction, and is shown to scale for large programs.

Zheng and Rugina [140] propose a demand-driven alias analysis for C. Similar to the work on demand-driven points-to analysis for Java [122, 121], they also formulate the computation of (alias) queries as a CFL-reachability problem. The aliasing relations in their analysis can be

described using two, mutually dependent, hierarchical state machines, one for memory aliases and the other for value aliases. A useful aspect of their approach is that it does not require building or intersecting points-to sets. Their technique has been shown to be very efficient in practice, which makes it a good candidate for interactive tools.

3.9 Incremental and Probabilistic Methods

In this section, we discuss the work on incremental points-to analysis and probabilistic points-to analysis. An incremental analysis allows for dynamic addition and/or deletion of a set of statements to the already analyzed program and processes the statements without having to analyze the complete program (original program plus new statements) from scratch. An incremental points-to analysis allows for on-the-fly addition or deletion of a points-to constraint over existing constraints with pre-computed points-to information for the existing constraints.

Yur et al. [137] propose an incremental flow-sensitive and context-sensitive points-to analysis algorithm to handle addition and deletion of single statements in a C program. For an incremental change, their worklist-based method identifies the affected region and updates the interprocedural control flow graph to reflect the change. As a next step, their method adds the relevant aliases onto the worklist which is reiterated to find the final aliasing solution.

Saha and Ramakrishnan [114] describe a framework based on logic programming for implementing various incremental and demand-driven program analyses formulated using deductive rules. They instantiate their framework for an incremental and demand-driven points-to analysis.

A definite or a non-probabilistic points-to analysis computes points-to information which may or must hold at various program points during the execution of the program. Such an analysis does not quantify the certainty with which a points-to fact would hold at a program point. A probabilistic points-to analysis, in contrast, assigns a probability with each points-to fact computed. The result of such an analysis can help optimize the runtime execution of a program, e.g., speculative execution of such a program can make intelligent decisions based on the likelihood of a points-to fact.

Hwang et al. [60, 16] introduce probabilistic points-to analysis. To identify the probabilities with which each points-to fact is generated and preserved, their approach first computes the

transfer functions for probabilistic data-flow analysis. The probability of each points-to fact is then computed using the transfer functions. Chen et al. [15] use the above probabilistic points-to analysis for speculative multithreading.

Silva and Steffan [23] propose a one-level flow-sensitive and context-sensitive probabilistic points-to analysis by encoding linear transfer functions as sparse matrices. They demonstrate that, even without edge-profiling information, their analysis can provide accurate probabilities for the points-to facts.

In the context of object-oriented programs, Sun et al. [124] propose probabilistic points-to analysis for Java. In contrast to the former work in the context of C programs, their work handles object-oriented features such as inheritance and polymorphism.

3.10 Analysis of Parallel Programs and Parallel Analyses

In this section, we discuss the work related to pointer analysis of multithreaded programs and parallel versions of pointer analysis itself. Due to numerous thread-interleavings possible in a multithreaded program, the analysis of such programs poses severe challenges from precision and scalability perspectives.

Rugina and Rinard [111, 112] propose an interprocedural, context-sensitive and flow-sensitive pointer analysis for multithreaded programs. Their method extracts thread interference to take into account the shared pointers accessed by parallel threads.

Salcianu and Rinard [116] propose a combined pointer and escape analysis for multithreaded programs. Their algorithm uses interaction graphs to analyze the interactions between threads and is compositional, i.e., it analyzes each method or thread once to extract a parameterized analysis result that can be specialized in a context.

There has been some work on parallelizing the pointer analysis algorithm itself. While some of the former approaches simply mention that their algorithms could be parallelized [66, 138, 110], the first parallel pointer analysis is proposed by Mendez-Lojo et al. [86]. They illustrate that inclusion-based points-to analysis can be formulated entirely in terms of graphs and graph-rewrite rules. Their algorithm exposes the amorphous data-parallelism in irregular applications.

Edvinsson et al. [30] propose parallel points-to analysis for object oriented programs. It

deals with different target methods of polymorphic function calls and independent control flow branches.

3.11 Application of Points-to Analysis

Pointer analysis is not an optimization; a client needs to use the computed points-to information for performing an optimization over the program. In this section, we discuss various clients which have been shown to make use of pointer analysis.

Livshits and Lam [80] propose an extended form of SSA, called IPSSA, to track pointers and apply it for finding buffer overruns and format string violations in C programs.

Milanova et al. [87] apply a pointer analysis [138] for precise call-graph construction. They find that for call-graph construction as a client, an inexpensive pointer analysis may provide precise enough information.

Wu et al. [133] propose element-wise points-to mapping for loop-based dependence analysis. An element-wise points-to mapping summarizes the relation between a pointer and the heap object it points to, for every instance of the pointer inside a loop and for every array element directly accessible through the pointer. They demonstrate that element-wise points-to information can significantly improve the precision of loop-based dependence analysis.

Avots et al. [4] use a context-sensitive, field-sensitive points-to analysis to detect security vulnerabilities in C programs. By assuming a restricted, but common usage C syntax, they improve the pointer analysis precision. They show that their optimistic pointer analysis can be used to reduce the overhead of a dynamic string-buffer overflow detector.

Buss et al. [11] propose an analysis space for pointer analysis based on ordering of program statements, modeling of conditionals and handling of strong updates. By choosing various values for these three configuration parameters, they show that one can design an analysis with the desired precision and scale. They apply the developed points-to analyses for bug finding and show that the precision of the underlying points-to analysis directly affects the precision of the bug finding tool.

Mock et al. [90] apply dynamic points-to information to improve the precision of static program slicing for C. They find that programs with many call sites that make calls through

function pointers experience a significant reduction in slice size when dynamic points-to information is used. However, for other programs which do not make much use of function pointers for calling functions, there is little reduction in slice size.

Ghiya and Hendren [41] use pointer information to develop a shape analysis for C programs. Their analysis can detect if a data structure has a tree-like structure, a directed acyclic graph or whether it is cyclic.

Tonella et al. [125] apply their flow-insensitive and context-insensitive points-to analysis for C++ to reaching definitions analysis and slicing.

Guyer and Lin [45, 46] apply a client-driven pointer analysis for C programs to several error detection problems. Their analysis has two passes. The first pass is a fast, low-precision points-to analysis to discover the precision demands of various parts of the program. The second pass uses this information along with the feedback from the client to run a customized precision policy on different parts of the program. Their analysis treats data objects in a flow-sensitive or a flow-insensitive manner and procedures in a context-sensitive or a context-insensitive manner depending upon the precision policy. They claim that typical clients need a small amount of extra precision applied to selected parts of each program and one can trade off precision for scalability for the remaining parts.

Silva and Steffan [23] propose a one-level flow-sensitive and context-sensitive probabilistic points-to analysis for speculative optimizations.

Orso et al. [95] classify data dependences in the presence of pointers and then make use of the classification for data-flow testing and to develop an incremental slicing algorithm.

Hind and Pioli [57] compare the effect of various pointer analyses on different clients including Mod/Ref analysis, live variable analysis, reaching definitions analysis, conditional constant propagation and dead code elimination.

In Chapter 4, we apply our bloom filter-based points-to analysis for Mod/Ref analysis. We find that the effect of false positives incurred by our approximate representation is very less on the client.

3.12 Evaluations and Quantifications

In this section, we discuss the work which focus on the experiments to obtain insights about the usage of pointers in programs and properties of a pointer analysis.

Hackett and Aiken [47] study four applications to identify common aliasing patterns that arise in practice. They find that almost all pointers are used as one of the following nine use-cases: *parent pointers* are references to data closer to the root of a data structure, *child pointers* are references to data stored deeper in a data structure, *shared immutable pointers* are multiple references to the data and all are used only for reading, *shared I/O pointers* are two references to the data where one is used only for writing and the other only for reading, *global pointers* are references to the globals, *index cursors* to support an additional index for a structure, *tail cursors* to hold the end point of an index, *query cursors* to read data internal to an index, and *update cursors* to write data internal to an index.

Lhotak and Hendren [77] present a framework of BDD-based context-sensitive points-to analyses for Java. Using their framework, they evaluate the precision of various context-sensitive analyses. Two of their main findings are (i) object-sensitive analyses are more precise than comparable variations of other approaches, and (ii) context-sensitive heap-abstraction improves precision more than extending the length of the context string.

Hind and Pioli [56, 57] compare different points-to analyses on C programs. The analyses vary in their use of control-flow information and their work quantifies the effect of varying flow-sensitivity on the analysis performance in terms of analysis time and precision. They also report their findings on how the points-to information computed by each of the analyses affects different clients, namely, Mod/Ref analysis, live variable analysis, unreachable code identification, reaching definitions analysis, dependence analysis, and conditional constant propagation. One of the main findings of their study is that a flow-sensitive pointer analysis offers only a small amount of additional precision over a flow-insensitive analysis [57]. They also find that the time and space efficiency of a client analysis improves as the pointer analysis precision is increased.

Das et al. [25] estimate the impact of scalable flow-insensitive and context-insensitive analyses on compiler optimizations. Their major finding is that limited forms of context-sensitivity and subtyping provide the same precision as algorithms with full context-sensitivity and subtyping.

Zhang et al. [139] present results of their combined analysis which uses program decomposition to apply different aliasing to independent program segments. They find that combined analysis allows application of a flow-sensitive analysis to segments of a program which is too large to be analyzed by a flow-sensitive analysis as a whole. They also find that points-to analysis is more efficient than an alias analysis.

Mock et al. [91] evaluate the degree of imprecision caused by static pointer analysis for C programs with respect to the actual behavior of pointers at run-time. They find that the pointer information produced by existing scalable static pointer analyses is far worse than the actual behavior observed at run-time. They advocate usage of profile data on pointer values to improve analysis precision.

Mock et al. [90] apply dynamic points-to information to improve the precision of static program slicing for C. They find that programs with many call sites that make calls through function pointers experience a significant reduction in slice size when dynamic points-to information is used. However, for other programs which do not make much use of function pointers for calling functions, there is little reduction in slice size.

Diwan et al. [28] evaluate three alias analyses based on programming language types for Modula-3. They find that type-compatibility alone yields a very imprecise alias analysis. However, if it is coupled with field-sensitivity, it significantly improves precision.

In the context of Java, Liang et al. [79] evaluate the precision of static reference analysis using profiling information. They demonstrate that modeling heap-allocated memory with the allocation site may be sufficiently precise for most allocation sites. They also find that static Andersen's analysis [3] can compute very precise information for some allocation sites, but can also compute very imprecise information for many allocation sites. Further, they illustrate that existing approaches may compute very imprecise points-to information for programs that use sophisticated data structures.

Ribeiro and Cintra [107] also investigate the sources of uncertainty in the points-to information computed by a static analysis. Their approach also makes use of the profiling information, but in contrast to the other works in this direction, they wish to quantify the amount of uncertainty that is intrinsic to the applications. They find that often static pointer analysis is very accurate, but for some benchmarks a significant fraction, up to 25%, of their accesses via pointer-dereferences cannot be statically disambiguated. They claim that the main reasons

behind this behavior is the use of pointer arithmetic and the fact that some control paths are not taken.

3.13 Points-to Analysis for Other Languages

In this section, we discuss various points-to analysis algorithms for other languages such as Java and Python.

Apart from C, one of the languages for which pointer analysis is developed is Java [63]. Since the pointers in Java are called references, the analysis is often termed as reference analysis. We mention the work in the context of Java below. Berndl et al. [6], Whaley and Lam [129] and Zhu and Calman [142] propose variants of points-to analysis algorithms using BDDs for Java. Sridharan et al. [122] propose demand-driven points-to analysis. Sridharan and Bodik [121] build upon their previous work [122] to propose a demand-driven, client-driven refinement-based context-sensitive points-to analysis. Sun et al. [124] propose probabilistic points-to analysis for object-oriented programs. Lhotak and Hendren [77] present a framework of BDD-based context-sensitive points-to analyses. Liang et al. [79] evaluate the precision of static reference analysis using profiling information.

Gorbovitski et al. [43] propose alias analysis for a dynamic object-oriented language, for program optimization by incrementalization and specialization. Incrementalization is a language optimization for reducing the cost of expensive queries. Specialization is an optimization technique for generic code. They instantiate their flow-sensitive and context-sensitive analysis for Python [102].

Tonella et al. [125] propose a flow-insensitive and context-insensitive points-to analysis for C++, which handles various object-oriented features like polymorphism, templates and dynamic binding. They show its effectiveness by applying it to reaching definitions analysis and slicing.

In contrast to traditional languages like C and Java, JavaScript [64] has dynamic features such as run-time modification of objects through addition of properties and updating of methods. Jang and Choe [62] propose the first points-to analysis for JavaScript. Their analysis can identify the use of a structure's field directly or via a property, to improve precision over a traditional Andersen's analysis [3].

Jovanovic et al. [65] propose a precise alias analysis for server-side scripting languages like PHP [101]. They target their analysis towards the unique reference semantics commonly found in scripting languages and apply it to detect web application vulnerabilities.

While the above work is related to dynamic languages, there has also been some work on dynamic pointer analysis, i.e., performing pointer analysis while the program is running. Salami and Valero [115] propose a dynamic interprocedural pointer analysis for multimedia applications using a memory disambiguation technique called dynamic memory interval test. Hirzel et al. [58] propose online pointer analysis for Java.

Since most alias analyses are formulated in terms of high-level language features, they cannot easily handle features such as pointer arithmetic and out-of-bound array references. To handle these issues, Debray et al. [26] propose alias analysis of executable code. In order to be practical, their algorithm trades off precision for memory requirement. They show that their analysis is able to provide non-trivial information about 30% to 60% of the memory references.

3.14 Chapter Summary

In this chapter we presented a survey of various pointer analysis methods. We classified the analyses in 12 categories and briefly discussed the work in each category.

Chapter 4

Points-to Analysis using Bloom Filter

4.1 Introduction

A major cause for the non-scalability of context-sensitive points-to analysis is its large memory requirement. Thus, with an objective of reducing the memory requirement, we propose to store points-to information in an approximate manner. We call an analysis that does not store points-to information in an approximate manner as *exact*. In order to be useful, an approximate representation should ensure *safety*. In other words, it should always store a superset of the points-to information computed by *exact*. This would ensure that a points-to fact (say, p points to q) identified by an exact method as exhibited by the program would always be included in the approximate analysis. That is, there should be no false negatives. The representation may, however, contain false positives, and may store a few spurious points-to facts. Thus, an alias query *DoAlias(p, q)*, checking whether pointers p and q alias with each other, may be answered affirmatively by our *approximate* analysis while an *exact* analysis may say otherwise. In order for our approach to be effective, these false positives should be as few as possible. Thus, our goal is to develop a data structure for storing points-to information in an approximate manner which reduces the memory requirement of the points-to analysis significantly while ensuring that there are no false negatives and minimizing the false positives. We achieve this goal using bloom filters [7].

A bloom filter is a compact and approximate representation (typically in the form of a bit

vector) of a set of elements. It trades off some precision for significant savings in memory. It is a lossy representation that can incur false positives, i.e., an element not in the set may be answered to be in the set. However, it does not have false negatives, i.e., no element in the set would be answered as not in the set. To maintain this property, the operations on a simple bloom filter are restricted so that items can only be added to the set but can never be deleted [1]. Our motivation for using bloom filters for context-sensitive flow-insensitive points to analysis stems from the following three key observations.

1. **Conservative static analysis.** As with any other compiler analysis, static points-to analysis tends to be conservative as correctness is an absolute requirement. Thus, in case of static may-points-to analysis, a pointer not pointing to a variable at run time can be considered otherwise, but not vice-versa. Hence a bloom filter that does not have false negatives, is a safe representation. However, such a representation may answer that a pointer points to a few extra pointees. This only makes the analysis less precise and does not pose any threat to correctness. Further, as a bloom filter is designed to efficiently trade off precision for space, it is an attractive representation to enable scalability of points-to analysis.

2. **Sparse points-to information.** The number of pointees that each context-wise pointer (pointer under a given calling context) actually points to is many orders of magnitude less than both the number of context-wise pointers and the total number of potential pointees. Hence, though the points-to set can potentially be very large; in practice, it is typically small and sparse. A bloom filter is ideally suited to represent data of this kind. When the set is sparse, a bloom filter can significantly reduce the memory requirement with a probabilistically low bound on loss in precision.

3. **Monotonic dataflow analysis.** As long as the underlying analysis uses a monotonic iterative data flow analysis, the size of the points-to set can only increase monotonically. This makes a bloom filter a suitable choice as monotonicity guarantees that there is no need to support deletions.

[1] Counting bloom filters [36] support limited number of deletions but they suffer from the drawback of allowing false negatives.

The above observations make bloom filter as a promising candidate for representing points-to information. However, using the bloom filter as originally proposed [7] is not efficient for a context-sensitive analysis.

We present the necessary background on bloom filter in the next section. We also discuss why a naive bloom filter is not suited for points-to analysis. In Section 4.3, we extend the basic bloom filter to a multi-dimensional bloom filter (*multibloom*) to enable efficient storage and manipulation of context-aware points-to information. The added dimensions correspond to pointers, calling contexts, dereferences, and hash functions. The first three dimensions, namely, pointer, calling contexts and dereferences, are required to efficiently support all the common pointer manipulation operations (`p = &q`, `p = q`, `p = *q` and `*p = q`) and the query operations `DoAlias(p, q)` and `DoAlias(p, q, c)` for context-insensitive and context-sensitive analyses respectively. The fourth dimension (hash functions) is essential to control loss in precision. A multibloom not only suits our purpose, but also gives clients an ability to control the trade-off between memory and precision. We extend the algorithm for context-sensitivity in Section 4.4. We evaluate the effectiveness of our approximate points-to analysis using multibloom in Section 4.5. We theoretically show and empirically observe that as the number of hash functions increases, the precision loss decreases. Further, with a relatively small number (16 or so) hash functions, we achieve precision loss lower than 1%. In effect, multibloom significantly reduces the memory requirement with a very low probabilistically-bound precision loss. The compact representation of points-to information allows the context-sensitive analysis to scale well with the program size. In order to evaluate how the precision loss of the underlying points-to analysis affect a client, we evaluate Mod/Ref analysis using our approximate points-to analysis in Section 4.7. We show that our points-to analysis using multibloom does not adversely affect the client and gives significant savings in memory requirement and analysis time. We conclude the chapter with a summary in Section 4.9.

4.2 Bloom Filter

A bloom filter is a probabilistic data structure used to store a set of elements and test the membership of a given element [7]. In its simplest form, a bloom filter is an array of N bits, with an associated hash function h. An element e belonging to the set is represented by setting

the k^{th} bit to 1, where $h(e) = k$. For instance, if the hash function is $h_1(e) = (3 * e + 5)$ mod 11, then for elements $e = 13$ and 20, the bits 0 and 10 are set. Membership of an element e is tested by using the same hash function. Note that element 2 also hashes to the same location as 13. This introduces false positives, as the membership query would return *true* for element 2 even if it is not inserted. Note, however, that there is no possibility of false negatives, since none of the bits are reset.

The false positive rate of a bloom filter can be reduced drastically by using multiple hash functions. Thus, if we use a second hash function in the above example, namely, $h_2(e) = (8 * e + 2) \mod 7$, then the elements $e = 13$, 20 and 2 get hashed to bits 1, 1 and 4 respectively. Note that a membership query for 2 would return *false* as location 4 corresponding to $h_2(2)$ is 0, even though location 0 corresponding to $h_1(2)$ is set. Thus, with two hash functions, the false positive between elements e_1 and e_2 can happen if and only if $h_1(e_1) = h_1(e_2)$ and $h_2(e_1) = h_2(e_2)$.

For a bloom filter of size N bits and with d hash functions, the false positive rate P, after n elements are inserted in the bloom filter, has been derived by Bloom [7] as below.

$$P = \frac{(1/2)^d}{(1 - \frac{nd}{N})} \tag{4.1}$$

This is under the assumption that the individual hash functions are *random* and different hash functions are *independent*. Note that for fixed values of n and N, P reduces exponentially with increasing the number of hash functions d.

Traditionally, sparse bitmaps [100] and Binary Decision Diagrams (BDD) [129] are the most prevalent data structures used to store points-to information. Sparse bitmaps have the desirable property that the time to insert elements and to check for their membership is independent of the number of elements added. BDDs improve upon sparse bitmaps by exploiting commonality across the information being stored, reducing the overall memory requirement. However, points-to analysis using BDDs is 2× slower than that using sparse bitmaps; whereas using sparse bitmaps consumes 5.5× more memory than using BDDs [49]. We show that bloom filter allows us to enjoy the best of both the worlds: it is as fast as sparse bitmaps and its storage requirement is much less than that of BDDs. This makes bloom filter a promising candidate for scaling points-to analysis.

4.2.1 Issues with a Naive Bloom Filter

A points-to tuple $\langle p, x, c\rangle$ represents a pointer p pointing to a variable x in calling context c. As defined in Section 2.2.3, a context is defined by a sequence of functions and their call-sites. A naive implementation using bloom filter would store context-sensitive points-to tuples by hashing the tuple $\langle p, x, c\rangle$ and setting that bit in the bloom filter. This simple operation takes care of statements only of the form $p = \&x$. Other pointer statements, like $p = q$, $p = *q$, and $*p = q$ require additional care. For example, for handling $p = q$ type of statements, the points-to set of q has to be copied to p. While bloom filter is very effective for existential queries, it is well-known that it is inefficient for universal queries like "*what is the points-to set of pointer p under context c?*".

One way to solve this problem is to keep track of the set of all pointees (objects). This way, the query $FindPointsTo(p, c)$ to find the points-to set for a pointer p under context c is answered by checking the bits that are set for each of the pointees. Although this is possible in theory, it requires storing all possible pointees, making it storage inefficient. Further, going through all the pointers every time to process a copy operation $p = q$ makes this strategy time-inefficient as well. Therefore, we propose an alternative design that has more dimensions than a conventional bloom filter in order to support the pointer operations.

4.3 Multi-dimensional Bloom Filter

Our proposed multi-dimensional bloom filter (*multibloom*) is a generalization of the basic bloom filter introduced in Section 4.2. It has 4 dimensions, one each for pointers, contexts, hash functions and a bit vector along the fourth dimension. It is represented as $mb[P][C][D][B]$. The configuration of a multibloom is specified by a 7-tuple $\langle P, C, D, B, M_p, M_c, H\rangle$ where P is the number of entries for pointers, C is the number of entries for contexts, D is the number of hash functions, B is the bit-vector size for each hash function[2], M_p is the hash function mapping pointers to an integer in the range $1..P$, M_c is the function mapping contexts to an integer in the range $1..C$ and H is the family of hash functions. The first 4 entries (P, C, D, B) denote the number of unique values that can be taken along each dimension. For example

[2]We assume that all hash functions use the same number of bits, although it is possible to have different sizes for each hash function.

$C = 16$ would mean that the multibloom has space for storing the pointee set for 16 different contexts in which a pointer is accessed. We will have to map every context of a given pointer to one among 16 entries. The total size of the structure is $Size = P \times\ C \times\ D \times\ B$. Functions M_p and M_c map the pointer p and context c to integers $Pidx$ and $Cidx$ in the range $[1..P]$ and $[1..C]$ respectively. A family of hash functions $H{=}(h_1, h_2, \cdots, h_D)$ map the pointee x to D integers $Hidx_1, Hidx_2, \cdots, Hidx_D$ respectively. These play the same role as the hash functions in the previous section.

Given a points-to tuple $\langle p, x, c\rangle$, it is stored into the multibloom as follows. Let $M_p(p) = Pidx, M_c(c) = Cidx$, and $h_i(x) = Hidx_i, \forall i \in [1..D]$. Then the tuple is added to multibloom by setting the following D bits to 1:

$$mb\,[Pidx]\,[Cidx]\,[i]\,[Hidx_i] = 1,\ \forall i \in [1..D]$$

It should be emphasized that a multibloom does not store complete context information explicitly. The context is hashed and accordingly appropriate multibloom bits are set. For a context-sensitive query the input context specified in the query is again hashed using the same hash function(s) and appropriate multibloom bits are checked. By not storing the context information explicitly, a multibloom achieves large savings in the memory requirement of the analysis, albeit at the expense of some loss of precision. Note that, in contrast, a sparse bitmap or a BDD is forced to store complete (possibly k-limited) context information explicitly.

Next we discuss how copy, load and store constraints are processed using multibloom.

4.3.1 Handling Copy Constraint

While processing $p = q$ type of statement under context c, all we need to do is to find the $B \times D$ source bits from the multibloom that correspond to pointer q under context c and bitwise-OR them with the $B \times D$ destination bits corresponding to pointer p under context c. This logically copies the pointees of q on to p without having to universally quantify all the pointees that q points to. The pseudo-code is given in Algorithm 2. Note that the operation $X \vee = Y$ in Line 7 logically performs the operation $X = X \vee Y$.

Example. Consider the program fragment given in the first column of Figure 4.1. Consider

Algorithm 2 Handling statement $p = q$ under context c in multibloom, with D hash functions and a B-bit vector

1: $Pidx_{src} = M_p\,[q]$
2: $Cidx_{src} = M_c\,[c]$
3: $Pidx_{dst} = M_p\,[p]$
4: $Cidx_{dst} = M_c\,[c]$
5: **for** $i = 1$ to D **do**
6: **for** $j = 1$ to B **do**
7: $mb\,[Pidx_{dst}]\,[Cidx_{dst}]\,[i]\,[j] \quad \vee = \quad mb\,[Pidx_{src}]\,[Cidx_{src}]\,[i]\,[j]$
8: **end for**
9: **end for**

statement	multibloom processing	comment
a = &x	a [ ][ ][1][ ][ ][ ][ ][ ]	set bit 2 corresponding to x.
b = &y	b [ ][ ][ ][ ][ ][ ][1][ ]	set bit 6 corresponding to y.
p = &a	p [1][ ][ ][ ][ ][ ][ ][ ]	set bit 0 corresponding to a.
d = b	d [ ][ ][ ][ ][ ][ ][1][ ]	bitwise−OR b's bucket.

Figure 4.1: Example program to illustrate points-to analysis using bloom filters

a multibloom having the following configuration

$$\langle P, C, D, B, M_p, M_c, H\rangle = \langle 6, 1, 1, 8, I, C_0, (h_1)\rangle$$

For illustration purpose, we consider a multibloom with a single hash function (D=1), a single context (C=1) and 8 bits per hash function. The map M_p is an identity function I that returns a different value for each pointer variable. C_0 maps every context to entry 0, since $C = 1$. The hash function h_1 is defined as $h_1(x) = 2,\ h_1(y) = 6,\ h_1(a) = 0$. As there is only one entry for context and each statement modifies one pointer, we illustrate the multibloom as 4 bloom filters. For clarity, we depict the multibloom as multiple bit-vectors in Figure 4.1. Initially, all the bits in the buckets of each pointer are set to 0. The state of the bloom filters after processing each constraint for the example code is shown in Column 2. To avoid clutter, empty cells indicate a zero bit.

The address-of constraints set appropriate bit corresponding to the hashed value of the

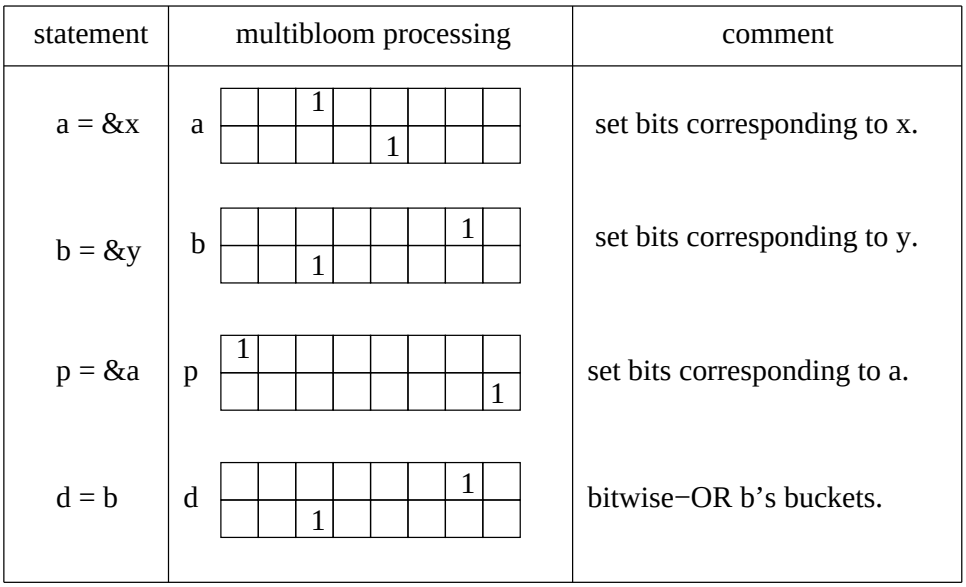


Figure 4.2: Example program using two hash functions

address-taken variable. The copy constraint `d = b` performs a bitwise-OR operation of `b`'s bucket with that of `d` and stores the result into `d`'s bucket.

Figure 4.2 shows the same example with two hash functions. The buckets corresponding to the two hash functions for a pointer are shown one below the other. Let the first hash function h_1 be defined as before and the second hash function h_2 is defined as $h_2(x) = 4$, $h_2(y) = 2$, $h_2(a) = 7$. Each address-of constraint of the form `p = &q` now sets two bits in the bloom filter for `p`, one corresponding to each hash function. A copy constraint `d = b` performs a bitwise-OR operation of each of the `b`'s buckets with the corresponding buckets of `d`, storing the result in the buckets of `d`. Thus, the bucket corresponding to h_1 (respectively h_2) of `b` is bitwise-ORed with the bucket corresponding to h_1 (respectively h_2) of `d` and the result is stored in the bucket corresponding to h_1 (respectively h_2) of `d`.

4.3.2 Handling Load and Store Constraints

There are two ways to handle statements of the form $*p = q$ and $p = *q$. One way is to extend the above strategy for copy constraints by adding more dimensions to the multibloom. This is extensible to multiple levels of indirection. This strategy would add more dimensions to our 4-dimensional bloom filter, one for each level of pointer dereference. Thus, each pointer dereference would be handled by a dimension and by appropriately copying bits from a dimension

(as explained below), one can handle load and store instructions. Clearly, this adds to storage and analysis time requirements. For S number of added dimension, the storage requirement would increase by a multiplicative factor of S bits and the time requirement would increase by a (small) constant amount of time required for an additional hash function calculation corresponding to the new dimension. The second way is to assume that multi-level pointers ($**p$, $***p$, and so on) point to the universal set of pointees and process the statements conservatively. This would make each multi-level pointer point to *all* the address-taken variables in the program, adding some spurious points-to information. However, the number of multi-level pointers is much less in programs compared to single-level pointers. Therefore, depending on the client analysis, one may be willing to lose some precision by following this conservative approach.

To obtain a good balance of storage requirement, analysis time and precision, we employ a combination of the above two techniques. We extend multibloom for two-level pointers ($**p$) and use the conservative strategy (universal set of pointees) for higher-level pointers ($***p$, $****p$ and so on). This conservative strategy results in little precision loss considering that on an average less than 1% of all dynamic pointer statements contain more than two levels of pointer indirections (obtained empirically).

Extending multibloom for two-level pointers makes it look like $mb[P][S][C][D][B]$ where S is the number of entries for pointers that are pointees of a two-level pointer. For higher-level pointers ($***p$ and above) an additional bit is set to indicate that the pointer points to the universal set of pointees. This approach makes the multibloom a 9-tuple $\langle P, S, C, D, B, M_p, M_s, M_c, H\rangle$ where M_s is the function mapping two-level pointers.

To handle load statement $p = *q$ where q is a two-level pointer, all the cubes $mb[q][s]$ (i.e., $C \times D \times B$ bits) corresponding to pointer q, for each $s = 1..S$ are bitwise-ORed to get a resultant cube. This cube is then bitwise-ORed with that of p, i.e., with $mb[p][1]$. This makes p point to the pointees pointed to by all pointers pointed to by q.

To handle store statement $*q = p$ where q is a two-level pointer, the cube $mb[p][1]$ of p is bitwise-ORed with each cube $mb[q][s]$ of q, for each $s = 1..S$. It makes each pointer pointed to by q point to the pointees pointed to by p.

Handling context-sensitive load/store statements requires a modification to address-of assignment $p = \&q$. If p is a two-level pointer, then to process the address-of statement in context

c, $D \times B$ bits of q are bitwise-ORed with $D \times B$ bits of p in the appropriate hash entry for q (see example below).

Example. Consider the program fragment given in the first column of Figure 4.3. Consider an extension of the multibloom in Section 4.3.1, with configuration

$$\langle P, S, C, D, B, M_p, M_s, M_c, H \rangle = \langle 6, 2, 1, 1, 8, I, h_s, -, (h) \rangle$$

Here, the map M_p is an identity function I that returns a unique value for each pointer variable. The hash function h is defined as $h(x) = 2, h(y) = 6$ and $h(a) = 0$. The mapping function h_s is defined as $h_s(x) = 0$ $h_s(y) = 1$ and $h_s(a) = 0$. Note that Figure 4.3 shows two buckets for each pointer although the number of hash functions is 1. This is because, S = 2. If the number of hash functions D had been 2, there would have been a total of $2 \times 2 = 4$ such buckets. Initially, all the bits in the buckets for each pointer are set to 0. The state of the bloom filters after each statement is processed is shown in the second column of Figure 4.3. Third column describes the multibloom operation.

For the address-of constraint a = &x, since $h_s(x) = 0$ and $h(x) = 2$, bit number 2 in the first bucket of a is set to 1. Then, x's buckets, which are all empty, are bitwise-ORed with the corresponding buckets of a. Similarly, the next address-of constraint $b = \&y$ is processed. The next address-of constraint $p = \&a$ is processed as follows. Since $h_s(a) = 0$, bucket 0 of p is selected. Further, since $h(a) = 0$, bit 0 of bucket 0 is set to 1. After this, the corresponding buckets of p and a are bitwise-ORed and stored in the buckets of p. This sets bit 2 of bucket 0 of p as shown in the third bloom filter of Figure 4.3. This extra operation makes sure that a's points to information is available with p when a dereference on p, i.e., $*p$, is performed. The next copy constraint $d = b$ is processed as before and bit 6 of bucket 1 of d is set, suggesting that now d points to y. The last constraint $d = *p$ is processed by bitwise-ORing the corresponding buckets of p and d. This makes sure that a's points to information, which was in the bucket 0 of p, is now available with d. Thus, now a membership query for the points-to fact $d \rightarrow x$ would be returned in affirmative, because $h_x(0)$, $h(x) = 2$ and bit 2 of bucket 0 of d is set to 1.

Note that the above strategy of using an additional dimension for two-level pointers can be extended to include more dimensions to accommodate higher-level pointers. The modified

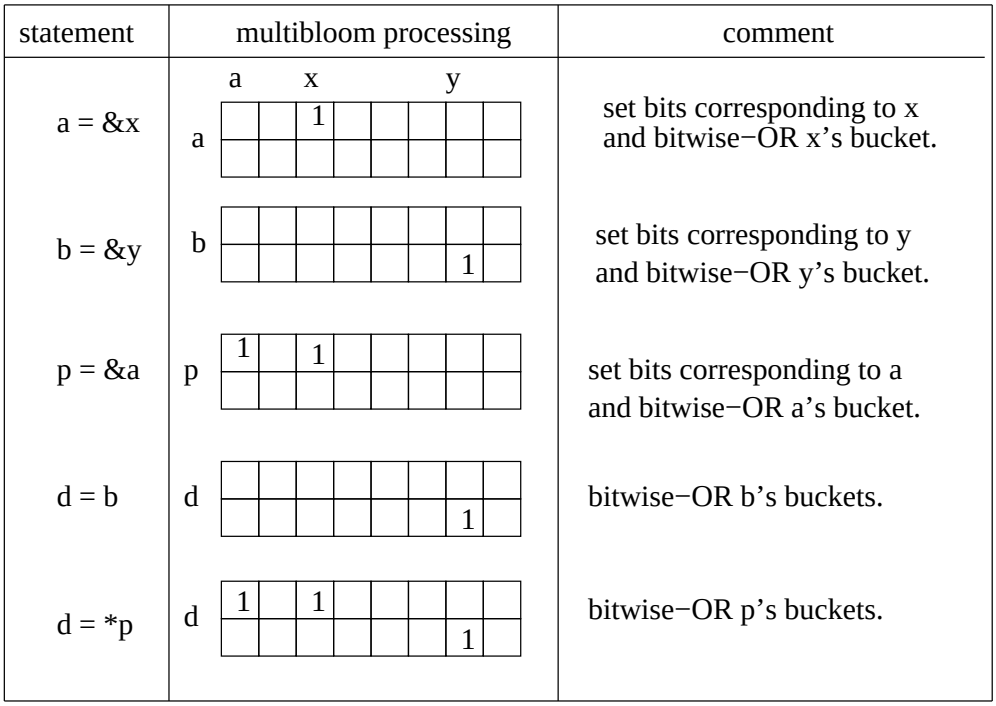


Figure 4.3: Example program to illustrate handling complex statements

pseudo-code for handling copy statement $p = q$ is given in Algorithm 3.

A flow-insensitive analysis iterates over the constraints until a fixed-point. Figure 4.4 shows our points-to analysis using multibloom using an extension of the example in Section 4.3.2. This example has an additional constraint $a = \&z$ and requires another iteration to reach the fixed-point. In this case, additionally, $h_s(z) = 1, h(z) = 4$. Thus, the address-of constraint $a = \&z$ sets bit 4 of bucket 1 of a. In the second iteration, the address-of constraint $p = \&a$ makes a's points-to information available to p, which sets bit 4 of bucket 1 of p. Later, d receives the corresponding points-to information from p making d point to z. The fixed-point is achieved at the end of the third iteration (not shown).

4.3.3 Extracting Information from Multibloom

In this section, we describe in detail the procedure for extracting points-to information from multibloom.

Checking for a points-to fact. To check if a pointer p points to an object o under a context c, the analysis maps p to its entry into the multibloom, maps c to its entry and determines

Algorithm 3 Modified algorithm for handling statement $p = q$ under context c

```
1: Pidx_q = M_p[q]
2: Cidx_q = M_c[c]
3: Pidx_p = M_p[p]
4: Cidx_p = M_c[c]
5: for s = 1 to S do
6:    for i = 1 to D do
7:       for j = 1 to B do
8:          mb[Pidx_p][s][Cidx_p][i][j] ∨= mb[Pidx_q][s][Cidx_q][i][j]
9:       end for
10:   end for
11: end for
```

Algorithm 4 Checking if p points to o under context c

```
1: Pidx = M_p[p]
2: Cidx = M_c[c]
3: for i = 1 to D do
4:    Hidx_i = h_i(o)
5:    if mb[Pidx][Cidx][i][Hidx_i] = 0 then
6:       return false
7:    end if
8: end for
9: return true
```

the hash value for each hash function. In order for the points-to fact to be present, *each* of the hashed bits in the mapped entries must be set to 1. This is because while inserting the points-to fact, the analysis must have set *each* of the hashed bits to 1 and it never reset any bit. However, it should be noted that some other points-to facts may have set *all* these bits and the analysis would falsely claim that the points-to fact is indeed present in the multibloom resulting in a false positive. This happens with a low probability as we demonstrate in Section 4.5. The procedure for checking a points-to fact is given in Algorithm 4.

Copying points-to set of a pointer. Points-to set of a pointer p can easily be copied to (or merged with that of) another pointer q by simply copying (bitwise-ORing) all the buckets of p to those of q. The procedure is the same as that for handling copy statement as shown in Algorithm 3.

Enumerating pointees of a pointer: *FindPointsTo(p).* Multibloom does not support

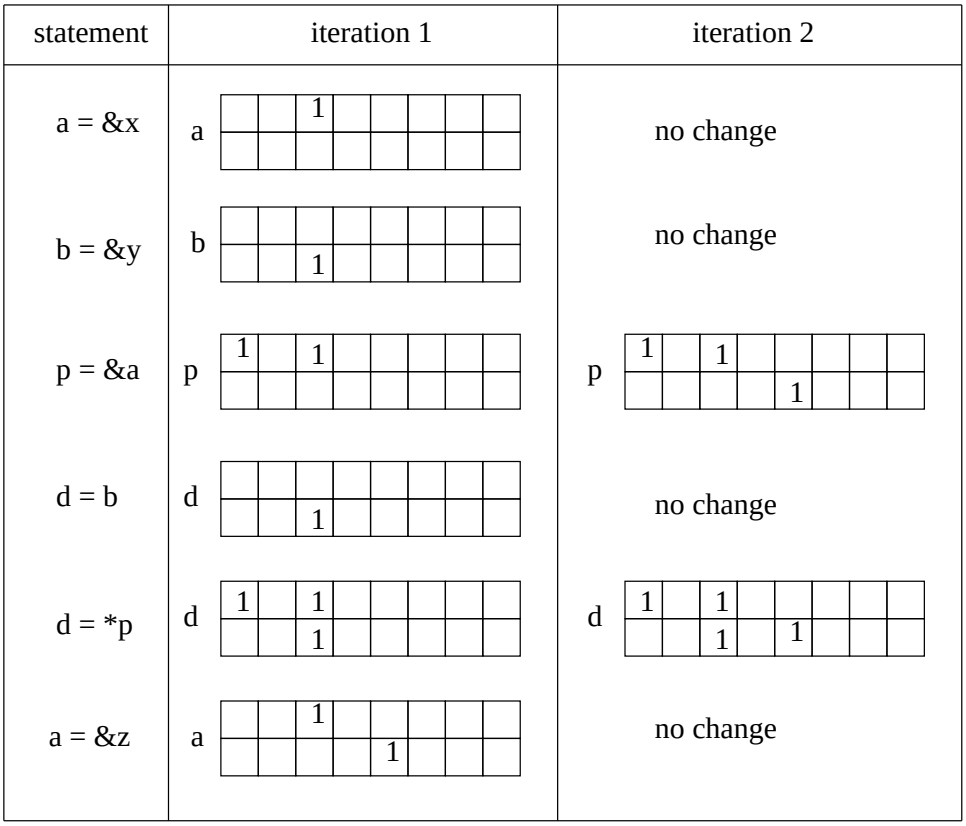


Figure 4.4: Example program to illustrate multiple analysis iterations

identifying which objects a pointer is pointing to. This is because it does not have a reverse mapping from a bit in a bucket to the corresponding pointee(s). As discussed in Section 4.2.1, maintaining this reverse mapping is inefficient. If a client analysis is interested only in checking if two pointers alias with each other rather than obtaining a detailed list of objects a pointer is pointing to, then an efficient procedure, discussed below, can be used.

Context-sensitive alias query: *DoAlias(p, q, c).* The ultimate goal of points-to analysis is to answer whether two pointers p and q alias with each other. A context-sensitive query is of type $DoAlias(p, q, c)$. To answer this query, for each hash function the algorithm needs to determine if the corresponding bit-vectors have at least one common bit with the value 1 indicating that both p and q potentially point to the same object and hence may alias. If no such bit exists for any one hash function, then p and q do not alias. The pseudo-code for handling context-sensitive query is given in Algorithm 5.

Algorithm 5 Handling context-sensitive query $DoAlias(p, q, c)$

```
1: Pidx_p = M_p[p]
2: Cidx_p = M_c[c]
3: Pidx_q = M_p[q]
4: Cidx_q = M_c[c]
5: for s = 1 to S do
6:    for i = 1 to D do
7:       hasPointee = false
8:       for j = 1 to B do
9:          if mb[Pidx_p][s][Cidx_p][i][j] = 1 and mb[Pidx_q][s][Cidx_q][i][j] = 1 then
10:            hasPointee = true
11:            break
12:         end if
13:      end for
14:      if hasPointee = false then
15:         return NoAlias
16:      end if
17:   end for
18: end for
19: return MayAlias
```

Algorithm 6 Handling context-insensitive query $DoAlias(p, q)$

```
for c = 1 to C do
   if DoAlias(p, q, c) = MayAlias then
      return MayAlias
   end if
end for
return NoAlias
```

Context-insensitive alias query: *DoAlias(p, q).* A context-insensitive query is of type $DoAlias(p, q)$. The query is answered by iterating over all possible values of the context c and calling the context-sensitive version of *DoAlias*: $DoAlias(p, q, c)$. If p and q alias in *any* context c, then it returns with an indication that p and q alias; otherwise they do not alias. The pseudo-code is shown in Algorithm 6. Note that enumerating all contexts means iterating over all possible *context* $c \in 1..C$ rather than actual calling contexts of a function. This makes the context-insensitive alias query efficient, as checking for the latter is many orders of magnitude higher than the former.

4.4 Context-sensitive Analysis

We extend Andersen's analysis [3] for context-sensitivity using an invocation-graph-based approach [32]. It enables us to disallow non-realizable interprocedural execution paths. We maintain a stack of function invocations, similar to that occuring at runtime, as we analyze the program starting from *main*. Thus, a *return* from a function always matches the function invocation at the top of the stack. This helps in mapping-unmapping of input arguments and return values of the function. In the presence of recursion, there are potentially infinite number of execution paths (contexts) and bounding the number is a major issue. We handle recursion by iterating over the cyclic parts of the call-chain and computing a fixed-point of the points-to tuples. Although this reduces analysis precision compared to a `k`-cfa [118] approach, which keeps track of `k` contexts inside recursion, the reduction is not substantial as we track complete contexts outside recursion. Our analysis is field-insensitive, i.e., we assume that any reference to a field inside a structure is to the whole structure. However, field-sensitivity does not pose any special challenges to our technique and our approach can be easily applied to field-sensitive points-to analysis. The context-sensitive version is outlined in Algorithm 7.

The algorithm takes four parameters: the function `f` to be processed, its calling context `cc` which includes `f`, the set of constraints `C` to be generated and the set of variables `V` to be created. The analysis starts by creating an entry in `V` for each global variable `g` as (`g`, {}) where {} denotes an empty context (not shown in the algorithm). It is then followed by the first call to the algorithm with parameters <*main*, {*main*}, C={}, V>. The procedure goes over all statements in the function and generates context-sensitive points-to constraints `C`. `C` is then evaluated as shown in the previous subsections. Lines 2–5 in Algorithm 7 create a new variable on encountering an *alloc* statement outside recursion. Lines 6–11 handle a non-recursive call. The first step is to add the callee to the call-chain followed by mapping the actual arguments to the formal arguments. The algorithm then recursively calls itself in Line 9 to process the invocation graph of the callee. The callee is analyzed the same way and the set of constraints `C` keeps getting updated. When the callee returns, the return value of the callee is mapped to the ℓ-value in the call statement. Finally, the calling context is updated by removing the callee. Lines 12–21 handle a recursive call by iterating over the cyclic call-chain and computing a fixed-point of the constraints in `C-cycle`. Note that the recursive call to Algorithm 7 in Line 17 uses the same call-chain. The fixed-point over the constraints `C-cycle` generated

Algorithm 7 Context-sensitive analysis

Require: Function `f`, call-chain `cc`, constraints `C`, variable set `V`

```
 1: for all statements s ∈ f do
 2:   if s is of the form p = alloc() then
 3:     if not inrecursion then
 4:       V = V ∪ {(p, cc)}
 5:     end if
 6:   else if s is of the form non-recursive call fnr then
 7:     cc.add(fnr)
 8:     add copy constraints to C for the mapping between actual and formal arguments
 9:     call Algorithm 7 with parameters < fnr, cc, C >
10:     add copy constraints to C for the mapping between the return value of fnr and ℓ-value
        in s
11:     cc.remove()
12:   else if s is of the form recursive call fnr then
13:     inrecursion = true
14:     C-cycle = {}
15:     repeat
16:       for all functions fc ∈ cyclic call-chain do
17:         call Algorithm 7 with parameters < fc, cc, C-cycle >
18:       end for
19:     until no new constraints are added to C-cycle
20:     inrecursion = false
21:     C = C ∪ C-cycle
22:   else if s is an address-of, copy, load, store statement then
23:     c = constraint(s, cc)
24:     C = C ∪ c
25:   end if
26: end for
```

in the cyclic call graph is then merged with `C` in Line 21. Lines 22–25 add the corresponding context-sensitive constraints for address-of, copy, load and store statements. A context-sensitive constraint contains variables in a particular context. For instance, a copy constraint is of the form $\mathtt{a}_{\mathtt{c}_1} = \mathtt{b}_{\mathtt{c}_2}$ where `a` and `b` are variables and $\mathtt{c}_1$ and $\mathtt{c}_2$ are contexts. The two sets, `C` and `V` are finally passed to a constraint solver. The reason for designing the analysis as a two-step process (generating constraints and solving them), rather than interleaving the two tasks, is to have a common constraint solving phase (with minor modifications) for context-sensitive and context-insensitive analyses.

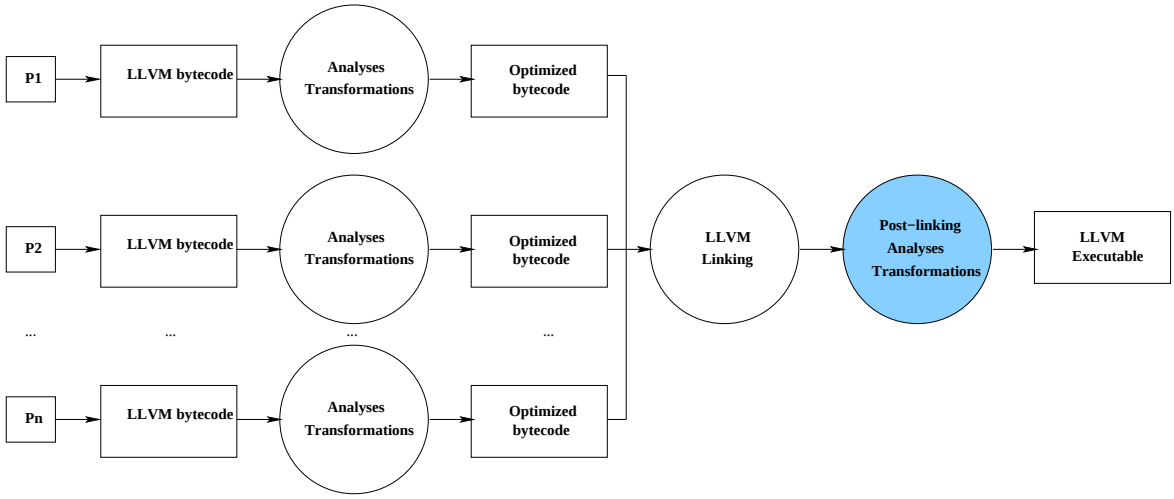


Figure 4.5: Placement of our analysis in LLVM compilation

4.5 Experimental Evaluation

Our implementation has been carried out in the LLVM compiler infrastructure [81] and the analysis is run as a post-linking phase. The block diagram of the complete compilation phase in LLVM is shown in Figure 4.5. The filled circle shows the placement of our analyses (all the pointer analyses in this thesis) in LLVM compilation. A (possibly partial) program $\texttt{P}_\texttt{i}$ is first compiled into an LLVM bytecode. A set of analyses and transformations analyzes and optimizes the bytecode. Several such partial programs $\texttt{P}_\texttt{i}$ are then linked by the LLVM linker which resolves external references. At this stage, post-linking optimizations and transformations are run on the linked bytecode. All our points-to analyses algorithms are implemented as post-linking analyses. This allows our analyses to take advantage of the resolved references to improve precision. The linking phase finally results into an executable image of the program components.

For bloom-filter-based analysis, we implement two points-to analyses, one which has an *exact* representation (without false positives) of the points-to set and the other uses our proposed multibloom representation. For an *exact* representation we store points-to information using sparse bitmaps, similar to Pereira and Berlin [100]. Both versions are implemented by extending Andersen's algorithm [3] for context-sensitivity as discussed in Section 4.4. Each aggregate (like arrays and structures) is represented using a single memory location, i.e., individual elements in an array and individual fields in a structure are not distinguished from other

Benchmark	KLOC	# Total Inst	# Pointer Inst	# Func
gcc	222.185	328,425	119,384	1,829
perlbmk	81.442	143,848	52,924	1,067
gap	71.367	118,715	39,484	877
vortex	67.216	75,458	16,114	963
mesa	59.255	96,919	26,076	1,040
crafty	20.657	28,743	3,467	136
twolf	20.461	49,507	15,820	215
vpr	17.731	25,851	6,575	228
eon	17.679	126,866	43,617	1,723
ammp	13.486	26,199	6,516	211
parser	11.394	35,814	11,872	356
gzip	8.618	8,434	991	90
bzip2	4.650	4,832	759	90
mcf	2.414	2,969	1,080	42
equake	1.515	3,029	985	40
art	1.272	1,977	386	43
httpd	125.877	220,552	104,962	2,339
sendmail	113.264	171,413	57,424	1,005
ghostscript	438.204	906,398	488,998	6,991
gdb	474.591	576,624	362,171	7,127
wine-server	178.592	110,785	66,501	2,105

Table 4.1: Benchmark characteristics

elements and fields. Both versions are optimized with online cycle elimination [34] which identifies pointer equivalent variables by checking for cycles dynamically in a constraint graph and offline variable substitution [108] which identifies pointer equivalent variables without running pointer analysis.

We evaluate the performance of the points-to analysis versions over 16 C/C++ SPEC 2000 benchmarks and five large open source programs: *httpd, sendmail, ghostscript, gdb* and *wine-server*. Their characteristics are given in Table 4.1. *KLOC* is the number of kilo lines of unprocessed source code, *Total Inst* is the total number of static LLVM instructions after optimizing at -O2 level, *Pointer Inst* is the number of static pointer-type LLVM instructions processed by the analysis and *Func* is the number of functions in the benchmark. The LLVM intermediate representations of SPEC 2000 benchmarks and open source programs were run using the *opt* tool of LLVM on an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM.

To quantify the loss in precision with a multibloom implementation, we use the *NoAlias*

percentage metric used in LLVM. It is calculated by making a set of alias queries for all pairs of pointer variables within each function in a program and counting the number of queries that return *NoAlias*. The *NoAlias* percentage metric is the ratio of the *NoAlias* count to the number of possible pointer pairs across all functions. Note that the *NoAlias* percentage need not (will not) be 100% even for the *exact* analysis. The *NoAlias* percentage metric represents, in some sense, the sparseness in the points-to pairs. The alias relationships between pointers become more and more sparse as the *NoAlias* percentage increases. Further, larger the *NoAlias* percentage, more precise is the analysis (upper bounded by the precision of the exact analysis). Since it covers all pairs of pointers in the program, the *NoAlias* percentage is a good way of estimating the precision loss due to false positives in the multibloom. We use *NoAlias* percentage normalized with respect to the exact analysis. Note that the normalized *NoAlias* percentage also represents the analysis precision. We define precision loss as below.

$$\text{Precision loss} = \left(1 - \frac{\textit{NoAlias} \text{ percentage of multibloom}}{\textit{NoAlias} \text{ percentage of exact}}\right)$$

We evaluate the performance of a multibloom for many different configurations and compare it with the exact implementation. In all evaluated configurations we allow the first dimension (P) to be equal to the number of unique pointers. This is done by choosing M_p to be an identity function that returns a unique value for each pointer. The number of pointers in a program is determined by making a quick initial pass over the program. The hash family H, the context mapper M_c and the pointer-location mapper M_s are hand-coded and are based on the hash functions at Arash Partow's website [96].

4.5.1 Performance of Exact Analysis: Baseline

We first present the results of our *exact* analysis that stores points-to information using sparse bitmaps and does not incur any false positives. This would allow us to present the remaining results of our approximate analysis using multibloom normalized with respect to the *exact* analysis. Normalized results would be easy to reason about and understand. Further, if necessary, the absolute performance numbers can be easily computed using the results of the *exact* analysis.

Table 4.2 shows the memory requirement in MB, precision as *NoAlias* percentage and

Benchmark	Memory (MB)	Precision (*NoAlias* %)	Analysis time (s)
gcc	2859	89.4	329.5
perlbmk	2133	93.6	143.4
vortex	1857	92.5	91.3
eon	1276	96.8	93.5
parser	478	98.0	35.4
gap	457	97.5	128.5
vpr	735	94.2	29.5
crafty	672	97.6	29.3
mesa	894	99.4	89.4
ammp	427	99.2	34.2
twolf	624	99.3	41.5
gzip	514	90.9	25.2
bzip2	633	88.0	23.3
mcf	403	94.5	22.4
equake	546	97.7	24.3
art	597	88.6	26.5
httpd	791	93.2	224.5
sendmail	914	90.4	172.7
ghostscript	1958	87.8	4384.2
gdb	2194	86.4	9338.2
wine-server	774	91.4	201.3
average	1035	93.6	737.5

Table 4.2: Performance of *exact* analysis.

analysis time in seconds for the *exact* analysis. On an average, each benchmark requires 1 GB of memory and over 12 minutes for its points-to analysis. 93.6% of the pointer pairs in the LLVM intermediate representation do not alias with each other and the remaining are indicated to be *may*-aliases.

Considering the benchmark characteristics in Table 4.1, we observe that roughly, larger benchmarks require more computation time and memory. However, program size alone, either as KLOC or the number of constraints, is not a good indicator of computational complexity of its points-to analysis. In general, the performance of points-to analysis also depends heavily on the underlying structure of the program and is affected by various subjective factors like complexity of the data structures manipulated and how much skewness is involved while accessing data, etc.

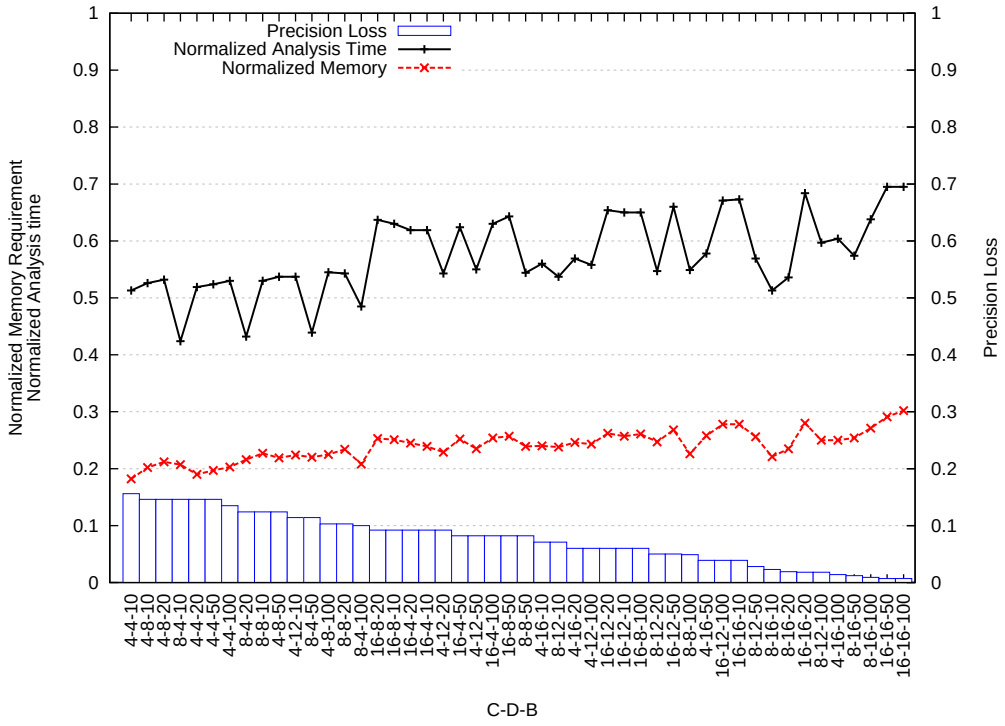


Figure 4.6: Overall effect of various configurations.

4.5.2 Performance of Multibloom: Overall Effect

We experimented with several combinations of parameters C, D, B: $C = (4, 8, 16)$, $D = (4, 8, 12, 16)$ and $B = (10, 20, 50, 100)$. Figure 4.6 shows the overall effect of all these configurations on precision, analysis time and memory requirement. X-axis indicates a configuration C-D-B. Y-axis represents memory requirement, precision loss and analysis time of a configuration, all normalized with respect to those of an *exact* analysis. Thus, the memory requirement and the analysis time of *exact* analysis are 100 each and its precision loss is 0. The configurations along X-axis are sorted based on their precision loss. For all the three quantities (memory, analysis time and precision loss), smaller values are better.

We observe that all the configurations we experimented with have normalized analysis time less than 70% of that of *exact*, with memory requirement ranging from 18% to 30%. The precision loss for these configurations varies from 0.7% to 15%. There are several configurations for which the precision loss is less than 5% with improvements both in terms of analysis time and memory requirement. For instance, the configuration 8-12-100 has a precision loss of 1.8% with normalized analysis time of 60% and normalized memory requirement of 25% compared

to *exact*.

These results show that, overall, multibloom offers great benefits in terms of analysis time and especially in terms of memory requirement trading off minimal amount of precision. Below, we study the effect of individual parameters on the performance of multibloom.

4.5.3 Effect of Parameter C

First we evaluate the sensitivity of our multibloom approach to variation in the number of context bins $C = 4$, 8 and 16. In this experiment, we keep the number of hash functions (D) fixed at 16, the bit-vector size (B) at 100 and the number of entries for two level pointers (S) at 5. The results are shown in Table 4.3. We observe that parameter C has only a marginal impact on normalized precision which improves from 98.6% to 99.3% with increasing C. The average analysis time varies from 60.4% to 69.5% of exact Andersen's method. There is also a proportional change in the memory requirement with a change of C.

From the performance numbers, we would like to conclude that C should only be increased when it is justified in terms of precision. This is because increasing C is going to result in a corresponding increase in the time and memory requirements. Thus, choosing a good value of C requires a trade-off between time/memory requirements and precision.

4.5.4 Effect of Parameter D

To estimate the effect of the number of hash functions, we kept other parameters fixed at $S = 5$, $C = 8$ and $B = 100$ varying $D = 4, 8, 12, 16$. The results are shown in Table 4.4. Amongst all the configuration parameters, D has the highest impact on performance. The precision varies from 90.0% to 99.1% with increasing D. The analysis time also increases from 48.5% to 63.8% of *exact*. The normalized memory requirement also varies in proportion with D ranging from 20.8% to 27.1%.

Choosing an appropriate value of D requires a trade-off between memory requirement and precision. Increasing D from 4 to 16 improves precision by 9.1%, but increases the normalized memory requirement and normalized analysis time by 6.3% and 15.3% respectively. Therefore, a judicious selection of D can help achieve a good balance between memory requirement and precision. In general, this balance is guided by the requirements of the client.

Benchmark	Normalized memory			Normalized precision			Normalized analysis time		
	C=4	C=8	C=16	C=4	C=8	C=16	C=4	C=8	C=16
gcc	37.8	42.0	48.1	99.2	99.3	99.6	49.0	49.4	50.0
perlbmk	22.6	23.5	27.0	98.2	99.6	100.0	70.0	70.8	71.9
vortex	11.2	12.4	14.8	98.6	99.5	99.8	95.7	97.2	99.5
eon	28.4	32.4	35.9	99.2	99.8	99.9	95.7	98.0	101.3
parser	26.4	31.2	36.8	99.6	99.9	100.0	78.4	86.9	104.5
gap	63.0	65.9	73.5	99.0	99.4	99.5	67.6	77.6	96.7
vpr	13.6	15.2	18.4	98.8	99.2	99.4	65.8	69.9	75.8
crafty	13.8	14.3	15.6	99.1	99.6	99.8	65.0	73.3	85.3
mesa	23.8	24.9	27.7	99.1	99.4	99.5	85.2	88.7	96.7
ammp	22.7	24.1	30.7	98.8	99.2	99.4	76.9	80.0	88.4
twolf	21.0	24.5	30.1	99.0	99.3	99.4	89.1	92.8	103.7
gzip	12.5	13.8	15.4	98.4	98.9	99.1	94.0	99.5	116.0
bzip2	10.0	10.7	11.2	97.7	98.0	98.2	93.2	100.3	114.7
mcf	15.6	16.9	17.4	99.3	99.5	99.5	91.7	95.6	105.1
equake	11.4	12.5	12.8	99.6	99.7	99.8	77.4	85.2	92.9
art	10.4	10.9	11.7	99.4	99.6	99.6	65.9	68.8	74.5
httpd	90.0	93.4	104.2	98.0	98.2	98.5	19.1	20.4	21.9
sendmail	45.3	48.4	54.3	98.0	98.4	98.8	11.6	11.8	12.1
ghostscript	66.2	67.5	70.7	97.6	98.1	98.5	58.7	60.1	61.6
gdb	82.7	88.0	93.1	96.8	97.4	97.8	31.6	31.8	32.1
wine-server	44.8	49.7	53.5	98.1	98.8	99.0	35.7	41.1	45.6
average	25.0	27.1	30.2	98.6	99.1	99.3	60.4	63.8	69.5

Table 4.3: Sensitivity to parameter C

4.5.5 Effect of Parameter B

Table 4.5 shows the effect of parameter B on the analysis performance. We fixed $C = 8$, $D = 16$ and $S = 5$ varying $B = 10, 20, 50, 100$. We found that, similar to C, B has only a marginal effect on performance. On an average the normalized memory requirement increases from 22.1% to 27.1% with only a small increase in normalized *NoAlias* percentage from 97.7% to 99.1%. There is also a corresponding increase in analysis time from 51.3% to 63.8% seconds.

As in the case of parameter C, we would like to conclude that parameter B should be increased only when precision is of prime importance over memory.

4.5.6 Effect of Parameter S

We evaluated the sensitivity of the analysis to parameter S. We kept other parameters fixed at $C = 8$, $D = 16$ and $B = 100$ and varied $S = 1, 2, 3, 4, 5$. The results are given in Table 4.6. We observe a small increase in the memory requirement with the increasing value of S from 25.5%

Benchmark	Normalized memory				Normalized precision				Normalized analysis time			
	D=4	D=8	D=12	D=16	D=4	D=8	D=12	D=16	D=4	D=8	D=12	D=16
gcc	33.4	34.5	38.0	42.0	84.4	92.2	97.2	99.3	46.6	47.3	47.9	49.4
perlbmk	19.5	21.0	22.5	23.5	85.1	92.8	96.9	99.6	54.7	63.3	68.2	70.8
vortex	8.5	10.0	11.0	12.4	86.1	93.7	98.4	99.5	89.6	92.5	94.7	97.2
eon	25.0	26.3	28.4	32.4	86.3	94.3	99.1	99.8	85.8	91.8	94.8	98.0
parser	20.1	22.2	26.6	31.2	90.3	95.3	99.1	99.9	57.3	63.8	75.3	86.9
gap	51.9	58.9	63.2	65.9	92.2	96.5	99.0	99.4	54.4	62.0	70.5	77.6
vpr	10.7	11.7	13.6	15.2	86.9	95.3	98.7	99.2	55.3	60.4	64.5	69.9
crafty	12.1	12.6	13.7	14.3	93.4	98.3	99.5	99.6	49.4	59.0	63.4	73.3
mesa	19.7	21.7	23.6	24.9	93.6	97.9	99.4	99.4	77.2	83.3	86.0	88.7
ammp	17.6	19.9	23.0	24.1	95.7	98.6	99.2	99.2	55.2	64.8	73.3	80.0
twolf	15.1	17.8	21.2	24.5	96.3	98.7	99.2	99.3	72.8	81.7	87.2	92.8
gzip	10.9	11.7	12.6	13.8	96.5	98.2	98.9	98.9	77.3	88.4	93.9	99.5
bzip2	8.7	9.3	10.1	10.7	97.0	97.6	98.0	98.0	85.8	93.0	100.3	100.3
mcf	13.9	14.6	16.1	16.9	98.6	99.2	99.4	99.5	68.8	78.6	92.0	95.6
equake	10.1	10.6	11.7	12.5	96.6	97.6	99.7	99.7	62.1	69.5	77.3	85.2
art	9.0	9.7	10.4	10.9	99.2	99.5	99.6	99.6	45.7	57.4	63.1	68.8
httpd	83.3	86.0	90.3	93.4	87.6	93.1	97.5	98.2	12.9	16.1	18.7	20.4
sendmail	40.3	43.2	45.3	48.4	84.6	92.1	97.3	98.4	6.8	9.0	10.2	11.8
ghostscript	53.7	60.2	64.5	67.5	84.1	89.2	96.0	98.1	51.9	55.2	57.7	60.1
gdb	58.7	64.5	77.1	88.0	79.8	87.8	94.5	97.4	30.8	31.3	31.6	31.8
wine-server	39.3	42.2	46.1	49.7	80.4	90.4	95.0	98.8	27.2	33.4	37.4	41.1
average	20.8	22.6	25.0	27.1	90.0	95.1	98.2	99.1	48.5	54.9	59.7	63.8

Table 4.4: Sensitivity to parameter D

Benchmark	Normalized memory				Normalized precision				Normalized analysis time			
	B=10	B=20	B=50	B=100	B=10	B=20	B=50	B=100	B=10	B=20	B=50	B=100
gcc	33.3	36.1	40.3	42.0	97.9	98.4	99.0	99.3	47.9	48.1	48.4	49.4
perlbmk	19.5	21.0	22.6	23.5	98.0	98.5	99.1	99.6	66.9	67.6	69.7	70.8
vortex	9.4	10.5	11.3	12.4	98.2	98.7	99.2	99.5	90.7	91.9	94.0	97.2
eon	27.2	29.4	31.1	32.4	98.7	99.0	99.4	99.8	91.3	93.8	95.6	98.0
parser	23.8	25.7	28.9	31.2	98.9	99.1	99.5	99.9	60.9	65.7	72.2	86.9
gap	53.2	58.0	62.4	65.9	98.7	99.0	99.2	99.4	60.5	63.9	69.5	77.6
vpr	11.7	12.8	13.7	15.2	98.4	98.9	99.0	99.2	52.6	56.4	60.8	69.9
crafty	10.6	11.5	12.8	14.3	98.7	99.1	99.3	99.6	47.7	51.8	60.0	73.3
mesa	20.7	22.4	23.6	24.9	98.7	99.0	99.2	99.4	65.7	70.0	77.2	88.7
ammp	20.6	21.5	22.7	24.1	90.5	90.8	99.1	99.2	57.0	60.2	66.3	80.0
twolf	17.1	18.3	21.8	24.5	98.8	99.0	99.3	99.3	78.3	81.9	85.5	92.8
gzip	11.7	12.1	13.0	13.8	98.4	98.6	98.7	98.9	71.7	77.3	88.4	99.5
bzip2	9.2	9.5	10.1	10.7	97.4	97.8	97.9	98.0	86.2	86.2	93.0	100.3
mcf	14.4	15.1	16.1	16.9	99.1	99.2	99.4	99.5	76.4	78.1	83.9	95.6
equake	10.8	11.4	11.7	12.5	99.4	99.5	99.7	99.7	62.1	69.5	77.3	85.2
art	9.4	9.7	10.4	10.9	99.2	99.4	99.6	99.6	54.4	57.4	60.1	68.8
httpd	84.6	87.6	90.1	93.4	96.8	97.3	97.9	98.2	16.4	17.6	18.4	20.4
sendmail	42.2	44.5	46.6	48.4	96.9	97.3	97.7	98.4	9.7	10.0	10.9	11.8
ghostscript	60.6	62.1	65.8	67.5	96.8	97.0	97.6	98.1	55.5	55.9	57.5	60.1
gdb	77.3	79.7	83.6	88.0	95.9	96.4	96.9	97.4	29.2	29.3	30.0	31.8
wine-server	34.8	36.8	43.4	49.7	97.5	97.9	98.2	98.8	27.4	29.1	32.2	41.1
average	22.1	23.5	25.4	27.1	97.7	98.1	98.8	99.1	51.3	53.6	57.4	63.8

Table 4.5: Sensitivity to parameter B

to 27.1%. There is a corresponding increase in the time requirement from 60.6% to 63.8%. However, we did not find much difference in the precision (less than 1% change). Therefore, we conclude that S only marginally affects the analysis precision. This is in line with our observation that programs typically have very few number of pointers at higher dereference levels.

4.5.7 Effect of Selected Configurations

In this section, we study the effect of a few carefully chosen configurations on the analysis performance. Specifically, we chose four configurations: *tiny* with C-D-B=4-4-10, *small* with C-D-B=8-8-10, *medium* with C-D-B=8-12-50 and *large* with C-D-B=8-16-100. For all the above configurations S=5. These configurations range over the spectrum of configurations and allow us to draw certain conclusions with respect to the use of a particular configuration for different applications. In addition to the overall effect studied in Section 4.5.2, we also study the effect on individual benchmarks.

The performance numbers for different configurations are reported in Table 4.7. The *tiny* configuration achieves a significant reduction in memory requirement and analysis time. Specifically, its memory requirement is only 18.2% and analysis time is 51.3%. However, it has a somewhat higher precision loss of 15.6%. The *tiny* configuration can be well suited for clients that have stringent memory and analysis time requirements but can tolerate some amount of precision loss.

At the other extreme, the *large* configuration achieves almost full precision, still yielding 73% improvement in the memory requirement and 36.2% improvement in the analysis time. With an exception of *bzip2*[3] where the *large* configuration of multibloom performs slightly worse, the configuration requires lesser time on all the benchmarks compared to the *exact* analysis. This configuration is greatly suited for clients that require high precision, at lower analysis time and memory requirement.

The *small* and the *medium* configurations prove to be excellent trade-off points. Their memory, analysis time and precision lie between those of *tiny* and *large* configurations. A client or a meta-analyzer may select a particular configuration to suit its needs.

[3]The absolute analysis time of *bzip2* for the *exact* method is only 23.3 seconds.

Benchmark	Normalized memory					Normalized precision					Normalized analysis time				
	S=1	S=2	S=3	S=4	S=5	S=1	S=2	S=3	S=4	S=5	S=1	S=2	S=3	S=4	S=5
gcc	33.5	34.0	34.5	38.2	42.0	98.0	98.4	98.7	99.1	99.3	49.1	49.2	49.3	49.4	49.4
perlbmk	22.8	23.0	23.2	23.4	23.5	98.2	98.4	99.1	99.4	99.6	70.4	70.6	70.7	70.8	70.8
vortex	12.0	12.1	12.2	12.3	12.4	99.0	99.1	99.3	99.4	99.5	95.7	96.2	96.6	97.2	97.2
eon	31.8	32.0	32.2	32.3	32.4	99.3	99.6	99.7	99.8	99.8	97.1	97.2	97.6	98.0	98.0
parser	26.2	27.6	28.9	30.1	31.2	98.9	99.2	99.7	99.9	99.9	83.6	84.4	85.7	86.9	86.9
gap	63.5	64.1	64.8	65.0	65.9	98.7	99.0	99.2	99.3	99.4	72.2	74.8	75.9	77.6	77.6
vpr	13.7	14.0	14.4	14.7	15.2	98.6	98.7	98.9	99.1	99.2	63.1	64.5	67.6	69.9	69.9
crafty	13.4	13.7	13.8	14.0	14.3	98.9	99.1	99.5	99.6	99.6	66.1	68.3	70.5	73.3	73.3
mesa	24.4	24.4	24.6	24.7	24.9	99.2	99.3	99.3	99.4	99.4	86.0	86.9	87.8	89.0	88.7
ammp	23.0	23.2	23.4	23.7	24.1	99.1	99.1	99.2	99.2	99.2	78.0	79.1	79.6	80.0	80.1
twolf	24.0	24.2	24.2	24.4	24.5	99.1	99.2	99.3	99.3	99.3	89.0	90.8	90.8	93.3	92.7
gzip	13.6	13.8	13.8	13.8	13.8	98.6	98.7	98.8	98.9	98.9	93.9	99.4	99.4	99.9	99.4
bzip2	10.3	10.4	10.4	10.6	10.7	97.7	97.9	98.0	98.0	98.0	93.0	100.2	100.2	96.5	100.2
mcf	16.6	16.6	16.6	16.6	16.9	99.5	99.5	99.5	99.5	99.5	88.0	89.9	91.9	96.5	95.7
equake	12.1	12.3	12.3	12.3	12.5	99.6	99.6	99.7	99.7	99.7	85.2	85.2	85.2	85.2	85.2
art	10.6	10.7	10.7	10.9	10.9	99.6	99.6	99.6	99.6	99.6	63.0	65.9	68.8	68.8	68.8
httpd	91.8	92.0	92.5	93.0	93.4	97.8	97.9	98.1	98.2	98.2	18.2	18.5	19.6	20.4	20.4
sendmail	47.4	47.6	48.0	48.2	48.4	98.2	98.2	98.4	98.4	98.4	10.5	11.5	11.6	11.8	11.8
ghostscript	61.8	63.8	65.5	66.6	67.5	97.3	97.4	97.6	97.9	98.1	59.3	59.6	59.9	60.1	60.1
gdb	84.1	84.4	85.4	86.6	88.0	95.9	96.3	96.8	97.1	97.4	31.7	31.8	31.8	31.8	31.8
wine-server	42.8	44.7	46.4	48.2	49.7	95.0	96.3	96.3	97.5	98.8	37.4	39.0	40.5	41.0	41.1
average	25.5	25.9	26.2	26.6	27.1	98.4	98.6	98.8	99.0	99.1	60.6	62.0	63.0	63.7	63.8

Table 4.6: Sensitivity to parameter S

Benchmark	Normalized memory				Normalized precision				Normalized analysis time			
	4-4-10 tiny	8-8-10 small	8-12-50 medium	8-16-100 large	4-4-10 tiny	8-8-10 small	8-12-50 medium	8-16-100 large	4-4-10 tiny	8-8-10 small	8-12-50 medium	8-16-100 large
gcc	12.7	18.6	23.4	42.0	84.5	86.5	96.4	99.3	35.5	38.5	41.6	49.4
perlbmk	10.2	13.1	14.7	23.5	82.6	86.6	98.1	99.6	53.4	55.2	59.5	70.8
vortex	6.4	7.8	8.2	12.4	83.5	87.8	97.3	99.5	70.5	72.7	78.2	97.2
eon	18.4	22.6	23.4	32.4	83.2	88.2	98.2	99.8	72.8	74.6	79.2	98
parser	19.2	21.6	24.7	31.2	82.9	88.4	98.6	99.9	79.4	80.2	82.5	86.9
gap	40.3	49.2	52.6	65.9	84.7	87.5	98.1	99.4	60.6	62.5	66.2	77.6
vpr	8.4	10.7	12.6	15.2	84.2	88.2	97.7	99.2	50.2	51.6	54.6	69.9
crafty	9.1	10.5	11.5	14.3	89.3	92.3	99.6	99.6	54.1	56.9	60.1	73.3
mesa	11.7	15.7	18.2	24.9	83.9	86.7	97.0	99.4	62.9	64.9	70.7	88.7
ammp	11.9	17.2	18.9	24.1	85.7	88.7	98.7	99.2	61.5	63.2	65.7	80
twolf	12.3	17.6	19.1	24.5	85.6	89.0	98.6	99.3	68.2	71.4	75.3	92.8
gzip	8.1	8.6	10.7	13.8	87.2	90.5	98.0	98.9	63.6	65.7	73.4	99.5
bzip2	7.2	8.1	8.8	10.7	88.4	90.8	96.5	98.0	77.4	79.6	84.1	100.3
mcf	8.8	10.2	11.9	16.9	90.1	93.4	99.4	99.5	66.2	68.6	72.6	95.6
equake	8.3	8.8	10.3	12.5	89.5	92.4	99.6	99.7	47.7	49.7	56.8	85.2
art	6.5	7.5	8.2	10.9	91.1	93.1	99.6	99.6	37.8	39.5	43.5	68.8
httpd	52.1	62.6	75.3	93.4	80.2	84.3	94.5	98.2	14.5	15.3	17.3	20.4
sendmail	22.1	29.6	33.5	48.4	81.3	83.7	95.2	98.4	8.6	8.6	9.2	11.8
ghostscript	34.3	44.9	49.6	67.5	77.6	80.6	93.6	98.1	42.0	41.1	44.7	60.1
gdb	49.3	59.4	66.6	88.0	76.7	79.9	92.7	97.4	21.7	22.6	25.3	31.8
wine-server	25.2	32.5	36.1	49.7	79.2	81.4	94.2	98.8	28.6	30.7	33.8	41.1
average	18.2	22.7	25.6	27.0	84.4	87.6	97.2	99.1	51.3	53.0	56.9	63.8

Table 4.7: Effect of select configurations on performance.

Benchmark	Time(sec)			Memory(MB)		
	exact	bddlcd	bloom	exact	bddlcd	bloom
gcc	329.5	17411.2	137.1	2859	2534	669
perlbmk	143.4	5879.9	85.4	2133	1723	314
vortex	91.3	4725.7	71.4	1857	1358	152
eon	93.5	2391.8	74.0	1276	1425	299
parser	35.4	618.3	29.2	478	345	118
gap	128.5	330.2	85.1	457	362	240
vpr	29.5	199.5	16.1	735	692	93
crafty	29.3	155.0	17.6	672	566	77
mesa	89.4	21.7	63.2	894	729	163
ammp	34.2	54.6	22.5	427	336	81
twolf	41.5	27.4	31.2	624	617	119
gzip	25.2	6.5	18.5	514	522	55
bzip2	23.3	4.7	19.6	633	588	56
mcf	22.4	32.0	16.3	403	389	48
equake	24.3	4.1	13.8	546	527	56
art	26.5	7.7	11.5	597	582	49
httpd	224.5	47.4	38.8	791	825	596
sendmail	172.7	117.5	15.9	914	851	306
ghostscript	4384.2	20612.8	1959.8	1958	1672	971
gdb	9338.2	24871.7	2362.6	2194	1859	1461
wine-server	201.3	36.7	68.0	774	690	279
average	737.5	3693.2	245.6	1035	914	295

Table 4.8: Time(seconds) and memory(MB) required for context-sensitive analysis

4.6 Comparison with Other Analyses

In this section we compare our bloom filter based approach with a Binary Decision Diagram (BDD) based approach. The BDD-based points-to analyses, referred to as *bddlcd*, is due to Hardekopf and Lin [49]. We extended the original implementation, obtained from the first author's website [51], for context-sensitivity. It uses Lazy Cycle Detection (LCD) as an optimization technique for deciding how frequently cycle detection algorithm should be invoked during points-to information computation. We use *medium* configuration for the comparison (see Section 4.5.7).

Table 4.8 shows the performance of Andersen's analysis [3], referred to as *exact*, of *bddlcd* and of our approach referred to as *bloom* on our set of benchmarks. All the points-to analysis methods are context-sensitive, flow-insensitive and field-insensitive.

We observe that, on an average, *bloom* is 3× faster than *exact* and 15× faster than *bddlcd*.

Except for small programs like *art, equake*, etc., *bloom* consistently performs better than the other two methods. The large improvement in analysis time is largely due to the fast, hash-based access functions in the bloom filter. Another reason for the significant analysis time reduction is due to the improvement in the spatial locality of the points-to information stored in the bloom filter. Further, various points-to operations are implemented as fast bitwise-OR operations.

Table 4.8 also shows the memory requirement of various methods. We observe that, on an average, *bloom* requires 3.5× less memory than *exact* and 3× less memory than *bddlcd*. Further, the memory requirement of *bloom* is consistently lower than both the other methods. Note that *bddlcd* is known for its space efficiency and, despite that, with minimal precision loss, our bloom filter based analysis offers greater savings in the memory requirement.

4.7 Mod/Ref Analysis as a Client

Next we analyze how the loss in precision in the points-to analysis due to false positives introduced by our multibloom method affects the client analyses. We use the Mod/Ref analysis as the client of our multibloom-based points-to analysis. For a query $GetModRef(callsite, pointer)$, the Mod/Ref analysis checks whether $callsite$ reads or modifies the memory pointed to by $pointer$. It has four outcomes:

- *NoModRef*: *call-site* does not read from or write to memory pointed to by *pointer*.
- *Ref*: *call-site* reads from the memory pointed to by *pointer*.
- *Mod*: *call-site* writes to (but does not read from) the memory pointed to by *pointer*.
- *ModRef*: *call-site* reads from and writes to the memory pointed to by *pointer*.

$ModRef$ is the most conservative outcome, and should be returned, when it is not possible to establish otherwise for a safe analysis. Consider two *Mod/Ref* analyses MR_1 and MR_2. The *NoModRef* percentage computed by each of them for the same program is upper bounded by that of an *exact* analysis on the program. If MR_1 computes a higher *NoModRef* percentage than that computed by MR_2, then we say that MR_1 is more precise than MR_2.

Figure 4.7 shows the percentage of queries answered *NoModRef* by the analysis. From the figure, it can be seen that the *NoModRef* percentage with multibloom is 90.7% of the exact

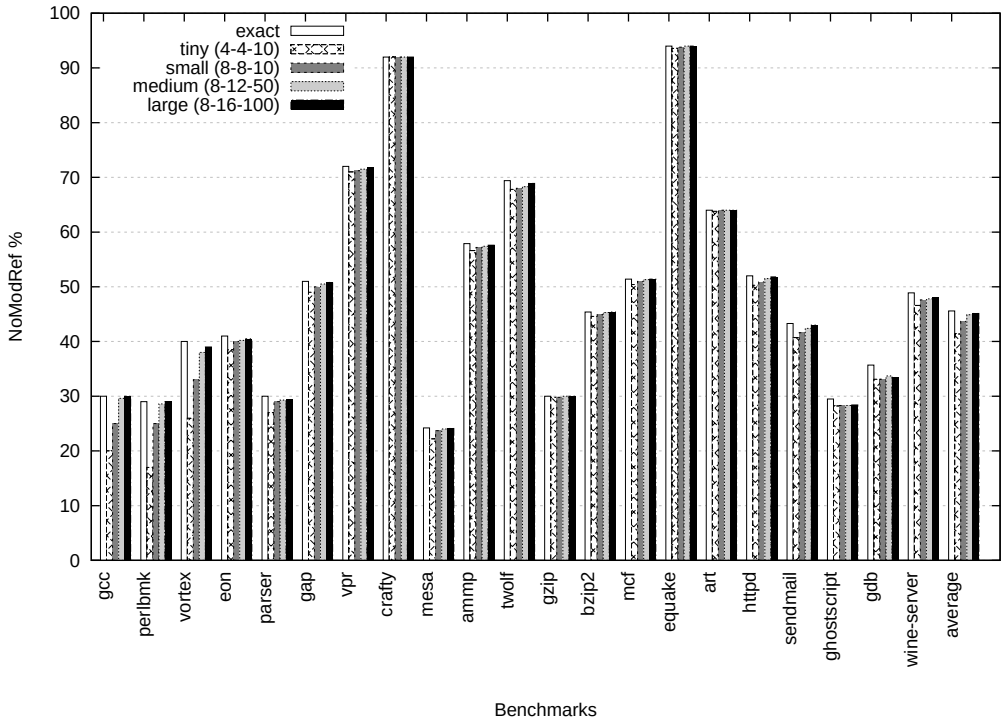


Figure 4.7: Mod/Ref client analysis.

analysis even with a *tiny* configuration. For *small* configuration, it improves further to 95.8%. With *large* configuration, the precision is upto 99%. Thus, a client can enjoy the benefits of reduced memory and analysis time incurring little precision loss. Further, even in some cases where the difference in precision was discernible when calculating *NoAlias* percentage, it is not so with a client analysis. For example, a *medium* multibloom matches the *exact* version when calculating *NoModRef* percentage for *gzip*. The *NoAlias* percentage showed a difference of 2.0% between *exact* and multibloom with *medium* configuration. Thus the loss in precision with an approximate representation may not even affect the client analysis.

4.8 Related Work

In this section we discuss the work related to the use of novel data structures for representing points-to information and the use of bloom filters in other application domains.

4.8.1 Methods using Novel Data Structures

Our points-to analysis uses a novel data structure (bloom filter) to store points-to information. Earlier approaches stored alias pairs explicitly [70]. Thus, if `p` and `q` are aliases, an earlier analysis would store the alias pair `(*p, *q)` explicitly. Storing alias pairs is storage-intensive. For instance, if `p, q, r` and `s` point to the same memory location, following is the set of alias pairs stored: {`(*p, *q), (*p, *r), (*p, *s), (*q, *r), (*q, *s), (*r, *s)`}. To reduce the storage requirement, compact representation has been proposed which stores only a few basic alias pairs explicitly and new alias pairs are derived based on dereference, transitivity and commutativity [18]. Later, a more crisp representation in the form of points-to pairs has been devised [31] which significantly reduced the storage requirement. Thus, if pointer `p, q, r` and `s` point to the same symbolic memory location `x`, following is the set of points-to pairs stored: {`(p, x), (q, x), (r, x), (s, x)`}. We store points-to information, instead of alias pairs, throughout our work.

Heintze and Tardieu [54] propose the use of sparse bitmaps for storing points-to information. Since for most programs, the points-to information is actually sparse — i.e., only a few pointers have a large number of pointees while most others have very few pointees — and accessing a sparse bitmap is very efficient, sparse bitmap is a desirable data structure for storing points-to information. It is used in GCC 4.1 [49]. However, bitmaps cannot take advantage of the commonality across various points-to sets, i.e., if the points-to sets of pointers `p` and `q` is mostly same with a few differences, a sparse bitmap would still store the duplicate points-to information for both `p` and `q`. Therefore, for a context-sensitive analysis, the use of bitmaps requires a large amount of memory.

Zhu [141] was the first one to observe that the vast amount of points-to information can be encoded in a space-efficient manner using binary decision diagrams (BDD) [8]. Until then, BDDs were used in symbolic model checking [9] and to represent large sets and maps [84]. Due to the storage efficiency, BDDs were quickly adapted for solving points-to analysis algorithms. Berndl et al. [6], Whaley and Lam [129] and Zhu and Calman [142] have proposed variants of points-to analysis algorithms using BDDs for Java. The scalability aspect of using BDDs is evident from the fact that as the number of contexts (and, in turn, the points-to information) grows, the storage requirement may come down [129].

However, the use of BDDs comes with a downside. Hardekopf and Lin [49] compared

the performance of BDD-based and bitmap-based points-to analyses. They found that a BDD-based implementation is, on an average, 2× slower than a sparse bitmap-based implementation, but uses 5.5× less memory.

Bloom filters offer the best-of-both-the-worlds: its access time is as low as that for bitmaps and its memory requirement is even below that of BDDs. Although it incurs a minimal amount of precision loss, a bloom-filter-based analysis is likely to scale well with program size both in terms of memory and analysis time.

4.8.2 Use of Bloom Filters in Other Applications

Bloom filter was also used in distributed networks to implement compact caches at proxy servers [36]. Since the routing tables stored at the proxy servers tend to get bigger and are needed to be periodically sent to other proxy servers for synchronization, a bloom-filter-based representation of the routing tables offers benefits both in terms of the storage requirement and the network bandwidth. Bloom filter was also used for distributed database joins [82]. The idea is to send a compact bloom filter based approximate representation of a table to another node in the network and perform an initial join locally on that node. Since due to the condition in the join command, the total size of this intermediate result would reduce, this intermediate join can then be sent back to the originating node. The originating node can then perform a join operation with this intermediate result to filter out some more tuples. The overall benefit is achieved by reducing the amount of data sent across the network. Gremillion [44] used bloom filters to improve the performance of differential files in a database environment. He uses bloom filters to check if two files contain the same data. Thus, if the bloom filter representations of the two files differ, the two files definitely contain different data. Manber and Wu [83] used bloom filters in checking validity of proposed passwords against previous passwords used and against dictionary words. This is done by maintaining a checksum and checking a new password's checksum against the old checksum. Similarly, a dictionary of words can be stored in an approximate manner in a bloom filter for fast hash-based lookup.

Bloom filters also underwent several advances. Mitzenmacher [89] and Fan et al. [36] proposed compressed bloom filters to enable more efficient transmission of bloom filter across servers on a network. The compression further reduces the storage requirement, but incurs a small cost in compression-decompression. However, its main usage is in reducing the network

traffic where the network latency dominates the compression-decompression time. For faster performance while using external storage, Manber and Wu [83] imposed a locality restriction on the hash functions used in a bloom filter. To support deletions, Fan et al. [36] proposed counting bloom filters. Counting bloom filters maintain a counter per data element, instead of a bit as in the case of a simple bloom filter. This counter is increased when an element which hashes to this location is added and reduced when the element is deleted. However, since the number of bits per counter is fixed, the number of additions and deletions should not exceed beyond a limit. Unless the number of bits is sufficiently large, a counting bloom filter suffers from the possibility of a false negative. To support multisets, i.e., to allow (and distinguish between) multiple occurrences of the same element in the bloom filter, Cohen and Matias [20] proposed spectral bloom filters.

We introduced multi-dimensional bloom filters, an extension of naive bloom filters [7], suited for queries in points-to analyses.

4.9 Chapter Summary

In this chapter we proposed a multi-dimensional bloom filter for storing points-to information. The proposed representation may introduce false positives, but it significantly reduces the memory requirement and provides a probabilistic lower bound on loss of precision. As our multibloom representation introduces only false positives, but no false negatives, it ensures safety for (may-)points-to analysis. We demonstrate the effectiveness of multibloom on 16 SPEC 2000 benchmarks and five real-world applications. Compared to an *exact* analysis, a multibloom configuration offers 75% reduction in memory requirement and 40% reduction in analysis time with less than 2% precision loss. We also showed that compared to a BDD-based analysis, a multibloom configuration is 15× faster and uses 3× less memory. Using Mod/Ref analysis as a client, we show that the effect of our approximate representation on the precision of this client is even less.

Unlike traditional data-structures for storing points-to information, like bitmaps and BDDs, bloom filters provide user a control on the memory requirement, yet giving a probabilistic lower bound on the precision loss. This control, coupled with huge savings in the memory requirement, makes bloom filters a promising data-structure for storing points-to information.

Chapter 5

Sound Randomized Points-to Analysis

5.1 Introduction

Precise points-to analysis is undecidable when dynamic memory allocation is allowed [103]. When dynamic memory allocation is disallowed, the problem (even context-insensitive, flow-insensitive) is still NP-Hard [59, 12]. These complexity results suggest that for a scalable points-to analysis, approximations cannot be avoided. In fact, several techniques proposed in literature employ such approximations either explicitly or implicitly. For instance, Steensgaard [123] proposed unification to design an almost-linear-time alias analysis, trading off precision for efficiency. Andersen [3] proposed an approximate points-to analysis based on inclusion of points-to information to achieve a polynomial-time algorithm. For context-sensitive analysis, Lattner et al. [75] proposed unification of contexts to achieve a scalable implementation at the cost of some precision. Our storage representation using multibloom (Chapter 4) approximately stores the points-to information using hashing.

In a similar spirit, in this chapter we propose to design a scalable algorithm using randomization, a powerful approximation technique. Despite using randomization, the analysis must preserve soundness, i.e., the points-to information computed by the analysis should be a superset of that computed by an existing sound points-to analysis. Our randomization technique works as follows. It applies a "more precise" analysis to a randomly chosen subset and a "less precise" analysis to the rest of the processing entities (e.g., points-to constraints). An important

challenge in developing randomized points-to analysis is how to summarize and compose the points-to information obtained from the less precise subset without compromising soundness. We develop such summarization and composition for various types of points-to analysis. By executing this analysis over a few runs of randomly-chosen processing entities, and applying summarization and composition, we get different results across the runs. By defining a suitable *meet* operator over the results, we get a good approximation to the most precise solution. Note that soundness is preserved in each run and if the meet operator is defined carefully, soundness is preserved in the final approximation as well.

This chapter is organized as follows. In Section 5.2 we describe our randomization technique in detail using unification-inclusion as the base. The important components of our technique, viz., selection, summarization and composition, for this analysis dimension are also explained in this section. In Section 5.3 we present our randomized context-sensitive algorithm using an example. Subsequently, in Section 5.4 we prove its soundness. The effectiveness of our approach is evaluated in detail in Section 5.5. We identify several configurations where significant improvements in analysis time can be obtained with minimal precision loss. We also study the effect of various configuration parameters on performance in this section. Finally, we conclude the chapter with a summary in Section 5.8.

5.2 Unification versus Inclusion

We explain our randomization approach using unification versus inclusion.

5.2.1 Overview of the Approach

Consider a flow-insensitive, context-insensitive and field-insensitive points-to analysis. We have two well-known methods for computing points-to information: Steensgaard's unification-based approach [123] and Andersen's inclusion-based approach [3]. Steensgaard's analysis runs in almost linear time in the number of constraints; whereas Andersen's analysis has a cubic running time complexity. However, Steensgaard's analysis is very imprecise compared to Andersen's analysis [59]. Further, since a unification constraint adds at least as much points-to information as that added by an inclusion constraint, the points-to information `U` computed by a unification-based approach is a superset of that computed by an inclusion-based analysis `I`, as depicted in

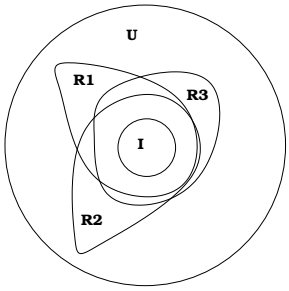


Figure 5.1: Unification versus Inclusion

Figure 5.1. Using set notation, it can be written as $\mathtt{I} \subseteq \mathtt{U}$.

Now consider a run of an analysis $\mathtt{R_1}$ that combines unification and inclusion. The proposed analysis divides the set of points-to constraints $\mathtt{C}$ into two disjoint subsets $\mathtt{C_{I_1}}$ and $\mathtt{C_{U_1}}$ such that $\mathtt{C_{I_1}} \cup \mathtt{C_{U_1}} = \mathtt{C}$ and $\mathtt{C_{I_1}} \cap \mathtt{C_{U_1}} = \phi$. A points-to constraint $\mathtt{c} \in \mathtt{C}$ is included in $\mathtt{C_{I_1}}$ with probability ρ_1 and in $\mathtt{C_{U_1}}$ with probability $\mathtt{1} - \rho_1$. The analysis processes constraints in $\mathtt{C_{I_1}}$ using inclusion and those in $\mathtt{C_{U_1}}$ using unification. Thus, the analysis processes each points-to constraint using inclusion with a probability ρ_1 and using unification with a probability $\mathtt{1} - \rho_1$. To compute a sound points-to solution, we require a way to *compose* the results of unification and inclusion which we describe in Section 5.2.2. Thus, at the end of the analysis, depending upon the probability ρ_1 of selecting a constraint, $\mathtt{R_1}$ computes a points-to information that is at least as precise as unification and at most as precise as inclusion. Therefore, using the same symbol for a run and its computed points-to information, we have $\mathtt{I} \subseteq \mathtt{R_1} \subseteq \mathtt{U}$.

If the probability ρ_1 is set to 0, then all the constraints are processed using unification and we get the same information as $\mathtt{U}$. By setting $\rho_1 = \mathtt{1}$, all the constraints are processed using inclusion and we get the same information as $\mathtt{I}$. By choosing an appropriate value (depending upon the client needs) for ρ_1, we can get an analysis with the desired precision and analysis time.

However, by doing a small amount of additional work, we can improve the analysis precision sharply. We can partition the constraint set $\mathtt{C}$ into $\mathtt{C_{I_2}}$ and $\mathtt{C_{U_2}}$ in a different manner using probability ρ_2 (ρ_2 may be same as ρ_1) and process them using inclusion and unification respectively. Thus, we get a run $\mathtt{R_2}$ that computes the points-to information in the same way as computed by run $\mathtt{R_1}$. Therefore, we get $\mathtt{I} \subseteq \mathtt{R_2} \subseteq \mathtt{U}$.

Continuing this way a few times, say $\mathtt{N}$ number of times, we obtain different-sized points-to

information $R_1 ... R_N$. Each run R_i satisfies the following set relationship

$$I \subseteq R_i \subseteq U \tag{5.1}$$

Now consider the following points-to set

$$R = \cap_{i=1}^{N} R_i \tag{5.2}$$

Thus, we have

$$R \subseteq R_i \tag{5.3}$$

Equation 5.3 implies that the run R is at least as precise as any of the runs R_i. From Equations 5.1 and 5.2, we obtain the following result.

$$R \subseteq U \tag{5.4}$$

Further, since $I \subseteq R_i$ for each i,

$$I \subseteq R = \cap_{i=1}^{N} R_i \tag{5.5}$$

From Equations 5.4 and 5.5, we have

$$I \subseteq R \subseteq U \tag{5.6}$$

In effect, by running the randomized analysis a few times, we obtain a points-to information that is likely to be more precise than any of the individual runs. By choosing appropriate values for ρ_i and N, we can get close to the precision of Andersen's inclusion-based analysis.

5.2.2 Selection, Summarization and Composition

Next, we describe three important operations, namely, selection, summarization and composition for the randomized unification-inclusion method. These operations need to be defined appropriately when we extend the randomized approach for other analysis dimensions such as flow-sensitivity, context-sensitivity and field-sensitivity. In case of unification-inclusion, the selection of program entity involves choosing points-to constraints; whereas for context-sensitivity, we select it to be a program function. Each constraint in the above analysis need not be processed separately. In fact, for optimization purposes, all the constraints chosen for unification

are grouped to form an aggregate `A`. The aggregate information computed for this group (of constraints) is later reused while processing the other set of constraints using inclusion.

After selection, the aggregate must compute a summary information corresponding to the precision level of the analysis dimension. In case of unification-inclusion, summarization step turns out to be an identity function. However, as we will see in the next section, summarization is an essential aspect for other analysis dimensions.

In defining a run in the previous section, we omitted an important detail. We mentioned that some constraints would be processed using unification and the others using inclusion. The two processing sequences need to be *composed* to generate a single sound points-to information. The composition involves processing the points-to constraints using unification and then operate on the remaining constraints using inclusion, starting with the points-to information computed using unification. The summarization and composition turn out to be simple for this analysis dimension. As we will see in Section 5.3, it could be complex in case of context-sensitivity. It should be noted that composing the two partial results is a necessary requirement for soundness. If the results of the less precise analysis and the more precise analysis are not composed, then the points-to analysis may not be able to compute certain points-to information, resulting in an unsound analysis.

5.2.3 Implementation Challenges

In addition to the above composition rule to ensure correctness, the approach also poses a certain implementation non-triviality. Unification would typically be implemented using a union-find data structure whereas inclusion may use bitmaps or binary decision diagrams (BDD) to store the points-to information. Thus, we need to convert the points-to information from one representation to another; more specifically, from union-find to bitmaps/BDD. One way to avoid the conversion and still achieve the effect of unification is to process each points-to constraint in `A` bi-directionally using inclusion, i.e., a constraint `a = b` will be processed using inclusion as `a = b` and `b = a`. However, this only adds to the analysis complexity and we lose the benefit of using a union-find data structure. Hence, we implement the conversion by keeping track of the leaf nodes (nodes that are not parents of any other node) in a union-find data structure and by traversing each alias group, i.e., a connected component in the union-find data structure, upwards (from a node to its parent) to convert the alias information to the points-to

Algorithm 8 Randomized Points-to Analysis using Unification and Inclusion

Require: program `P` as a list of points-to constraints
Ensure: each variable in `P` has a set of variables indicating its points-to set

```
1:  for run = 1..N do
2:    choose ρ^ui_run
3:    A = {} {A contains constraints to be processed using unification}
4:    for all points-to constraints e ∈ P do
5:      e.inclusionflag = getRandom(ρ^ui_run)
6:      if e.inclusionflag is reset then
7:        A = A ∪ {e}
8:      end if
9:    end for
10:   A_summary = summarize(A)
11:   repeat
12:     for all points-to constraints e ∈ P do
13:       if e ∈ A then
14:         union-find = process_U(A_summary)
15:         R_A = convert(union-find)
16:         R_run = R_run ∪ R_A
17:       else
18:         R_run = R_run ∪ process_I(e)
19:       end if
20:     end for
21:   until fixed point
22: end for
23: R = ∩_run R_run
24: return R
```

information.

5.2.4 The Algorithm

The complete randomized analysis for unification-inclusion is given in Algorithm 8. The `for` loop at Line 1 is executed `N` times for `N` runs of the analysis. The selection step identifies a points-to constraint as the program entity to be processed. The `for` loop at Line 4 selects constraints to be processed using unification with probability $\rho^{\texttt{ui}}_{\texttt{run}}$ where `run` $\in$ `1..N` and `ui` denotes the analysis dimension as unification-inclusion. The function `summarize()` at Line 10 creates a summary of the selected points-to constraints. For unification-inclusion, it can either be an identity function or can represent the constraints in an optimized manner in order to process them faster. The `repeat-until` loop from Lines 11–21 iteratively computes the points-to information until a fixed-point. In every iteration, depending upon the selection,

Algorithm 9 Randomized Context-sensitive Points-to Analysis

Require: program `P` as a list of functions to be processed

```
 1: for run = 1..N do
 2:   A = select(P)
 3:   A_summary = summarize(A)
 4:   repeat
 5:     for all functions e ∈ P do
 6:       if e ∈ A then
 7:         R_run = R_run ∪ compose(process(A_summary))
 8:       else
 9:         R_run = R_run ∪ process(e)
10:       end if
11:     end for
12:   until fixed-point
13: end for
14: R = ∩R_run, ∀run
15: return R
```

the constraint is processed using inclusion (Line 18) by function $\texttt{process}_{\texttt{I}}()$ or the summary information is processed using unification (Line 14) by function $\texttt{process}_{\texttt{U}}()$. The function `convert()` implements the composition rule to convert the points-to information in union-find data structure to a bitmap as explained in the previous subsection. Finally, at Line 23, the points-to information from different runs is intersected to get a more precise points-to information `R`.

5.3 Randomized Context-sensitivity

In this section, we illustrate how the randomization technique can be applied to context-sensitivity. The trade-off between precision and analysis time (and memory) gives a solid use-case for randomizing the context-sensitive analysis. Unlike unification-inclusion where the processing entity was a points-to constraint, in this case, the processing entity would be a function signature (or a declaration). The randomized context-sensitive analysis, based on selection, summarization and composition, is given in Algorithm 9. The functions `select()`, `summarize()` and `compose()` are given in Algorithm 10.

The selection step selects a set of functions to be processed in a context-insensitive manner and groups them into a set. The summarization step adds copy edges from actual arguments of the callers to the corresponding formal arguments of all the selected functions. It also adds

Algorithm 10 Selection, Summarization and Composition for Randomized Context-sensitivity

```
function select(P) {
for all functions e ∈ P do
  if e.csflag is reset then
    A = A ∪ {e}
  end if
end for
return A
}
function summarize(A) {
for all functions e ∈ A do
  for all callers c of e do
    for all actual-formal argument pairs (aa, fa) do
      add copy edge from aa to fa
    end for
    for all return value pairs (ar, fr) do
      add copy edge from fr to ar
    end for
  end for
end for
return A
}
function compose(R) {
for all functions e ∈ A do
  R_e = points-to information of e out of R
  for all points-to facts r ∈ R_e do
    r ⊢ r_e
  end for
end for
return R
}
```

copy edges from the formal return values of the callee to the actual return values of the caller. Note that this merges information across callers and makes the analysis context-insensitive. The composition simply tags (represented as $\vdash$) all the points-to information computed in each of the selected functions `e` with a context information of "`e`" alone. This means the calling context is ignored and only the current context "`e`" is available to the callees of `e`. The context-sensitive analysis may make use of any mechanism to process functions (summary based [35] or cloning based [129, 75] etc.).

Summarization has the effect of merging various nodes in an invocation graph. The merged nodes correspond to the functions processed in a context-insensitive manner. Merging reduces

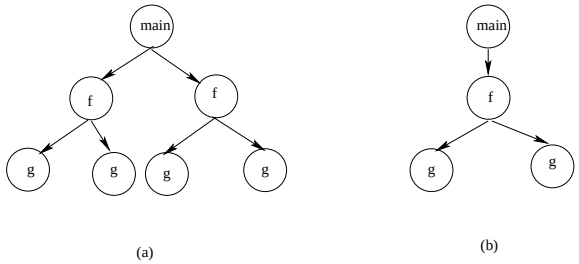


Figure 5.2: Context-sensitivity (a) Original invocation graph (b) Modified invocation graph with function `f` summarized

```
main() {
 1: f(&x, &y);
 2: f(&z, &w);
 3:
}

f(type *a, type *b) {
 4: if (...) {
 5:    a.f = b;
 6:    a = &x;
    } else {
 7:    p = &a;
 8:    c.f2 = *p;
    }
 9: g(&z);
10: g(&y);
}

g(type *d) {

}
```

Figure 5.3: Example to illustrate randomized context-sensitivity

the invocation graph size and thus the processing complexity. Figure 5.2 pictorially shows the effect of summarization on an invocation graph.

Example 5.1. Consider the program fragment shown in Figure 5.3. The program's `main` function calls function `f` twice with different parameters. The function `f` has a branch using `if-else` construct followed by two calls to function `g` with different parameters. The function `g` is empty.

The points-to information computed by a context-insensitive (also, flow-insensitive and field-insensitive) analysis for the above program is:

$\mathtt{R_{CI}} = [\mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{z}, \mathtt{y}, \mathtt{w}\}, \mathtt{b} \rightarrow \{\mathtt{y}, \mathtt{w}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{z}, \mathtt{y}, \mathtt{w}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}, \mathtt{d} \rightarrow \{\mathtt{y}, \mathtt{z}\}]$

The points-to information computed by a context-sensitive (but, flow-insensitive and field-insensitive) analysis is as below.

$\mathtt{R_{CS}} = [$

$\mathtt{along}\ 1-9 : \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{d} \rightarrow \{\mathtt{z}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}$

$\mathtt{along}\ 2-9 : \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{z}, \mathtt{w}\}, \mathtt{b} \rightarrow \{\mathtt{w}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{z}, \mathtt{w}\}, \mathtt{d} \rightarrow \{\mathtt{z}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}$

$\mathtt{along}\ 1-10 : \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{d} \rightarrow \{\mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}$

$\texttt{along } 2-10 : \texttt{a} \rightarrow \{\texttt{x}, \texttt{z}, \texttt{w}\}, \texttt{b} \rightarrow \{\texttt{w}\}, \texttt{c} \rightarrow \{\texttt{x}, \texttt{z}, \texttt{w}\}, \texttt{d} \rightarrow \{\texttt{y}\}, \texttt{p} \rightarrow \{\texttt{a}\}$

]

Let function `f` be selected by the randomized analysis to be processed in a context-insensitive manner. Figure 5.2 pictorially shows the effect of summarization on the invocation graph of this example program. Note that compared to four contexts in a context-sensitive analysis (number of paths in the invocation graph of Figure 5.2(a)), our randomized context-sensitive analysis contains only two contexts (number of paths in the invocation graph of Figure 5.2(b)). The merging of contexts invokes merging of the points-to information across the calls at labels 1 and 2. Thus, the pointees `x` and `z` are added to the points-to set of argument `a` of function `f`. Similarly, the pointees `y` and `w` are added to the points-to set of argument `b` of function `f`. However, since function `g` is processed in a context-sensitive manner, the argument `d` continues to have different points-to information across the two calls to itself from function `f`. After applying Algorithm 9 specialized with Algorithm 10, we get

$\texttt{R}_{\texttt{randomCS}} = [$

$\texttt{along } 1-9, 2-9 : \texttt{a} \rightarrow \{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}\}, \texttt{b} \rightarrow \{\texttt{y}, \texttt{w}\}, \texttt{c} \rightarrow \{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}\}, \texttt{d} \rightarrow \{\texttt{z}\}, \texttt{p} \rightarrow \{\texttt{a}\}$

$\texttt{along } 1-10, 2-10 : \texttt{a} \rightarrow \{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}\}, \texttt{b} \rightarrow \{\texttt{y}, \texttt{w}\}, \texttt{c} \rightarrow \{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}\}, \texttt{d} \rightarrow \{\texttt{y}\}, \texttt{p} \rightarrow \{\texttt{a}\}$

]

Note that $\texttt{R}_{\texttt{CS}} \subseteq \texttt{R}_{\texttt{randomCS}} \subseteq \texttt{R}_{\texttt{CI}}$. Thus, the precision of the randomized context-sensitive points-to analysis is between those of the deterministic context-insensitive and deterministic context-sensitive analyses.

5.4 Soundness

In this section we prove that our randomized context-sensitive analysis is sound. For proving soundness, it is sufficient to prove that it computes points-to information which is a superset of that computed by Andersen's analysis (extended for context-sensitivity). In other words, we need to prove that if $\texttt{R}_{\texttt{CS}}$ is the points-to information computed by context-sensitive Andersen's analysis `CS`, $\texttt{R}_{\texttt{randomCS}}$ is the points-to information computed by randomized context-sensitive analysis `randomCS`, and `r` is a points-to fact, then $\texttt{r} \in \texttt{R}_{\texttt{CS}} \implies \texttt{r} \in \texttt{R}_{\texttt{randomCS}}$.

The proof is based on the soundness of the summarization and the composition steps which we prove next. Summarization is sound if the points-to information computed by summarizing

a set of functions in the input program is a superset of that computed by a context-sensitive analysis of any context-length. Composition is sound if composing points-to information results in over-approximating it, i.e., the composed points-to information is a superset of that computed by a context-sensitive analysis of any context-length.

Lemma 5.1. *The function* `summarize()` *is sound.*

Proof. We assume that `CS` is `k`-context-sensitive, i.e., it uses the last `k` callers in the call-chain of a function `f` to determine `f`'s context, and $\texttt{k} \geq \texttt{0}$. We make use of the fact that for two values $\texttt{k}'$ and $\texttt{k}''$ of `k`, the following holds.

$$\texttt{k}' \geq \texttt{k}'' \implies \texttt{R}_{\texttt{k}'} \subseteq \texttt{R}_{\texttt{k}''} \tag{5.7}$$

Thus, as the value of `k` reduces, the context-sensitive points-to information computed goes on increasing. This is because by reducing the context-length, the analysis is able to differentiate between a smaller number of contexts, which results in the merging of context information beyond `k` callers of a function. This merging is similar to the one described in Figure 5.2. As a special case, for $\texttt{k} = \texttt{0}$, the information from all the contexts for a function gets merged and is essentially the one computed by an interprocedural context-insensitive points-to analysis. Thus,

$$\texttt{R}_{\texttt{0}} = \texttt{R}_{\texttt{CI}} \tag{5.8}$$

The function `summarize()` in Algorithm 10 adds copy edges between the formal and actual arguments and return values for *all* the callers of a function. This merges the points-to information across all the callers without distinguishing between contexts. Therefore, the points-to information computed by summarization (for the selected set of functions) is the same as that computed by a context-insensitive analysis for the set of functions.

$$\texttt{R}_{\texttt{summarize}} = \texttt{R}_{\texttt{CI}} \tag{5.9}$$

From Equations 5.8 and 5.9, we get

$$\texttt{R}_{\texttt{summarize}} = \texttt{R}_{\texttt{0}} \tag{5.10}$$

From Equations 5.7 and 5.10, we get

$$\mathtt{R}_{\mathtt{k'}} \subseteq \mathtt{R}_{\mathtt{summarize}} \tag{5.11}$$

In other words, the points-to information computed by summarizing the select set of functions is always a superset of that computed by a context-sensitive analysis of any context-length. This proves the claim. □

Lemma 5.2. *The function `compose()` is sound.*

Proof. The function `compose()` in Algorithm 10 simply specializes each of the context-insensitive points-to facts with a zero-length context, i.e., it simply instantiates the context information for a points-to fact to contain a zero-length context. From Equations 5.8, since the context-insensitive points-to information is the same as the context-sensitive points-to information with a zero-length context, the specialization operation does not result in a loss of information. Therefore, composition is sound. □

While Lemmas 5.1 and 5.2 show that the soundness is preserved for the function calls, the soundness of the points-to information computation and propagation is still not proved. We prove that next.

Theorem 5.3. *Algorithm 9 computes a sound points-to information.*

Proof. We prove that for any points-to fact `r`, $\mathtt{r} \in \mathtt{R}_{\mathtt{CS}} \implies \mathtt{r} \in \mathtt{R}_{\mathtt{randomCS}}$. For the sake of contradiction, assume that $\mathtt{r} \in \mathtt{R}_{\mathtt{CS}}$ and $\mathtt{r} \notin \mathtt{R}_{\mathtt{randomCS}}$, i.e., the points-to fact is computed by Andersen's analysis but is not computed by the randomized context-sensitive analysis. Since summarization and composition do not result in any loss of information (from Lemmas 5.1 and 5.2), the only possibility is that `r` is not computed by the function `process()` in Algorithm 9. However, the implementation of the `process()` function remains the same in `CS` and in `randomCS`. This means `r` does not computed by `process()` due to another points-to fact `r′`, on which the computation of `r` depends, but `r′` is missing from the computed points-to information $\mathtt{R}_{\mathtt{randomCS}}$. Using the same argument for `r′` as that for `r`, let `r″` be the points-to fact on which `r′` depends and which is missing from $\mathtt{R}_{\mathtt{randomCS}}$. Continuing the argument, let $\overline{\mathtt{r}}$ be the first points-to fact that does not depend upon any other points-to fact for its

computation, but is missing from the computed points-to information $\mathtt{R}_{\mathtt{randomCS}}$. The only non-dependent facts in points-to analysis are those computed by address-of constraints ($\mathtt{p} = \&\mathtt{q}$). However, whether present in aggregate A or not, i.e., whether processed in a context-insensitive manner or not, each address-of constraint computes the same points-to information. Thus, $\mathtt{R}^{\mathtt{addressof}}_{\mathtt{CS}} = \mathtt{R}^{\mathtt{addressof}}_{\mathtt{randomCS}}$.

Since, all the points-to facts $\overline{\mathtt{r}}$ are computed in the randomized analysis, the only other way a fact $\mathtt{r}, \mathtt{r}\prime$ or $\mathtt{r}\prime\prime$ is not computed by randomCS is because the program functions, the processing of which generates $\overline{\mathtt{r}}$ and $\mathtt{r}$, say, $\mathtt{f}_{\overline{\mathtt{r}}}$ and $\mathtt{f}_{\mathtt{r}}$, are in different partitions, and the information is not getting propagated from one partition to another. Recall that the functions in one partition, referred to as A, are processed in a context-insensitive manner, while those in the other partition are processed in a context-sensitive manner. Based on this, we have two cases.

Case 1: $\mathtt{f}_{\overline{\mathtt{r}}} \in \mathtt{A}$ and $\mathtt{f}_{\mathtt{r}} \notin \mathtt{A}$.

Since $\mathtt{f}_{\overline{\mathtt{r}}} \in \mathtt{A}$, it gets processed in a context-insensitive manner. From Lemma 5.1, $\mathtt{R}_{\mathtt{f}_{\overline{\mathtt{r}}}} \supseteq \mathtt{R}_{\mathtt{f}_{\mathtt{CS}}}$, i.e., processing the function in a context-insensitive manner, using function process() in Line 7 of Algorithm 9 generates at least the information as computed by processing the function in a context-sensitive manner. Therefore, the only way the points-to fact $\overline{\mathtt{r}}$ does not get propagated to the other partition (via $\mathtt{R}_{\mathtt{run}}$) is if the function compose() drops it. However, from Lemma 5.2, the composition does not lose any information. Thus, $\overline{\mathtt{r}}$ gets added to $\mathtt{R}_{\mathtt{run}}$ on Line 7 of Algorithm 9, which is eventually (in a later iteration) available to the other partition in Line 9. The function process() on Line 9 will then compute the dependent points-to fact r.

Case 2: $\mathtt{f}_{\overline{\mathtt{r}}} \notin \mathtt{A}$ and $\mathtt{f}_{\mathtt{r}} \in \mathtt{A}$.

Since $\mathtt{f}_{\overline{\mathtt{r}}} \notin \mathtt{A}$, the function gets processed in a context-sensitive manner similar to CS. Once the fact $\overline{\mathtt{r}}$ is computed, it is added to $\mathtt{R}_{\mathtt{run}}$ in Line 9 of Algorithm 9. Eventually, in a later iteration, the points-to fact is made available to the other partition A. Since the points-to fact $\overline{\mathtt{r}}$ is present, its dependent points-to facts are computed by function process() on Line 7. By Lemma 5.1 and 5.2, the number of points-to facts computed by randomCS may be more than that computed by CS. However, all the facts r computed by CS are definitely computed by randomCS. Therefore, the information propagates successfully across partitions in this case also.

Generalizing the above two cases for any points-to fact $\mathtt{r}$, we conclude the following.

$$\mathtt{r} \in \mathtt{R_{CS}} \implies \mathtt{r} \in \mathtt{R_{randomCS}}$$

This proves the soundness of Algorithm 9. □

5.4.1 Remark

With the soundness guarantee, one may be tempted to view the technique as an application of Las Vegas randomization [21]. A Las Vegas algorithm, unlike a Monte Carlo algorithm [21], is a randomized algorithm which is guaranteed to give a correct (i.e., sound) result. However, it should be noted that a Las Vegas randomized algorithm may explore the complete search space (even if exponential) for reaching a sound result and, in the worst case, may degenerate to a deterministic behavior. In contrast, our technique is solely guided by the probability $\rho^{\mathtt{ad}}_{\mathtt{run}}$ of the analysis dimension and will not behave in a deterministic fashion unless explicitly specified (by making $\rho^{\mathtt{ad}}_{\mathtt{run}} = \mathtt{1}$. More importantly, irrespective of the degree of randomization, our technique always guarantees a sound result.

5.5 Experimental Evaluation

As before, we evaluate our approach using 16 SPEC 2000 C/C++ benchmarks and five large open source programs, namely *httpd, sendmail, ghostscript, gdb* and *wine-server*. We use LLVM framework [81] for all our experiments (refer to the block diagram in Figure 4.5 in Chapter 4). All the experiments are carried out on the same platform, with an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM.

Our randomized analysis stores points-to information in sparse bitmaps. The baseline approach uses Andersen's points-to analysis [3]. It also performs optimizations like offline variable substitution [108] and online cycle elimination [34]. We show the effect of our randomized approach specifically for context-sensitive analysis which poses major challenge in terms of scalability for large programs. Our empirical results also reveal that the randomized approach does not yield significant benefits for other analysis dimensions, for which the analysis times are

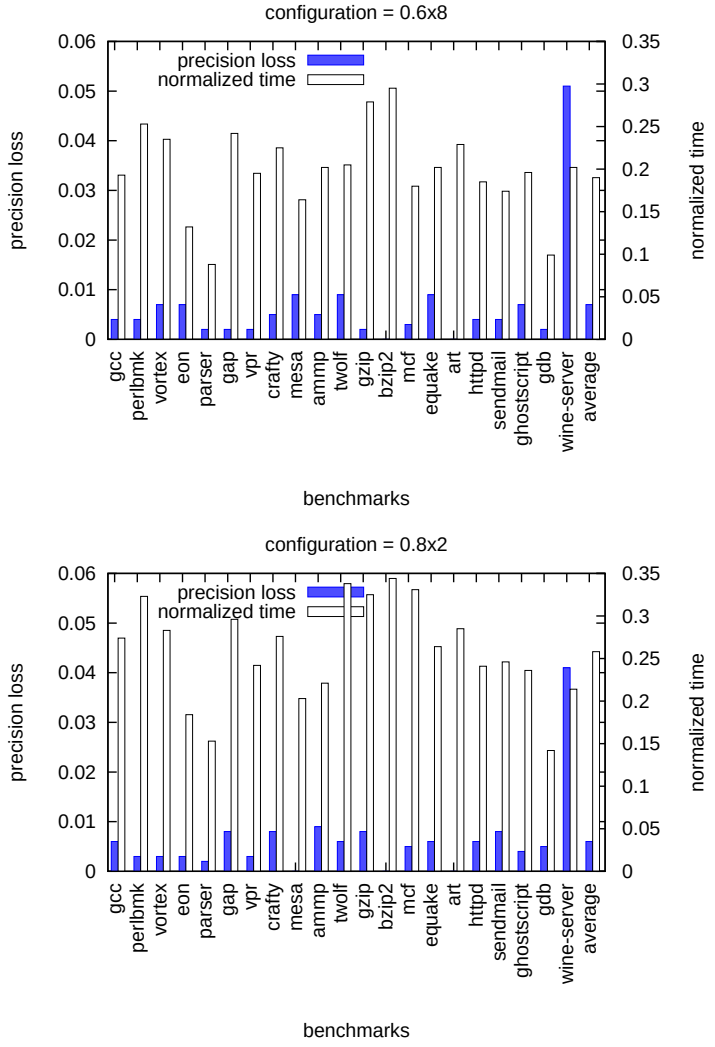


Figure 5.4: Effect of two representative configurations

already very small (see Section 5.6). Hence we focus on context-sensitive, flow-insensitive, field-insensitive analysis in this section. Context-sensitivity is implemented as an invocation-graph based approach as detailed in Section 4.4 (Chapter 6).

5.5.1 Overall Effect of Representative Configurations

We first illustrate the effect of two representative configurations on each benchmark. The configurations are: (i) selection probability $\rho = 0.6$, number of runs $N = 8$ represented as '0.6x8' and (ii) selection probability $\rho = 0.8$, number of runs $N = 2$ represented as '0.8x2'. Figure 5.4 shows the results for these two configurations. Left y-axis represents the precision loss while

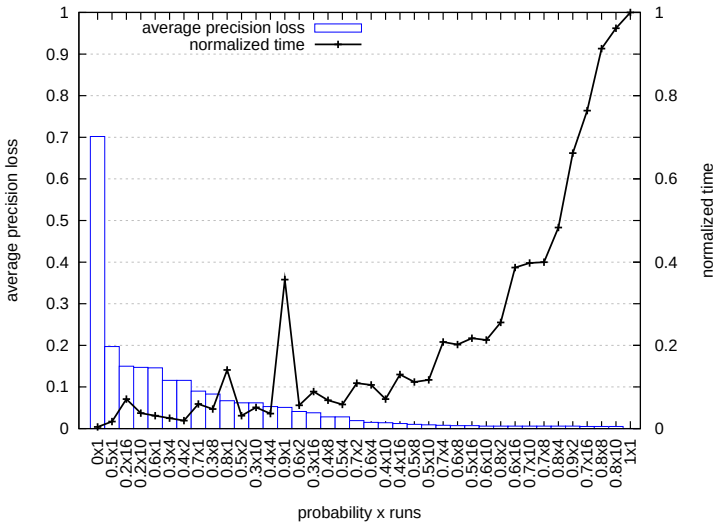


Figure 5.5: Overall effect across various configurations

right y-axis represents the normalized analysis time. The analysis time is normalized with respect to that of the most precise analysis, i.e., Andersen's analysis extended for context-sensitivity. Precision is defined as the ratio of the size of points-to set (of all pointer variables) in Andersen's analysis (extended for context-sensitivity) to that of the randomized approach for a given configuration. Precision loss is simply 1 - precision. We calculate the normalized analysis time as the ratio of the analysis time of the randomized approach to the analysis times of Andersen's analysis for each benchmark.

We observe that for each of the two configurations, our randomized analysis is almost three times faster with 20 out of 21 benchmarks being 99% as precise as Andersen's analysis. This shows significant reduction in analysis time (by a factor of 3x – 5x) with negligible precision loss. These two are not the only configurations that exhibit this benefit. After an extensive empirical evaluation of 54 configurations ($\rho^{\texttt{cs}}_{\texttt{run}} \in [\texttt{0.1...0.9}]$ and $\texttt{N} \in [\texttt{1}, \texttt{2}, \texttt{4}, \texttt{8}, \texttt{10}, \texttt{16}]$), we identified 36 configurations that give normalized analysis time less than 1. The effect of these configurations across all benchmarks is shown in Figure 5.5 sorted on precision loss. We plot the average precision loss (arithmetic mean of precision loss across all benchmarks) and the normalized analysis time (for all the benchmarks). The normalized analysis time is obtained as the ratio of the sum of the analysis times for all benchmarks in the randomized approach to that in Andersen's method. The labels on x-axis are of the form `a x b` where `a` is the probability of selection $\rho^{\texttt{cs}}_{\texttt{run}}$ and `b` is the number of times the analysis is run. Thus, for context-sensitive

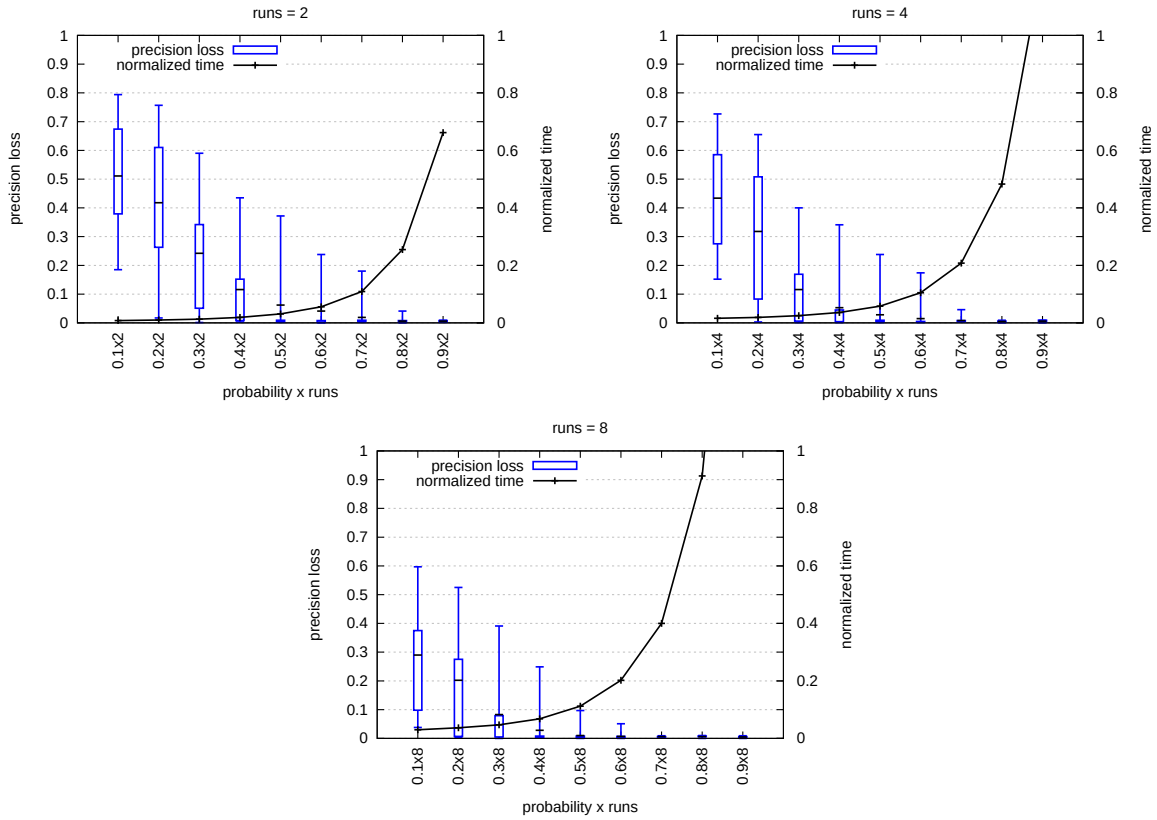


Figure 5.6: Effect of selection probability ρ

randomized analysis, 0x1 represents completely context-insensitive analysis while 1x1 represents a context-sensitive analysis. We observe that for several configurations, the randomized analysis requires normalized analysis time of less than 0.3, while still achieving an average precision loss less than 5%.

5.5.2 Effect of Selection Probability

Next we evaluate the impact of configuration parameters. Figure 5.6 shows the result of varying selection probability $\rho^{\mathtt{cs}}_{\mathtt{N}}$ for 2, 4, 8 runs. The graph shows a single line representing the normalized analysis time on the right y-axis and the precision loss on the left y-axis as a box plot with five values: minimum, $25^{\mathtt{th}}$ percentile, average, $75^{\mathtt{th}}$ percentile and maximum. We observe that with the increase in the selection probability, average precision improves significantly. While configurations with low selection probability ($\rho^{\mathtt{cs}} < 0.3$) result in very high precision loss ($\geq 30\%$), configurations with $\rho^{\mathtt{cs}} \geq 0.5$ achieve an average precision loss of

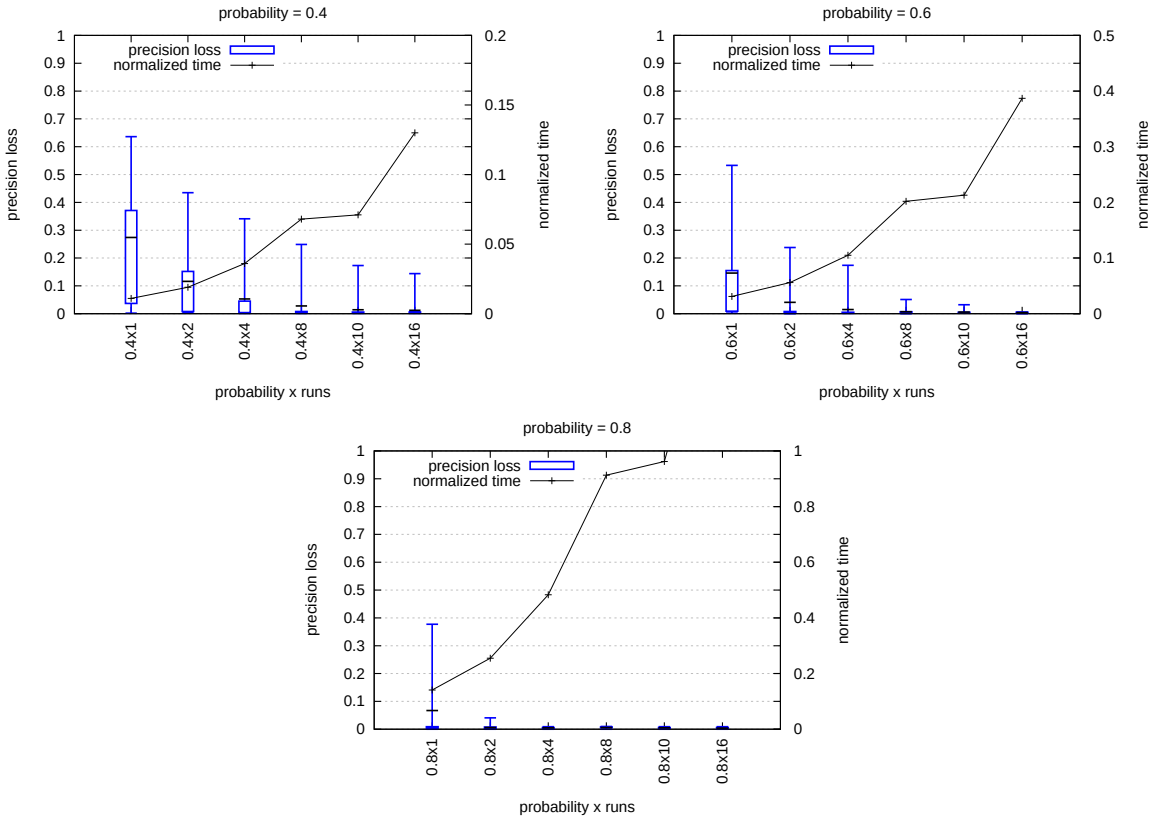


Figure 5.7: Effect of number of runs N

less than 5%, while the normalized analysis time remains less than 0.2. Specifically, for the configuration 0.7x4, the average and maximum precision loss are 0.3% and 0.8% respectively, while the normalized analysis time is 0.2. The configurations 0.6x8 and 0.7x8 achieve very high precision with normalized analysis times less than 0.4. When the number of runs is increased to 8, the normalized analysis time increases beyond 1 for selection probability above 0.8. However, there are several combinations of the selection probability and the number of runs that achieve an average precision loss of less than 2% with the analysis time ranging from 0.2 – 0.5 times that of the baseline Andersen's analysis. These combinations are the ones that form the configuration *sweet spots* which would prove beneficial to the clients that can afford to trade off a small amount of precision for a large improvement in analysis time.

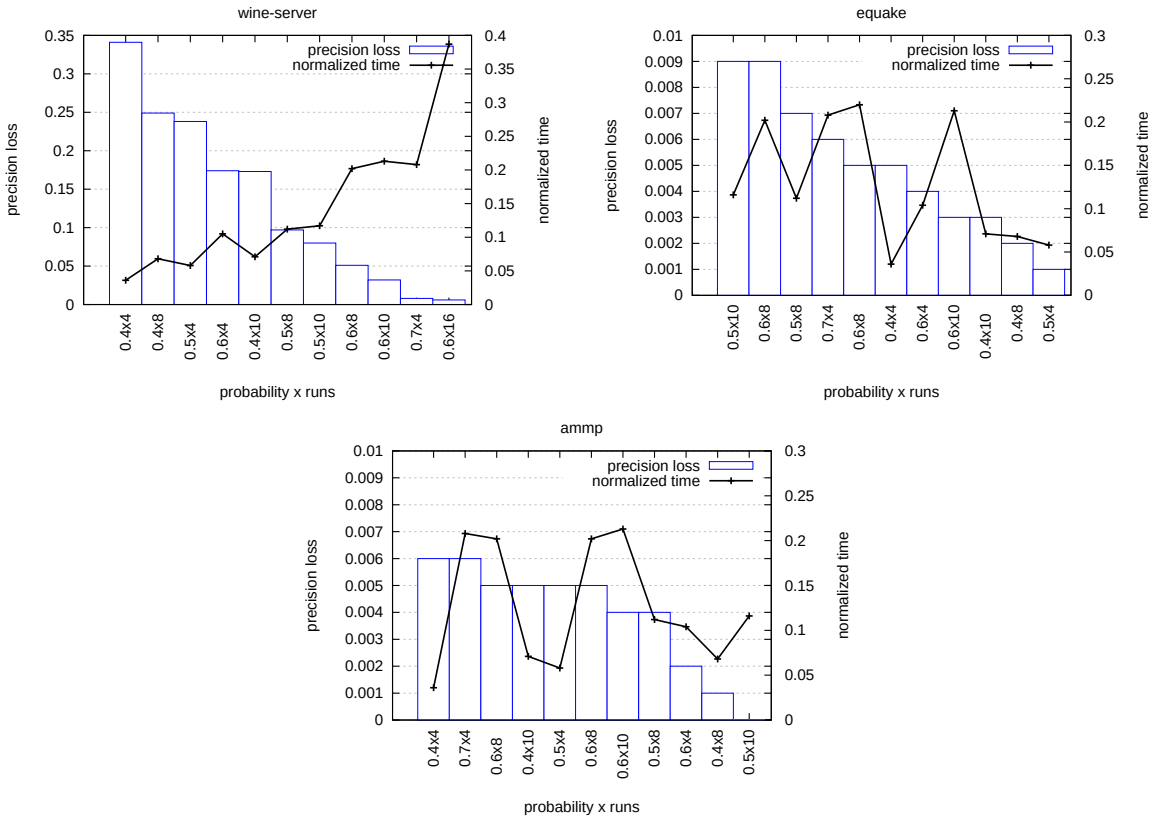


Figure 5.8: Effect of some configurations over programs with high precision loss

5.5.3 Effect of Number of Runs

Figure 5.7 shows the effect of varying the number of runs N for a fixed $\rho^{\texttt{cs}}_{\texttt{run}}$ where $\rho^{\texttt{cs}}_{\texttt{run}} \in [0.4, 0.6, 0.8]$. Note that scales of the right y-axis differ across the plots. For a given probability, increasing N improves precision with a linear increase in analysis time. These results also reveal some interesting configurations that achieve a very precise analysis (with an average precision loss of 0.05) with significant reduction in analysis time (greater than 70%).

5.5.4 Benchmarks with High Precision Loss

The box plots in Figure 5.6 and 5.7 show that while the average precision loss is very low for most of the configurations, the maximum precision loss and the last quartile are relatively higher. This means that 25% of the benchmarks, i.e., 5 out of 21, still incur a precision loss of 1 – 10% for some of the good configurations (e.g., 0.5x8, 0.7x4, etc.) and 1 – 2% for some very good configurations (e.g., 0.6x8, 0.7x8, etc.). We investigate these benchmarks in this section.

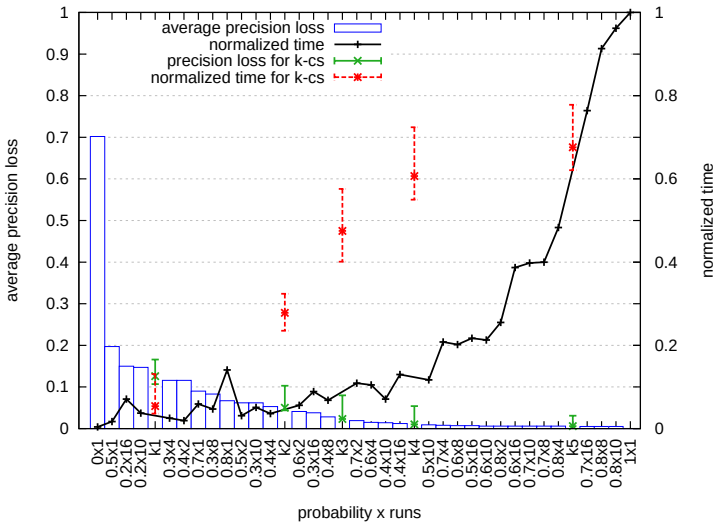


Figure 5.9: Randomized versus k-context-sensitive analysis

To analyze such programs, we study the effect of a few configurations over three benchmarks that have a higher precision loss, namely, *wine-server*, *equake* and *ammp*. The results are given in Figure 5.8. We observe, for *wine-server*, that there exist configurations (like 0.7x4 and 0.6x16) that provide a precision loss of less than 1% with normalized analysis time less than 0.4. Thus, even for those programs that incur a high precision loss in one configuration, there are configurations for randomized approach that achieve very high precision but still with significant reduction in the analysis time. Therefore, it is essential to use a proper configuration for a given program. We address this issue in Section 5.5.7 by designing an adaptive algorithm that chooses appropriate configuration in each run to achieve a scalable performance.

5.5.5 Comparison with k-Context-sensitivity

A k-context-sensitive analysis keeps track of last k callers of a function in the call-stack. The context beyond k callers is ignored. Thus, $k = 0$ implies a context-insensitive analysis. With increasing k, the context information tracked also increases which results in a higher analysis precision and usually a higher analysis time. Both randomized context-sensitivity and k-context-sensitivity can be viewed as merging of nodes in an invocation graph (see Figure 5.2). However, the difference lies in identification of nodes for the merge operation. In case of k-context-sensitivity, the nodes beyond k callers are merged irrespective of how much precision is lost due to merging or irrespective of the non-merged (tracked) nodes offering little precision

improvement. In contrast, randomized context-sensitivity merges nodes randomly in the invocation graph. It too, does not consider the precision offered by not merging nodes; but being random, it is not susceptible to a specific configuration that would arbitrarily skew its benefits in terms of analysis time and precision. In fact, in practice, randomization works much better than a deterministic k-context-sensitive approach as shown in Figure 5.9. The plot is essentially Figure 5.5 with results of k-context-sensitivity superimposed for k = 1,2,3,4,5, denoted by k1, k2, k3, k4, k5 respectively. Note that the plot is sorted based on average precision loss of randomized context-sensitivity results. The precision-loss plot for k-context-sensitivity also includes the maximum and minimum precision-loss values, and the analysis time for k-context-sensitivity also shows the 25^{th} and 90^{th} percentile.

We observe that the analysis for $k = 1$, i.e., the analysis that tracks only one caller, denoted as k1, requires normalized analysis time of only 0.054. However, it results in 12.6% (maximum 29.2%) precision loss. The precision loss is higher in several benchmarks where a separate memory allocation wrapper function is used for customized memory or error management (e.g., *gcc, perlbmk, parser, twolf, httpd, sendmail, ghostscript, gdb* etc.). The precision loss reduces considerably with increasing k. However, the normalized analysis time sharply goes on increasing. $k = 2$ and 3 offer a nice sweet-spot between analysis time (0.28 and 0.48) and precision (95% and 98%).

However, randomized context-sensitivity steadily outperforms k-context-sensitivity. Several configurations (like 0.4x10, 0.6x8, etc.) have less than 1% precision loss with analysis time less than 0.2. This suggests that merging nodes randomly in an invocation graph achieves higher benefits than strictly merging callers beyond k levels.

5.5.6 Effect of Mixed Randomization

In our experiments so far, the selection probability $\rho^{\texttt{cs}}_{\texttt{run}}$ was fixed across different runs. However, as shown in Figures 5.6 and 5.7, although precision improves with the number of runs, the gain in precision reduces after a few runs. In order to check if a selection probability varying across runs can lead to lower precision loss without significantly increasing the analysis time compared to a fixed value of $\rho^{\texttt{cs}}_{\texttt{run}}$, we study the effect of mixed randomization. For this, we carefully identify the selection probabilities that together require normalized analysis time less than 1. Here, a configuration `a-b-c-d` represents selection probabilities of `a, b, c` and `d`

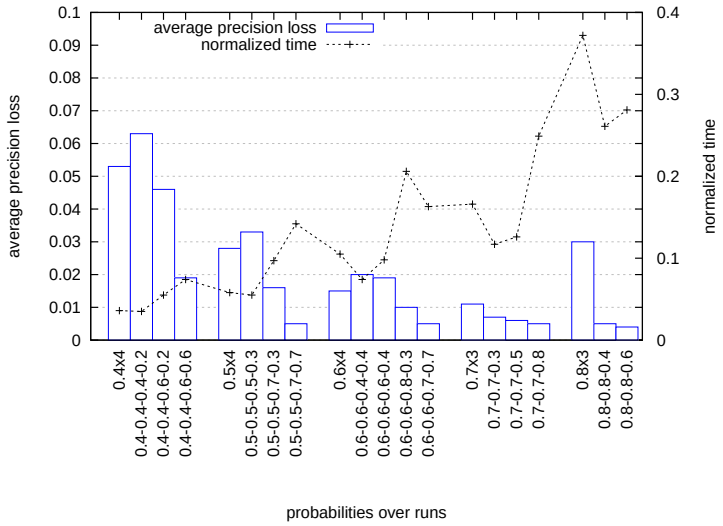


Figure 5.10: Effect of mixed randomization

in successive runs. We present only those results here that have an average precision loss of less than 10%. Figure 5.10 shows average precision loss (left y-axis) and analysis time (right y-axis) for various configurations (x-axis) across all the benchmarks. The configurations are partitioned according to ρ_1^{cs} of the first run where $\rho_1^{\mathsf{cs}} \in [0.4 - 0.8]$. The first bar in each partition recalls the effect of the relevant configuration with a fixed selection probability. The rest of the bars in each partition are sorted on precision loss. We observe that in each case, a mixed configuration gives a nice trade off between analysis time and precision compared to the corresponding fixed configuration. For instance, the configuration 0.5-0.5-0.7-0.7 reduces average precision loss from 3% to less than 1% while being more than 6× faster than the exact context-sensitive analysis.

5.5.7 Adaptive Analysis

Choosing an appropriate selection probability for a run to analyze a given program is a non-trivial task. It depends upon various program characteristics like number of constraints, number of contexts, etc. A randomly chosen configuration may take a large amount of time or may end up being very imprecise. Therefore, we develop an adaptive approach for choosing an appropriate amount of randomization beyond two runs. The idea is to start with a selection probability that in general gives a good trade-off between analysis time and precision. After

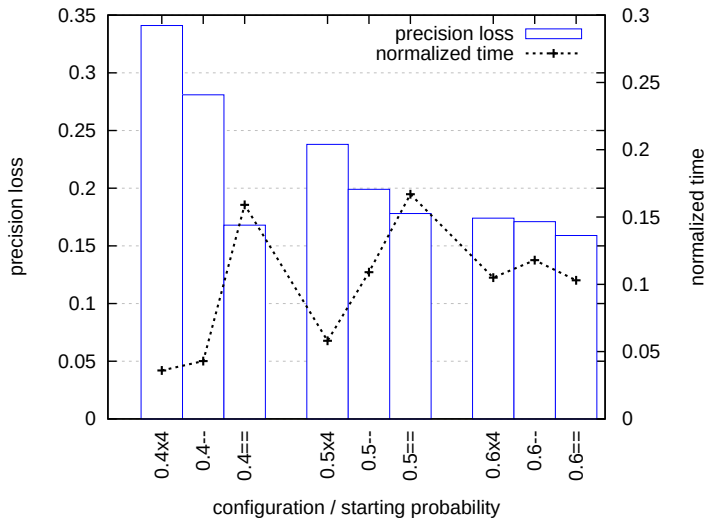


Figure 5.11: Adaptive analysis

two runs, depending upon the change in the points-to information, the next selection probability is calculated. This process is repeated in the succeeding runs. The initial two runs have the same selection probability $\rho^{cs}_{1,2} \in [0.4, 0.5, 0.6, 0.7, 0.8]$. If the amount of points-to information changes by more than $\texttt{threshold}_{\texttt{max}}$%, then the randomized approach continues with the same selection probability. If the amount of points-to information changes by less than $\texttt{threshold}_{\texttt{min}}$%, then it shows that having another run will not change a lot of points-to information and we stop the analysis (note that the soundness is not compromised). We empirically set $\texttt{threshold}_{\texttt{max}}$ and $\texttt{threshold}_{\texttt{min}}$ to 20% and 2% respectively. When the amount of points-to information changes between the two thresholds, we have two options (represented, for example, as 0.4af and 0.4av respectively): (i) increase the selection probability with a fixed amount, say 0.1. (ii) increase the selection probability with a variable amount proportional to the changed points-to information (e.g., increase by 0.07 for 18% change whereas increase by 0.04 for 5% change). Figure 5.11 shows the results averaged over all the benchmarks after implementing the above two options, along with the corresponding result for a fixed selection probability for all the runs represented, for example, as 0.4x4. The graph is once again partitioned according to the initial selection probability. We observe that both the adaptive versions improve upon the precision. For instance, for the starting selection probability of 0.4, the precision loss reduces by around 15% and 55% for the two adaptive versions. Note that the selection probabilities in the second run onwards may differ across benchmarks. Similar to the

mixed randomization, adaptive analysis reveals several interesting configurations which can be used to trade off analysis time versus precision. The configuration 0.8x2 changes less than 2% information in the second run, and hence the adaptive analysis stops after two iterations, but still achieves an average precision loss less than 1%.

5.6 Other Analysis Dimensions

In this section we discuss how our randomization technique can be applied to other analysis dimensions. Specifically, we discuss randomized flow-sensitive points-to analysis and randomized field-sensitive points-to analysis. In each case, appropriate selection, summarization and composition operations need to be modeled. For flow-sensitivity, the selection step selects basic blocks that are to be processed in a less precise way, i.e., in a flow-insensitive manner and groups them in an aggregate `A`. The summarization step updates the incoming and the outgoing edges for such basic blocks to come to and go out of `A`. The composition steps specializes the computed flow-insensitive points-to information to become flow-sensitive, i.e., it tags the information as that computed at `A`.

For randomizing field-sensitivity, the selection step selects statements with field-accesses to be processed in a field-insensitive manner. The summarization step rewrites all field-accesses to be to the aggregate. The composition step is empty as aggregate accesses are considered valid in a field-sensitive analysis.

It is possible to randomize multiple analysis dimensions simultaneously. For instance, a randomized context-sensitive, field-sensitive analysis may choose a set of program functions to be processed in a context-insensitive manner and, at the same time, may choose to process certain field-accesses in a field-insensitive manner.

5.7 Related Work

We presented a bloom-filter based points-to analysis in Chapter 4 and a randomized points-to analysis in this chapter. Both the methods provide a mechanism to alter their configuration parameters to achieve a trade-off between precision and analysis time. A few analyses [117, 49, 46] provide such a trade off. For instance, Hardekopf and Lin [49] propose eager, lazy and hybrid cycle detection. Depending upon the frequency of cycle detection, the gain in the

analysis time by collapsing cycles is traded off for the overhead of cycle detection. Guyer and Lin [46] propose a client-driven pointer analysis that adjusts precision according to the client needs.

The work by Shapiro and Horwitz [117] is very close to our work on randomized analysis. It proposes an algorithm based on multiple out-degrees for the nodes in the points-to graph (instead of unity as in Steensgaard's analysis [123]), so that the precision and the analysis time can be tuned between those of Steensgaard's analysis [123] and Andersen's analysis [3]. The work also involves *multiple runs* to compose the results of various runs. However, the work is illustrated specifically for flow-insensitive, context-insensitive and field-insensitive analysis. In contrast, we illustrate how our technique can be applied to various analysis dimensions. Further, randomization forms the basis of our technique, whereas the approximation in their work is either client-driven or based on heuristics, and does not involve any randomization.

The work on *program decomposition* [138, 110] has some resemblance to our randomization technique. Program decomposition involves analyzing a program to create groups of program elements (say statements) such that a different kind of analysis could be run on each of them. It is based on the assumption of skewness in typical programs wherein a small part of a program adds a large number of data flow facts. In contrast, our randomization technique does not depend upon the skewness. Further, program decomposition can be applied in a complementary manner to identify the set of program elements to be processed with a different precision level. This would add some determinism to our technique.

The work on *combined analysis* [139] is similar in spirit to our work, although it involves no randomization. It requires an equivalence relation on the program names to partition variables. Each program segment induced by the equivalence relation is then analyzed with a different alias analysis. Our technique is more general and does not require an equivalence relation using decomposition.

To the best of our knowledge, ours is the first work on sound randomized points-to analysis.

5.8 Chapter Summary

A randomized algorithm is typically simpler than its deterministic counterpart and achieves a good result in substantially less amount of time. However, typical randomized algorithms for

program analysis often compromise the analysis soundness. This also reduces the applicability of such analyses. For instance, randomization has been mainly used only in testing and program debugging. We proposed a sound randomized technique to scale pointer analysis in this chapter. Our method selectively applies different kinds of analyses on different partitions of the program entities to be processed and then composes the results carefully to get a sound approximation to the points-to solution. We illustrated the technique to develop a randomized context-sensitive points-to analysis. Our empirical evaluation using a set of 21 benchmarks revealed several configurations that achieve less than 5% precision loss with over 50% reduction in context-sensitive analysis time. Based on our analysis of the results, we developed an adaptive randomized points-to analysis which can be used for a program for which the right configuration is unknown. We hope that the technique would prove useful to other program analyses.

Chapter 6

Points-to Analysis as a System of Linear Equations

6.1 Introduction

In order to achieve scalability, one way is to formulate the points-to analysis problem P into a well-studied problem Q, solve Q using the most efficient methods available in literature and map the solution of Q back to the solution of P. If the conversions between P and Q can be done efficiently, then we can hope for an overall efficient solution for points-to analysis.

With this high-level picture in mind, we observe that finding a flow-insensitive solution of points-to constraints in an iterative manner is similar in spirit to obtaining a solution to a system of linear equations. Each equation defines a constraint on the feasible solution and a linear solver progressively approaches the final solution in an iterative manner. Similarly, every points-to statement forms a constraint on the feasible points-to information and every iteration of a points-to analysis *refines* the points-to information obtained over the previous iteration. We exploit this similarity to *map* the input source program to a set of linear constraints, solve it using a standard linear equation solver and *unmap* the results to obtain the points-to information. As we show in the next section, a naive approach of converting points-to statements into a linear form faces several challenges due to (i) the distinction between ℓ-value and r-value in points-to statements, (ii) multiple dereferences of a pointer and (iii) the same variable being defined in multiple statements. We address these challenges with novel mechanisms based on prime factorization of integers.

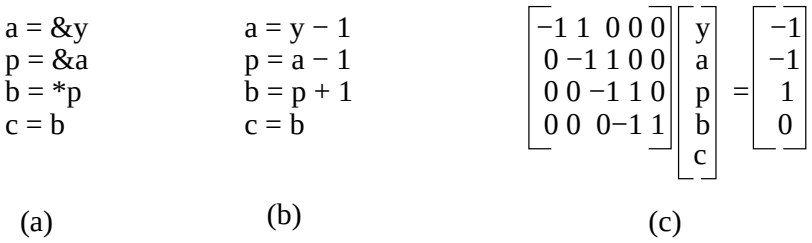


Figure 6.1: Example to illustrate points-to analysis as a system of linear equations

In the following section, we describe a simple method to convert a set of points-to statements into a set of linear equations. Using it as a baseline, we discuss several challenges such a method poses. Consequently, in Section 6.3, we introduce our novel transformation that addresses all the discussed issues and present our points-to analysis algorithm in Section 6.4. We prove the soundness and precision of the algorithm in Section 6.5. We describe how our novel representation of points-to information can improve client analyses in Section 6.6. Finally, in Section 6.7, we demonstrate the scalability promise in our technique by running our analysis on our benchmark suite consisting of SPEC 2000 benchmarks and five large open source programs, namely, *httpd*, *sendmail*, *ghostscript*, *gdb* and *wine-server*. We discuss related work in Section 6.8. We conclude the chapter with a summary in Section 6.9.

6.2 Naive Approach

In this section, we develop a simple transformation to convert points-to constraints into linear equations and explain its limitations.

Consider the set of C statements given in Figure 6.1(a). Let us define a transformation that translates &x into $(\mathtt{x} - 1)$, `*x` to `(x + k)`, `**x` to `(x + 2k)`, etc., and $*^{\mathtt{n}}\mathtt{x}$ into $\mathtt{x} + \mathtt{nk}$, where `n` is the number of '$*$'s used in dereferencing. If we choose `k = 1` then `*x` is translated to `(x + 1)`, `**x` to `(x + 2)` and so on. A singleton variable without any operations is copied as it is, thus, `x` translates to `x`. The transformed program with `k = 1` now looks like Figure 6.1(b). This becomes a simple linear system of equations that can be written in matrix form $\mathtt{AX} = \mathtt{B}$ as shown in Figure 6.1(c).

This small example illustrates several interesting aspects. First, the choices of transformation functions for '$*$' and '&' are not independent, because '$*$' and '&' are complementary

operations by language semantics, which should be carried to the linear transformation. Second, selecting $\mathtt{k} = 0$ is not a good choice because we lose information regarding the address-of and dereference operations, resulting in loss of precision in the analysis. Third, every row in matrix A has at most two non-zero elements, i.e., every equation has at most two unknowns. All the entries in matrices A and B are 0, 1 or -1.

Solving the above linear system using a solver yields the following parameterized result: $\mathtt{y} = \mathtt{r}, \mathtt{a} = \mathtt{r} - 1, \mathtt{p} = \mathtt{r} - 2, \mathtt{b} = \mathtt{r} - 1, \mathtt{c} = \mathtt{r} - 1$.

From the result, we can quickly conclude that a, b and c are aliases because they have the same value. Further, since the value of p is one smaller than that of a, we say that p points to a, and in the same manner, a points to y. This conclusion is derived due to our transformation of a dereference $*\mathtt{x}$ as $\mathtt{x} + 1$.

Andersen's analysis [3] over our example program in Figure 6.1(a) gives the following points-to information.

$\mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{y}\}$

We see that our naive analysis of the example using a linear solver as explained above computes all the points-to information obtained using Andersen's analysis. Thus, for this example, the linear solver method gives a safe solution.

6.2.1 Issues

The naive transformation illustrated above suffers from the following issues.

Imprecision in the analysis. Note that our naive solver added a few spurious points-to pairs. Specifically, because p is one smaller than that of b as well as c, the following facts are also derivable from the result: $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}\}$. This means that our naive approach, although *sound*, computes an over-approximation of the information computed by Andersen's analysis [3]. Therefore, the first-cut approach described above gives an *imprecise* result.

Cyclic dependences. The naive method illustrated above can handle cyclic dependences for a type-safe language. For instance, a set of statements like $\mathtt{a} = \mathtt{b}; \mathtt{b} = \mathtt{c}; \mathtt{c} = \mathtt{a}$ can be easily transformed into a linear system. However, it cannot handle cyclic dependences that do not respect type-safety. For instance, a set of statements $\mathtt{a} = \&\mathtt{b}; \mathtt{b} = \mathtt{c}; \mathtt{c} = \&\mathtt{a}$ generates a system

of equations that does not have any solution. A minimal example of such a weakly-typed cyclic dependence is $\mathtt{a} = \&\mathtt{a}$. Being a type-unsafe language, C allows coercions of this form. Our naive method would transform such a statement into the equation $\mathtt{a} = \mathtt{a} - 1$ which does not have a solution.

More formally, each equation in the system of equations transformed using the naive method is of the form $\mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}}$ where $\mathtt{b_{ij}} \in \{0, 1, -1\}$. We can build a constraint graph[1] $\mathtt{G} = (\mathtt{V}, \mathtt{E}, \mathtt{w})$ where

$\mathtt{V} = \{\mathtt{a_1}, ..., \mathtt{a_n}\} \cup \{\mathtt{a_0}\}$, $\mathtt{w}(\mathtt{a_i}, \mathtt{a_j}) = \mathtt{b_{ij}}$, $\mathtt{w}(\mathtt{a_0}, \mathtt{a_i}) = 0$ and

$\mathtt{E} = \{(\mathtt{a_i}, \mathtt{a_j}) : \mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}}$ is a constraint$\} \cup \{(\mathtt{a_0}, \mathtt{a_1}), ..., (\mathtt{a_0}, \mathtt{a_n})\}$.

The above linear system has a feasible solution *iff* the corresponding constraint graph has no cycle with negative weight [21]. A negative cycle corresponds to a type-unsafe situation. For instance, the type-unsafe sequence of statements $\mathtt{a} = \&\mathtt{b}; \mathtt{b} = \mathtt{c}; \mathtt{c} = \&\mathtt{a}$ generates a negative weight cycle when tranformed into a linear system as shown above. However, a linear solver would not output any solution for a system with such a cycle. Our modified algorithm (presented below) uses appropriate variable renaming that allows a standard linear solver to solve such equations.

Inconsistent equations. The above naive approach also fails for multiple assignments to the same variable. For instance, $\mathtt{a} = \&\mathtt{x}; \mathtt{a} = \&\mathtt{y}$ is a valid program fragment. However, $\mathtt{a} = \mathtt{x} - 1, \mathtt{a} = \mathtt{y} - 1$ does not form a consistent equation system unless $\mathtt{x} = \mathtt{y}$. This issue is addressed in the context of *bug-finding* by Ganapathy et al. [39]. However, since they [39] allowed false positives as well as false negatives, the approach taken was to progressively find and remove *irreducibly inconsistent sets (IIS)* in order to obtain a feasible solution which can then be solved using a linear solver. However, since we want our static points-to analysis to be *safe*, we cannot afford our algorithm to generate false negatives. Therefore, we cannot simply find an IIS out of the inconsistent points-to equations and solve them.

One way of handling inconsistent equations is to assume that all the variables involved in more than one assignment point to a universal set of pointees. This would make the system feasible but at the cost of precision.

Yet another way to solve the issue is to convert the program into Static Single Assignment

[1]Note that this constraint graph is different from the one defined in Section 2.5.

(SSA)[52] form which would guarantee single assignment to each variable (although it would not guarantee feasibility of the solution). However, the same issue resurfaces when we consider the same set of assignment across multiple iterations of the analysis. For instance, if we convert `a = x` and `a = y` into SSA form to get `a1 = x` and `a2 = y`, at the end of the first iteration both `a1` and `a2` would have some value computed which needs to be carried forward in the second iteration. However, that cannot be done because doing so can result into an inconsistent set of equations (across iterations) if the points-to sets of `x` and `y` have changed, in which case, it would be different instantiations of `x` and `y`. Therefore, we need a mechanism which is an extension of SSA that works across multiple iterations. Our solution works irrespective of whether the program is in SSA form or not.

Nonlinear system of equations. One way to handle inconsistent equations is to multiply the constraints having the same unknown to generate a non-linear set of equations. Thus, `a = &x` and `a = &y` would generate a non-linear constraint $(\texttt{a} - \texttt{x} + 1)(\texttt{a} - \texttt{y} + 1) = 0$. The above non-linear equation defines a second order curve that has zeros at $(\texttt{x} - 1)$ and $(\texttt{y} - 1)$, defining the points-to set of `a`. Adding another pointee to `a` increases the degree of the curve and has an additional zero. The set of all zeros for all curves defines the feasible region (actually, a set of discrete points) of the set of equations. The solution to the system would be the minimal set of zeros obtained using the non-linear constraints which would be calculated using an iterative analysis until a fixed-point over zeros is reached. However, non-linear analysis is often more expensive than a linear analysis. Further, maintaining integral solutions across iterations using standard techniques is a difficult task.

As shown in the next section, it is possible to take care of inconsistent equations without disturbing the linearity of the system.

Equations versus inequations. The inclusion-based analysis semantics for a points-to statement `a = b` imply points-to-set(a) $\supseteq$ points-to-set(b). Transforming the statement into an equality $\texttt{a} - \texttt{b} = 0$ instead of an inequality can be imprecise, as equality in mathematics is bidirectional. Thus, in mathematics, `a = b` also implies `b = a`, which is not the case according to the inclusion-based analysis semantics. In fact, that is the semantics of a unification-based

analysis. For inclusion-based analysis, we need to transform a statement $\texttt{a} = \texttt{b}$ into an inequality constraint $\texttt{a} - \texttt{b} \geq 0$. It is easy to verify, however, that if a set of linear constraints contains each *ℓ-value* exactly once *and* this holds across iterations of the analysis, the solution sets obtained using inequalities and equalities would be the same. We exploit this observation in our algorithm and use linear equalities in our approach.

Dereferencing. As per our first-cut approach, transformations of points-to statements $\texttt{a} = \&\texttt{b}$ and $*\texttt{a} = \texttt{b}$ would be $\texttt{a} = \texttt{b} - 1$ and $\texttt{a} + 1 = \texttt{b}$ respectively. According to the algebraic semantics, the above equations are equivalent, although the two points-to statements have different semantics. This happens because $*\texttt{a}$ is a points-to expression generating an *ℓ-value* whereas $\texttt{a} + 1$ is an algebraic expression generating an *r-value*. This naive transformation can easily add spurious points-to information. Further, dereferencing pointer `a` in a *store* statement $*\texttt{a} = \texttt{b}$ may generate multiple *ℓ-values* which cannot be easily captured using a system of equations. This necessitates one to take care of the *store* constraints separately. Note that *load* constraints of the form `a=*b` also generate multiple *r-values* but do not pose an issue for the solver.

6.3 Prime-Factorization Approach

We solve the issues with the above naive approach with a modified mechanism. Our approach is iterative, and in each iteration, it goes through four major steps, viz., (i) preprocessing, (ii) solving the linear system of equations, (iii) post-processing and (iv) evaluating *store* constraints. A limitation of our approach is that it can handle only upto `k` levels of dereferencing, for a fixed `k`. This is not a serious limitation, as, in practice, programs contain only 3 or 4 levels of dereferencing (as in ***p and ****q) and no more. We illustrate our approach using the following example.

$\texttt{a} = \&\texttt{x}; \texttt{b} = \&\texttt{y}; \texttt{p} = \&\texttt{a}; \texttt{c} = *\texttt{p}; *\texttt{p} = \texttt{b}; \texttt{q} = \texttt{p}; \texttt{p} = *\texttt{p}; \texttt{a} = \texttt{b}.$

6.3.1 Step 1: Preprocessing

First, we move *store* constraints, i.e., constraint of the form `*p = q`, from the set of equations to a set of generative constraints (as they *generate* more linear equations) that are processed specially. We proceed with the remaining non-*store* constraints.

Second, all constraints of the form $\mathtt{v} = \mathtt{e}$ are converted to $\mathtt{v} = \mathtt{v_{i-1}} \oplus \mathtt{e}$. Here, $\mathtt{v_{i-1}}$ is the value of the variable $\mathtt{v}$ obtained in the last iteration. Initially, $\mathtt{v} = \mathtt{v_0} \oplus \mathtt{e}$. This transformation ensures monotonicity required for a flow-insensitive points-to analysis. The operator $\oplus$ would be explained shortly. $\mathtt{v_0}$ is a constant, since it is already computed from the previous iteration.

Next, we assign unique prime numbers from a select set $\mathcal{P}$ to the right-hand side expression in each address-of constraint. We defer the definition of $\mathcal{P}$ to a later part of this subsection. In the above example, let $\&\mathtt{x}, \&\mathtt{y}$ and $\&\mathtt{a}$ be assigned arbitrary prime numbers, say $\&\mathtt{x} = 17; \&\mathtt{y} = 29$; and $\&\mathtt{a} = 101$. The addresses of the remaining variables $(\mathtt{b}, \mathtt{p}, \mathtt{q}, \mathtt{c})$ are assigned a special sentinel value χ. Further, all the variables of the form $\mathtt{v}$ and $\mathtt{v_i}$ are assigned an initial r-value of χ. Thus, $\mathtt{x}, \mathtt{y}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{p}, \mathtt{q}$ and $\mathtt{x_0}$, $\mathtt{y_0}$, $\mathtt{a_0}$, $\mathtt{b_0}$, $\mathtt{c_0}$, $\mathtt{p_0}, \mathtt{q_0}$ equal χ. We keep two-way maps of variables to their r-values and addresses. This step of assigning prime numbers is performed only once in the analysis. In the rest of the chapter, the term "address of a variable" refers to the prime number assigned to it by the preprocessing step of our analysis.

Next, a dereference $*\mathtt{q}$ in every load statement $\mathtt{p} = *\mathtt{q}$ is replaced by expression $\mathtt{q_{i-1}} + 1$ where $\mathtt{i}$ is the current iteration. Therefore, $\mathtt{c} = *\mathtt{p}$ becomes $\mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0}{+}1)$ and $\mathtt{p} = *\mathtt{p}$ becomes $\mathtt{p} = \mathtt{p_0} \oplus (\mathtt{p_0} + 1)$. Note that by generating different versions of the same variable in this manner, we remove cyclic dependences altogether. This happens because the variables $\mathtt{v_i}$ are never *defined* explicitly in the constraints. Hence they are not dependent on any other variable. The renaming is only symbolic and appears only for exposition purposes. Since values from only the previous iteration are required, we simply make a copy $\mathtt{v_{copy}}$ for each variable $\mathtt{v}$ at the start of each iteration.

Last, we rename multiple occurrences of the same variable as an *ℓ-value* in different constraints to convert it to an SSA-like form. For each such renamed variable $\mathtt{v'}$, we store a constraint of the form $\mathtt{v} = \mathtt{v'}$ in a separate merging constraint set. Thus, assignments to variable $\mathtt{a}$ in $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a} = \mathtt{b_0}$ are replaced as $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a'} = \mathtt{b_0}$ and the constraint $\mathtt{a} = \mathtt{a'}$ is added to the merging constraints set. The constraints now look as follows.

Linear constraints: $\mathtt{a} = \mathtt{a_0} \oplus \&\mathtt{x}; \mathtt{b} = \mathtt{b_0} \oplus \&\mathtt{y}; \mathtt{p} = \mathtt{p_0} \oplus \&\mathtt{a}; \mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0} + 1);$

$$\mathtt{q} = \mathtt{q_0} \oplus \mathtt{p}; \mathtt{p'} = \mathtt{p_0} \oplus (\mathtt{p_0} + 1); \mathtt{a'} = \mathtt{a_0} \oplus \mathtt{b}.$$

Generative constraints: $*\mathtt{p} = \mathtt{b}$.

Merging constraints: $\mathtt{a} = \mathtt{a'}; \mathtt{p} = \mathtt{p'}$.

Substituting the r-values and the primes for the addresses of variables, we get

$\mathtt{a} = \chi \oplus 17; \mathtt{b} = \chi \oplus 29; \mathtt{p} = \chi \oplus 101; \mathtt{c} = \chi \oplus (\chi + 1);$

$\mathtt{q} = \chi \oplus \mathtt{p}; \mathtt{p}' = \chi \oplus (\chi + 1); \mathtt{a}' = \chi \oplus \mathtt{b}.$

χ and $\oplus$. We unfold the mystery behind the values of χ and $\oplus$ now. The rationale behind replacing the address of every address-taken variable with a prime number is to have a *non-decomposable* element defining the variable. We make use of *prime factorization* of integers to map a value to the corresponding points-to set. The first trivial but important observation towards this goal is that any pointee of any variable has to appear as address taken in at least one of the constraints. Therefore, the only pointees any non-null pointer can have would exactly be the address-taken variables. Thus, a *composition* $\mathtt{v} = \mathtt{v_i} \oplus \mathtt{v_j} \oplus ...$ of the primes $\mathtt{v_i}, \mathtt{v_j}, ...$ representing address-taken variables defines the pointer `v` pointing to all these address-taken variables. The composition is defined by operator $\oplus$ and it defines a lattice over the finite set of all the pointers and pointees (Figure 6.2). The top element $\top$ defines a composition of all address-taken variables ($\mathtt{v_0} \oplus \mathtt{v_1} \oplus ... \oplus \mathtt{v_n}$) and the bottom element $\bot$ defines the empty set. Since we use prime factorization, $\oplus$ becomes the multiplication operator $\times$ and χ is the identity element, i.e., `1`. The reason behind using $\oplus$ and χ as placeholders is that it is possible to use an alternative lattice with different $\oplus$ and χ and achieve an equivalent transformation (as long as the equations remain linear).

Since every positive integer has a unique prime factorization, we guarantee that the value of a pointer uniquely identifies its pointees. For instance, if $\mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}$ and $\mathtt{b} \rightarrow \{\mathtt{y}, \mathtt{z}, \mathtt{w}\}$, and we assign arbitrary primes to $\&\mathtt{x}, \&\mathtt{y}, \&\mathtt{z}, \&\mathtt{w}$ as $\&\mathtt{x} = 11, \&\mathtt{y} = 19, \&\mathtt{z} = 5, \&\mathtt{w} = 3$, then the values of `a` and `b` would be calculated as $\mathtt{a} = 11 \times 19 = 209$ and $\mathtt{b} = 19 \times 5 \times 3 = 285$. Since, `209` can only be factored as 11×19 and `285` does not have any other factorization than $19 \times 5 \times 3$, we can obtain the points-to sets for pointers `a` and `b` from their prime factors.

Unfortunately, prime factorization is not known to be polynomial [69]. Therefore, for efficiency reasons, our implementation keeps track of the prime factors explicitly. We use a prime-factor-table for this purpose which stores all the prime factors of a value. We initially store all the primes p corresponding to the address-taken variables as $\mathtt{p} = \mathtt{p} \times 1$.

Thus, the system of equations now becomes

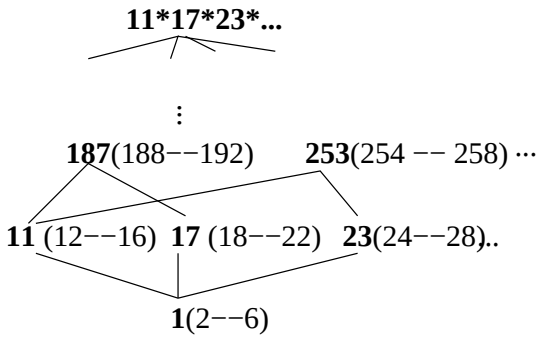


Figure 6.2: Lattice over the compositions of primes guaranteeing five levels of dereferencing

Linear constraints: $\mathtt{a} = 17; \mathtt{b} = 29; \mathtt{p} = 101; \mathtt{c} = 2; \mathtt{q} = \mathtt{p}; \mathtt{p}' = 2; \mathtt{a}' = \mathtt{b}$.

Generative constraints: $*\mathtt{p} = \mathtt{b}$.

Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

6.3.2 Step 2: Solving the System

Solving the above system of equations using a standard linear solver gives us the following solution.

$\mathtt{a} = 17, \mathtt{b} = 29, \mathtt{p} = 101, \mathtt{c} = 2, \mathtt{q} = 101, \mathtt{p}' = 2, \mathtt{a}' = 29$.

6.3.3 Step 3: Post-processing

Interpreting the values in the above solution obtained using a linear solver is straightforward except for those of $\mathtt{c}$ and $\mathtt{p}'$ (2 is not chosen to be one of the primes.). In the simple case, a value $\mathtt{v} + \mathtt{k}$ denotes $\mathtt{k}^{\mathtt{th}}$ dereference of $\mathtt{v}$. To find $\mathtt{v}$, our method checks each value ϑ in $(\mathtt{v} + \mathtt{k}), (\mathtt{v} + \mathtt{k} - 1), (\mathtt{v} + \mathtt{k} - 2), ...$ in the prime-factor-table. For the first ϑ that appears in the prime-factor-table, $\mathtt{v} = \vartheta$ and $\mathtt{k}' = \mathtt{v} + \mathtt{k} - \vartheta$ represents the level of dereferencing. We obtain the prime factors of ϑ from the table, which would correspond to the addresses of variables, reverse-map the addresses to their corresponding variables, then obtain the r-values of the variables from the map whose prime factors would denote the points-to set we want for expression $\mathtt{v} + \mathtt{k}$. Another level of reverse mapping-mapping would be required for $\mathtt{k} = 2$ and so on. We explain dereferencing method later (Algorithm 13).

Note that since our method can handle only a limited number of dereferences ($\mathtt{k}$), the number of iterations required in the dereferencing step is also limited (and is typically small in practice). Therefore, in the example, the value 2 of the variables $\mathtt{c}$ and $\mathtt{p}'$ is represented as

$\mathtt{1}+\mathtt{1}$ where the second 1 denotes a dereference and the first 1 is the value of the variable being dereferenced. In this case, since $\mathtt{v}=\mathtt{1}$, which is the sentinel χ, its dereference results in an empty set and thus, both c and $\mathtt{p}'$ are assigned a value of 1.

Selection of primes. In general, a value ϑ may be interpreted as $\mathtt{v_1}+\mathtt{k_1}$ as well as $\mathtt{v_2}+\mathtt{k_2}$, if the values $\mathtt{v_1}$ and $\mathtt{v_2}$ happen to be close to each other. To avoid this ambiguity, the ranges $(\mathtt{v_1}...\mathtt{v_1}+\mathtt{k})$ and $(\mathtt{v_2}...\mathtt{v_2}+\mathtt{k})$ must be non-overlapping for all possible $\mathtt{v_1}$ and $\mathtt{v_2}$ where k corresponds to the maximum level of dereferencing allowed by the analysis. This is accomplished by a careful selection of the prime numbers representing the address-taken variables. Our analysis selects primes offline and guarantees that a certain k number of dereferences will never overlap with one another. Our prime number set $\mathcal{P}$ is also defined for this specific k. More specifically, for any prime numbers $\mathtt{p}\in\mathcal{P}$, the products of any one[2] or more p are distance more than k apart. Thus, $|\mathtt{p_i}-\mathtt{p_j}|>\mathtt{k}$ and and $|\mathtt{p_i}*\mathtt{p_j}-\mathtt{p_l}|>\mathtt{k}$ and $|\mathtt{p_i}*\mathtt{p_j}-\mathtt{p_l}*\mathtt{p_m}|>\mathtt{k}$ and $|\mathtt{p_i}*\mathtt{p_j}*\mathtt{p_l}-\mathtt{p_m}*\mathtt{p_n}*\mathtt{p_o}|>\mathtt{k}$ and so on. Note that $\mathcal{P}$ needs to be computed only once and can be done offline, i.e., prior to running our analysis. Also, typically, the number of dereferences in real-world programs is very small (<5). The lattice for the prime number set $\mathcal{P}$ chosen for $k=5$ is shown in Figure 6.2. Here, the values in bold are the values that are attained by variables. The bracketed values beside a value ϑ in bold-face denote possible dereferencings of a variable which is assigned the value ϑ. For instance, (12,13,...,16) denote possible dereferencings of a variable which is assigned the value 11 for `k = 1..5`.

The next step is to merge the points-to sets of renamed variables, i.e., evaluating merging constraints. This changes a and p as $\mathtt{a}=\mathtt{17}\times\mathtt{29}$ and $\mathtt{p}=\mathtt{101}\times\mathtt{1}=\mathtt{101}$.

After merging, we discard all the renamed variables.

Thus, at the end of the first iteration, the points-to set contained in the values is:
$x\rightarrow\{\},y\rightarrow\{\},a\rightarrow\{x,y\},b\rightarrow\{y\},c\rightarrow\{\},p\rightarrow\{a\},q\rightarrow\{a\}$.

6.3.4 Step 4: Evaluating Special Constraints

The final step is to evaluate the generative constraints and generate more linear constraints. In the first iteration, the store constraint $*\mathtt{p}=\mathtt{b}$ generates the copy constraint $\mathtt{a}=\mathtt{b}$ which already exists in the system. Thus, no new linear constraints are generated. Note that the generative

[2]product of one number is the number itself.

constraints set is retained as more constraints may need to be added in further iterations. At the end of each iteration, our algorithm checks if any variable value is changed since the last iteration. If yes, then another iteration is required.

6.3.5 Subsequent Iterations

The constraints, ready for iteration number two, are

Linear constraints: $\mathtt{a} = \mathtt{a}_1 \times \&\mathtt{x}; \mathtt{b} = \mathtt{b}_1 \times \&\mathtt{y}; \mathtt{p} = \mathtt{p}_1 \times \&\mathtt{a}; \mathtt{c} = \mathtt{c}_1 \times (\mathtt{p}_1 + 1);$
$\mathtt{q} = \mathtt{q}_1 \times \mathtt{p}; \mathtt{p}' = \mathtt{p}_1 \times (\mathtt{p}_1 + 1); \mathtt{a}' = \mathtt{a}_1 \times \mathtt{b}.$

Generative constraints: $*\mathtt{p} = \mathtt{b}$.

Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Here, $\mathtt{v}_1$ is the value of the variable $\mathtt{v}$ obtained in iteration 1. Thus the constraints to be solved by the linear solver are:

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 101 + 1, \mathtt{q} = 101 \times \mathtt{p},$

$\mathtt{p}' = 101 \times (101 + 1), \mathtt{a}' = 17 \times 29 \times \mathtt{b}.$

The linear solver offers the following solution.

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 102, \mathtt{q} = 101 \times 101 \times 101,$

$\mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29 \times 29 \times 29.$

The solver returns each value as an integer (e.g., 8381) and not as factors (e.g., $17 \times 29 \times 17$). Our analysis finds the prime factors using the prime-factor-table.

Post-processing over the values starts with *pruning the powers* of the values containing repeated prime factors, as they do not add any additional points-to information to the solution. Thus, we obtain

$\mathtt{a} = 17 \times 29, \mathtt{b} = 29, \mathtt{p} = 101, \mathtt{c} = 102, \mathtt{q} = 101, \mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29.$

The next step is to dereference variables to obtain their points-to sets. Since, 17, 29, and 101 are directly available in prime-factor-table, the values of $\mathtt{a}, \mathtt{b}, \mathtt{p}, \mathtt{q}, \mathtt{a}'$ do not require a dereference. In case of $\mathtt{c}$, 102 is not present in the prime-factor-table, so the next value 101 is searched for, which indeed is present in the table. Thus, $(102 - 101)$ dereferences are done on 101. Further, 101 reverse-maps to $\&\mathtt{a}$ and $\mathtt{a}$ forward-maps to the r-value 17×29. Hence $\mathtt{c} = 17 \times 29$, suggesting that $\mathtt{c}$ points to $\mathtt{x}$ and $\mathtt{y}$.

The value of $\mathtt{p}'$ is an interesting case. The solution returned by the solver (10302) is neither

a prime number, nor a short offset from the product of primes. Rather, it is a product of a prime and a short offset of the prime. We know that it is the value of variable $\mathtt{p}'$ whose original value was $\mathtt{p}_1 = 101$. This original value is used to find out the points-to set contained in value 10302. To achieve this, our method (always) divides the value obtained by the solver by the original value of the variable. Thus, we get $10302/101 = 102$. Our method then applies the dereferencing algorithm on 102 to get its points-to set, which, as explained above for c, computes the value 17×29 corresponding to the points-to set $\{\mathtt{x}, \mathtt{y}\}$. This updates $\mathtt{p}'$ to $101 \times 17 \times 29$.

Such a division is not only required for a load statement, but also for a regular copy statement. This is because of our use of prime-factor-table for finding the factors of a number. If the linear solver assigns a value $\mathtt{x} \times \mathtt{y}$ to a variable v, the value may not be present in the table and we will not know its factors. However, one of x and y would definitely be present (as original value of v prior to the current iteration) and we can obtain both x and y as factors by dividing the value by the original value $\mathtt{v}_\mathtt{i}$ of the variable v.

It should be emphasized that our method never performs the computationally expensive primality testing. It only does a look-up, subtraction (corresponding to a dereference) and division. After prime-factor-table is populated initially with a set of primes as a multiple of the prime and unity, and updated with the factors of the compositions obtained in each iteration of the analysis, a lookup in the table suffices for primality testing. Further, it is easy to verify that the division is integral.

The next step is to evaluate the merging set to obtain the following.

$\mathtt{a} = 17 \times 29 \times 17 \times 29, \mathtt{p} = 101 \times (101 \times 17 \times 29)$.

On pruning this gives

$\mathtt{a} = 17 \times 29, \mathtt{p} = 17 \times 29 \times 101$.

Thus, at the end of the second iteration, the points-to sets are

$\mathtt{x} \rightarrow \{\}, \mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}\}$.

Executing the final step of evaluating the generative constraints, we obtain two additional linear constraints: $\mathtt{x} = \mathtt{b}, \mathtt{y} = \mathtt{b}$.

Following the same process, at the end of the third iteration we get

$\mathtt{x} = 29, \mathtt{y} = 29, \mathtt{a} = 17 \times 29, \mathtt{b} = 29, \mathtt{c} = 17 \times 29, \mathtt{p} = 17 \times 29 \times 101, \mathtt{q} = 17 \times 29 \times 101$

which corresponds to the points-to set

$\mathtt{x} \rightarrow \{\mathtt{y}\}, \mathtt{y} \rightarrow \{\mathtt{y}\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}$

and no new linear constraints are added.

The fourth iteration makes no change to the values of the variables suggesting that a fixed-point solution is reached.

6.4 The Algorithm

First, we describe our approach for field-insensitive, flow-insensitive and context-insensitive points-to analysis. Our points-to analysis is outlined in Algorithm 11. To avoid clutter, we have removed the details of pruning of powers, which is straightforward. The analysis takes a set of constraints `C` and a set of variables `V` used in `C` as inputs. An important data structure is the prime-factor-table which is implemented as a hash-table, mapping a key to a set of prime numbers that form the factors of the key. Insertion of the tuple ($\mathtt{a} \times \mathtt{b}$, `a`, `b`) assumes existence of `a` and `b` in the table (our analysis guarantees that). This is done by combining the prime factors $\in \mathcal{P}$ for `a` and `b` from the table. The product of their unique prime factors is then stored into the table (which may be smaller than $\mathtt{a} \times \mathtt{b}$).

Lines 1–3 initialize each variable to the sentinel value $\chi = 1$. Before iterative procedure begins, the analysis goes over the constraints to evaluate address-of constraints by assigning a distinct prime to each address-taken variable (Lines 5–9). Our algorithm also moves store constraints to the generative constraints set (Lines 10–12) and transforms load constraints (Lines 13–14). Each variable has a two-way mapping to its r-value and address (&`x`). If the variable is an address-taken variable, its address would be represented by a unique odd prime number; otherwise, it would be `1`. Its r-value is a composition of addresses of its pointees. This mapping allows us to dereference variables during post-processing. In the algorithm, for the sake of readability, we use `v` to mean its r-value. As a part of initialization, our algorithm also updates the prime-factor-table to denote that a prime number could not be dereferenced, since one of its factors is `1` (Line 8). This helps us not invoke a primality test for a value. Address-of constraints are removed from `C` prior to the iterative analysis (Line 9).

The iterative analysis begins with making a copy of each variable's r-value in Lines 19–21. In Lines 22–31, the constraint variables are renamed to *use* the copies ($\mathtt{v}_{\mathtt{copy}}$) and to *define* new

Algorithm 11 Points-to analysis as a system of equations

Require: set C of points-to constraints, set V of variables
Ensure: each variable in V has a value indicating its points-to set

```
1:  for all v ∈ V do
2:    v = 1
3:  end for
4:  for each constraint c in C do
5:    if c is an address-of constraint a = &b then
6:      address-of(b) = nextprime()
7:      prime-factor-table.insert(a× address-of(b), a, address-of(b))
8:      a = a× address-of(b);
9:      C.remove(c)
10:   else if c is a store constraint ∗a = b then
11:     generative-constraints.add(c)
12:     C.remove(c)
13:   else if c is a load constraint a = ∗b then
14:     c = constraint(a = b + 1)
15:   end if
16: end for
17:
18: repeat
19:   for all v ∈ V do
20:     v_copy = v
21:   end for
22:   for all c ∈ C of the form v = e do
23:     renamed = defined(v)
24:     if renamed == 0 then
25:       c = constraint(v = v_copy × e)
26:     else
27:       c = constraint(v^renamed = v_copy × e)
28:       merge-constraints.add(constraint(v = v^renamed))
29:     end if
30:     ++defined(v)
31:   end for
32:   V = linear-solve(C)
33:   for all v ∈ V do
34:     v = interpret(v, v_copy, V, prime-factor-table) {Algo. 13}
35:   end for
36:   for all c ∈ merging-constraints of the form v_1 = v_2 do
37:     prime-factor-table.insert(v_1 × v_2, v_1, v_2)
38:     v_1 = v_1 × v_2
39:   end for
40:   for all c ∈ generative-constraints of the form ∗a = b do
41:     S = get-points-to(a, prime-factor-table) {Algo. 12}
42:     for all s ∈ S do
43:       C.add(constraint(s = b))
44:     end for
45:   end for
46: until V == set(v_copy)
```

Algorithm 12 Finding points-to set

Require: Value v, prime-factor-table
1: S = {}
2: P = get-prime-factors(v, prime-factor-table)
3: **for all** p ∈ P **do**
4: S = S∪ {reverse-address(p)}
5: **end for**
6: return S

variables ($\mathtt{v}^{\text{renamed}}$, i.e., $\mathtt{v}', \mathtt{v}''$, ...). When a new variable is defined, a new merge-constraint to unify its value with that of the original variable is generated in Line 28. The mapping `defined(v)` tracks how many times a variable v has been renamed. An initial value of zero indicates that the variable has not been renamed yet. The renaming helps in making the equations consistent. The set of equations thus obtained is then solved using a standard linear solver to get a solution, mapping each variable in V to an r-value (Line 32).

The important step of interpreting the solution is done in Lines 33–35 using Algorithm 13. The algorithm checks for an entry of a variable's value in the prime-factor-table to see if it is a valid composition of primes. If yes, then no dereferencing is required and the value is returned as it is (Lines 3–4 of Algorithm 13). Otherwise, the value is divided by the original value of the variable at the start of the iteration ($\mathtt{v}_{\mathtt{copy}}$). The division is guaranteed to be integral since right hand side of each equation defining v was multiplied by $\mathtt{v}_{\mathtt{copy}}$ (Lines 25 and 27 of Algorithm 11). If the quotient is not found in prime-factor-table then it implies that one or more dereferences are required. A linear downward search from the value of the variable is performed in the prime-factor-table (Lines 11–13). The number of entries visited in the process corresponds to the number of dereferences to be performed. The dereferencing is done by unmapping from the primes corresponding to the addresses of variables and then mapping the variables to their r-values (Line 19 of Algorithm 13). The above procedure is performed for all pointees of the variable (Line 18 of Algorithm 13). The value of *prod* denotes the prime composition of pointees obtained at a dereference level. The composition obtained at the end of this procedure denotes the new pointees computed in the current iteration. This, multiplied by $\mathtt{v}_{\mathtt{copy}}$, is the new value of v.

After interpreting the values, Lines 36–39 of Algorithm 11 evaluate the merging constraints. Merging is done by multiplying the original value of the variable with the new value (Line 38). prime-factor-table is updated to reflect the new product (Line 37). Finally, Lines 40–45 add

Algorithm 13 Interpreting values

Require: Value v, Value v_{copy}, set of variables V, prime-factor-table

```
1:  if v == 1 then
2:    return v
3:  else if v ∈ prime-factor-table then
4:    return v
5:  else if v/v_copy ∈ prime-factor-table then
6:    prime-factor-table.insert(v, v_copy, v/v_copy)
7:    return v
8:  else
9:    v = v/v_copy
10:   k = 1
11:   while (v − k) ∉ prime-factor-table do
12:     ++k
13:   end while
14:   v = (v − k)
15:   for i = 1 to k do
16:     S = get-points-to(v, prime-factor-table) {Algo. 12}
17:     prod = 1
18:     for all s ∈ S and s ≠ 1 do
19:       r = reverse-lvalue(s)
20:       prod = prod × r
21:       prime-factor-table.insert(prod, prod/r, r)
22:     end for
23:     v = prod
24:   end for
25: end if
26: return v × v_copy
```

copy constraints to C by evaluating generative constraints. The expanded set C is then evaluated in the next iteration. The iterative analysis proceeds until r-values of all variables match those of v_{copy} (Line 46) which indicates a fixed-point.

Both Algorithms 11 and 13 make use of Algorithm 12 for computing points-to set of a pointer. It finds the prime factors of the r-value of the pointer (Line 2) followed by an unmapping from the primes to the corresponding variables (Line 4). Note again that the prime factors are obtained from the prime-factor-table and not by any factorization algorithm.

At the end of Algorithm 11, the r-values of variables in V denote their computed points-to sets. C is no longer required. If a client does not need a pointer's points-to set and queries only for alias information of pointers, then the prime-factor-table can also be freed.

6.4.1 Solution Properties

Following properties can be derived based on the values of the pointers.

Property 1: If the r-value of a pointer is a prime, then it is a *must* points-to relation, else it is considered to be a *may* points-to relation.

Property 2: If a pointer has an r-value of `1`, then it is a *non-pointer*. A non-pointer is a variable with empty points-to set [108].

Property 3: If the r-values of two pointers are the same, then they are *pointer-equivalent*. Two pointers are pointer-equivalent if their points-to sets are the same [48].

Property 4: Two variables are *location-equivalent* when address of one variable divides some pointer's r-value and it simultaneously holds that the product of the addresses of the two variables also divides the r-value. Two variables are location-equivalent if both of them always appear together in the points-to sets of pointers, i.e., if one variable is pointed to by a pointer, then the other variable is also pointed to by the same pointer [48].

Pointer-equivalence and location-equivalence are useful to reduce the number of variables tracked during a points-to analysis, and help in making the analysis efficient.

6.4.2 Implementation Issues

Similar to other works on finding linear relationships among program variables [22, 92], our analysis suffers from the issue of large values. Since we store points-to set as a multiplication of primes, the resulting values quickly go beyond the integer range of 64 bits. For very small dereference sizes (upto 16), the transformation works within 64 bits. However, since our analysis is flow-insensitive, the dereference sizes of pointers are well above a few tens. Hence we are required to use an integer library (GNU MP Bignum Library [42]) that emulates integer arithmetic over large unsigned integers. Fortunately, our analysis requires only simple arithmetic operations such as addition, subtraction, multiplication and division which can be implemented acceptably fast.

6.5 Soundness and Precision

Soundness states that our algorithm identifies every points-to fact identified by an inclusion-based analysis. Precision states that our analysis does not compute a (proper) superset of the

information compared to an inclusion-based analysis.

We first prove three essential properties of the solution to the system of linear equations: feasibility, uniqueness and integrality. Feasibility means that the solution exists. Uniqueness means that only one solution exists. Integrality means that the solution is integral. These properties are added to validate our approach of solving a set-based points-to analysis as a system of linear equations.

Property 1: *Feasibility.*

Proof: A system of linear equations, as discussed in Section 6.2.1 has a feasible solution if and only if the corresponding constraint graph has no cycle with negative weight [21]. By renaming the variable occurring in multiple assignments as $\mathtt{a}', \mathtt{a}'', ...$, we guarantee at most one definition per variable. Further, all constants involved in the equations are positive. Thus, there is no negative weight cycle in the constraint graph (see Section 6.2.1). In fact, there is neither a cycle nor a negative weight. This guarantees a feasible solution to the system.

Property 2: *Uniqueness.*

Proof: A variable attains a unique value if it is defined exactly once. In each iteration, which corresponds to one run of the linear solver, the variable renaming creates copies of multiply-defined variables ensuring that each variable is defined only once. Further, our algorithm initializes all the variables to the value of $\chi = \mathtt{1}$. This removes the problem of infinite number of solutions. For instance, let the system have only one constraint: $\mathtt{a} = \mathtt{b}$. In general, this system has infinite number of solutions because `b` is not restricted to any value. In our analysis, we initialize both (`a` and `b`) to `1`. This ensures a unique solution in each iteration of the analysis.

In other words, in each iteration, every equation defines a single variable and no variable is defined multiple times due to renaming; thus, when the system of equations is transformed into a matrix form, it has full row rank. A full row rank indicates that all the rows of the coefficient matrix are linearly independent. Further, by initializing all the undefined variables to $\chi = \mathtt{1}$, our algorithm ensures that there are no free parameters in the system. When the definitions of these originally undefined variables are also added to the system of equations, the coefficient matrix continues to have a full row rank. Thus, the rank of the matrix is equal

to the number of variables. A fundamental theorem in linear algebra is that a solution to a system of equations is unique if and only if its rank equals the number of variables [21]. Thus, in each iteration, our analysis ensures a unique solution to the transformed system of equations.

Property 3: *Integrality.*

Proof: We are solving (and not optimizing) a system of equations that involves only addition, subtraction and multiplication over positive integers ($\mathtt{v_i}$ and constants). Further, each equation is of the form $\mathtt{v} = \mathtt{v_i} \times \mathtt{e}$ where both $\mathtt{v_i}$ and $\mathtt{e}$ are integral. Hence the system guarantees an integral solution.

We now prove soundness and precision of our analysis.

Lemma 1.1: *The analysis in Algorithm 11 is monotonic.*

Proof: Every address-taken variable is represented using a distinct prime number. Second, every positive integer has a unique prime factorization. Thus, as far as the representation of points-to information is concerned, it does not lead to a precision loss. Multiplication by an integer corresponds to including addresses represented by its prime factors. Division by an integer maps to the removal of the unique addresses represented by its prime factors. Multiplying the equations by $\mathtt{v_{copy}}$ in iteration $\mathtt{i}$ (Lines 25 and 27) thus ensures encompassing the points-to set computed in iteration $\mathtt{i-1}$. The division operation is performed only in Algorithm 13 (Line 9) which is guaranteed to be without a remainder. However, the product is restored in Line 26, and hence there is no information loss. Thus, no points-to information is ever killed and we guarantee monotonicity.

Lemma 1.2: *Address-of statements are transformed safely.*

Proof: The effect of address-of statement is computed by assigning the prime number of the address-taken variable to the r-value of the destination variable (Lines 5–9 of Algorithm 11). Since this represents each address-taken variable in a unique manner, the transformation is safe.

Lemma 1.3: *Variable renaming is sound.*

Proof: According to the set-constraint based semantics, for a variable $\mathtt{a}$ and expressions $\mathtt{e_i}$,

statements $\mathtt{a} = \mathtt{e}_1, \mathtt{a} = \mathtt{e}_2, ..., \mathtt{a} = \mathtt{e}_n$ mean $\mathtt{a} \supseteq \mathtt{e}_1$, $\mathtt{a} \supseteq \mathtt{e}_2, ..., \mathtt{a} \supseteq \mathtt{e}_n$ which implies $\mathtt{a} \supseteq (\mathtt{e}_1 \cup \mathtt{e}_2 \cup ... \cup \mathtt{e}_n)$. Renaming gives $\mathtt{a}' = \mathtt{e}_1, \mathtt{a}'' = \mathtt{e}_2, ..., \mathtt{a}'^n = \mathtt{e}_n$ which adds constraints $\mathtt{a}' \supseteq \mathtt{e}_1, \mathtt{a}'' \supseteq \mathtt{e}_2, ..., \mathtt{a}'^n \supseteq \mathtt{e}_n$ which implies $(\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^n) \supseteq (\mathtt{e}_1 \cup \mathtt{e}_2 \cup ... \cup \mathtt{e}_n)$. Merging the variables as $\mathtt{a} = \mathtt{a}', \mathtt{a} = \mathtt{a}'', ..., \mathtt{a} = \mathtt{a}'^n$ adds constraint $\mathtt{a} \supseteq (\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^n)$. By transitivity of $\supseteq$, $\mathtt{a} \supseteq (\mathtt{e}_1 \cup \mathtt{e}_2 \cup ... \cup \mathtt{e}_n)$. Thus, variable renaming is sound.

Corollary 1.1: *Copy statements are transformed safely.*

Lemma 1.4: *Store statements are transformed safely.*

Proof: We define a points-to fact `f` to be *realizable* by a constraint `c` if evaluation of `c` *may* result in the computation of `f`. `f` is *strictly-realizable* by `c` if for the computation of `f`, evaluation of `c` is a *must*. For the sake of contradiction, assume that there is a valid points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ that is strictly-realizable by the store constraint $\mathtt{a} = *\mathtt{p}$ and that does not get computed in our algorithm. Since the store statement, added to the generative constraint set, adds copy constraints $\mathtt{a} = \mathtt{b}_1, \mathtt{a} = \mathtt{b}_2, ..., \mathtt{a} = \mathtt{b}_n$ where $\mathtt{p} \rightarrow \{\mathtt{b}_1, \mathtt{b}_2, ..., \mathtt{b}_n\}$ at the end of an iteration after points-to information computation and interpretation is done, the contradiction means that $\mathtt{x} \notin (*\mathtt{b}_1 \cup *\mathtt{b}_2 \cup ... \cup *\mathtt{b}_n)$. This implies, $(\mathtt{x} \notin *\mathtt{b}_1) \wedge (\mathtt{x} \notin *\mathtt{b}_2) \wedge ... \wedge (\mathtt{x} \notin *\mathtt{b}_n)$. This suggests that the pointee `x` propagates to the pointer `a` via some other constraints, implying that the points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ is not strictly-realizable by $\mathtt{a} = *\mathtt{p}$, contradicting our hypothesis.

Lemma 1.5: *Decomposing an r-value of* `p` *into its prime factors, unmapping the addresses as the primes to the corresponding variables, and mapping the variables to their r-values corresponds to a pointer dereference* $*\mathtt{p}$.

Lemma 1.6: *Load statements are transformed safely.*

Proof: For a `k`-level dereference $*^{\mathtt{k}}\mathtt{v}$ in a load statement, every '*' adds 1 to `v`'s r-value. Thus, for a unique $\mathtt{v} + \mathtt{k}$, the evaluation involves `k` dereferences. Lines 15–24 of Algorithm 13 do exactly this, and by Lemmas 1.3 and 1.5, load statements compute a safe superset.

Theorem 1: *The analysis is sound with respect to an inclusion-based analysis for a dereferencing level k.*

Proof: Follows from Lemma 1.1—1.6 and Corollary 1.1.

Lemma 2.1: *Address-of statements are transformed precisely.*

Proof: Address of every address-taken variable is represented using a distinct prime value. Further, in Lines 5–9 of Algorithm 11, for every address-of statement $\mathtt{a} = \&\mathtt{b}$, the only primes that $\mathtt{a}$ is multiplied with are those corresponding to the addresses of the variables represented by $\mathtt{b}$.

Lemma 2.2: *Variable renaming is precise.*

Proof: Since each variable is defined only once and by making use of Lemma 1.3, $\mathtt{a} = (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$.

Lemma 2.3: *Copy statements are transformed precisely.*

Proof: From Lemma 2.2 and since for a transformed copy statement $\mathtt{a} = \mathtt{a_{copy}} \times \mathtt{b}$, only the primes computed as the points-to set of $\mathtt{a}$ so far (i.e., $\mathtt{a_{copy}}$) and those of $\mathtt{b}$ are included. This inclusion is guaranteed to be unique due to the uniqueness of prime factorization. Thus, the analysis does not include any spurious pointees for $\mathtt{a}$.

Lemma 2.4: *Store statements are transformed precisely.*

Proof: We prove this by contradiction. Assume that a points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ is computed spuriously by evaluating a store constraint $\mathtt{a} = *\mathtt{p}$ in Algorithm 11. This means at least one of the following copy constraints computed the fact: $\mathtt{a} = \mathtt{b_1}, \mathtt{a} = \mathtt{b_2}, ..., \mathtt{a} = \mathtt{b_n}$ where $\mathtt{p} \rightarrow \{\mathtt{b_1}, \mathtt{b_2}, ..., \mathtt{b_n}\}$. Thus, at least one of the copy constraints is imprecise. However, Lemma 2.3 falsifies the claim.

Lemma 2.5: *Load statements are transformed precisely.*

Proof: The number of dereferences denoted by $\mathtt{v} + \mathtt{k}$ is the same as that denoted by $*^{\mathtt{k}}\mathtt{v}$. By Lemma 1.5 and 2.3 and by the observation that Algorithm 13 does not include any extra pointee in the final dereference set.

Theorem 2: *The analysis is precise with respect to an inclusion-based analysis for a dereferencing level k.*

Proof: Follows from Lemma 2.1–2.5.

Algorithm 14 Alias query

Require: Pointers p, q
1: v_p = get-rvalue(p)
2: v_q = get-rvalue(q)
3: return gcd(v_p, v_q) $\neq 1$

Theorem 3: *Our analysis computes the same information as an inclusion-based points-to analysis for a dereferencing level k.*

Proof: Immediate from Theorems 1 and 2.

6.6 Client Analysis

In this section, we illustrate how our prime-factorization based points-to analysis can be effectively used in some of the client analyses.

Several clients (e.g., constant propagation, parallelism extractors, etc.), which use points-to analysis, query for alias information in the form *alias*(p, q). An alias query *alias*(p, q) returns true if pointers p and q share any pointee, and false otherwise. Using our prime-factorization approach, this query can be easily answered by finding the greatest common divisor (GCD) of values v_p and v_q, which are the integral values computed by our points-to analysis algorithm for pointers p and q. If the GCD is 1, the pointers do not alias; otherwise, p and q have a common (prime) factor and the pointee denoted by the factor is pointed by both p and q. Hence, they alias. Algorithm 14 outlines the alias query.

It works because GCD corresponds to the common pointees across the two pointers. This can be easily observed from the lattice of values (Figure 6.2).

The second example that we consider is the Mod/Ref analysis, which we used as a client analysis in Chapter 4. As mentioned in Section 4.7, the Mod/Ref analysis checks if a function call reads or writes to a (symbolic) memory location. A useful form of Mod/Ref analysis is to find out which global (or heap) variables are only referenced and not modified by any pointers in the program. This helps in determining the read-only globals and is immensely helpful in parallelization of programs. A first step towards this goal is to find all pointers pointing to a given set of (global or heap) variables. Searching for the variables of interest in a sparse pointee list of a pointer would be costly. In contrast, if the underlying representation is using a

composition of primes to represent points-to information, then we simply need to check whether for each variable `v` of interest, the address-value of `v` divides the r-value of the pointer. Thus, a representation based on prime factorization helps in speeding up the client analyses (offset by the cost of emulating large integer arithmetic).

The third example that we consider is program slicing [126]. To compute a slice with respect to a variable `v` at a program point, one may encounter a statement with a pointer dereference `*p`. Although slicing is typically flow-sensitive, the pointer analysis can be flow-insensitive and check whether the pointer dereference may point to `v`. If the points-to information is stored as proposed above, then one simply needs to check whether the r-value of `p` is divisible by the address (prime number) of `v` which is much faster compared to looking up a pointee `v` in the sorted points-to set of `p`. Thus, prime-factorization is an attractive alternative for storing points-to information.

6.7 Experimental Evaluation

We evaluate the effectiveness of our approach using the same set of benchmarks that we used in Chapter 4, namely, a set of 16 SPEC C/C++ benchmarks and five large open source programs, *httpd, sendmail, ghostscript, gdb* and *wine-server*. As before, we have implemented our method in the LLVM compiler framework [81] as a post-linking phase (refer to the block diagram in Figure 4.5 in Chapter 4). When run on an input program, our method generates a set of linear equations from the program's points-to constraints. For solving the equations, we use C++ language extension of CPLEX® solver from IBM ILOG toolset [61]. A reader may be interested to know that we did not get good performance out of other linear solvers like Matlab. All the experiments are carried out on the same platform, with an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM.

We compare our approach, referred to as *linear*, with the following implementations. We consider both context-sensitive (*cs*) and context-insensitive (*ci*) versions.

- *anders*: This is the base Andersen's algorithm [3] which we used to assess the effectiveness of our bloom-filter based approach (Chapter 4, Section 4.5).

- *bddlcd*: This is the *Lazy Cycle Detection* (LCD) algorithm implemented using Binary Decision Diagrams (BDD) from Hardekopf and Lin [49] which we used to assess the

effectiveness of our bloom-filter based approach (Chapter 4, Section 4.5)

- *bloom*: The bloom filter method (as described in Chapter 4) stores context-sensitive points-to information in a bloom filter. We recall here that this representation is approximate and results in false-positives and introduces some loss in precision. For our experiments, we use the *medium* configuration which results in roughly 3% of precision loss for the chosen benchmarks.

All the above implementations are considered in both context-insensitive (*ci*) and context-sensitive (*cs*) forms. The context-sensitive version is implemented as discussed in Chapter 4 in Section 4.4. We also compare a context-insensitive, flow-insensitive and field-insensitive version of our prime-factorization based analysis (referred to as *linear-ci*) with Deep Propagation [100].

- *deep*: This is the context-insensitive deep-propagation method from [100]. This method propagates points-to information in the constraint graph to all the reachable nodes along a path, before the other paths are considered. It uses a sparse bitmap representation to store points-to sets.

6.7.1 Analysis Time

The analysis times in seconds required for each benchmark by different methods are given in Table 6.1 for context-insensitive analysis and in Table 6.2 for context-sensitive analysis. The analysis time is composed of reading an input points-to constraints file, applying the analysis over the constraints and computing the final points-to information as a fixed-point.

Context-insensitive analysis. As seen from Table 6.1, the timing numbers for context-insensitive version of our approach are moderately low, requiring 15 seconds per benchmark on an average and a maximum of 143 seconds for *gdb*. More specifically, *linear-ci* outperforms the *exact* versions *anders-ci, bddlcd-ci* and *deep-ci*. *linear-ci* is 5× faster than *anders-ci*, 3.7× faster than *bddlcd-ci* and 2.8× faster than *deep-ci*. Although *bloom-ci* performs significantly better than *linear-ci*, *linear-ci* is more precise than *bloom-ci* as *bloom-ci* is an approximate analysis. A large part (around 70%) of the analysis time in *linear-ci* is spent in solving the equations in different iterations. This suggests that optimizing the constraint solving in the

Benchmark	anders-ci	bddlcd-ci	bloom-ci	deep-ci	linear-ci
gcc	151.618	5.154	3.028	1.740	13.243
perlbmk	65.969	3.078	1.374	1.744	5.863
vortex	1.457	2.282	0.964	0.116	2.850
eon	29.625	2.339	1.209	11.701	6.172
parser	0.831	2.020	0.477	0.176	1.715
gap	6.689	2.527	1.148	0.092	4.576
vpr	0.465	1.290	0.443	0.024	0.750
crafty	0.453	1.528	0.451	0.004	0.551
mesa	1.029	2.231	0.985	0.248	2.657
ammp	0.372	1.347	0.365	0.032	0.651
twolf	0.614	2.056	0.543	0.032	1.250
gzip	0.221	0.955	0.228	0.004	0.158
bzip2	0.199	0.889	0.208	0.004	0.125
mcf	0.175	1.228	0.183	0.004	0.153
equake	0.176	0.856	0.185	0.004	0.057
art	0.167	0.643	0.177	0.004	0.176
httpd	58.624	1.856	0.824	53.727	21.358
sendmail	37.276	1.521	0.796	12.729	8.126
ghostscript	425.362	343.579	4.636	207.030	95.665
gdb	852.622	758.473	6.221	587.829	142.583
wine-server	62.545	45.512	2.992	8.165	6.339
average	80.785	56.255	1.307	42.162	15.001

Table 6.1: Time required in seconds for context-insensitive analysis

linear solver may help in improving the analysis time of *linear-ci* further.

Context-sensitive analysis. From Table 6.2, we observe that *linear* outperforms both *anders* and *bddlcd*: *linear* is 1.76× faster than *anders* and almost 9× faster than *bddlcd*. Most of the time lost by *bddlcd* is due to the complex BDD operations that need to be performed for maintaining points-to information. *bloom* is 2× faster than *linear* and its higher performance is attributed to fast hash functions used to store the context-sensitive points-to information, instead of going over the absolute contexts. Further, the spacial locality offered by a *multi-bloom* greatly enhances the analysis speed. However, it should be noted here that *bloom* is an approximate analysis and has around 3% precision loss in these applications (Chapter 4) compared to *anders*, *bddlcd* and *linear*. We believe that the analysis time of *linear* can be further improved by taking advantage of sharing of tasks across iterations and by exploiting properties of simple linear equations in the linear solver.

Benchmark	Time(sec)				Memory(MB)			
	anders	bddlcd	bloom	linear	anders	bddlcd	bloom	linear
gcc	329.5	17411.2	137.1	189.7	2859	2534	669	2473
perlbmk	143.4	5879.9	85.4	124.6	2133	1723	314	2041
vortex	91.3	4725.7	71.4	66.3	1857	1358	152	1741
eon	93.5	2391.8	74.0	102.7	1276	1425	299	942
parser	35.4	618.3	29.2	53.2	478	345	118	434
gap	128.5	330.2	85.1	91.4	457	362	240	452
vpr	29.5	199.5	16.1	46.2	735	692	93	673
crafty	29.3	155.0	17.6	50.7	672	566	77	646
mesa	89.4	21.7	63.2	62.5	894	729	163	625
ammp	34.2	54.6	22.5	23.3	427	336	81	413
twolf	41.5	27.4	31.2	24.9	624	617	119	652
gzip	25.2	6.5	18.5	22.1	514	522	55	642
bzip2	23.3	4.7	19.6	21.6	633	588	56	558
mcf	22.4	32.0	16.3	23.4	403	389	48	489
equake	24.3	4.1	13.8	20.9	546	527	56	579
art	26.5	7.7	11.5	21.3	597	582	49	522
httpd	224.5	47.4	38.8	86.4	791	825	596	814
sendmail	172.7	117.5	15.9	91.6	914	851	306	922
ghostscript	4384.2	20612.8	1959.8	3144.2	1958	1672	971	1852
gdb	9338.2	24871.7	2362.6	4384.4	2194	1859	1461	2259
wine-server	201.3	36.7	68.0	142.5	774	690	279	637
average	737.5	3693.2	245.6	418.8	1035	914	295	970

Table 6.2: Time(seconds) and memory(MB) required for context-sensitive analysis

6.7.2 Memory

The memory requirement in MB of each analysis for the set of benchmarks is given in Table 6.3 for context-insensitive analysis and in Table 6.2 for context-sensitive analysis.

Context-insensitive analysis. From Table 6.3, it can be observed that our approach *linear-ci* requires 131 MB on an average (maximum 924 MB for *gdb*) which is quite reasonable for today's desktops/servers. Although *bddlcd-ci* and *bloom-ci* implementations require substantially less amount of memory than *linear-ci*, we observe that *linear-ci* requires 1.8× less memory than *anders-ci*, and 2.9× less memory than *deep-ci*. Major memory saving in *linear-ci* happens as the approach does not store the points-to sets explicitly. The mappings and the reverse mappings between variables, their r-values and address-values are the data structures that consume a large portion of the memory (around 60%). The prime-factor-table consumes most of the remaining space.

Benchmark	anders-ci	bddlcd-ci	bloom-ci	deep-ci	linear-ci
gcc	1269	27	6	83	125
perlbmk	669	17	3	100	97
vortex	10	18	2	16	24
eon	383	64	3	248	137
parser	8	3	1	4	7
gap	87	6	2	8	7
vpr	1	1	1	2	2
crafty	1	1	1	1	5
mesa	3	4	2	14	16
ammp	1	2	1	3	2
twolf	3	3	1	4	3
gzip	1	2	1	1	2
bzip2	1	2	1	1	2
mcf	1	1	1	1	1
equake	1	1	1	1	2
art	1	1	1	1	1
httpd	469	49	5	674	327
sendmail	353	64	5	256	183
ghostscript	547	177	6	2871	672
gdb	631	218	6	3556	924
wine-server	444	94	4	185	222
average	233	36	3	382	131

Table 6.3: Memory required in MB for context-insensitive analysis

Context-sensitive analysis. From Table 6.2, we observe that our approach *linear* requires slightly (6%) more memory than *bddlcd*. The *bddlcd* method, which is known for its space efficiency, requires 914 MB on an average. *linear* also requires around 6% less memory than *anders*. The *bloom* method, with its lossy representation, uses the minimum amount of memory.

Thus, we observe that *linear* offers benefits both in terms of analysis time and memory requirement and helps in achieving a scalable implementation.

6.7.3 Comparison with Bitwise Operations[3]

In order to assess the effect of a standard linear solver for solving points-to constraints, we implemented a variant called *nrec*. *nrec* is a version which is a mix of *linear* and *anders*. It converts the initial set of constraints (except the store constraints) into a non-recursive SSA-like

[3]We acknowledge the reviews of the anonymous reviewer who suggested this comparison.

Benchmark	Context-insensitive				Context-sensitive			
	Time(sec)		Memory(MB)		Time(sec)		Memory(MB)	
	linear-ci	nrec-ci	linear-ci	nrec-ci	linear-cs	nrec-cs	linear-cs	nrec-cs
gcc	13.243	8.322	125	274	189.7	193.3	2473	2743
perlbmk	5.863	3.509	97	152	124.6	132.3	2041	2558
vortex	2.850	2.692	24	32	66.3	68.2	1741	1866
eon	6.172	7.835	137	189	102.7	95.1	942	977
parser	1.715	1.141	7	8	53.2	53.2	434	446
gap	4.576	3.604	7	9	91.4	81.9	452	481
vpr	0.750	1.391	2	2	46.2	37.3	673	703
crafty	0.551	0.436	5	5	50.7	43.6	646	704
mesa	2.657	1.949	16	20	62.5	58.3	625	663
ammp	0.651	0.423	2	2	23.3	22.6	413	435
twolf	1.250	0.887	3	3	24.9	22.1	652	692
gzip	0.158	0.135	2	1	22.1	22.0	642	670
bzip2	0.125	0.073	2	1	21.6	18.4	558	595
mcf	0.153	0.086	1	1	23.4	22.0	489	490
equake	0.057	0.095	2	1	20.9	19.6	579	585
art	0.176	0.147	1	1	21.3	20.9	522	536
httpd	21.358	16.598	327	427	86.4	73.8	814	853
sendmail	8.126	5.744	183	215	91.6	85.9	922	1201
ghostscript	95.665	57.363	672	806	3144.2	3216.3	1852	2052
gdb	142.583	102.526	924	1027	4384.4	4492.1	2259	2758
wine-server	6.339	6.472	222	274	142.5	126.3	637	656
average	15.001	10.544	131	164	418.8	424.0	970	1077

Table 6.4: Time and memory comparison with bitwise operations

format, creating generative constraints as in *linear*. These non-recursive constraints are then evaluated similar to *anders* to generate new points-to information. As part of post-processing, the generative constraints are evaluated to merge the points-to information for a variable from its copies and to generate more constraints using store constraints. This process is repeated until a fixed-point. The points-to information is stored in sparse bitmaps. Comparing against *nrec* enables us to better evaluate the effect of using a linear solver in *linear*.

Time and memory requirements of *linear* and *nrec* are given in Figure 6.4. In case of context-insensitive analysis, we find that *nrec-ci* outperforms *linear-ci* and is around 30% faster. This suggests that although solving a set of linear equations using an external solver is practically fast, it has overheads, which can be superseded by a standard constraint solver. However, as evident from the table, the memory requirement of *nrec-ci* is also more by 25% than that of *linear-ci*. Once again, the major memory savings in case of *linear-ci* occur due to not storing

the points-to information explicitly, offset by the additional storage requirement to represent big integers.

In contrast, in case of context-sensitive analysis, the behavior of benchmarks differs across the two methods. For most of the smaller benchmarks (e.g., *vpr, crafty, mesa, ammp, ...*) *nrec-cs* requires slightly lesser time than that required by *linear-cs*. However, for the large benchmarks (e.g., *gcc, perlbmk, vortex, ghostscript* and *gdb*), the analysis time of *nrec-cs* is considerably more than that required by *linear-cs*. It is interesting to see that for the same set of large benchmarks, the memory requirement of *nrec-cs* is also considerably high. These observations suggest that *linear* is better suited for context-sensitive analysis of large programs. For context-sensitive analysis, the cost of solving the constraints is much higher compared to the cost of invoking an external linear solver. Although the exact reason for better performance of the IBM ILOG solver would require drilling down the proprietary solver code, the solver's documentation [61] also confirms that the efficiency of the solver improves as the data-set increases in size. We believe that the solver is able to take advantage of the simple equations and perform optimizations, which are not being done by *nrec*. The advantage of these optimizations becomes visible when the data-set size is large, which happens in case of context-sensitive analysis of large programs. The reason for higher memory requirement of *nrec* is a clearly due to different data representation. The results clearly indicate that storing large points-to sets as a single (big) integer is storage-efficient compared to a bitmap representation.

6.8 Related Work

We formulated the points-to analysis problem as that of solving a system of linear equations. An important use of linear algebra in program analysis has been to compute affine relations among program variables [92]. They describe an interprocedural flow-sensitive analysis which determines identities that are valid among the program variables whenever control reaches a program point. Their analysis computes all polynomial relations of bounded degree precisely in time linear in the program size and polynomial in the number of occurring variables. Cousot and Halbwachs [22] applied abstract interpretation for discovering equality or inequality constraints among program variables. However these methods are not applicable to pointer dereferences. Fecht and Seidl [37] proposed an SML (Standard Markup Language) [119] based

solver for computing a partial approximate solution for a general system of equations used in logic programs. Their analysis considers as few variables as necessary to compute the values of variables of interest and assumes no specific properties of the right-hand-side expression of an equation. Another area where analyses based on linear systems has been used is in finding security vulnerabilities. Ganapathy et al. [39] proposed a context-sensitive light-weight analysis modeling string manipulations as a linear program to detect buffer-overrun vulnerabilities. Dor et al. [29] presented a tool C String Static Verifyer (CSSV) to find string manipulation errors. CSSV converts a program written in a restricted subset of C into an integer program with assertions. A violation of an assertion signals a possible vulnerability. It uses Das's one-level flow algorithm [24] to perform pointer analysis. In another method, Esparza et al. [33] proposed Newtonian Program Analysis as a generic framework to solve iterative program analyses using Newton's method. They propose Newton's method as an efficient alternative to Kleene's iterative method of finding a join-over-all-paths (JOP) solution to the system of linear equations over basic blocks.

To the best of our knowledge, our work is the first one modeling points-to analysis as a system of linear equations and using prime factorization to store the points-to information.

We empirically showed that our context-sensitive analysis is, on an average, 8.8× faster than BDD-based Lazy Cycle Detecton [49] and is 1.8× faster than an optimized Andersen's analysis [3]. Further, our context-insensitive analysis is, on an average, 3.7× faster than BDD-based Lazy Cycle Detecton [49], 5.3× faster than an optimized Andersen's analysis [3], and 2.8× faster than Deep Propagation [100]. In case of memory requirement, we find that our context-sensitive analysis requires almost the same memory as Andersen's analysis [3], whereas the context-insensitive version of our analysis, on an average, requires 1.8× less memory than Andersen's analysis [3], 2.9× less memory than Deep Propagation [100], but 3.6× more memory than the BDD-based analysis [49].

6.9 Chapter Summary

In this chapter, we proposed a novel approach to transform a set of points-to constraints into a system of linear equations using prime factorization. We overcame the technical challenges by partitioning our inclusion-based analysis into a linear solver phase and a post-processing

phase that interprets the resulting values and updates points-to information accordingly. The novel way of representing points-to information as a composition of primes allowed us to keep the equations linear in every iteration. We show that our analysis is sound and precise with respect to an inclusion-based analysis for a fixed dereference level. Using a set of 21 programs, we showed that our approach is not only feasible, but is also competitive to the state-of-the-art solvers. More than the performance numbers reported here, the main contribution of this work is the novel formulation of points-to analysis as a linear system based on prime factorization.

Chapter 7

Prioritizing Constraint Evaluation for Efficient Points-to Analysis

7.1 Introduction

It is well-known that for a flow-insensitive points-to analysis, the order in which the points-to constraints are evaluated does not affect the fixed-point computed. In this chapter we study the effect of evaluating constraints in a particular order to reach the same fixed-point. Our goal is to devise a dynamic constraint ordering to compute the fixed-point of the points-to information as efficiently as possible.

A flow-insensitive analysis iterates over a set of points-to constraints until a fixed-point is obtained. Typically, the flow of points-to information is represented using a constraint graph `G`, in which a node denotes a pointer variable and a directed edge from node `n1` to node `n2` represents propagation of points-to information from `n1` to `n2`. Each node is initialized with the points-to information computed by evaluating the *address-of* constraints. Edges are added to `G` initially by *copy* constraints and then by *complex* (*load* and *store*) constraints as the analysis progresses. This is because the edges introduced by *complex* constraints depend upon the availability of points-to information at nodes which, in turn, depends upon the propagation. Thus, as the analysis performs an iterative progression of the points-to information propagation, new edges get introduced in `G` due to the evaluation of the *complex* constraints, resulting in the computation of more and more points-to information at its nodes. When no more edges and no more points-to information can be computed, `G` stabilizes and a fixed-point (points-to

Algorithm 15 Points-to Analysis using Constraint Graph

Require: set `C` of points-to constraints
1: Process address-of constraints
2: Add edges to constraint graph G using copy constraints
3: **repeat**
4: Propagate points-to information in G
5: Add edges to G using load and store constraints
6: **until** `fixed-point`

information at the nodes) is reached. The information can then be used by various clients. An outline of this analysis is given in Algorithm 15.

Techniques have been developed for efficient *propagation* of the points-to information across the edges of a constraint graph, i.e, Line 4 of Algorithm 15. Online cycle elimination [34] detects cycles in `G` on-the-fly and collapses all the nodes in a cycle into a representative node. Cycle collapsing is possible because all the nodes in a cycle eventually contain the same points-to information. This significantly reduces the number of pointers tracked and speeds up the overall analysis. Wave and Deep Propagation [100] techniques perform a topological ordering of the edges and propagate difference in the points-to information in breadth-first or depth-first manner respectively. These propagation orders significantly improve the analysis times. In yet another method, various heuristics like Greatest Input Rise, Greatest Output Rise, and Least Recently Fired [67] work on the amount and recency of information computed at various nodes in the constraint graph to achieve a quicker fixed-point.

All of the above techniques essentially focus on the propagation order (Line 4 of Algorithm 15) and prioritize the order in which the points-to propagation takes place. In other words, these techniques work on the constraint graph *after* an edge is added. However, these techniques do not attempt to dictate which evaluation order of the constraints (Line 5 of Algorithm 15) would prove more beneficial for faster points-to information computation. Specifically, there are two aspects of the constraint evaluation that are hitherto not exploited in literature: (i) how many edges a constraint adds, and (ii) where in `G` a constraint adds edges. We observe that both these parameters are important and can significantly influence the speed of convergence of the fixed-point computation. Intuitively, an analysis should give more priority to a constraint that adds *more edges* and that adds *edges which appear early in the topological ordering* of `G`. A constraint that adds more edges provides more opportunities for points-to

information propagation. A constraint that adds edges *earlier* in the constraint graph reduces the number of unnecessary propagations of the points-to information compared to the edges that appear *later* in the topological ordering. We address both of these aspects using a single heuristic in this chapter. It should be noted that neither the propagation order nor the constraint evaluation order changes the fixed-point of the points-to solution. Thus, the analysis precision is not affected by these techniques.

We develop a framework that deals with the priority ordering of the points-to constraints and the propagation of points-to information. The two criteria mentioned above give rise to a priority assigned to each constraint. The priority is dynamic in nature and can change as the analysis progresses. Our prioritized analysis not only evaluates constraints in the priority order, but also evaluates certain constraints repeatedly based on priority. The result is a skewed evaluation of important and useful constraints early and in a repeated manner to reach the fixed-point solution faster.

The chapter is organized as below. We first prove that finding a sequence of the points-to constraints that ensure reaching the fixed-point in an optimal number of steps in a flow-insensitive inclusion-based analysis is NP-Complete (Section 7.2). We then explain our priority-based greedy technique using an example (Section 7.3). In Section 7.4, we develop a priority based analysis framework. The framework is general and can be used for other static analyses. In Section 7.5, we instantiate our framework by defining *constraint priority* based on the structure of and the number of points-to facts changed by a constraint. Our constraint priority framework is quite generic and can be applied to different points-to analyses. In Section 7.6, we evaluate the effectiveness of our approach by applying it on top of the state-of-the-art algorithms (Andersen's analysis [3], BDD-based Lazy Cycle Detection [49], Deep Propagation [100] and Bloom Filters (Chapter 4)) for our benchmark suite. We discuss related work in Section 7.7. We conclude the chapter with a summary in Section 7.8.

7.2 Optimal Constraint Ordering

Given our observation that prioritizing the processing of points-to constraints in Line 5 of Algorithm 15 may improve the analysis time, it raises the question: *does there exist an order in which the constraints should be processed which can ensure optimal number of steps for*

reaching the fixed-point? For instance, given a set of points-to constraints

$\mathtt{a} = \&\mathtt{x}; \mathtt{b} = \&\mathtt{y}; \mathtt{p} = \&\mathtt{q}; *\mathtt{q} = \mathtt{b}; *\mathtt{p} = \mathtt{a}$,

it takes two iterations to reach the fixed-point (third iteration is required to confirm the fixed-point), if they are processed in the above order. The processing for each iteration is given in the following table.

Constraint	New points-to information added in iteration 1	New points-to information added in iteration 2
$\mathtt{a} = \&\mathtt{x}$	$\mathtt{a} \rightarrow \mathtt{x}$	
$\mathtt{b} = \&\mathtt{y}$	$\mathtt{b} \rightarrow \mathtt{y}$	
$\mathtt{p} = \&\mathtt{q}$	$\mathtt{p} \rightarrow \mathtt{q}$	
$*\mathtt{q} = \mathtt{b}$		$\mathtt{x} \rightarrow \mathtt{y}$
$*\mathtt{p} = \mathtt{a}$	$\mathtt{q} \rightarrow \mathtt{x}$	

However, processing the above set of constraints in the following order ensures fixed-point in one iteration (second iteration is required to confirm the fixed-point).

$\mathtt{a} = \&\mathtt{x}; \mathtt{b} = \&\mathtt{y}; \mathtt{p} = \&\mathtt{q}; *\mathtt{p} = \mathtt{a}; *\mathtt{q} = \mathtt{b}$.

The processing is given in the following table.

Constraint	New points-to information added in iteration 1
$\mathtt{a} = \&\mathtt{x}$	$\mathtt{a} \rightarrow \mathtt{x}$
$\mathtt{b} = \&\mathtt{y}$	$\mathtt{b} \rightarrow \mathtt{y}$
$\mathtt{p} = \&\mathtt{q}$	$\mathtt{p} \rightarrow \mathtt{q}$
$*\mathtt{p} = \mathtt{a}$	$\mathtt{q} \rightarrow \mathtt{x}$
$*\mathtt{q} = \mathtt{b}$	$\mathtt{x} \rightarrow \mathtt{y}$

Since existing techniques decouple propagation and evaluation of the *complex* constraints, they do not reorder the constraints, which results in requiring multiple iterations to reach the fixed-point.

In this section, we prove that computing such a sequence even for a restricted scenario where only copy constraints are allowed is NP-Complete.

Theorem 1. *Computing the flow-insensitive inclusion-based points-to solution in an optimal number of steps from a set of copy constraints is NP-Complete.*

Proof. In order to prove this, we reduce Set Cover problem [21] to points-to analysis. Consider a Set Cover instance SC($\mathcal{U}, \mathcal{S}$, K) with universe $\mathcal{U}$ and a set $\mathcal{S}$ of subsets $\mathcal{S}_i$. The decision version of the Set Cover problem states that given a universe $\mathcal{U}$ and a set $\mathcal{S}$ with subsets $\mathcal{S}_i$ possibly having elements in common, whether there exists a set of K subsets whose union contains all the elements contained in any $\mathcal{S}_i$. Formally, given $\mathcal{U}$ and $\mathcal{S}$, a cover is a subfamily $\mathcal{F} \subseteq \mathcal{S}$ of sets whose union is $\mathcal{U}$. The problem SC($\mathcal{U}, \mathcal{S}$, K) is known to be NP-Complete [21].

We reduce SC($\mathcal{U}, \mathcal{S}$, K) to PTA(C, S, K) which is a flow-insensitive points-to analysis over a set of points-to (copy) constraints C with an initial points-to information and the fixed-point defined with respect to pointer S. The decision version of PTA checks whether the constraints in C can be evaluated in a manner such that the fixed-point with respect to S is reached in K steps. A step indicates evaluation of a constraint.

The reduction is performed as below. For each element $s \in \mathcal{S}_i$, we create an initial points-to information $\mathcal{S}_i \rightarrow \{s\}$, i.e., $\mathcal{S}_i$ points-to s. For each set $\mathcal{S}_i$, we create a copy statement $\mathcal{S} = \mathcal{S}_i$. Note that there are no dependence cycles ($\mathtt{a} = \mathtt{b}, \mathtt{b} = \mathtt{a}$) in the constraints and the fixed-point can be obtained without iterating over the constraints. This transformation from SC($\mathcal{U}, \mathcal{S}, K$) to PTA($C$, S, K) is linear in the number of sets and the number of elements. Thus, SC polynomially reduces to PTA.

If an efficient (polynomial) solution exists for PTA, then the solution can be mapped back to SC. Thus, if there is a sequence of K steps to obtain the fixed-point, and since the fixed-point would contain all the points-to information, then the subsets $\mathcal{S}_i$ corresponding to the chosen copy constraints ($\mathcal{S} = \mathcal{S}_i$) would cover all the elements $s \in \mathcal{U}$ that correspond to the points-to facts in the fixed-point, forming the set cover.

Similarly, if a set cover of size K exists, then no other subset would be able to add any new points-to information to $\mathcal{S}$ and the K subsets would form an optimal sequence of K steps to obtain the fixed-point over the constraints.

Thus, PTA is NP-Hard. (a)

It is easy to see that a given sequence of K constraints can act as a polynomial time verifier to check if PTA evaluates to a fixed-point. Thus, PTA is in NP. (b)

From (a) and (b), PTA is NP-Complete. □

7.3 Prioritized Computation of Constraints

Since finding an optimal constraint ordering is NP-Complete, we propose a greedy heuristic, based on constraint priority, for efficient computation of points-to constraints. We first explain our priority-based approach using an example, then discuss various priority schemes followed by our prioritization framework. Our priority based approach may be viewed similar to the maximum benefit approach in the online set cover problem [5].

7.3.1 Example

Consider the program fragment given in Figure 7.1(a). Let the initial points-to information due to the address-of constraints be $\texttt{a} \rightarrow \{\texttt{a}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}\}$ and $\texttt{p} \rightarrow \{\texttt{b}, \texttt{c}, \texttt{d}\}$. Figure 7.1(b) illustrates the constraint graph `G` at the end of different iterations of Deep Propagation [100]. As described in Algorithm 15, Deep Propagation uses the constraint graph with the propagation order similar to a depth-first search mechanism, i.e., points-to information is propagated along the complete subtree (or graph) starting from a child of a node before another child of the node is considered. For simpler exposition, we assume that online cycle elimination [34] is not performed.

A node is represented as $\texttt{n}\{\texttt{P}\}$ where `n` is a pointer and $\{\texttt{P}\}$ is its points-to set computed so far. Directed edge from node $\texttt{n}_1$ to $\texttt{n}_2$ represents the propagation of points-to information from $\texttt{n}_1$ to $\texttt{n}_2$. As the behavior of the pointers $\texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}$ is the same in this example, we use a single node to represent all of them. Prior to Iteration 1, edges `d` to `e` and `a` to `b` are added to `G` using copy constraints (refer Algorithm 15). Iteration 1 starts with propagating the points-to set {`a,q,r,s,t`} from node `a` to `b`. Since the points-to set of `d` is empty, no information flows to `e`. The next step of processing *complex* constraints adds the edges as indicated below.

`*e = c`: none.

`c = *a`: `a` to `c`, `qrst` to `c`.

`*a = p`: `p` to `a`, `p` to `qrst`.

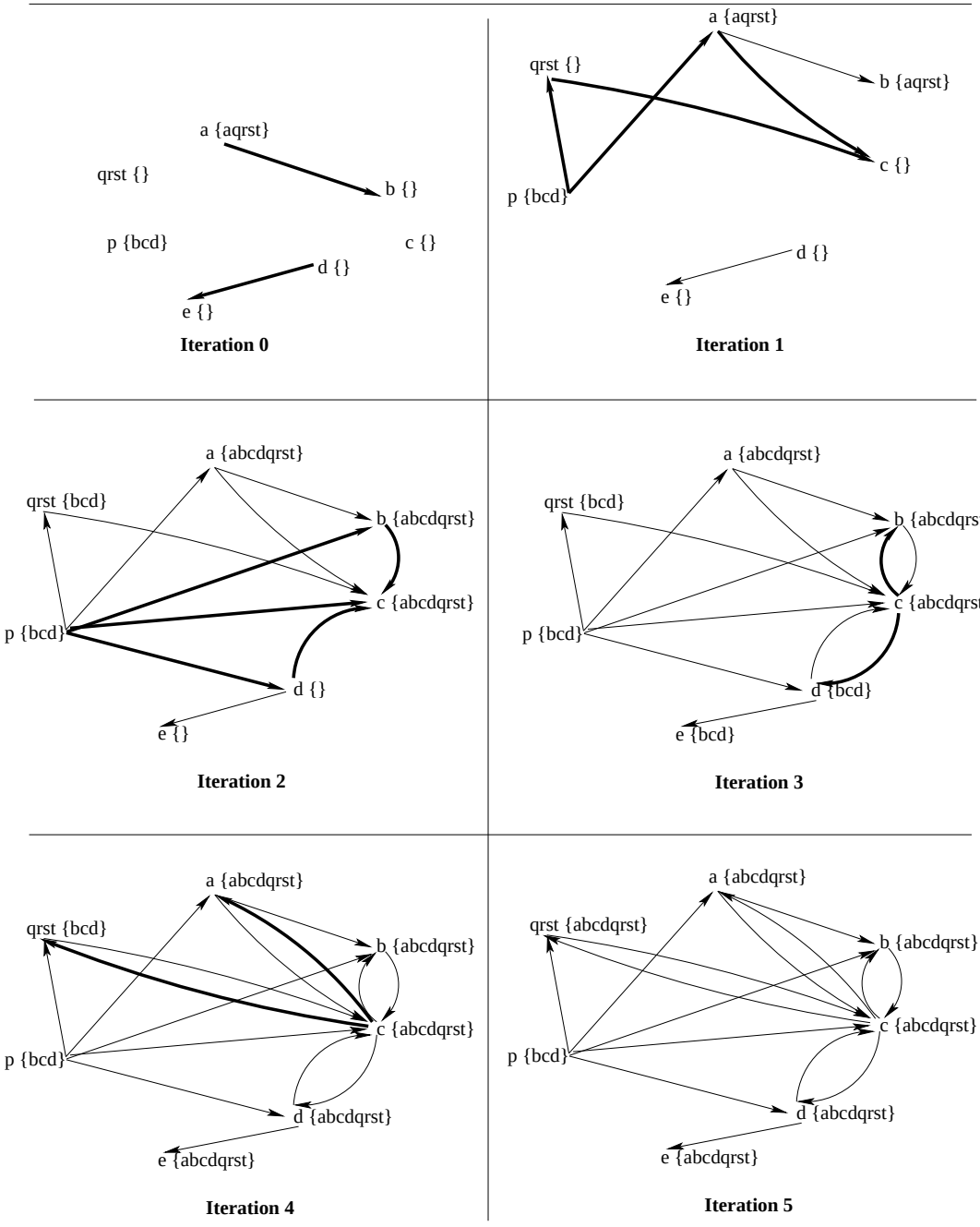
The new edges introduced in this iteration are shown as thick lines. Thus, at the end of the first iteration, the points-to information computed at different nodes as well as the constraint graph are shown in Figure 7.1(b), Iteration 1.

The analysis continues propagating more points-to information and then adding more edges (shown as thick lines) in each iteration until it reaches the fixed-point in Iteration 5. The final

Input constraints *e = c, c = *a, e = d, b = a, *a = p

Fixed processing order e = d, b = a; *e = c, c = *a, *a = p

(a)



(b) **Deep Propagation**

Figure 7.1: (a) Input constraints and fixed constraint ordering for Deep Propagation (b) Constraint graphs for Deep Propagation

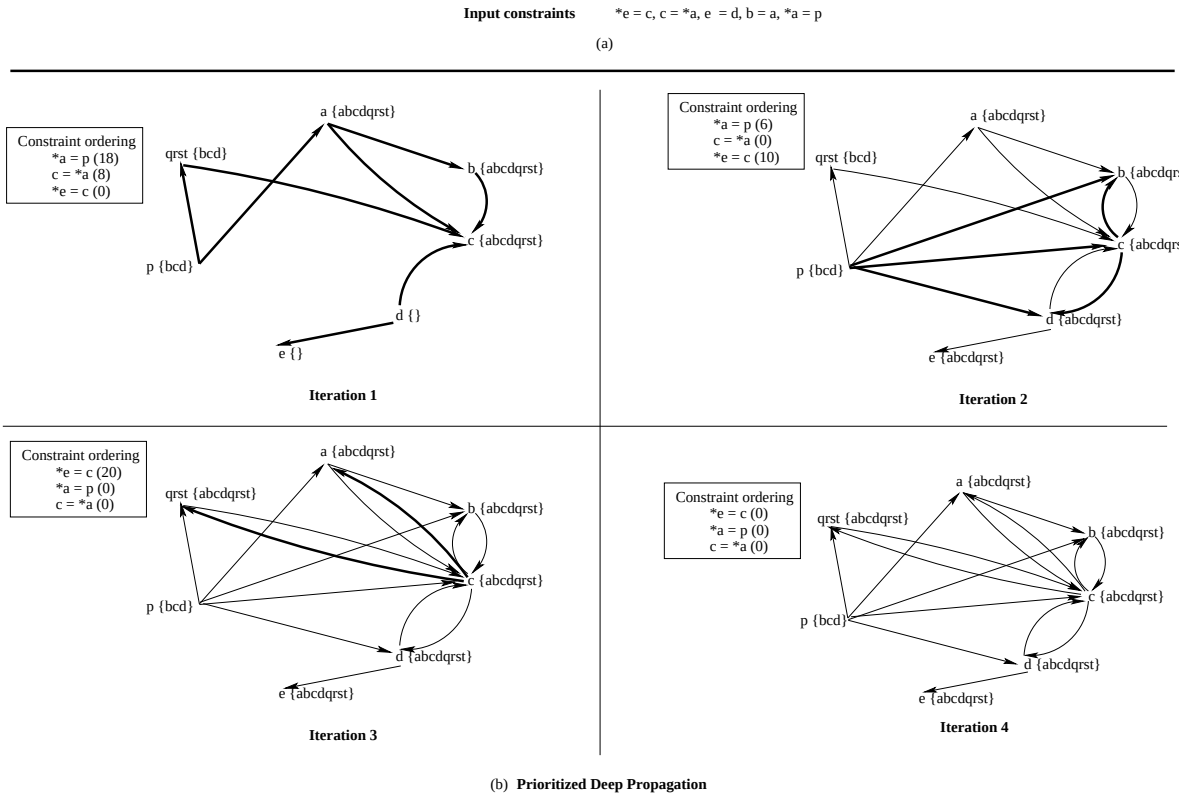


Figure 7.2: (a) Input constraints (b) Constraint graphs for Prioritized Deep Propagation

points-to information computed at the nodes by Deep Propagation is shown in Figure 7.1(b) Iteration 5. In all the iterations of the analysis, a fixed ordering of the constraints is used, which is typically the order in which the constraints appear in the program (Figure 7.1(a)).

Next, we explain how a prioritized version of Deep Propagation would reorder the constraints and hence improve the fixed-point computation. Our priority scheme can use various mechanisms for ordering the constraints. We use a mechanism wherein the priority of a constraint is the number of new points-to facts it adds in the previous iteration. Thus, the constraint priority is dynamic and may change across iterations. At the start of the analysis, i.e., before Iteration 1, the constraints can be ordered using any ordering, including the program order. In this example, we choose to use a dependence order. That is, a constraint $\mathtt{c}_1$ gets more priority over another constraint $\mathtt{c}_2$ if $\mathtt{c}_1$ may *define* a variable that $\mathtt{c}_2$ *uses*. Thus, $\mathtt{p} = \mathtt{q}$ gets higher priority over $\mathtt{r} = \mathtt{p}, \mathtt{r} = *\mathtt{p}, *\mathtt{p} = \mathtt{r}$. The constraint ordering at the start of Iteration 1 is shown in the top drawing of Figure 7.2(b). The value in parentheses following a constraint is the number of new points-to pairs the constraint adds in that iteration. For instance, the

constraint `*a = p` adds 18 new points-to pairs to `G` in Iteration 1.

As in the case of unprioritized Deep Propagation above, prior to Iteration 1, the copy constraints `e = d` and `b = a` add edges `d` to `e` and `a` to `b` respectively. Iteration 1 of our prioritized approach adds directed edges and computes points-to information as shown in the top drawing of Figure 7.2(b). Specifically, the constraint `*a = p` adds edges from `p` to `qrst` and `p` to `a`. This allows for addition of the points-to set `b,c,d` to those of `qrst`, `a` and `b`, i.e., 18 new points-to pairs. Similarly, the constraint `c = *a` adds 4 new edges: from nodes `qrst, a, b, d` to `c` and 8 new points-to pairs: $\texttt{c} \rightarrow \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}\}$[1]. The last constraint `*e = c` does not add any edges or new points-to information. Contrasting the state of `G` in Iteration 1 of the prioritized Deep Propagation with that of Deep Propagation, we observe that two additional edges are added in the prioritized analysis, namely `b` to `c` and `d` to `c`. Thus, compared to Iteration 1 of Deep Propagation, Iteration 1 of the prioritized version adds the following additional points-to information to the solution: $\texttt{a}, \texttt{b}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t} \rightarrow \{\texttt{b}, \texttt{c}, \texttt{d}\}, \texttt{c} \rightarrow \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}\}$. In general, our priority-based analysis enables addition of more edges to the constraint graph early resulting in more possibilities for early propagation of points-to information. Further note that if the constraint `*a = p` is evaluated twice, then the edges from `p` to `b,c,d` would be added, making provision for propagation of more points-to pairs. We exploit this fact for skewed evaluation in our algorithm (Section 7.5).

Our priority-based analysis framework keeps track of the number of points-to facts that are newly added by each constraint evaluation and accordingly assigns priority to the constraint. Multiple constraints may receive the same priority forming clusters of constraints. A customary way of representing various priorities is using levels. Thus, constraints which add `i` new points-to facts are assigned a priority level $\texttt{P}_\texttt{i}$. As the analysis progresses, constraints are mapped to different priority levels. As an example, since $\texttt{c} = *\texttt{a}$ adds 8 new points-to pairs in Iteration 1, it is moved to $\texttt{P}_8$ (to be used in Iteration 2). Similarly, the constraint $*\texttt{a} = \texttt{p}$ is moved to $\texttt{P}_{18}$. The constraint $*\texttt{e} = \texttt{c}$ remains at $\texttt{P}_0$.

The prioritized ordering of the constraints at the start of Iteration 2 (shown in Figure 7.2(b) Iteration 2) remains the same as in Iteration 1. The new edges added (thick lines) and the

[1]We use a right-arrow ($\rightarrow$) to indicate a points-to relation, whereas, an edge in the constraint graph is worded as *from node x to node y*.

points-to information computed at various nodes are as shown in Figure 7.2(b). In this iteration, the constraint $*\mathtt{a} = \mathtt{p}$ adds 6 new points-to pairs ($\mathtt{d}, \mathtt{e} \rightarrow \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$), the constraint $\mathtt{c} = *\mathtt{a}$ adds no new points-to information and the constraint $*\mathtt{e} = \mathtt{c}$ adds 10 new points-to pairs ($\mathtt{d}, \mathtt{e} \rightarrow \{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$). Note that the edges `c` to `b` and `c` to `d` get added in this iteration, whereas in the case of the original Deep Propagation, the same edges are added in Iteration 3 (Figure 7.1(b)). The points-to information at node `e` depends upon the information propagated via this edge (pointee set $\{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$). The third drawing of Figure 7.2(b) shows the prioritized ordering of the constraints at the start of Iteration 3, sorted by the number of points-to pairs each constraint added in Iteration 2. The points-to information computed is as shown in Figure 7.2(b). In this iteration, only $*\mathtt{e} = \mathtt{c}$ adds 20 new points-to pairs ($\mathtt{qrst} \rightarrow \{\mathtt{a}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}\}$).

The method converges after Iteration 4.

As shown in the example, a priority-based analysis evaluates constraints in such an order that it enables addition of more edges and more useful edges early in the constraint graph to ensure quick fixed-point computation. The propagation of the points-to information via these edges is done by the underlying analysis (Andersen's method [3] or Deep Propagation [100], etc.) which is not dictated by our method. The above example also suggests that the fixed-point computation of an analysis can be improved by going beyond the conventional mechanism of treating all the complex constraints with the same priority. The above example illustrated two different priority schemes – one based on the dependence across constraints based on *def-use* chain and another based on the amount of information a constraint changes. We carefully categorize different priority schemes and formalize the prioritization framework in the next section.

7.4 Prioritization Framework

We now formalize our notion of a priority based framework. A prioritized framework $\mathcal{R}$ is a 4-tuple.

$$\mathcal{R} = \langle C, \mathcal{P}, \mathcal{F}, \leq \rangle, \text{ where}$$

- C is the set of input constraints,
- $\mathcal{P}$ is the set of priority levels of size $N_{\mathcal{P}}$,

- $\mathcal{F} = \{f_1, f_2, ..., f_k\}$ is a family of priority functions. In the i^{th} iteration, the analysis uses one of f_j for some $j \in 1..k$. Then, a constraint $c \in C$ is assigned a priority p if $f_j(c) = p$, and

- $\leq$ is a partial order defined on the priority levels assigned to a pair of constraints c_x and c_y at the i^{th} iteration. If $f_i(c_x) = p_x$ and $f_i(c_y) = p_y$, then $p_x \leq p_y$ *iff* p_y is at a higher priority level than p_x.

Examples. $\mathcal{R} = \langle C, \mathcal{P}, \mathcal{F}, \leq \rangle$ defines Andersen's inclusion based analysis, where $\mathcal{P} = \{p_0\}$ and $f(c) = p_0$ for all $f \in \mathcal{F}$ and $c \in C$.

$\mathcal{R} = \langle C, \mathcal{P}, \mathcal{F}, \leq \rangle$ is a linguistic prioritized framework with $f \in \mathcal{F}$ defined as $f(c) \equiv |\textit{dependents(c)}|$ where $|\textit{dependents(c)}|$ gives the number of dependent constraints over c using *def-use* chains (as used for ordering constraints before Iteration 1 of the example in Section 7.3).

7.4.1 Priority Schemes

There are several ways in which the points-to constraints can be prioritized. We classify them into two types.

- *Linguistic scheme:* In this scheme, the constraints are prioritized based on their structure and the constraint variables. For instance, the priority mechanism in the last subsection for ordering constraints prior to Iteration 1 is a linguistic scheme. Another linguistic scheme may prioritize all load and store constraints over copy constraints.

- *Effect-driven scheme:* In this scheme, the constraints are prioritized based on the evaluation effects, e.g., the number of times a constraint gets evaluated or the number of points-to facts it adds (as in the example above).

It is possible to come up with a hybrid scheme that uses a combination of the above two schemes. For instance, one could assign different priority levels based on an effect-driven scheme, and a linguistic scheme can be used to order the constraints within the same priority level.

Algorithm 16 Prioritized Points-to Analysis

Require: set C of points-to constraints
1: process address-of constraints and remove from C
2: add edges to G using copy constraints and remove from C
3: sort C using dependence order
4: partition the constraints in different priority levels
5: **repeat**
6: **for all** level ∈ Highest priority level .. Lowest priority level **do**
7: times = 0
8: **repeat**
9: **for all** c ∈ P_{level} **do**
10: diff = evaluate(c)
11: new-level = priority-level(diff)
12: $P_{level} = P_{level} \setminus \{c\}$
13: $P_{new\text{-}level} = P_{new\text{-}level} \cup \{c\}$
14: **end for**
15: **until** inner fixed-point or ++times > threshold
16: **end for**
17: **until** outer fixed-point

7.5 The Algorithm

We instantiate our prioritization framework with a set of two functions. The first function f_1 is used for the first iteration whereas the other function f_2 is used for the subsequent iterations of the analysis. Thus, $\mathcal{F} = \{f_1, f_2\}$. The function f_1 assigns priority to a constraint according to its depth in the dependence graph of constraints. Thus, it uses a linguistic scheme to define a constraint priority. For instance, if c_1 defines a variable that c_2 uses, then c_1 gets higher priority than c_2. Note that one could use any suitable priority function of choice. The function f_2 assigns a priority level to a constraint c in iteration $i+1$ according to the amount of points-to information newly added by c in iteration i. The complete analysis developed using the prioritization framework is given in Algorithm 16. The function `evaluate()` in Line 10 implements a single iteration of the points-to information computation and propagation using any method like Andersen's analysis [3], Deep Propagation [100] or Lazy Cycle Detection [48].

Similar to Algorithm 15, our algorithm first processes the address-of and copy constraints (Lines 1–2). Line 3 finds the dependence across constraints and and Line 4 partitions them in different priority levels depending upon the topological ordering of the nodes.

The `repeat-until` loop at Lines 5–17 iterates through various constraints at various priority levels until none of the constraint evaluations changes the points-to information, suggesting that

the fixed-point is reached. Each iteration of this loop corresponds to the different iterations of the points-to analysis illustrated in Figure 7.2 of Section 7.3.

Constraints in each priority level are processed, starting from the highest priority level, in the `for` loop (Lines 6–16). In Line 10, the points-to information is computed using an underlying points-to analysis method. The method returns a single integer suggesting the amount of new points-to information computed. Based on the returned integer value, a new priority level is assigned to the constraint (Line 11). If the new priority level is the same as the current priority, the same constraint may get processed again shortly, as it has changed the points-to information. The `repeat-until` loop (Lines 8–15) shows that the constraints in the same priority level get processed repeatedly until an inner fixed-point (fixed-point for the constraints within the same priority level). This essentially allows the skewed processing of some constraints, as they get evaluated more often than others.

Lines 12 and 13 remove a constraint from its current priority level and put it in a new level, if its new priority level is different from the current one. Note that the new priority level computed is directly proportional to the change in the points-to information. This essentially means that the constraints which add more new points-to pairs are given higher priority. Since adding more edges typically results in the propagation of more points-to information, constraints that add more edges get higher priority.

We explain the computation of the new priority level at Line 11 next. The number of priority levels used in our method requires to be carefully chosen. Keeping this number same as the difference (`diff`) in the points-to information may require a large number of priority levels to be considered, as a few constraints may change hundreds of points-to facts. Moreover, this, in most cases, is unnecessary, as the fixed-point computation benefits from clusters of constraints having approximately the same priorities, rather than an isolated high-priority constraint. Therefore, we combine a range of priority values into a single priority level. This *bucketization* proves helpful in skewed evaluation of constraints in a priority level.

Further, bucketization exploits an important observation about constraint solving (as a side-effect): several interdependent constraints, at different times during the analysis, modify the same amount of points-to information. We observed this empirically and on inspection, realized that after initial warm-up, several of the *load* and *store* constraints, of the form $\texttt{p} = *\texttt{q}$ and $*\texttt{p} = \texttt{q}$, start adding a fixed number of points-to information. Therefore, interdependent

constraints, due to bucketization, get grouped at the same priority level. Thus, iterating over these interdependent constraints helps in reaching the fixed-point faster.

However, the importance of iterating over the constraints at the same priority level should not be over-emphasized. Due to the cyclic nature of (self or transitive) dependence, in most cases, it suffices to iterate only twice over the constraints at a priority level. The number of indirections present in hand-written programs is typically quite small. Hence, looping beyond two iterations gradually reduces the gain (the amount of points-to information added). Therefore, the `repeat-until` loop from Lines 8–15 iterates for at most `threshold` number of times over the constraints at the same priority level. The condition `inner fixed-point` takes care of not iterating an $(\mathtt{i}+1)^{\mathtt{th}}$ time if the $\mathtt{i}^{\mathtt{th}}$ iteration does not change the points-to information.

Since our analysis evaluates constraints from higher to lower priority order and since a constraint priority changes dynamically, it could lead to an interesting paradoxical situation. Consider constraints $\mathtt{c}_1$ and $\mathtt{c}_2$, which are at priority level $\mathtt{P}_\mathtt{i}$ at the start of an iteration `i`. In that iteration, let $\mathtt{c}_1$'s priority increase and let it *jump* to a priority level above $\mathtt{P}_\mathtt{i}$. Let the constraint $\mathtt{c}_2$, in contrast, *jump* to a priority level below $\mathtt{P}_\mathtt{i}$ by adding less amount of points-to information. Since the `for` loop at Line 6 traverses the priority levels from the highest to the lowest priority order, the constraint $\mathtt{c}_1$, which has a higher priority, does not get evaluated more than once in iteration i. However, the constraint $\mathtt{c}_2$ which has a lower priority, gets evaluated again in iteration i at the lower priority level. Intuitively, we expect $\mathtt{c}_1$ to be evaluated more number of times than $\mathtt{c}_2$. However, due to the nature of our algorithm, exactly the reverse may happen. An intuitively better way would be to *evaluate the current highest priority constraint*. In order to evaluate the effect of evaluating the current highest priority constraint, we implemented our analysis using a priority queue (see Section 7.6.6 for details). Thus, we modify Lines 6 to 16 of Algorithm 16 to extract an element from a priority queue which returns the constraint with maximum priority. After evaluating the constraint, the constraint is pushed back into the queue with the updated priority. This guarantees that, at each stage, a constraint with the highest priority is evaluated. We observed that our current implementation outperforms the priority queue based implementation by a large margin (over 33%). This suggest that choosing a strict priority ordering does not achieve the maximum performance. Alternative approaches to handle priority, other than the ones discussed here, could be proposed. We leave this aspect as a future work.

We remark that the priority scheme does not require the constraint graph to be constructed. The algorithm works on constraints rather than on individual pointers.

7.6 Experimental Evaluation

We evaluate the effectiveness of prioritized points-to analysis using our benchmark suite consisting of 16 SPEC C/C++ benchmarks and five large open source programs (*httpd*, *sendmail*, *gdb*, *wine-server* and *ghostscript*). As before, we evaluate the impact of our prioritization approach on Andersen's analysis [3] referred to as *anders*, Lazy Cycle Detection (LCD) [49] referred to as *bddlcd*, Bloom Filter based points-to analysis (Chapter 4) referred to as *bloom* and Deep Propagation [100] referred to as *deep*.

To recall, *deep* is context-insensitive while other implementations are context-sensitive; further, all methods are flow-insensitive and field-insensitive. Context-sensitivity is implemented using an invocation-graph based approach [32], as discussed in Section 4.4. For each method *a*, we denote its prioritized version as *p-a* (e.g., *p-anders, p-bloom* etc.). All prioritized versions implement effect-driven scheme with all the optimizations described in Section 7.5 with the number of priority levels set to 203 (see Section 7.6.4 for a discussion on selecting the number of priority levels). The experiments are carried out on the same platform, with an Intel Xeon machine with 2 GHz clock, 4 MB L2 cache and 4 GB RAM.

7.6.1 Analysis Time

The analysis times (in seconds) of various methods are shown in Table 7.1. Comparing *anders* versus *p-anders*, we observe a 13%–41% reduction in the analysis time (average 33%) due to prioritized scheduling of points-to constraints. This emphasizes the importance of a good constraint order.

In case of *bddlcd*, we observe a larger benefit due to prioritization (44% on an average, excluding *gdb*). The benefit is an outcome of an interplay between *bddlcd* algorithm and prioritized scheduling. Cycle detection benefits from evaluating the cyclic constraints together which change an equal number of points-to pairs and hence get grouped into the same priority level. The prioritized scheduling approach evaluates all of them in close-proximity, often giving correct hints to the cycle detection mechanism, resulting in an overall efficient analysis. In case

Benchmark	Context-sensitive						Context-insensitive	
	anders	p-anders	bddlcd	p-bddlcd	bloom	p-bloom	deep	p-deep
gcc	329.5	286.5	17411.2	7984.5	137.1	119.2	1.740	1.176
perlbmk	143.4	98.4	5879.9	3159.5	85.4	72.6	1.744	1.396
vortex	91.3	69.7	4725.7	3397.2	71.4	63.2	0.116	0.088
eon	93.5	79.3	2391.8	1515.1	74.0	52.4	11.701	2.320
parser	35.4	26.4	618.3	331.0	29.2	26.1	0.176	0.072
gap	128.5	85.0	330.2	186.8	85.1	72.9	0.092	0.044
vpr	29.5	20.1	199.5	95.6	16.1	11.9	0.024	0.008
crafty	29.3	22.1	155.0	91.6	17.6	13.4	0.004	0.004
mesa	89.4	65.1	21.7	12.1	63.2	58.9	0.248	0.108
ammp	34.2	23.3	54.6	31.4	22.5	19.5	0.032	0.012
twolf	41.5	33.8	27.4	13.5	31.2	25.5	0.032	0.016
gzip	25.2	14.9	6.5	3.1	18.5	12.4	0.004	0.004
bzip2	23.3	14.0	4.7	3.9	19.6	18.3	0.004	0.004
mcf	22.4	17.1	32.0	18.4	16.3	16.3	0.004	0.004
equake	24.3	17.2	4.1	3.7	13.8	11.3	0.004	0.004
art	26.5	19.2	7.7	4.1	11.5	10.0	0.004	0.004
httpd	224.5	193.3	47.4	24.8	38.8	32.8	53.727	23.722
sendmail	172.7	136.2	117.5	96.6	15.9	11.6	12.729	10.613
ghostscript	4384.2	3183.8	20612.8	12372.0	1959.8	1652.4	207.033	126.140
gdb	9338.2	5847.3	24871.7	OOM	2362.6	1941.3	587.829	294.066
wine-server	201.3	147.3	36.7	23.5	68.0	55.0	8.165	5.488
average	737.5	495.2	3693.2	1468.4*	245.6	204.6	42.162	22.157

* The average is calculated ignoring the *OOM* entry.

Table 7.1: Analysis time (seconds)

of *gdb*, the prioritized LCD version goes out of memory (see discussion in Section 7.6.2).

In case of *bloom*, both the versions (*bloom* and *p-bloom*) analyze all the benchmarks successfully to completion, with *p-bloom* achieving 16% reduction in the analysis time.

The execution times of context-insensitive analysis (*deep* and *p-deep*) are significantly lower due to the relatively lower computational requirements of the context-insensitive algorithm. But even in this case, introducing prioritization results in an improvement that is either smaller for benchmarks which require few hundred milliseconds analysis time or not observed (for the benchmarks where the analysis time is a few milliseconds). However, for the larger benchmarks such as *httpd, ghostscript* and *gdb*, the improvements are significant, resulting in more than 50% reduction in the analysis time. The reason is quite similar to that in case of *bddlcd*. Online cycle detection implemented as part of *deep* benefits from prioritized scheduling. However,

Benchmark	Context-sensitive						Context-insensitive	
	anders	p-anders	bddlcd	p-bddlcd	bloom	p-bloom	deep	p-deep
gcc	2859	2174	2534	3651	669	664	83	73
perlbmk	2133	1878	1723	2940	314	312	100	93
vortex	1857	1553	1358	2030	152	146	16	16
eon	1276	907	1425	1882	299	295	248	66
parser	478	419	345	488	118	112	4	4
gap	457	397	362	472	240	238	8	8
vpr	735	688	692	973	93	93	2	2
crafty	672	600	566	716	77	76	1	1
mesa	894	825	729	1069	163	161	14	14
ammp	427	372	336	478	81	80	3	2
twolf	624	485	617	837	119	115	4	4
gzip	514	446	522	685	55	54	1	1
bzip2	633	582	588	856	56	57	1	1
mcf	403	379	389	505	48	49	1	1
equake	546	501	527	847	56	56	1	1
art	597	524	582	852	49	49	1	1
httpd	791	686	825	1252	596	594	674	425
sendmail	914	799	851	1297	306	303	256	224
ghostscript	1958	1644	1672	2389	971	969	2871	2364
gdb	2194	1635	1859	OOM	1461	1363	3556	2765
wine-server	774	615	690	1100	279	274	185	149
average	1035	862	914	1266	295	289	382	296

* The average is calculated ignoring the *OOM* entry.

Table 7.2: Memory requirement (MB)

another artifact of *deep* facilitates higher benefits with prioritized scheduling. Deep Propagation works on the topological ordering of the directed *copy* edges across pointer nodes in the constraint graph. Effect-driven prioritization of constraints adds *copy* edges that propagate (approximately) the same number of points-to constraints in an iteration, resulting in most of the propagation path available for deep propagation.

7.6.2 Memory

The memory requirements (in MB) for various methods are shown in Table 7.2. In general, one would expect the memory requirements to remain almost the same. However, the internal structure and implementation of the algorithm play a key role in the overall memory requirement.

The *p-anders* method consistently requires significantly less memory than *anders*, with an

average memory reduction of 16.7%. The memory savings are largely due to the difference propagation (propagating only the difference in the points-to sets across a constraint-graph edge) and the use of temporary data structures during constraint evaluation. Instead of keeping small difference information for propagation across iterations as in *anders*, our effect-driven prioritized scheme combines difference information together and propagates them along complete paths in consecutive evaluations. This benefit is similar in spirit to what Deep Propagation achieves over Wave Propagation [100].

Both the *bloom* and *p-bloom* methods complete successfully on all the benchmarks and do not use difference propagation. Hence the memory requirements are quite similar (295 MB versus 289 MB on an average). The small drop observed in the memory requirement for the prioritized version is due to the reduction in the temporary data structures used for holding points-to information during propagation.

The *p-deep* method outperforms *deep* in terms of the memory requirement by 22.5% on an average. Memory savings are largely due to difference propagation.

On the other hand, the *p-bddlcd* method requires 38.5% more memory than *bddlcd*. In fact, in case of *gdb*, *p-bddlcd* runs out of memory whereas the non-prioritized version completes successfully. The increase in the memory requirements in *p-bddlcd* is due to the nature of *bddlcd* algorithm. Unlike other algorithms discussed here, *bddlcd* is worklist based. A constraint may get added to the worklist while its another instance is already present. Thus, the worklist size is not bound by the total number of points-to constraints. Having a prioritized scheme requires multiple such worklists to be created, pushing different instances of the same constraint into different worklists based on the current priority of the constraint. Thus, using multiple worklists increases the amount of memory consumed. Thus, in case of *bddlcd*, introducing prioritization decreases the analysis time though at the expense of increased memory requirement.

7.6.3 Overall Effect

We counted the average number of new points-to facts generated by each constraint for *anders* and *p-anders* in each iteration[2]. We present results for four representative benchmarks, namely, *vortex, art, vpr* and *gap* in Figure 7.3. The results of other benchmarks are similar. We observe that in all cases, the prioritized version computes points-to facts in earlier iterations and thus

[2]Due to constraints getting evaluated multiple times in *p-anders*, the notion of iteration is not well defined.

computes fixed-point faster compared to the non-prioritized version. As a specific example, in case of *vortex*, due to prioritizing appropriate constraints, our *p-anders* method computes the points-to facts earlier and reaches the fixed-point in 8 iterations compared to 10 as in the *anders* method.

7.6.4 Effect of Bucketization

First, we experimented with several values for the number of priority levels. The sensitivity of the analysis time (execution time to complete the points-to analysis) to the number of priority levels (buckets) for *p-anders* is shown in Figure 7.4. Note that the values are normalized with respect to *anders*. To avoid clutter, we show the effect on only four representative benchmarks *ghostscript*, *gdb*, *perlbmk* and *gzip* along with the average over all the benchmarks listed in Table 4.1. We observe that the analysis time steadily reduces with the increasing number of buckets. However, the number of buckets should not be arbitrarily increased. It is important to keep related constraints together so that an inner fixed-point over the related constraints would be beneficial (refer to a discussion in Section 7.5). Using too many priority levels may move related constraints in different priority levels and would reduce the benefit of the inner fixed-point. Further, after a point, increasing the number of buckets starts giving diminishing returns.

7.6.5 Effect of Skewed Evaluation

As certain constraints may get evaluated multiple number of times, it is important to measure the effect of this skewed evaluation. In Algorithm 16, we removed the threshold check for inner fixed point (Line 15) and allowed each priority level to be evaluated until fixed-point. We measured the total amount of new points-to information added by each inner iteration across all the priority levels. Figure 7.5 shows the effect over our benchmark suite. X-axis shows the inner iteration number. Y-axis shows the percentage of new points-to information added at an iteration. We observe that 61% of the total points-to information is added in the first iteration while a total of 93% points-to information is added in the first two iterations. The later iterations (iteration 3 onwards) add only a very small amount of information and, in the interest of overall analysis time, we restrict the threshold for inner iterations to 2.

We would like to note that values of the configuration parameters play an important role

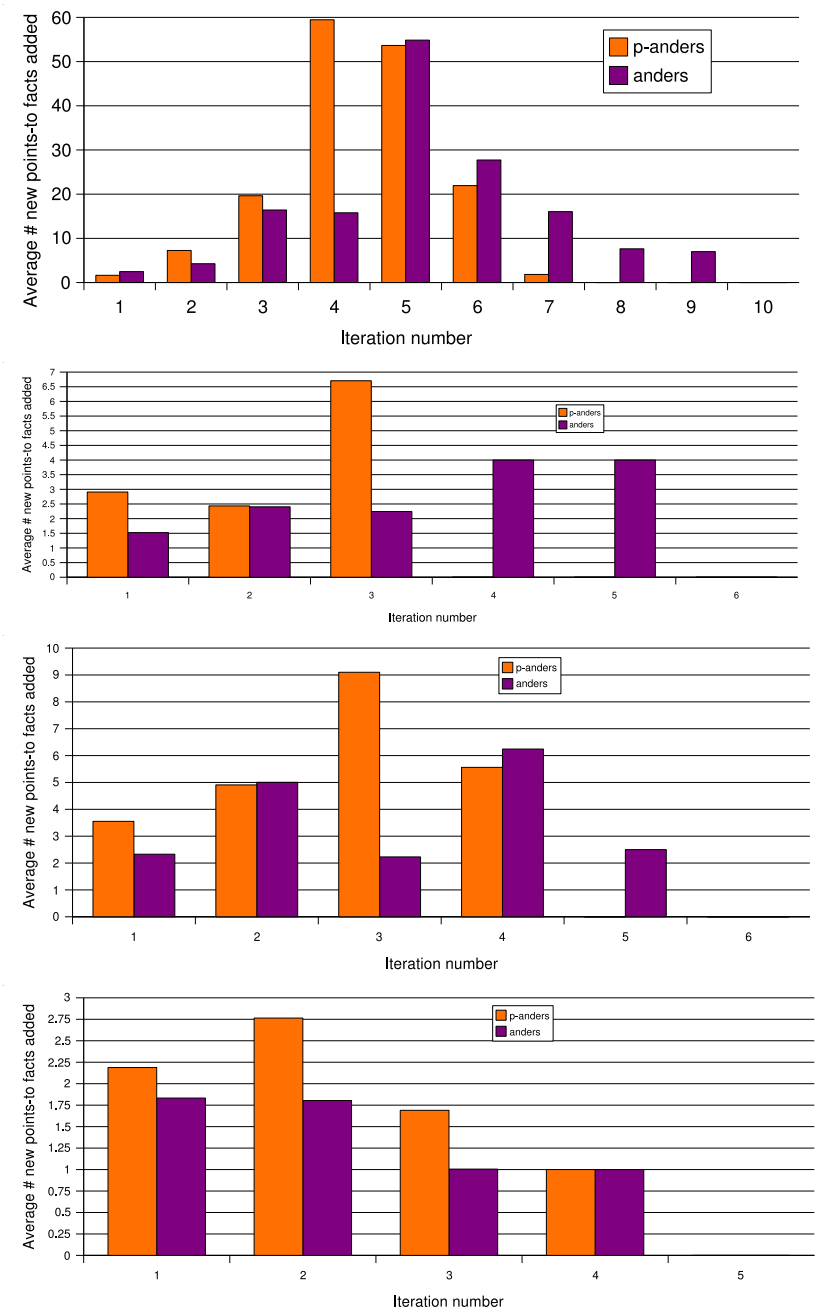


Figure 7.3: Effect of prioritization for *vortex, art, vpr* and *gap* respectively

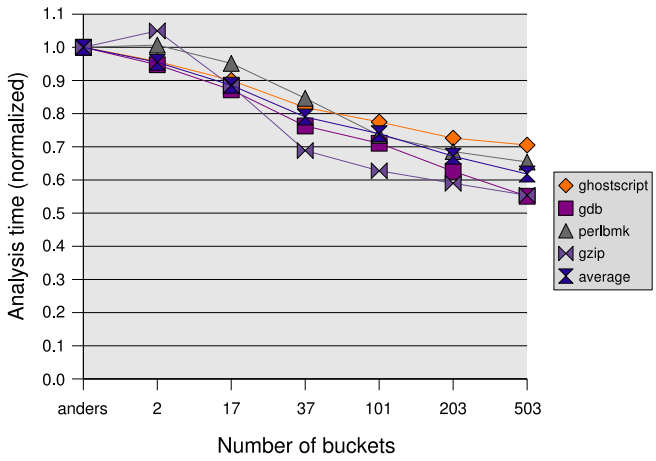


Figure 7.4: Effect of bucketization

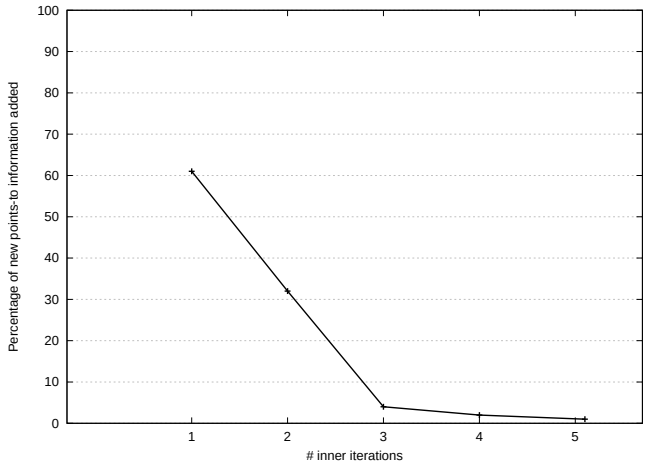


Figure 7.5: Effect of skewed evaluation

in the analysis efficiency and must be chosen carefully. However, our experience suggests that the parameters vary according to the program characteristics and there is no simple rule to arrive at the optimal values for all the programs.

7.6.6 Comparison with Priority Queue

In all our experiments the buckets are implemented as a hashtable wherein the priority level acts as the key. One may argue that the prioritization can possibly be more efficiently implemented as a priority queue. Theoretically, a priority queue incurs, on an average, an O(log n)

Benchmark	p-anders	p-anders-pq
gcc	151.618	952.368
perlbmk	65.969	552.173
vortex	1.457	11.735
eon	29.625	224.621
parser	0.831	24.827
gap	6.689	13.975
vpr	0.465	2.381
crafty	0.453	2.879
mesa	1.029	5.645
ammp	0.372	1.883
twolf	0.614	3.725
gzip	0.221	0.852
bzip2	0.199	0.255
mcf	0.175	0.188
equake	0.176	0.222
art	0.167	0.190
httpd	58.624	407.571
sendmail	37.276	286.395
ghostscript	425.362	503.515
gdb	852.622	5843.766
wine-server	62.545	541.532
average	80.785	446.700

Table 7.3: Comparison with Priority Queue: Analysis Time (seconds)

complexity for each insertion and removal of an element. In our hashtable-based implementation, insertion and removal are O(1) operations, given a reference to a constraint. To study how the two implementations perform in practice, we developed a priority-queue-based analysis with C++ STL. The analysis time for various benchmarks required for our hash-table based implementation (*p-anders*) and priority queue based implementation (*p-anders-pq*) are given in Table 7.3. We observe that the priority-queue-based Andersen's analysis (*p-anders-pq*) is 5× slower than our hash-table-based analysis. The *p-anders* method requires 81 seconds on an average for analyzing a benchmark, whereas the *p-anders-pq* method requires 447 seconds on an average. This shows that the priority levels implemented as a hashtable perform better than those implemented as a true priority queue.

7.7 Related Work

We proposed a framework for prioritizing points-to constraint evaluation. To the best of our knowledge, no prior work has focused on prioritizing points-to constraints. However, there exists some work on efficient propagation of points-to information in a constraint graph. Pearce et al. [99] propose difference propagation to propagate only the difference in the points-to information across nodes. In order to be efficient, they maintain only an approximate difference across the end points of an edge. Wave and Deep Propagation [100] propagate points-to information in breadth-first and depth-first manner respectively for efficient analysis. The technique uses (exact) difference propagation [99] for propagating only the changed points-to information. They show that both the techniques can significantly improve the propagation time and Deep Propagation can greatly reduce the memory requirement of the analysis. Kanamori and Weise [67] propose several heuristics for choosing a node-ordering for points-to information propagation, e.g., Greatest Input Rise, Greatest Output Rise and Least Recently Fired. Pearce et al. [99] find that the Least Recently Fired strategy works very well in practice over other heuristics especially for large programs. Our work is related, but deals with constraint evaluation ordering rather than points-to information propagation. It can be easily combined with any propagation related optimization for enjoying joint benefits.

Our prioritization mechanism may be viewed to be similar to an online set cover problem in which sets are chosen to maximize the benefit [5] or to minimize the cost [2] in every iteration.

We empirically showed that, on an average, our prioritization mechanism reduces the running time of optimized Andersen's analysis [3] by 33%, of BDD-based Lazy Cycle Detection [49] by 44%, of bloom filter based analysis (Chapter 4) by 16%, and of Deep Propagation [100] by 48%. Our approach also improves the memory requirement of the algorithms which use difference propagation. On an average, our prioritization mechanism reduces the memory requirement of Andersen's analysis [3] by 16.7%, and of Deep Propagation [100] by 22.5%.

7.8 Chapter Summary

In this chapter, we proposed a prioritized order of processing constraints in the points-to analysis method to improve its efficiency. First, we proved that finding an optimal sequence

of points-to constraints for even a restricted flow-insensitive version is NP-Complete. Subsequently, we identified two new dimensions for evaluating points-to constraints: how many edges a constraint adds and where in the constraint graph it adds edges. Based on this observation, we presented a prioritization framework for evaluating a set of points-to constraints. We illustrated the generality of the proposed framework by implementing prioritized versions of Andersen's analysis, Lazy Cycle Detection using BDD, Bloom-filter based analysis and Deep Propagation. We instantiated the framework with a hybrid priority scheme based on the *use-def* chains of variables and the amount of points-to information a constraint changes on evaluation. Experimental evaluation shows that the presented priority scheme can greatly benefit the state-of-the-art algorithms to reach a fixed-point faster. In addition to improving the analysis time, the proposed approach also reduces the memory requirement of the algorithms that use difference propagation.

While the framework is illustrated in the context of points-to analysis, the idea of prioritized evaluation is general and applicable to other static and dynamic analyses. We believe that further work on prioritizing constraints can open up interesting possibilities for performing optimizations.

Chapter 8

Conclusions and Future Work

8.1 Summary

This thesis proposed several novel approaches to improve the scalability of context-sensitive pointer analysis. We formalized and experimentally validated their practical usefulness and scalability.

We proposed a multi-dimensional bloom filter for storing points-to information. The proposed representation, though, may introduce false positives, significantly reduces the memory requirement and provides a probabilistic lower bound on loss of precision. As our multibloom representation introduces only false positives, but no false negatives, it ensures safety for the points-to analysis. We demonstrated that compared to an *exact* analysis, a multibloom configuration offers, on an average, 75% reduction in memory requirement and 40% reduction in analysis time with less than 2% precision loss. We also showed that compared to a BDD-based analysis, a multibloom configuration is 15× faster and uses 3× less memory. Using Mod/Ref analysis as a client, we showed that the effect of our approximate representation on the precision of the client is even less. Unlike traditional data-structures for storing points-to information, like bitmaps and BDDs, bloom filter provides user a control on the memory requirement, yet giving a probabilistic lower bound on the precision loss.

Next, we presented a randomized, yet sound, technique for scaling points-to analysis. Our method selectively applies different kinds of analyses on different partitions of the program entities to be processed and then composes the results carefully to get a sound approximation to the points-to solution. We illustrated the technique to develop a randomized context-sensitive

points-to analysis. Our empirical evaluation revealed several configurations that achieve less than 5% precision loss with an average 50% reduction in context-sensitive analysis time. Based on our analysis of the results, we developed an adaptive randomized points-to analysis which can be used for a program for which the right configuration is unknown.

Next, we proposed a novel approach to transform a set of points-to constraints into a system of linear equations using prime factorization. We overcame the technical challenges by partitioning our inclusion-based analysis into a linear solver phase and a post-processing phase that interprets the resulting values and updates points-to information accordingly. The novel way of representing points-to information as a composition of primes allowed us to keep the equations linear in every iteration. We showed that our analysis is sound and precise with respect to an inclusion-based analysis for a fixed dereference level. Using a suite of benchmarks, we showed that our approach is not only theoretically feasible, but is also practically viable. On an average, our context-sensitive analysis is 8.8× faster than BDD-based Lazy Cycle Detecton [49] and 1.8× faster than an optimized Andersen's analysis [3].

Finally, we proposed a prioritized order of processing constraints in the points-to analysis method to improve its efficiency. First, we proved that finding an optimal sequence of points-to constraints for even a restricted flow-insensitive version is NP-Complete. Subsequently, we presented a prioritization framework for evaluating a set of points-to constraints. We illustrated the generality of the proposed framework by implementing prioritized versions of Andersen's analysis, Lazy Cycle Detection using BDD, bloom-filter based analysis and Deep Propagation. We instantiated the framework with a hybrid priority scheme based on the *use-def* chains of variables and the amount of points-to information a constraint changes on evaluation. We demonstrated that the presented priority scheme can greatly benefit the state-of-the-art algorithms to reach a fixed-point faster. Thus, our technique improves the analysis time of Andersen's method by 33%, of Lazy Cycle Detection by 44%, of bloom-filter based analysis by 16%, and of Deep Propagation by 45% on an average. In addition to improving the analysis time, the proposed approach also reduces the memory requirement of the algorithms that use difference propagation. Thus, our technique reduces the memory requirement of Andersen's analysis by 16.7%, and of Deep Propagation by 22.5% on an average.

8.2 Future Work

In this section, we mention a few directions along which our work can be extended.

- *Flow-sensitive points-to analysis using bloom filters:* We explored bloom filters for flow-insensitive points-to analysis. A flow-sensitive points-to analysis is more memory intensive than its flow-insensitive counterpart. Since the use of bloom filters to store points-to information enables large savings in the memory requirement, it could be very well suited for a flow-sensitive analysis. However, a basic bloom filter operation does not support deleting an element (resetting a bit), which would be required to implement a *kill* set of a flow-sensitive analysis. A variant of bloom filter, called counting bloom filter [36], supports a limited number of element deletions, but beyond the limit, it may result into a false negative. Therefore, some innovative ways of managing the kill-set would be required to implement a flow-sensitive points-to analysis using bloom filters.

- *Applying randomization to other program analyses:* We presented our randomization technique for a context-sensitive points-to analysis. However, the technique is general and can possibly be applied to other analysis dimensions and other program analyses. For instance, consider an analysis $\mathcal{A}_1$ of multi-threaded programs which considers all the feasible interleavings of the threads, versus an analysis $\mathcal{A}_2$ which analyzes the threads flow-insensitivity. Obviously, $\mathcal{A}_1$ is more precise and time-consuming than $\mathcal{A}_2$. However, by applying our randomization technique, a random set of threads could be chosen to be processed in a more precise manner while the remaining could be chosen to be processed in a less precise manner and by composing the two results, one may get a sound approximation to the analysis solution. Depending upon the degree of randomization, the analysis precision would vary from that of the more precise to the less precise analysis.

- *Exploiting the structure of linear constraints:* We reduced the pointer analysis problem to that of solving a system of linear equations. It would be interesting to see the effect of various optimizations proposed for the set of linear equations on the efficiency of our analysis. Specifically, in our transformation, each equation has a special structure: each equation contains not more than two variables, the left hand side variable has a coefficient of unity and the only constant term that can appear is 1 (since the constraints

are normalized; see Section 2.4.1). Due to this special structure, it may be possible to apply some linear algebra techniques to optimize solving the system of equations.

- *Points-to analysis as a network flow problem:* Another interesting direction would be to map the points-to analysis problem to another well-studied problem and take advantage of the algorithms and optimizations developed for the problem. For instance, one may reduce points-to analysis to a network flow problem since the former is essentially propagation of information across nodes in a constraint graph. Such reductions may open up avenues for asymptotically better points-to analysis algorithm.

- *Exploring other prioritization mechanisms for constraint evaluation:* Our prioritization framework was instantiated using one greedy heuristic which depends upon the amount of new points-to information added by a constraint. However, it is very much possible to devise a new prioritization scheme to be plugged into our framework. For instance, calculating the priority of a constraint in the current iteration could be done based on all the previous iterations rather than only the last iteration. A comparative study of such schemes would let us better understand the role of constraint evaluation in points-to analysis and may lead us to a highly efficient algorithm.

To conclude, we believe that our work has advanced the state-of-the-art on pointer analysis.

References

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 129–140, New York, NY, USA, 2003. ACM.

[2] N. Alon, B. Awerbuch, and Y. Azar. The Online Set Cover Problem. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 100–105, New York, NY, USA, 2003. ACM.

[3] L. O. Andersen. Program Analysis and Specialization for the C Programming Language, PhD Thesis, DIKU, University of Copenhagen, 1994.

[4] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving Software Security with a C Pointer Analysis. In *Proceedings of the 27th international Conference on Software Engineering*, ICSE '05, pages 332–341, New York, NY, USA, 2005. ACM.

[5] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time (Extended Abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 519–530, New York, NY, USA, 1996. ACM.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to Analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[7] B. H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13:422–426, July 1970.

[8] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[9] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic Model Checking for Sequential Circuit Verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(4):401 –424, apr 1994.

[10] M. G. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 234–250, London, UK, 1995. Springer-Verlag.

[11] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards. A Novel Analysis Space for Pointer Analysis and its Application for Bug Finding. *Sci. Comput. Program.*, 75:921–942, November 2010.

[12] V. 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*, POPL '03, pages 115–125, New York, NY, USA, 2003. ACM.

[13] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant Context Inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. ACM.

[14] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of Points-To Analysis of Java in the Presence of Exceptions. *IEEE Trans. Softw. Eng.*, 27:481–512, June 2001.

[15] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. *SIGPLAN Not.*, 38:25–36, June 2003.

[16] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Interprocedural Probabilistic Pointer Analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15:893–907, October 2004.

[17] B.-C. Cheng and W.-M. W. Hwu. Modular Interprocedural Pointer Analysis using Access Paths: Design, Implementation, and Evaluation. In *Proceedings of the ACM SIGPLAN*

2000 Conference on Programming Language Design and Implementation, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[18] J.-D. Choi, M. Burke, and P. 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*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[19] W. Choi and K.-M. Choe. Cycle Elimination for Invocation Graph-Based Context-Sensitive Pointer Analysis. *Inf. Softw. Technol.*, 53:818–833, August 2011.

[20] S. Cohen and Y. Matias. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms, McGraw Hill, 2001.

[22] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[23] J. Da Silva and J. G. Steffan. A Probabilistic Pointer Analysis for Speculative Optimizations. In *Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 416–425, New York, NY, USA, 2006. ACM.

[24] M. Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.

[25] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 260–278, London, UK, 2001. Springer-Verlag.

[26] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[27] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond K-Limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 230–241, New York, NY, USA, 1994. ACM.

[28] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-Based Alias Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM.

[29] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 155–167, New York, NY, USA, 2003. ACM.

[30] M. Edvinsson, J. Lundberg, and W. Löwe. Parallel Points-to Analysis for Multi-Core Machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM.

[31] M. Emami. A Practical Inter-Procedural Alias Analysis for an Optimizing/Paralleling C Compiler, Master thesis, School of Computer Science, McGill University, 1993.

[32] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[33] J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian Program Analysis. *J. ACM*, 57:33:1–33:47, November 2010.

[34] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on*

Programming Language Design and Implementation, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[35] M. Fähndrich, J. Rehof, and M. Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.

[36] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8:281–293, June 2000.

[37] C. Fecht and H. Seidl. An Even Faster Solver for General Systems of Equations. In *Proceedings of the Third International Symposium on Static Analysis*, pages 189–204, London, UK, 1996. Springer-Verlag.

[38] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[39] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 345–354, New York, NY, USA, 2003. ACM.

[40] GCC, http://gcc.gnu.org/.

[41] R. Ghiya and L. J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[42] GNU MP Integer Library, http://gmplib.org/.

[43] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle. Alias Analysis for Optimization of Dynamic Languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM.

[44] L. L. Gremillion. Designing a Bloom Filter for Differential File Access. *Commun. ACM*, 25:600–604, September 1982.

[45] S. Z. Guyer and C. Lin. Client-Driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[46] S. Z. Guyer and C. Lin. Error Checking with Client-Driven Pointer Analysis. *Sci. Comput. Program.*, 58:83–114, October 2005.

[47] B. Hackett and A. Aiken. How is Aliasing Used in Systems Software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 69–80, New York, NY, USA, 2006. ACM.

[48] B. Hardekopf and C. Lin. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2007.

[49] B. Hardekopf and C. Lin. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[50] B. Hardekopf and C. Lin. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[51] Ben Hardekopf, http://www.cs.utexas.edu/users/benh/.

[52] R. Hasti and S. Horwitz. Using Static Single Assignment Form to Improve Flow-Insensitive Pointer Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 97–105, New York, NY, USA, 1998. ACM.

[53] N. Heintze and O. Tardieu. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 24–34, New York, NY, USA, 2001. ACM.

[54] N. Heintze and O. Tardieu. Ultra-Fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[55] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.*, 21:848–894, July 1999.

[56] M. Hind and A. Pioli. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, SAS '98, pages 57–81, London, UK, 1998. Springer-Verlag.

[57] M. Hind and A. Pioli. Which Pointer Analysis Should I Use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[58] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast Online Pointer Analysis. *ACM Trans. Program. Lang. Syst.*, 29, April 2007.

[59] S. Horwitz. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Trans. Program. Lang. Syst.*, 19:1–6, January 1997.

[60] Y.-S. Hwang, P.-S. Chen, J. K. Lee, and R. D.-C. Ju. Probabilistic Points-to Analysis. In *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, LCPC'01, pages 290–305, Berlin, Heidelberg, 2003. Springer-Verlag.

[61] ILOG Toolkit, http://www.ilog.com/.

[62] D. Jang and K.-M. Choe. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1930–1937, New York, NY, USA, 2009. ACM.

[63] Java Programming Language, http://www.java.com/.

[64] JavaScript Programming Language, http://www.javascript.com/.

[65] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop on Programming*

Languages and Analysis for Security, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[66] V. Kahlon. Bootstrapping: A Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[67] A. Kanamori and D. Weise. Worklist Management Strategies for Dataflow Analysis, MSR Technical Report, MSR-TR-94-12, 1994.

[68] U. Khedker and B. Karkare. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78791-4_15.

[69] D. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1997.

[70] W. Landi. Undecidability of Static Analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, December 1992.

[71] W. Landi. Interprocedural Aliasing in the Presence of Pointers, PhD thesis, Rutgers University, 1992.

[72] W. Landi and B. G. Ryder. Pointer-Induced Aliasing: A Problem Taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[73] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[74] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural Modification Side Effect Analysis with Pointer Aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 56–67, New York, NY, USA, 1993. ACM.

[75] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[76] O. Lhotak. A Tour of Pointer Analysis, Summer School on Theory and Practice of Language Implementation, University of Oregon, 2009.

[77] O. Lhoták and L. Hendren. Evaluating the Benefits of Context-Sensitive Points-to Analysis using a BDD-Based Implementation. *ACM Trans. Softw. Eng. Methodol.*, 18:3:1–3:53, October 2008.

[78] D. Liang and M. J. Harrold. Efficient Points-to Analysis for Whole-Program Analysis. *SIGSOFT Softw. Eng. Notes*, 24:199–215, October 1999.

[79] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the Precision of Static Reference Analysis using Profiling. In *Proceedings of the 2002 ACM SIGSOFT international Symposium on Software Testing and Analysis*, ISSTA '02, pages 22–32, New York, NY, USA, 2002. ACM.

[80] V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 317–326, New York, NY, USA, 2003. ACM.

[81] The LLVM Compiler Infrastructure, http://llvm.org.

[82] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 149–159, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[83] U. Manber and S. Wu. An Algorithm for Approximate Membership Checking with Application to Password Security. *Inf. Process. Lett.*, 50:191–197, May 1994.

[84] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly Representing First-Order Structures for Static Analysis. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 196–212, London, UK, 2002. Springer-Verlag.

[85] V. Martena and P. S. Pietro. Alias Analysis by Means of a Model Checker. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 3–19, London, UK, 2001. Springer-Verlag.

[86] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel Inclusion-Based Points-to Analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM.

[87] A. Milanova, A. Rountev, and B. G. Ryder. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engg.*, 11:7–26, January 2004.

[88] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005.

[89] M. Mitzenmacher. Compressed Bloom Filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 144–150, New York, NY, USA, 2001. ACM.

[90] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving Program Slicing with Dynamic Points-to Data. *SIGSOFT Softw. Eng. Notes*, 27:71–80, November 2002.

[91] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 66–72, New York, NY, USA, 2001. ACM.

[92] M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.

[93] R. Muth and S. Debray. On the Complexity of Flow-Sensitive Dataflow Analyses. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 67–80, New York, NY, USA, 2000. ACM.

[94] R. Nasre. Approximating Inclusion-based Points-to Analysis. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 66–73, New York, NY, USA, 2011. ACM.

[95] A. Orso, S. Sinha, and M. J. Harrold. Classifying Data Dependences in the Presence of Pointers for Program Comprehension, Testing, and Debugging. *ACM Trans. Softw. Eng. Methodol.*, 13:199–239, April 2004.

[96] A. Partow. General Purpose Hash Function Algorithms, http://www.partow.net/programming/hashfunctions/.

[97] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient Field-Sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.*, 30, November 2007.

[98] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient Field-Sensitive Pointer Analysis for C. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 37–42, New York, NY, USA, 2004. ACM.

[99] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Control*, 12:311–337, December 2004.

[100] F. M. Q. Pereira and D. Berlin. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 126–135, Washington, DC, USA, 2009. IEEE Computer Society.

[101] PHP: Hypertext Preprocessor, http://www.php.net/.

[102] Python Programming Language, http://www.python.org/.

[103] G. Ramalingam. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 16:1467–1471, September 1994.

[104] V. Raman. Pointer Analysis – A Survey, CS203 UC Santa Cruz, 2004. http://www.soe.ucsc.edu/ vishwa/publications/Pointers.pdf.

[105] D. Rayside. Points-to Analysis, 2005. http://www.cs.washington.edu/homes/mernst/teaching/6.883/lectures/points-to.pdf.

[106] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[107] C. G. Ribeiro and M. Cintra. Quantifying Uncertainty in Points-to Relations. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 190–204, Berlin, Heidelberg, 2007. Springer-Verlag.

[108] A. Rountev and S. Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.

[109] A. Rountev, A. Milanova, and B. G. Ryder. Points-to Analysis for Java using Annotated Constraints. *SIGPLAN Not.*, 36:43–55, October 2001.

[110] E. Ruf. Partitioning Dataflow Analyses using Types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 15–26, New York, NY, USA, 1997. ACM.

[111] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM.

[112] R. Rugina and M. C. Rinard. Pointer Analysis for Structured Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 25:70–116, January 2003.

[113] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

[114] D. Saha and C. R. Ramakrishnan. Incremental and Demand-Driven Points-to Analysis using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005. ACM.

[115] E. Salamí and M. Valero. Dynamic Memory Interval Test vs. Interprocedural Pointer Analysis in Multimedia Applications. *ACM Trans. Archit. Code Optim.*, 2:199–219, June 2005.

[116] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[117] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-to Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[118] O. G. Shivers. Control-Flow Analysis of Higher-Order Languages, PhD Thesis, Carnegie Mellon University, 1991.

[119] SML, http://en.wikipedia.org/wiki/Standard_ML.

[120] P. Sotin and B. Jeannet. Precise Interprocedural Analysis in the Presence of Pointers to the Stack. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 459–479, Berlin, Heidelberg, 2011. Springer-Verlag.

[121] M. Sridharan and R. Bodík. Refinement-Based Context-Sensitive Points-to Analysis for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[122] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-Driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[123] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[124] Q. Sun, J. Zhao, and Y. Chen. Probabilistic Points-to Analysis for Java. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 62–81, Berlin, Heidelberg, 2011. Springer-Verlag.

[125] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow Insensitive C++ Pointers and Polymorphism Analysis and Its Application to Slicing. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 433–443, New York, NY, USA, 1997. ACM.

[126] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[127] J. Whaley. Program Analysis using BDDs, Talk at MIT, 2005.

[128] J. Whaley and M. S. Lam. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 180–195, London, UK, 2002. Springer-Verlag.

[129] J. Whaley and M. S. Lam. Cloning-based Context-Sensitive Pointer Alias Analysis using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[130] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[131] Static Program Analysis, http://en.wikipedia.org/wiki/Static_program_analysis.

[132] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs.

In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[133] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise Points-to Analysis for Loop-Based Dependence Testing. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 262–273, New York, NY, USA, 2002. ACM.

[134] Q. Wu. Survey of Alias Analysis, http://www.cs.princeton.edu/ jqwu/Memory/survey.html.

[135] S. H. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.

[136] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 218–229, New York, NY, USA, 2010. ACM.

[137] J.-s. Yur, B. G. Ryder, and W. A. Landi. An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st international Conference on Software Engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM.

[138] S. Zhang, B. G. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step Toward Practical Analyses. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 81–92, New York, NY, USA, 1996. ACM.

[139] S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with Combined Analysis for Pointer Aliasing. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, pages 11–18, New York, NY, USA, 1998. ACM.

[140] X. Zheng and R. Rugina. Demand-Driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[141] J. Zhu. Symbolic Pointer Analysis. In *ICCAD 2002. IEEE/ACM International Conference on Computer Aided Design, 2002.*, pages 150 – 157, nov. 2002.

[142] J. Zhu and S. Calman. Symbolic Pointer Analysis Revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.