Static pointer analysis in intermediate representation for compilation optimizations

Irini Stavrakantonaki
Computer Science Department
University of Crete

A thesis submitted for the degree of
Master of Science

July 2016

Declaration of Authorship

I, *Irini Stavrakantonaki*, declare that this thesis titled, "Static Pointer Analysis In Intermediate Representation for Compilation Optimizations" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Master degree at University of Crete.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Author: _____
Irini Stavrakantonaki

The work reported in this thesis has been conducted at the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship, including funding by the European Commission through project ASAP (FP7 STREP #619706).

University of Crete
Department of Computer Science

Dissertation submitted by

Irini Stavrakantonaki

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

Author: _____
Irini Stavrakantonaki

Examination Committee: _____
Angelos Bilas, Professor, University of Crete

_____
Polyvios Pratikakis, Researcher, CARV-ICS, FORTH, Advisor

_____
Anthony Savidis, Professor, University of Crete

Departmental approval: _____
Antonis Argyros, University of Crete, Director of Graduate Studies

Heraklion, July 2016

To all those *known-unknown* people
who convinced *me*
that *human analysis* is far more
complex and rewarding than
pointer analysis.

Acknowledgements

Points-to analysis is regarded as a challenging and tough subject, and without being supported by many people around me, I would not even have arrived here. Graping this great opportunity, *as I feel that the end of this thesis marks also the end of a whole chapter in my life*, I would like to express my heartfelt thanks to the following people for having helped shaping my life, *even without their knowledge*, inspired and morally supported me during both my undergraduate and postgraduate studies.

My family for keep encouraging me in all of my pursuits and inspiring me to follow my dreams. Their emotional (and financial) support was enormous, and -of course- I will never be able to reward. I always knew that you wanted the best for me. Now that the days on my 'family' house are coming to an end, and you will 'get rid of me', hope your life will be calmer and quieter!

I must thank my advisor, Polyvios Pratikakis, whom I have worked with over the last two and a half years for his untiring support and guidance, for all the opportunities given to me even from the undergraduate years, for sharing his broad knowledge and for the many interesting discussions. Discussions that introduced me to the exciting world of static analysis, provided me with many valuable knowledge and kept me focused and excited during the year I worked on the project.

To my dear members of CARV laboratory: Christy, Foivos, Nikos, Maria, Giannis, and Lena, who made the everyday office life way more exciting and enthusiastic. My special thanks to Christy, the usual receiver of my questions, for her general help, for listening, offering me advice and of course for the many general discussions, tennis matches, drinks and laughs. I cannot but thank also Foivos for being always willing to help and share his deep, hacking (and not only) knowledge. Christy and Foivos also took the stale task of reviewing versions of my thesis and I thank them honestly. (I owe you. You deserve a bigger thank you than this. I will remember what you did. I appreciate the timing of your help. You helped right when I needed it most.)

Of course, I cannot but thank my friends in life without whom it would not be possible to bear life in Iraklio for another two years. This journey would not have been possible without them. Flora, Java, Panito, Lefa, Jasmin, Kotso, Tsek, Giorgo, Giorgo, Dimitri and Eleni, you deserve a big *Thankialisaurous*. Because a big thanks just isn't enough. Be careful though, he has big teeth. My honest thankings also to all the people that I have met these last two years, that changed me, offered me genuine free lessons of life and with whom we shared many unforgettable moments. To all my friends around Europe from AEGEE group, and tennis academy my deeply thankings.

(Please, note that the order of thankings is an outcome of a random generator, no special meaning on it.)

Abstract

Pointer analysis is a major component of most static program analyses. Static pointer analysis has many uses like increasing the precision and efficiency of compiler optimizations, driving auto-parallelization, and bug-finding tools. Usually pointer analysis implementations target one given source language and take advantage of specific features of that language to increase precision, *e.g.*, the type system, package system, or object inheritance mechanism. This, however, makes each implementation give different results with respect to precision and performance, and often requires a new implementation for each front-end of a multi-language compiler system. Conversely, implementing a pointer analysis for a compiler intermediate representation makes it more generic, while sacrificing precision that could have been achieved by working on the source-language abstraction level. This creates an interesting trade-off, where the optimal point would achieve maximum precision for the lowest-level language that is possible.

This thesis presents the design and implementation of a static pointer analysis for the LLVM intermediate representation language. We implemented an extension of Andersen's pointer analysis algorithm, broadly used to compute such information due to its scalability. The analysis is computed directly from the intermediate representation of LLVM, making it applicable to all languages that compile to the LLVM intermediate representation. The analysis is type-based, inter-procedural, flow-insensitive, and context-insensitive, following most applications of Andersen's algorithm, with the extension of field-sensitivity for increased precision on struct types. Our algorithm is capable of responding to a full variety of alias analysis queries and can provide substantially more precision than the standard algorithm due to its field-sensitivity. Evaluated using three different suites of 20 open-source C programs ranging from 1K to 230K lines of source code, we demonstrate our technique and provide a qualitative evaluation towards other algorithmic approaches.

Abstract

Η ανάλυση δεικτών είναι σημαντικό συστατικό των περισσότερων στατικών αναλύσεων. Η στατική ανάλυση δεικτών χρίζει πολλών εφαρμογών, όπως η αύξηση της ακρίβειας και της αποδοτικότητας των βελτιστοποιήσεων στους μεταγλωττιστές προγραμμάτων, οι αυτόματες παραλληλοποιήσεις κώδικα, και τα εργαλεία εντοπισμού λαθών. Συνήθως οι υλοποιήσεις ανάλυσης δεικτών βασίζονται σε γλώσσα πηγαίου κώδικα κι εκμεταλλεύονται τα χαρακτηριστικά των γλωσσών για να αυξήσουν την ακρίβεια των απαντήσεων τους, όπως πχ. το σύστημα τύπων. Ωστόσο, αυτό επιτρέπει σε διαφορετικές υλοποιήσεις να δώσουν διαφορετικά αποτελέσματα βάση της ακρίβειας κι της απόδοσης τους, καθιστώντας μ' αυτόν τον τρόπο, απαραίτητη την υλοποίηση διαφορετικού αλγορίθμου για κάθε διαθέσιμο front-end ενός πολυγλωσσικού μεταγλωττιστή. Αντίθετα, η ανάλυση δεικτών για μια γλώσσα ενδιάμεσης αναπαράστασης είναι πιο ευρεία, θυσιάζοντας, όμως, την ακρίβεια που δίνει μια γλώσσα πηγαίου κώδικα. Συνεπώς, βρισκόμαστε μπροστά σ' έναν ενδιαφέρον συμβιβασμό, όπου ο βέλτιστος τρόπος επιτυγχάνει τη μέγιστη ακρίβεια για τη `χαμηλότερη' δυνατή αναπαράσταση γλώσσας.

Η εργασία αυτή παρουσιάζει τη σχεδίαση κι υλοποίηση μιας ανάλυσης δεικτών για τη γλώσσα αναπαράστασης του LLVM. Υλοποιήσαμε μια επέκταση της ανάλυσης δεικτών τύπου Andersen, που χρησιμοποιείται ευρέως για τον υπολογισμό τέτοιας πληροφορίας κυρίως λόγω της κλιμακωσιμότητα (scalability) της. Η ανάλυση υπολογίζεται κατευθείαν στη γλώσσα αναπαράστασης του LLVM, καθιστώντας την εφαρμόσιμη σε όλες τις γλώσσες που μπορούν να μεταφραστούν σ'αυτή την αναπαράσταση. Σε όρους στατικής ανάλυσης, η ανάλυση είναι ένας type-based, interprocedural, flow-insensitive, και context-insensitive αλγόριθμος, ακριβώς όπως οι περισσότερες εφαρμογές του αλγορίθμου Andersen, με την προσθήκη όμως του field-sensitivity για περισσότερη ακρίβεια σε τύπους δυναμικών δομών. Ο αλγόριθμος είναι σε θέση να ανταποκριθεί σε μια πλήρη σειρά ερωτημάτων δεικτών και να παρέχει σημαντικά μεγαλύτερη

ακρίβεια απ' ότι ο κλασσικός αλγόριθμος, λόγω του field-sensitivity. Αξιολογήθηκε χρησιμοποιώντας 3 διαφορετικές σουίτες από 20 προγράμματα C ανοικτού κώδικα που κυμαίνονται από χίλιες μέχρι 230 χιλιάδες γραμμές πηγαίου κώδικα, αποδεικνύοντας την αποτελεσματικότητα της τεχνικής και παρέχοντας μια ποιοτική αξιολόγηση προς άλλες αλγοριθμικές προσεγγίσεις.

Contents

List of Figures

Chapter 1

Introduction and Motivation

Static analysis is of critical importance for software quality assurance. To minimize software defects developers complete software verification activities in order to ensure the quality of increasingly complex software systems. Static analysis is an automatic testing method that detects and prevents potential bugs without even running the program. Instead, the process executes the program in an 'abstract' way, by (over/under)-approximating all possible behaviors of the program. It provides an understanding of the code and can also help to measure code complexity, confirm agreement with coding standards and verify that the software will not fail at run-time.

The presence of pointers (e.g., function pointers, object accesses, closures, virtual methods) on programs complicates the analysis, as it is hard to know where pointers are pointing to at compile time by a simple inspection of the program statements. To obtain such information, static analysis depends on points-to analysis (also called pointer analysis) that resolves above issues by computing *points-to sets*: set of locations to which a pointer may point. This computation makes pointer analysis a fundamental technique for compiler analysis used to determine the indirect memory references that result from the use of pointers and pointer-based data structures.

We contribute to compiler optimizations by delivering a novel pointer analysis that uses the infrastructure of the state-of-the-art LLVM compiler. Indicatively, LLVM performs more than 60 different optimization algorithms. This indicates compilers' great need for effective and correct pointer analyses, as imprecise information is a major impediment to back-end optimizations that depend on pointer information.

This dissertation presents a precise pointer analysis algorithm based on intermediate code to achieve better compiler optimizations on C programs. The analysis over-approximately determines the properties of the program, the relations among its variables and delivers back a points-to graph that demonstrates the total of the generated points-to sets.

1.1 Goals & Contributions of this Thesis

To overcome the limitations of high-level pointer analysis, we design a pointer analysis that operates at low intermediate representation code instead of the source code. Our analysis is a flow-, context- insensitive and field-sensitive pointer analysis, and handles all the features of C programming language.

We present the core algorithm that is build on top of intermediate representation code (assembly-like), that is designed for "downward" instructions from high level languages (introduced in Chapter 4), and present a comparison with other common algorithms that are either built on top of intermediate or source code. We believe these results will prove valuable to designers of other static analyses for C.

More specifically, we target to deliver an analysis that accurately predicts every points-to set relation at every pointer dereference; distinguishes different fields of data structures; increases the overall understanding of program behavior; investigates potential trade-offs between analyses on source code and analyses on intermediate code; scales to POSIX, SPEC 2000 and SPEC CPU 2006 benchmark suites; and finally, provide compilers with a new efficient pointer analysis that can be regarded as the basis of further optimizations. To satisfy these goals we use LLVM modular compiler and develop a precise pointer analysis that can scale to large programs of 200k lines of code.

1.2 Benefits of Low-level Analysis

This section states the advantages of implementing an analysis on a low level intermediate representation code over the source code, as traditionally developers of analyses choose.

The result of a source code pointer analysis is annotated usually on the intermediate representation and can be maintained by following code transformations. However, by performing analysis at this level, we may face the problem of dealing with out-of-date pointer information after the performed code transformation, giving at the end a a less effective memory optimization. In addition to, high-level pointer analysis cannot be used at link-time or run-time, where the source code is unavailable.

Instead, the low-level analysis is fully source-language independent. It can be performed regardless of any source language and compiler. This compiler independence allows the analysis to be applied to the code that is executed on the back-end; so, the correctness of the analysis is not affected by optimizations, as they are performed on an earlier stage.

Regardless these, analysis on low level IR code is still harder to implement and remains a challenge to provide the same result but with better quality than source code analysis.

```
1  a = myarray[k];
2  myarray[l] = b;
3  c = myarray[k];
4  e = c + d;
```

(a) Without optimization

```
1  a = myarray[k];
2  myarray[l] = b;
3
4  e = a + d;
```

(b) With optimization

Figure 1.1: Pointer information makes optimization possible

1.3 Clients of Pointer Analysis

An absolute evidence of the importance of pointer analysis lies on its clients. Many of them find pointer information an essential input to deliver better code, specially when the underlying points-to analysis is accurate. Perhaps the most important applications today are in software engineering tools and for optimizing compilers.

1.3.1 Pointers and Software Engineering Tools

Several kinds of software engineering tool applications have the need of pointer information. Typically we meet pointer analysis on applications like call graphs construction [46], dependence analysis and optimization [6], slicing [48, 53, 26] and design pattern detection [40].

Program slicers [48, 53], debuggers [8, 17] and software visualizers [27] are used broadly to aid program understanding and help to the detection of bugs provoked by memory leaks, wild pointer and security holes, thus they absolutely need accurate pointer information.

Static analysis tools [7, 13] and model checkers [25, 21] are generally used for bug finding and identifying security vulnerabilities. In languages such as C/C++ and Java, pointer analysis is constantly found in all these tools, where it forms a foundation for other analyses (e.g., data-flow analysis).

1.3.2 Pointers and Compiler Optimizations

Compilers adopt pointer analysis techniques to optimize programs with pointers as it provides memory dependence information, crucial for many optimizations. We demonstrate, here, some of these optimizations that are clients of pointer analyses. We also provide examples to confirm the need for pointer analysis on more effective program optimizations.

Figure 1.1(a) shows an example for a part of code that has no possible optimization without given alias information and Figure 1.1(b) shows its adaptation if alias information is provided. Assuming that A and B do *not* alias, we can eliminate redundant load/store, and schedule code as shown in (b).

Overall, the basic compiler optimizations are: *register allocation*, *loop invariant code mo-*

tion, *common sub-expression elimination*, *dead code elimination*, *live variables*, *instruction scheduling*, *redundant load*, and *store elimination*.

1.4 Organization of this Thesis

The rest of this thesis is organized as follows.

Chapter 2 provides a broad survey over pointer analysis methods and practices, gives a background on the analysis, its concepts, the existing techniques and generated results, and describes some of the related work on state-of-art pointer analysis algorithms existing in the literature. Chapter 3 gives a theoretical description of our analysis. Chapter 4 follows with the full technical implementation of our technique. Chapter 5 presents the empirical evaluation between our analysis algorithms and other state-of-the-art algorithms in the literature, and presents our limitations and strengths. Chapter 6 is based on the found limitations of the previous chapter speculates on future work. Finally, Chapter 7 concludes the thesis by recapping our contributions.

Chapter 2

A Survey of Pointer Analysis Methods

This chapter introduces the theoretical background knowledge and related work in the field of pointer analysis. We do not provide a thoroughly detailed background on static analysis, but instead we focus on key terminology only used in this thesis.

We first introduce the fundamental concepts and terminology complicated in pointer analysis research, as well as some of the traditional approaches applied to perform pointer analysis and their drawbacks. Then, we refer to some well-known pointer analysis algorithms in the literature that -we believe- are closely related to our work given their importance in program analysis and software engineering.

2.1 Pointer Analysis In Theory

Pointer analysis is a static analysis technique that aims to determine the points-to relationships in the heap, as well as the addresses in the heap of the program that every variable can refer to. The result of the analysis, (often) in the form of dependence edges, represents all the pointer information of the program.

Designing a pointer analysis algorithm can bring big challenges, with many different methods that trade accuracy for efficiency. Though, still the issue of delivering a scalable accurate analysis is considered an open problem. The greatest challenge is to report a small number of pairs of pointers (points-to sets) while remaining correct, in order to improve accuracy.

2.1.1 Pointer Analysis Methods

There are many design options regarding implementing a pointer analysis that can ensure the accuracy of a pointer analysis. Some of these are:

- intra-procedural or inter-procedural;
- context sensitivity or insensitivity;
- flow sensitivity or insensitivity;
- field sensitivity or insensitivity.

The level of accuracy of the points-to set, stated by client applications, differs greatly from analyses that are intra-procedural to inter-procedural, with or without flow, context or field sensitivity. The next sections explain in more detail the differences between these approaches.

2.1.1.1 Intra-procedural vs Inter-procedural

In intra-procedural analysis, each function is analyzed locally by creating a summary of points-to sets. Parameters, globals and pointers returned by function calls are ignored, instead only local variables affects the result.

In contrast, an inter-procedural analysis gathers information from the entire program. Each function is analyzed with regard to the incoming parameters and globals defined or/and used in that function.

Intra-procedural analyses are fast, but inaccurate. Whereas, inter-procedural analyses are slow but can provide with crucial information in many cases, like detecting memory leaks, as pointers cannot leak if their life cycle is within a function.

2.1.1.2 Context-sensitive vs Context-insensitive

A context-sensitive analysis analyses all the callees for each caller by separating calls that map to different contexts. In contrast, a context-insensitive analysis ignores the callers and are regarded less effective than the former ones. Consider a simple program in Listing 2.1:

```
1 void *foo(void* q) {
2   return q;
3 }
4 int a, b;
5 int *p = foo(&a);
6 int *q = foo(&b)};
```

Listing 2.1: Context sensitivity example

An insensitive analysis, as it ignores the callers, connects both calls to the same variables, would conclude that q and p are alias, and can point either to a or b. In contrast, a sensitive analysis separates different calls of foo and would answer that p and q are not aliases.

2.1.1.3 Flow-sensitive vs Flow-insensitive

A flow sensitive analysis considers the order of statements and computes one answer for every program point.

A flow-insensitive analysis ignores the order of statements, and computes one answer for the entire program. Consequently, it produces a smaller points-to set.

In summary, flow-insensitive analyses are fast, but inaccurate, whereas, flow-sensitive analyses are slow and in general ad-hoc.

```
1 void foo(int q) {
2   x = 4;
3   ...
4   x = 5;
5 }
```

Listing 2.2: Flow sensitive example

Consider a simple program in Listing 2.2: a flow-sensitive analysis would decide that x has the value 4 after the first assignment and the value 5 after the second assignment. In contrast, a flow-insensitive would answer that x has either value 4 or 5.

2.1.1.4 Field Sensitive vs Field Insensitive

Field sensitive analyses distinguishes different fields of structures by using unique variables to model each field, while field insensitive analyses regard the aggregates as one, and model each one with a single variable, discarding all field information. As a conclusion, they produce smaller points-to sets than the sensitive algorithms, thus, they are less effective, but faster. Example in Listing 2.3 clarifies this:

```
1 int *a, b, c, d;
2 struct STR{int *first, int *second;}
3
4 STR str1, str2;
5 str1.first = &b;
6 str1.second= &d;
7 str2.first = &c;
8 a = str1.first;
```

Listing 2.3: Field sensitive example

A field-sensitive algorithm decides that a pointer variable has a points-to set only with first field, in contrast to the insensitive algorithm that answers that a has a set with str1 elements: b and d, which is in fact wrong.

Program Statement	Relation	Generated Edge
x = &y	address-of	x points-to y
x = y	copy	if y points-to w, then x points-to w
x = *y	load	if y points-to z and z points-to w, then x points-to w
*x = y	store	if x points-to z and y points-to w, then z points-to w
x = y + o	offset	if y offset o points-to w, then x points-to w

Table 2.1: Graph construction: how program statements contribute to graph edges

2.2 Selection of Well-known Algorithms

This section lists briefly some well-known algorithms of every pointer analysis method. We will describe in more detail the Andersen-style algorithm as our method is basically based on the extension of this technique.

2.2.1 Context-, Flow- Insensitive (CIFI) Algorithms

In this group of algorithms we meet the well-known Andersen's [5] and Steensgaard's [42] analyses. These are, by general admittance, the simplest and most common types of pointer analysis algorithms: insensitive to both flow and context. Although, this kind of algorithms is mainly used by the commercial compilers until today.

Fahndrich's algorithm [15] is an improved (by orders of magnitude) Andersen-style analysis. Extra to Andersen's, it collapses cycling constraints into a single node and propagates only on selective constraints.

Rountev in [39] also suggests an optimization over Andersen's analysis, by collapsing cyclical constraints that give the benefit of ignoring duplicate computations.

2.2.1.1 Andersen's Inclusion-based Pointer Analysis

Andersen's [5], perhaps the most well-known pointer analysis algorithm, has been adopted by state-of-the-art compilers such as GCC and Open64, mainly due to its scalability. It was embraced also by LLVM platform until the 2.6 version but till then it has been removed. It consists also the main focus of this thesis. Andersen-style (i.e. inclusion-based) pointer analysis is a flow-insensitive, context-insensitive analysis.

Andersen's analysis initially translates statements of the input program to constraint sets than encode all possible may points-to relations. Table 2.1 summarizes the map between statements and constraints. Then, it builds and solves a set-constraint directed graph, named *points-to graph*, that models the whole heap of the entire input program.

In the generated graph, the vertices represent the pointer locations in the program. An edge between two vertices, a->b, represents a points-to relation between a and b, meaning

that a pointer variable points-to b at a program point. Theses edges are created by the pointer-manipulation instructions in the program, as shown in Table 2.1.

For enabling field-sensitivity, we introduce the 'offset edges'. Every field of a struct is mapped to a unique variable. For a field-insensitive algorithm, this instruction does not need handling, therefore it returns the pointer to the struct, so every access to its fields gets the same pointer.

Andersen's analysis provides a good trade-off between efficiency and precision. It has played a major role in many other analyses used for many different purposes, including program slicing [31], precise pointer analysis [23, 33], bug detection [45], inter-procedural SSA analysis [9].

The size of the points-to graph is regarded as a great evidence of the accuracy of the analysis. That is because points-to sets stand as a measure of analysis precision; smaller ones mean more precise results for pointer analysis, thus more effective program analysis in general. Therefore, fewer sets on the graph, represent, in general, a better analysis.

2.2.2 Context-, Flow- insensitive, Field-sensitive (FA) Algorithms

Flow-insensitive, context-insensitive, field-sensitive (FA) algorithm have exactly the same characteristics as our approach. FA is a pointer analysis similar to the well-known pointer analyses by Andersen and Steensgaard [42], but extended with field-sensitivity. For instance, for an aggregate str with two pointer fields first and second, Andersen's analysis creates a single points-to set for both *(str.first) and *(str.second), while, the FA analysis creates different points-to sets for *(str.first) and *(str.second).

Pearce et al. in [36] propose a FA analysis based on source code that treats field accesses unsoundly: pointers to aggregate types are casted to structurally unrelated types, making possible to recognize wrong transfers. In their implementation, function arguments are mapped to numbers. Thus, in an indirect function call, we access them with the offset to that function variable. Similarly, the external library functions are mapped to hard-coding function tags and auxiliary variables replace the nested pointer dereferences.

Yong in [54] gives an evaluation of the precision difference between field sensitive and insensitive analysis, and find that "distinguishing individual fields of structs is important". They prove so in their evaluation: the average pointer sets produced by field-insensitive analysis are at least twice as large as the sets produced by field-sensitive analysis (10x larger in the worst case).

2.2.3 Context-, Flow- sensitive (CSFS) Algorithms

The algorithms that are both context and flow sensitive (CSFS) are considered the most precise pointer analysis algorithms.

Emami [14] proposed a very precise and accurate CSFS points-to analysis, extended with field-sensitivity, that maps each stack variable to a unique location set. Although this analysis does not manage to scale to large programs.

Wilson and Lam [51] suggested also a CSFS that gathers points-to information for each function. Their approach scales well to programs of five thousand lines of code.

Chaterjee et al. [11] , following Wilson and Lam model [51], introduced summaries for each function by using a single safe transfer function. This technique showed improvements regarding space but failed to scale over five thousand lines of code.

Cheng and Hwu [12] with their technique of partial context-sensitivity demonstrated scalability to hundreds of thousands of lines of code.

2.2.4 Context-sensitive Flow-insensitive (CSFI) Algorithms

These are perhaps the most rare pointer analyses. Algorithms that are sensitive to the context but remain insensitive to the flow of the program. There are many CSFI algorithms implemented [16]. Some of them are using binary decision diagrams (BDD) [50, 57] to efficiently solve the alias analysis problem in an accurate way.

2.3 Pointer Analysis and LLVM compiler

Due to the fact that our design uses LLVM compiler framework, in this section we refer to already implemented analyses that are either already adopted by LLVM or use its framework to depend their analysis -as we intend to do.

2.3.1 Implemented Analyses on LLVM

-anders-aa An -anders-aa pass implementing the well-known "Andersen's algorithm" for inter-procedural analysis was available in the LLVM infrastructure until version 2.6. Although, it was removed on later versions due to numerous bugs and its low maintainability. This algorithm was a flow-insensitive, context-insensitive, and field-insensitive analysis that is widely believed to be fairly precise.

-basicaa This is LLVM's default basic alias analysis, an intra-procedural flow-insensitive analysis that collapses all address-taken variables.

-ds-aa The -ds-aa pass implements the 'full Data Structure Analysis' (DSA) algorithm: a modular, context-sensitive, flow-insensitive, field-sensitive alias analysis with full-heap cloning. The quite scalable DSA alias analysis can respond to various alias analysis queries, and provide context-sensitive information as well. Though, there is no support yet in LLVM for must-alias information.

2.3.2 Implemented Analyses over LLVM

Many pointer analyses are using the compiler infrastructure of LLVM to depend on their analyses. These algorithms are designed upon the intermediate representation of LLVM - its key feature.

LevPA pointer analysis [55] is a flow-, context- sensitive algorithm for C programs that scales well to millions of lines of code. LevPA pre-computes points-to information directly on LLVM IR and then analyzes pointers 'level by level', according to the generated points-to levels of the points-to graph, with intention to perform a fast and accurate full parse flow-sensitive analysis still in a flow-insensitive style.

Lhotak and Chung [29] propose a points-to analysis with efficient strong updates by computing points-to information directly on the LLVM IR of a program. They suggest an efficient strong update analysis that combines flow-insensitive and flow-sensitive points-to analysis.

Hardekopf and Lin [23] with their flow-sensitive pointer analysis compute points-to information directly on the LLVM IR and provide a precise algorithm that generates a sparse value-flow graph that captures the def-use chains of all variables. or Wave implemented by Pereira and Berlin [37] is a useful pointer analysis solver. Andersen's pointer analysis can be written easily by choosing an appropriate solver like Wave.

SVF [44] is a static analysis tool, built upon LLVM 3.7, that enables scalable and precise inter-procedural dependence analysis for C programs. It uses a field-sensitive Andersen's pointer analysis to perform its pre-analysis phase. And can generally accept points-to information generated by any other pointer analysis.

Ye, Sui and Xue in [43] proposed a selective flow-sensitive pointer analysis for multi-threaded C programs. Their algorithm (FSAM) is built on top of SVF [44] and bases on its pre-computed value-flow information. Then, FSAM applies a series of thread interference analysis phases for multi-threaded C programs.

NeonGoby [52] is a static analysis tool f effectively detecting errors in alias analysis implementations. It currently checks alias analyses implemented on the LLVM framework. It has identified 29 bugs in two popular alias analysis (-ds-aa and anders-aa). NeonGoby can be regarded as a great tool for verifying about the correctness of alias analyses.

2.4 Summary

As we demonstrate, various optimizations, like code motion and parallelization, get an excessive support from pointer analysis. We think it is still really promising and deserves to be further studied, the design of analyses that fit particular optimization needs and provide specific pointer information. Discovering the lack of an Andersen-style analysis on the LLVM platform, we focus on this thesis on delivering a novel Andersen-based, context-, flow-insensitive, field-sensitive analysis mainly designed for this powerful, state-of-the-art compiler. The next chapters introduce the algorithm used (Chapter3) and its implementation (Chapter4).

Chapter 3

Pointer Analysis Algorithm

In this chapter, we first present the low-level intermediate representation language on which we perform the pointer analysis. We then illustrate the algorithmic method we use in our implementation and its fundamental features.

3.1 Analysis Outline

We develop a type-based analysis to identify the points-to sets of a program and detect which pointers may point to the same memory locations at run-time. Broadly, our proposed algorithm is an inclusion-based (i.e. Andersen-style [5]), flow-insensitive, context-insensitive, field-sensitive algorithm for low-level code. It combines elements from the pointer analysis implemented by Pratikakis in [38], appropriated modified to work on low-level intermediate representation. Pratikakis introduced a context-sensitive, inclusion-based points-to analysis to characterize locks and locations in the program, and this context-sensitivity stands as a key difference to our context-insensitive approach.

3.2 The Core Intermediate Language

```
e ::= v | x | e1 e2 | (e1, e2) | e.j |
      let f = e1 in e2 | f^i | fix^i f : t.e
v ::= n | λx : t.e | (v1, v2)
t ::= void | int | t x t | t→t | ref^l
l ::= l
C ::= 0 | {c} | C ⋂ C
c ::= t ≺ t' (subtyping) |
      l ≺ l' (location flow)
```

Listing 3.1: A core language

Here, we present the core basis of the intermediate language our pointer analysis is conducted on. For the generation of the constraints, as we describe in next section, we use a type system, that proves judgments of the form $C; \Gamma \vdash e : \tau$. Meaning that expressions e

have type τ under type assumptions Γ and constraint-set C.

Figure 3.1 presents our core language based on lambda calculus. C is a set of constraints c. Within the type rules, the judgment $\mathsf{C} \vdash \mathsf{c}$ indicates that c can be proven by the constraint set C. It contains the basic variables x, functions $\lambda\mathsf{x} : \mathsf{t.e}$, and function calls $\mathsf{e\ e}$. Types t include void, integer int, pairs $\mathsf{t} \times \mathsf{t}$, functions $\mathsf{t} \rightarrow \mathsf{t}$ and references (ref^i) types. For modeling data structures, such as structs and unions, we add Pairs types (e, e). To enable field-sensitivity, we introduce $\mathsf{e.j}$ for accessing fields. $\mathsf{e.j}$ returns the jth element of the pair $(\mathsf{j} \in 1, 2)$. Functions declarations are stated by the expression: $\lambda\mathsf{x} : \mathsf{t.e}$, where x is the formal argument of the function and t the return value. Function calls are typed with t→t expression. The language also contains expressions of $\mathsf{let\ f = e1\ in\ e2}$ which map the name f to expression $\mathsf{e1}$ in the scope of expression $\mathsf{e2}$. To support recursion, we add the type fix^i $\mathsf{f : t.e1}$ that recursively maps the name f to expression $\mathsf{e1}$ in the scope of $\mathsf{e1}$.

```
Constants ::= {c1, c2, ..}
Variables ::= {v1, v2, ..}
Pointers  ::= {p1, p2, ..}
Instructions(i)   ::=
  Allocate memory        | p = alloc(v)
  Copy memory            | p = y z
  Call function          | p = call(F)
  Load into pointer      | p = *p1
  Store from pointer     | *p = p1
  phi-function           | p = phi(p1:l1, p2:l2)
  Return function        | return ; return(c)
  Arithmetic operation   | v = v1 op v2
```

Listing 3.2: Instruction set of our language of pointers

Finally, the syntax ref^l annotates an occurrence of pointer reference with a label l. We use a label to specifically refer to each occurrence of a pointer in a program. This way, the analysis can discriminate pointer types and handle the addresses they point-to at run-time by associating labels with types. As a result, when we recognize a points-to relation among two variables, there is also a flows-to relation among their labels. This flow-to relation implies the 'location flow' which is demonstrated on the generated points-to graph with a new edge.

We can, hence, describe and define types by equations, which are translated based on the instructions of the program (a simplified instruction set of our language is illustrated in Listing 3.2). While we parse the equations generated by the constraint-solving of the program, we type every variable name with a new label name.

As our method supports field-sensitivity, we set different labels for every field of the structures we identify, as well. We manipulate as well the cyclic linked data structures. We represent these data structures abstractly as special sets of equations. Thus, each equation relates a variable (meaning a memory address) to a node of the data structure that is mainly

```
1 if (even) then x := 0
2 else x := 1
3 return x
```

(a) Without SSA

```
1 if (true) then x1 := 0 else x2 := 1
2 x := phi(x1, x2)
3 return x
```

(b) With SSA

Figure 3.1: SSA-form example

represented by a type applied to variables. Consequently, we can identify a potential cyclic data structure and not continue the typing of the structure but return its initial type as already defined.

Note that apart from the usual memory allocation and pointer arithmetic instructions that our language of pointers supports, we also contain ϕ-functions to ensure the Static Single Assignment (SSA) property of our language. The following section describes SSA in detail.

3.2.1 SSA form

By terminology, to be in valid Static Single Assignment (SSA) form, each variable can only be defined by a single definition. This way, we create new variable names at each program point, and our analysis can infer new information. Thus, each of the alloc, copy, call, load, store, return, pointer arithmetic, and intersections instructions define new variables, whose names are associated with information. For example, the memory allocation instruction $p = \mathsf{alloc}(y)$ copies y to p, and binds p to a memory chunk.

The rule of SSA is broken if the same variable is defined in two different control-flow paths. To resolve this issue, the concept of ϕ-functions is brought in. The ϕ-function returns the value of the argument that is set on the control-flow path that was taken.

In the a example of Figure 3.1, x is defined in two different control-flow paths. To transform this into SSA form (b), we need to add a ϕ-function (the right example).

The value that ϕ(x1, x2) returns depends solely on which of the two control flow paths it has taken. If it passes through the true branch of the if statement, the resulting value is x1, otherwise the resulting value is x2.

3.3 Subtyping

We also support standard subtyping rules in our language, similar to those presented by Cardelli [10]. Subtyping is a key feature in languages dealing with pointers that proves the relation between types.

The fundamental subtyping rule says that 'if $\tau 1$ is a *subtype* of $\tau 2$ ($\tau 1 \preccurlyeq \tau 2$), then a program can use the value of $\tau 1$ instead of the value of $\tau 2$'. Also, 'if e is of type τ, and τ is

a subtype of τ', then e is also of type τ' '. To formalize the requirements on subtyping, we define a rule of subsumption. According to this, if $\tau < \sigma$ then an expression of type τ has type σ:

$$\frac{\Gamma \vdash \epsilon : \tau \ \tau < \sigma}{\Gamma \vdash \epsilon : \sigma}$$

Struct types We apply subtyping to struct types. Its type includes a set of all the types of the fields together with their labels. For example, we assume a struct Line with two pointers x, y as fields and a struct *Triangle* with three pointers *x, y, z* as fields. As *Triangle* contains all the fields of *Line* with the same type, we say that *Triangle* is a subtype of *Line* (Line $\preccurlyeq$ *Triangle*). In this case we say that we '*conflate*' the labels of the fields, meaning that we create a '*flow-to*' relation among the locations of every elementary field type of the data structures.

According to this, the subtyping rule for structs τ_1 and τ_2 generates l $\preceq$ l' for their labels and applies subtyping in both elements of the structs:

$$\frac{\mathrm{C} \vdash \mathrm{l} \preceq \mathrm{l'} \qquad \mathrm{C} \vdash \tau_1 \preceq \tau_1\text{'} \qquad \mathrm{C} \vdash \tau_2 \preceq \tau_2\text{'}}{\mathrm{C} \vdash \tau_1 \times^{l} \tau_2 \preceq \tau_1\text{'} \times^{l'} \tau_2\text{'}}$$

Conflating all the field types of the structs enforces our analysis to be correct, as it cannot report any false negative, or lose any flow of the program, though, it suffers in precision and performance. For cases of huge structs e.g., with 100 and more fields, we need to conflate *every* field of the struct, even if it is not read/written in any other program point. This would cost both on entries to Γ but also on extra edges on the points-to graph. This can be addressed by using lazy annotation (see Chapter 6 for a more detailed description).

Function types Also different functions are allowed to be in a subtyping relation. Considering two function types $\tau 1 \rightarrow \tau 2$ and τ'1 $\rightarrow \tau$'2, we subtype between the argument and the result types of the two functions. Specifically, we subtype the arguments of the second function to the arguments of the first, and the result of the first function to the result of the second.

According to this, the subtyping rule for functions generates l $\preceq$ l' constraint for their labels and also applies the subtyping in the argument and the return type of the functions:

$$\frac{\mathrm{C} \vdash \mathrm{l} \preceq \mathrm{l'} \qquad \mathrm{C} \vdash \tau_1\text{'} \preceq \tau_1 \qquad \mathrm{C} \vdash \tau_2 \preceq \tau_2\text{'}}{\mathrm{C} \vdash \tau_1 \rightarrow^{l} \tau_2 \preceq \tau_1\text{'} \rightarrow^{l'} \tau_2\text{'}}$$

Reference types Subtyping is also applied to pointer reference types. In fact, a pointer type τ **ref** is a subtype of τ' **ref** if and only if $\tau = \tau$'. Following, the subtyping rule for

references generates l $\preceq$ l' and applies the rule for its elementary location as well:

$$\frac{C \vdash l \preceq l' \qquad C \vdash \tau \preceq \tau' \qquad C \vdash \tau' \preceq \tau}{C \vdash \mathrm{ref}^{l}(\tau) \preceq \mathrm{ref}^{l'}(\tau')}$$

Recursive types Recursive types are presented in our core language, since it provides typing recursive functions and data structures. For example, a list of elements of type τ (a τ list) is either empty or it is a pair of a τ and a τ list.

```
τ list = unit + (τ x τ list)
```

This is a recursive equation for a list. In parallel, we introduce a recursive type constructor: μt.τ This means that type variable t is bound in τ.

The tricky thing on implementing the subtyping recursive types is to manage to break the circle of annotation and subtyping of recursive types. We perform so by holding in a structure the initial type. This way, when we traverse it again we return the initial annotated type and avoid of falling into infinite loops of subtyping and annotating.

As a result, subtyping recursive types aids analysis to not lose the flow of the program, providing more accuracy on the results. Though it costs on number of labels, as it annotates with new labels the fields it visits.

3.4 The Type-Based Analysis

As stated above, and also described in Andersen's algorithm (see Chapter 2), our analysis mainly describes and defines types by equations, which are translated based on the statements of the program. It applies approximations that leaves only an unordered set of assignment statements which is sufficient enough to represent the original program. The result is a similar program that executes every assignment after the other. In other words, we generate constraints from the code. Pointer assignments are viewed as constraints and are decomposed into four instruction types: address-of, copy, store and load (see Table 3.1). We make use of set-constraints, a method pioneered by Aiken [4] and Heintze [24]. Then, by parsing the equations generated by the constraint-solving of the program, we type every pointer variable with a new label name, as our type system implies. The language that we use is summarized by the following constraints:

$$p \supseteq q \mid p \supseteq \{q\} \mid p \supseteq *q \mid *p \supseteq q \mid *p \supseteq \{q\}$$

Thus, if p and q are constraint variables, and the constraint set is p $\supseteq$ x it means that p *points to* x. Also, the assignments hold constraint information. Implicating that a simple assignment like $x = y$, means that x takes on the value of y or in other words that the value of x includes the value of y.

Instruction Type	Syntax	Translation
Alloca	a = &b	loc(b) $\in$ pts(a)
Assign	a = b	pts(a) $\supseteq$ pts(b)
Load	a = *b	$\forall$ v $\in$ pts(b) : pts(a) $\supseteq$ pts(v)
Store	*a = b	$\forall$ v $\in$ pts(a) : pts(v) $\supseteq$ pts(b)

Table 3.1: Effects of instructions to the graph

According to this method, we can regard function calls as pure assignment statements. Doing so, we remove part of the control flow and simplify the initial program. Specifically, in function calls, we assign the actual arguments to formal arguments and the result of the function to the result of the call, according to the typing rule of applications:

$$\frac{\Gamma \vdash \mathrm{e_1} : \mathrm{T} \rightarrow \mathrm{T'} \qquad \Gamma \vdash \mathrm{e_2} : \mathrm{T_2}}{\Gamma \vdash \mathrm{e_1\ e_2} : \mathrm{T'}}$$

For instance, in a function call like $x = foo(y)$, we should expect a set of assignments $p = y$ and $x = r$, where p stands as the parameter of foo and r the returning value of foo.

All these generated assignment statements create maps of variables to sets of *labels*. These maps are the points-to sets. The goal of the pointer analysis is to detect the points-to sets of every program variable and to manage to do so with the smallest possible size of points-to sets. For instance, suppose that we have a program with the expressions: p = & x; q = &p; *q = &y.

As a result, the points-to sets are: $< *p, x >, < *q, p >, < **q, x >, < **q, y >$. Table 3.1 summarizes (in simplicity) how we translate all the equations to create the points-to sets. $pts(v)$ represents the points-to set of a variable v, whereas $loc(v)$ represents the memory location stood for v.

We represent all this points-to set information, in a data structure called points-to graph. This graph has one node for each pointer variable. For each points-to relation, there is a directed edge from the pointee to the pointer. This way, the edges of the graph shows us the constraints between variables of the program.

In the following sections we describe the data structures of Γ and points-to graph, used to hold and represent all the pointer information of the analysis.

3.4.1 The Gamma

Our pointer analysis goes over the entire code of the program, associating pointer variables with specific labels. As implied by our type system, it is Γ that gathers all the type assumptions (variables and pointers) of the given program. Therefore, we say that Γ is an environment mapping variables x to types. Pointers that are stored in Γ represent the nodes of the points-to graph. These nodes will be also referred here as labels or rhos.

In our implementation, Γ is represented as a simple data structure of a symbol table that assembles all the type information. Every function holds its own local Γ environment. Whenever we type a new symbol, we first check if exists in Γ. Variables which are added for the first time in Γ, hold information associated with their register name, their annotated type, the program scope and the Γ we store them into. Pointer variables are mapped with their label, as well.

3.4.2 The Points-To Graph

The points-to graph (PTG) is the fundamental structure used in our Andersen-style algorithm. It is built up *on-the-fly*, implying that the points-to analysis concludes automatically which methods are called at each call-site. The PTG just models the heap and the state of the pointer variables.

The vertices of the PTG graph represent pointer variables, and the directed edges represent the constraints between variables. Specifically, an edge indicates a may points-to relationship for any point in the program. If an edge goes from variable x to y, then y's points-to set must include x's points-to set.

The analysis satisfies the constraints expressed in the graph by propagating the variables' points-to sets along the directed edges of the graph. Thus, for each edge x $\rightarrow$ y, we have that pts(y) $\leftarrow$ pts(x), where the left arrow represents an update on the set. As mentioned above (in 3.2), direct constraints (a = b - assign instructions), generate simple directed edges from b node to a node. In instructions that initiate constraints, like alloca (a = & b), it comes that node a's points-to set includes *loc(b)*.

As the analysis updates variables' points-to sets, it adds on-the-fly new edges to the graph to represent constraints of load and store instructions that can be determined using the new points-to information. If variable b's points-to set is updated, then for each load instruction (a = *b) a $\supseteq$ $*$b and each v $\in$ pts(b), we add a new edge v $\rightarrow$ a. Similarly, for each store instruction, $*b = a$, we add a new edge a $\rightarrow$ v.

If there is a virtual function call over local variable x, and if we know already that x can point to variable y, then the called function is looked up in the Γ and a new edge is created from the call site to this function. The PTG depicts the 'flow' points-to information between actual and formal parameters of functions. For example, if there is an edge between call-site callMe and function myFunction, then we infer an assignment to the i-th formal argument of myFunction from the i-th actual argument at callMe, for every i.

PTG depicts also field-sensitivity's attributes of labeling different fields of structures. For example, if there is a field access of a structure which is stored in a variable, then, an edge from this specific field to the stored value is created. In case, we assign values of one

field of a struct to another, then we conflate their types, as described above.

A complete points-to graph allows the computation of the points-to sets, and the queries on the final alias information from the sets. For instance, if a pair (p,q) belongs to the points-to set, then p *may* point to q, and for any pair (r,s) not found in the set, r *must* not point to s.

Chapter 4

Pointer Analysis Implementation

In this chapter we give a detailed description of the implementation of our analysis which is based on Low Level Virtual Machine (LLVM)'s [28] type system and light-weight intermediate representation (IR). The choice is driven by the fact that LLVM is a state of the art, modern, modular compiler, with a well-maintained and documented code-base. Basically, our analysis is a fork from the original LLVM repository.

Our flow-insensitive, context-insensitive, field-sensitive pointer analysis is implemented in C++ and makes use of LLVM compiler infrastructure which translates from several front-ends (e.g. C, C++, etc.), into an intermediate representation (LLVM IR). Our implementation uses this LLVM IR to reason about pointer information of C programs.

4.1 Integration with LLVM

In this section we present the Low Level Virtual Machine (LLVM) [28] which we use on our analysis and describe the way we interact with it.

LLVM is a state-of-the-art compiler framework, introduced by Chris Lattner and Vikram Adve in 2000 at the University of Illinois. It contains 3 compilation stages, as shown in Figure 4.1. These three stages are the front-end, the optimizer and the back-end. The *front-end* parses source language and produces LLVM IR code. The *optimizer* transforms IR into an optimized equivalent one. And the *back-end* takes IR and produces machine code optimized for a specific CPU (e.g., X86, Power-PC, ARM). Obviously, LLVM's advantage is that it can support multiple source languages and target architectures. It can easily support a new source language without the need of writing a new optimizer and target language for it.

Mainly, our analysis is an addition of a new Pass on LLVM. We benefit of LLVM's design as a reusable library, instead of the classic model of a monolithic command-line compiler, like Java Virtual Machine (JVM) and .net system.

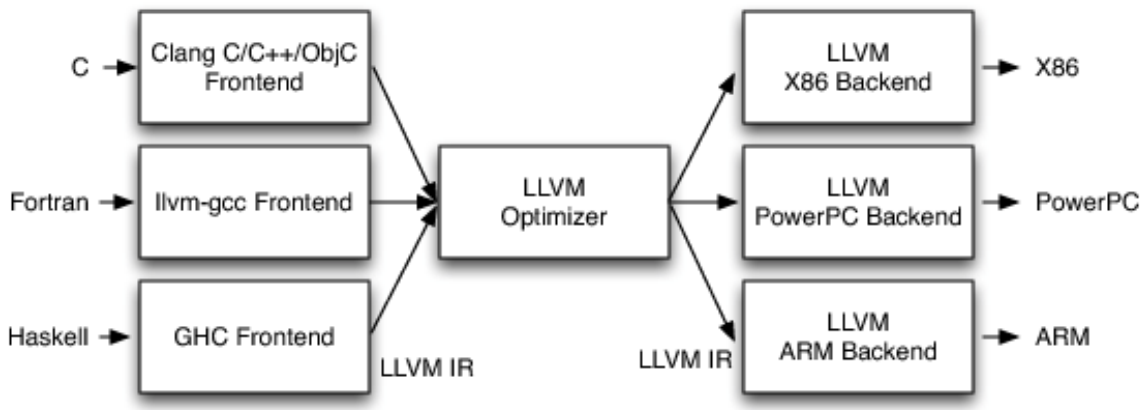

Figure 4.1: LLVM Architecture (picture from Lattner's book)

All the LLVM core libraries depend on a LLVM code representation, known as the LLVM Intermediate Representation ("LLVM IR"), which is used during all phases of LLVM compilation design and is regarded as a central architectural feature of LLVM. The IR is a Static Single Assignment (SSA) representation, generated by the front-end, that supplies low-level operations with a capability of expressing also "all" high-level languages in a neat way. The LLVM IR cannot be regarded as a type-safe language, as its type system allows several casts, calling functions with incorrect signatures, accessing invalid memory, etc. This unsafeness of IR, though, costs to our approach as we lose useful type information.

Our implementation takes a program in LLVM IR format as input and determines the pointer information and relations among locations. Our pass conforms to the standard LLVM passes, so it can be used by any other build-in analysis in LLVM. By default, the Pass generates a points-to graph. This information can be extracted from another Pass that takes ours as a prerequisite and make alias queries using our public interface.

The analysis supports the majority of LLVM instructions and handles all the LLVM types except the Vector type, which is not a supported type in C programming language in which we focus our implementation on. In the following sections we refer in detail to the mapping between LLVM language and type system to our translation and new (annotated) type system.

4.2 How We Implement Andersen-based Analysis on LLVM

To begin with the main implementation, we first refer to the creation of the new type system that we depend on, and describe how LLVM IR instructions effect the generation of points-to relations and contribute on the built of the points-to graph.

Initially, our analysis traverses once through the given program and collects all the

function prototypes. We need these prototypes for the typing of (library) function calls. Then, on a second traversal, it detects all the memory objects like globals, heap and stack allocated objects, and stores them into an environment that gathers all the type information, named Γ. Then, it proceeds on identifying constraints in the program by traversing it. It looks for pointer assignments and other statements that effect the points-to graph. For instance, an assignment expression like "x = y" indicates that x can point to anything that y can point to. Constraints gathered can handle copies, loads, and stores and address taking instructions.

The graph is updating on alloca, malloc, bitcast, load, store, phi, and get element pointer instructions of LLVM. In the following sections, we refer in detail to each of these instructions.

4.2.1 The Novel LLVM-based Type System

Following a type-based pointer analysis technique, we need to depend on a type system. For simplicity issues, we map complex LLVM types to equivalent type definitions in a new simple and small type system. This new representation of the LLVM type system is named 'annotated type system'. It is equivalent to LLVM's and conforms LLVM's coding standards and typical architecture.

Table 4.1 illustrates all the original LLVM types. LLVM's original type system consists of *Primitive* and *Derived* types. The *Primitive* types include integer, floating point, label and void types. The *Derived* types allow a programmer to represent types, such as arrays, structs and pointers and are build up by primitive types and other derived types.

Following, the annotated types include numbers, pointers, voids, functions and structs. Table 4.2 shows the mapping of LLVM types to our annotated types (TType). Note here that the original void * types of C language are translated to T_Pointers of T_Number types in our type system. This is due to the fact that void * types are casted as i8* or i32* pointer types in LLVM system. Therefore, a void *, on our type system is actually a T_Number pointer.

4.2.2 Typing LLVM Language

In Chapter 3, we describe how we collapse program statements into simple equations, here we map these statements to pure LLVM IR instructions and show how they contribute and effect on our points-to graph. By parsing the IR code, we parse and type every instruction explicitly. Thus, in our implementation, we say that we 'type' every LLVM IR instruction.

The core of the LLVM instruction set is its *commands*, which include the usual suite of

Class	Type	Syntax	Description
	Integer	iN where N is a literal integer	An integer type with bit width N.
	Floating Point	float, double, x86_fp80, fp128, ppc_fp128	32 bit floating point number, LLVM supports other floating point formats
Primitive Types	Label	label	The label type represents code labels.
	Void	void	The void type does not represent any value and has no size.
	Array	[10 x i32]	An array with ten 32 bit integers.
	Function	i32 (i32) *	A function that takes a 32 bit integer as an argument and returns a 32 bit integer
	Pointer	i32*	A pointer to a 32 bit integer
Derived Types	Structure	{i32, i32 (i32)*}	A structure with a 32 bit integer in the first field and a function in the second
	Vector	<3 x float>	A vector with three 32 bit floating point numbers.
	Opaque	opaque	Unknown content, used for example as a placeholder in a structure.

Table 4.1: LLVM Primitive and Derived Types

LLVM Type	Annotated Type
Floating Point	T_Number
Integer	
Array	T_Pointer
Pointer	
Void	T_Void
Metadata	
Label	
Function	T_Function
Struct	T_Struct

Table 4.2: Mapping of LLVM Types to Annotated Types

Instruction	Effect on Points-To Graph
alloca	A vertex is created for the allocated variable and for the unique stack label it corresponds to. An edge is created from the variable vertex to the stack vertex. For instance, $p = \&q$ creates a point-to relation between {p} and {q}.
load	A vertex is created for the variable receiving the value. Edges are created from this vertex to the targets of all of the other variable's targets.
store	For a call which stores the value of variable x into the spot where variable y points to, edges are created from all targets of y to all targets of x.
get element pointer	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of the RHS variable vertex.
cast	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of the RHS variable vertex.
phi	A vertex is created for the variable receiving the value. Edges are created from this vertex to all targets of each RHS variable vertex.
ret	Edges are created between the value assigned to this instruction and the output tau (the returning type) of the function.
select	For a call that either assigns its first argument to the result value or the second one, edges are created from both the first and the second value argument to the variable receiving the value.
call	A vertex is created for the variable receiving the value. Edges are created from the returning value of the function called to this result value. Also, edges are created from the vertices representing the actual arguments of the call to the vertices of the formal arguments of the called function.

Table 4.3: LLVM Instructions and how they affect the points-to graph

binary arithmetic operations, memory accessors, heap operations, stack allocation, conversion operations and calls. It consists of 52 instructions, classified in *8* different *groups* and has *32* different *op-codes*. Its small size guarantees its simplicity and efficient design, but, as our implementation proves, it consists an impediment for precise type information.

LLVM supports a Static Single Assignment form (SSA), exactly as our code language in Chapter 3, which means that each variable (a typed register) is assigned exactly once. This fact enables our algorithm to provide a 'strong updates' feature on the points-to analysis.

Table 4.3 summarizes how main LLVM instructions affect the construction of our points-to graph. Note, that some instructions like 'fence', 'cmpxchg', 'atomicrmw' are not supported in our implementation. In the following sections we describe deliberately the typing of every instruction on LLVM 3.6 version.

Terminator Instructions are used to mark the end of the basic blocks. All the terminator instructions included in the LLVM are:

ret	*br*	*switch*	*indirectbr*
invoke	*resume*	*unreachable*	

From the above instructions, we type only the 'ret' instruction, as the rest do not offer us any useful pointer information. The 'ret' instruction returns control flow from a function back to the caller. It has two syntax forms: ret void or ret <type> <value>. For example, it can be found with the following syntax: $ret\ \%struct.x * null$ in which struct.x is the type and null the value.

When our analysis traversal over the IR encounters a 'ret' instruction, we know we are inside a function scope. So, in order to not lose the flow of the program, we subtype the type of 'ret' instruction to the return type of the function.

Binary Operations Instructions

Binary operators are responsible for most of the computations in a program. They need two operands of the same type to perform an operation on them, and produce a result with a single value and he same type as the operands. The operators operate on values of integer type, floating point type and vectors. Binary operations have the following syntax:

```
<result_value> = opcode_name <type> <operand 1>, <operand 2>
```

The list of opcodes for binary instructions used in the LLVM is:

add	*fadd*	*fsub*
mul	*fmul*	*udiv*
sdiv	*fdiv*	*urem*
srem	*frem*	

The type <type> in the first seven instructions must be integer or vector of integer values and in the rest instructions floating point or vector of floating point values.

Bitwise Binary Operations Instructions execute bit manipulation. Both two arguments to the operation have to be of the same integer type. The result of the calculation has the same type as the operands and can be integer or vector of integer values. Bitwise Binary Operations have the following syntax:

```
<result_value> = opcode_name <type> <operand 1>, <operand 2>
```

All the opcodes for bitwise binary instructions used in the LLVM are:

shl	*lshr*	*ashr*
and	*or*	*xor*

All these instructions have exactly the same syntax, hence, we type them with the same way. As the result of the instruction is an integer or a vector of integers, there is no pointer information that we need to add to the graph, so there is no need for any subtyping or conflation. We only add to the Γ mapping the annotated result type of the instruction.

Vector Operations Instructions manipulate memory on vectors. In our implementation we did not cover vector types, as we focused only on C programming semantics. Indicatively, these instructions are:

extractelement *insertelement* *shufflevector*

Aggregate Operations Instructions handle aggregate values. LLVM contains the following list of instructions:

extractvalue *insertvalue*

extractvalue The 'extractvalue' instruction has the following syntax:

```
<result_value> = extractvalue <aggregate_type> <value>,
                 <index>{, <index>}*
```

The <result_value> holds the value of the aggregate (struct or array) at the position specified by the indices. All the indices are constant values. Since the value indexed isn't a

pointer, the first index is zero (0). Instruction implies that at least one index must be specified and that array indices must be in bounds. For example, the following extractvalue instruction has an aggregate type i64, i64* on value %tmp1 and intends to extract the first element of it (index is 0):

```
%tmp14 = extractvalue { i64, i64* } %tmp13, 0$
```

Our typing method retrieves the stored type of %tmp13 from the Γ environment (we know that it is already stored as it is on the right side of the equation) and extracts from it the field that the index points to. In this case, we return back the type of the first element of %tmp13. So, in our example it returns the type of the value i64. Then, this type is mapped to the variable of result_value and is stored again in the Γ.

insertvalue The 'insertvalue' instruction has the following syntax.

```
<result_value> = insertvalue <aggregate_type> <value>,
                 <type> <element>, <idx> {, <idx>}*
```

It inserts a value into a member field in a aggregate type at the position specified by the constant indices <idx>. The <result_value> has the same type with <value> type. We follow again here exactly the same method as in typing extractvalue instruction.

Memory Instructions model memory accesses in LLVM. The LLVM supports several commands for working with heap-allocated data structures and for reading, writing and allocating memory to transfer data between memory and the virtual registers. The following instructions are responsible for the memory management in LLVM:

alloca	*load*	*store*
fence	*cmpxchg*	*atomicrmw*
getelementptr		

(Note that the instructions fence, cmpxchg and atomicrmw are not supported in our typing system, as we are not interested in atomic operations.)

malloc and alloca instructions allocate array-structured regions of memory. They take a type parameter that rules layout and padding of the elements of the region, and an integral size that defines the number of elements. They both return a pointer to the freshly allocated region.

alloca The 'alloca' instruction is used to allocate memory on the stack frame of the current function. After the function's execution the allocated memory is automatically released. It has the following syntax.

```
<result_value> = alloca <allocated_type> [, <type> <num_elements>]
```

```
[, align <alignment>]
```

It allocates space equal to sizeof (<allocated_type>) * <num_elements>. If <num_elements> is omitted, it defaults to one. The type of the resulting value is a pointer to the <allocated_type> argument. The 'alloca' has also an optional alignment, which is a constant value.

'alloca' follows the subtyping rule of references. Thus, we create a node on the graph for the result_value type and an edge from this to the node of the annotated allocated_type.

load The 'load' instruction reads from memory and it has the following basic syntax:

```
<result_value> = load <type>* <ptr_value>
```

It takes a single operand that specifies the location of memory from which to load. To read the actual integer, we use the instruction load: $\%reg = load\ i32*\ \%regptr$ This loads the actual value into %reg.

Loads follow the subtyping rule of sub-dereferencing. Thus, we annotate the type, map it to the result_value and add this pair on the Γ environment.

store The 'store' instruction writes to memory and it doesn't produce any value. It has the following basic syntax:

```
store <type> <value>, <type>* <pointer_value>
```

The <value> operand indicates the value that will be stored and the <pointer_value> operand the memory location to store it. The type of the second operand must be a pointer to the type of the first operand. For example, for writing a value into %reg, we write:

```
store i32 42, i32* %regptr
```

Stores follow the subtyping rule of assignments. Hence, we subtype the type of value to the elementary type of pointer_value.

getelementprt The 'getelementptr'(GEP) instruction makes address computations only and, unlike load and store instructions, it does not access memory. It is used also for addressing fields inside arrays or structures. GEP follows the syntax:

```
<result_value> = getelementptr <type>* <ptr_value> {,
        <integer_type> <idx>}*
```

The operand <ptr_value> is always a value of a pointer type and is the base for the computations. Subsequent types can be arrays, vectors, and structs. Note that subsequent types being indexed into can *never* be pointers, since that would require loading the pointer before continuing the calculation. The rest of the arguments indicate which of the elements of the object are indexed. It returns a pointer to the specified element.

As it is generally admitted even by the engineers of LLVM platform, the GEP instruction is often seen as quite 'confusing' [2]. In general, GEP indexes into a structured data type by computing an offset pointer from another given pointer based on its type and a list of indices that describe a path into the datatype. For instance, on the following example we use GEP to typecast to a pointer:

```
@global_str = constant [13 x i8] c"Hello World!\00"
%t = getelementptr [13 x i8]* @global_str, i64 0, i64 0
```

The first argument to GEP is the pointer to the global string variable. The first index, i64 0, is required to step over the pointer to the global variable. Because the first argument to the GEP instruction must always be a value of type pointer, the first index steps through that pointer. A value of 0 means 0 elements offset from that pointer. The second index, i64 0, is used to select the 0th element of the string.

Our typing method retrieves the new annotated type returned by GEP ptr_value and assigns it as a the type of result_value.

Conversion Operations Instructions are used to convert a value of one type into a value of an other type. Conversion operations have the following syntax:

```
<result_value> = opcode_name <type1> <value> to <type2>
```

Conversion operations take a single operand and a type. They perform several bit conversions on the operand <value> and return a value with type <type2>. The operators that convert between integer and float types can also operate on entire vectors of values. The *opcode_name* can be one of the following instructions:

trunc..to	*zext..to*	*sext..to*
fptrunc..to	*fpext..to*	*fptoui..to*
fptosi..to	*uitofp..to*	*sitofp..to*
ptrtoint..to	*inttoptr..to*	*bitcast..to*

Our typing method handles all the opcodes on the same way as the syntax of the instruction remains the same. Thus, it types the right value of the instruction and assigns this new annotated type to the result_value. In order to type the right value, it subtypes type1 to type2, hence, a new edge on the graph is created between these two types.

Other Instructions contain all the "miscellaneous" instructions:

icmp	*fcmp*	*phi*
select	*call*	*va_arg*
landing_pad		

(Note that we do not implement landing_pad and va_arg instructions)

cmp (icmp, fcmp) instructions are used to compare two integer type or floating point type values. The result of the comparison of these two values is a one bit integer. Both follow the same syntax:

```
<result> = opcode <cond> <ty> <op1>, <op2>
```

Our typing method as it does not handle pointer types, it just annotates the type on the right side of the instruction and maps it to the result value. Hence, we do not lose flow of the program.

phi instruction, as already mentioned in 3.2.3.1, enforces the SSA representation by correspond to the ϕ-node. It has the following syntax:

```
<result> = phi <type> [ <val0>, <label0> ], ...
```

The first argument <type> defines the type of the incoming values. The second argument is a list of pairs of values and labels and each pair corresponds to one of the predecessor blocks of the current basic block. The phi instruction returns one of the incoming values depending on which basic block control flow came from. Phi instructions appear at the beginning of a basic block and there must be no non-phi instructions between a phi and the start of a block. Phi nodes may refer to themselves (meaning we have loops) and may select undefined (undef) values for certain in-edges.

Our typing method creates subtyping relations between the op1 or op2 types and the result type. We also handle extreme cases when we have uninitialized pointers on loops. This would mean that val0 and val1 may have not an annotated type store in the Gamma yet, as these variable are expected to be added on next instructions (after phi). For general cases, we subtype both op1 and op2 to the result type. Thus, we do not lose regardless the execution path it will be chosen. The points-to graph, then, at a phi instruction adds two edges from the argument types both to result type.

select instruction is used to select one value based on a condition without the need of branching.

```
<result> = select selty <cond>, <ty1> <val1>, <ty2> <val2>
```

The 'select' instruction needs an 'i1' value or a vector of 'i1' values expressing the condition, and two values of the same first class type. If val1/val2 are vectors and the condition is a scalar, then the entire vectors are selected and the not individual elements. If the condition evaluates to True, 'select' returns the first value argument, otherwise it returns the second value argument.

Similarly to phi instruction, we subtype from both ty1 and ty2 types to the result type. So, the graph is again updated with two new edges between argument types and result.

call instruction represents a simple function call. All functions are defined with their type.

```
<result> = [tail] call [cconv] [ret attrs] <ty> [<fnty>*]
    <fnptrval>(<function args>) [fn attrs]
```

Calls follow the typing rules of applications. Therefore, we subtype the actual arguments to the formal arguments of the called function and the return type of the function to the result type of call instruction. So, in order to type the calls, we retrieve from the Γ the called function (all function types are declared on the global scope) and create the subtyping relations between its arguments and return type with the ones of the callee.

In cases of indirect calls, there is a different syntax. For example:

```
%call = call i32 %fn(i32 %b.0)
```

In the case of library functions, as the arguments are unknown (typed as 'varargs') we insert an 'extra' argument with void* type when we first type their prototypes on the first pass traversal. So, in case of a library function call we subtype all the actual arguments to this void* type. To our knowledge, this adds more conflation on our method, but we remain correct and not lose any program flow.

4.3 Summary

The support of LLVM IR in our implementation is being shown that it gives many benefits. Above all, we take great advantage of the Static Single Assignment form for low-level code. Its feature of assigning different variables on every program statement, assures us that every variable has one reaching definition. By this, our method gets more simplified, as we do not need to re-annotate variable types on different occurrences of the same variable - as it is defined only once in a program. May this conclude to a larger environment of types, but in fact, gives an extra efficiency to our method. Furthermore, the abstract addresses of IR as well as the strongly connected function calls also simplify our work.

On the other side, we need much of effort in order to understand where the pointers are and this is usually done by a "look like" behavior. Hard-coding is used in the most of cases when typing the LLVM type system, proving that it may be light-heavy and efficient but it lacks in type-safety and support of type information, issues that prove to be strong impediments on our method. The main examples are the typing of get element pointer and extractvalue instructions. Moreover, we face issues with no type information, thus no collapsing inputs by types and also with infinite paths of pointers.

Chapter 5

Experimental Evaluation

In this section we demonstrate the evaluation results and a broad comparison with other existing analyses. We implemented and evaluated the presented analysis on 17 benchmarks of 3 different suites. We present a quantitative evaluation for our analysis by solving the generated points-to graph using bddbddb [50]. The primary goal of our empirical evaluation is to quantify the speed and space of our proposed method. Secondly, we want to verify whether analyses on intermediate representation (IR) code can provide more efficiency than the ones performed on source level code. Specifically, we measure the construction and solving time of graph and the total number of points-to sets.

5.1 Experimental Set-up

To evaluate the effectiveness of our approach and make the comparison with related work more direct, we set up experiments using specific benchmarks collected from disparate sources, including 5 POSIX threads applications, 4 SPEC 2000 benchmarks and 8 SPEC CPU 2006 benchmarks, that range from 1.2 KLOC to 230.5 KLOC. The characteristics of the reported benchmarks are shown in Table 5.1.

Five multi-threaded POSIX programs that successfully compiled on LLVM gathered from sourceforge.net to support the evaluation towards Locksmith [38] : *Aget* is a wget clone, a FTP client in which multiple threads download chunks of a file. *Ctrace* is a sample program of ctrace library used for tracing the execution of multi-threaded programs; *Pfscan* is a parallel multi-threaded file scanner; *Smtprc* is an open mail relay scanner that looks for potential configuration problems and *Knot* is a multi-threaded web server.

The SPEC 2000 benchmarks are relatively large and are elected to aid study the scalability and robustness of our algorithmic approach and comparison with Lhotak's analysis [29]. The suite is divided into integer and floating-point categories. Our evaluation was based on the integer benchmark programs (CINT 2000), all written in C. *164.gzip* is a com-

pression tool. *176.gcc* is a C programming language compiler. *181.mcf* is a combinatorial optimization. *256.bzip2* is a compression tool.

The SPEC CPU 2006 suite also contains benchmarks from real life applications that exercise corner cases of CPUs, memory systems and compilers - especially C++ compilers. This suite contains two categories of benchmarks: integer and floating point. For our evaluation we depended on a variety from both categories that compiled successfully in order to better support the comparison with Lhotak [29] and Li [55] analyses. The integer benchmarks we chose: *400.perlbench* is a cut-down version of Perl v5.8.7 scripting language. *401.bzip2* is a version of the compression tool bzip2. *429.mcf* is a derived of MCF, a program used for single-depot vehicle scheduling in public mass transportation. *433.milc* is simulator for physics and quantum chromodynamics. *445.gobmk* is the artificial intelligence game of Go. *464.h264ref* is a video compression tool. The floating-point benchmarks we chose: *470.lbm* is a simulator for computational fluid dynamics. *482.sphinx3* is the widely known speech recognition system, Sphinx-3.

The configuration of the computer used for the experimental work is: Processor: Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz; RAM: 8 GB; Cache Memory: L1 32KB, L2 4096KB; Operating System: Ubuntu 14.04.1 LTS x86-64; LLVM: version 3.4.

We use CIL v.1.3.6 merger to combine all the code for every benchmark into a single C file. We remove any comments and redundant declarations as well. We then compile all the C files with Clang 3.4 compiler by passing arguments for supporting POSIX threads (-pthread), for enabling mcount instrumentation (-pg), for generating source level debug information and debug line number tables (-g and -gline-tables-only) and for emitting LLVM IR code for .s and .o files (-S -emit-llvm). We then pass the generated bitcode (.ll) file to LLVM to run our pass. We do not use any built-in optimization to generate the code and enable the option for assigning names to anonymous instructions (-instnamer), as well.

To estimate the attributes of the generated graph, we depend on the back-end of the state-of-the-art bddbddb tool [49] to solve and measure the graph. In section 5.3 we describe exactly the way we use bddbddb tool for our evaluation.

5.2 Pointer Analysis Metrics

In this section, we present the metrics chosen for evaluating the performance and precision of the presented pointer analysis. We also refer to the tools and methods with which we retrieve these metrics.

5.2.1 Accuracy Metrics

For the evaluation of the analysis precision we basically measure the distribution of sizes of the points-to sets. In specific, we gather metrics for: the average points-to set size; the maximum points-to set size; the cardinality of the points-to set for each node on the graph; the total size of the points-to graph (a commonly good proxy for the overall precision of the analysis) and the total size of the Γ environment. We believe these five metrics give a reliable picture of the accuracy of the analysis.

We measure the sizes of the graph and Γ environment by adding a 'Statistics' pass on LLVM, so that easily count the entries to both structures. For the rest of the metrics we base on bddbddb (see following section for more details on its use).

5.2.2 Performance Metrics

Performance is demonstrated on our evaluation with four metrics: the construction time of points-to graph; the solving time of the points-to graph; the total time of the analysis and solving; the total number of warnings raised for the construction of the points-to graph. Even though, times are the ultimate performance metric for all analyses, they suffer from implementation and environment factors bias. In all of our experiments the times shown are means of ten runs.

The construction time of the graph is directly measured with the help of LLVM. The solving time time of the graph is again measured by bddbddb and the number of raised warnings is a 'Statistic' we add on our LLVM Pass in order to count the times we warn for loss of precision on the analysis mainly due to conflation of pointers.

5.3 Using bddbddb

We use bddbddb [50] as an external graph solver in order to measure the metrics, such as the *solving time of the graph*, the *cardinality of the points-to sets* as well as their *average* and *maximum* sizes. By providing to bddbddb the edges of the graph, we query on it how much time it takes to solve (i.e. traverse) the whole graph and also which are the points-to sets of every node of the graph.

bddbddb is a Datalog [47] interpreter, which can automatically translate program analyses expressed in Datalog into Binary Decision Diagrams (BDD). It also stands as a query engine by supporting relational algebra operations and database queries on program analyses by using BDDs. Thus, it can track value and type flow in an accurate way. For these reasons, given tool's efficiency, bddbddb is our primary choice for value (i.e. edge) tracking on top of our graph.

bddbddb, initially, generates the relations of the input program in order to depict them in BDDs and answer queries on the program. In our case, we have already these relations, so we only depend on the back-end part of bddbddb to solve these relations (edges of the graph).

Traditionally, bddbddb with the help of Joeq tool, generates a set of relation and domain files, written in Datalog depending on the input Java program. The generated files by Joeq represent the needed relations of the input program. The original Joeq generated files have a format of 3 different kind of clauses: *domains* (or else named as *facts*); *constraints* (indicate the restrictions that facts must specify) and *rules* (comprise the query). The *constraints* are either marked as input or output. The '*input*' relations come from an input file that has the name of the relation. The '*output*' relations are then written to other files with their name. If we use '*outputtuples*' as well, then the relations will be written to a file with one tuple per line. Similarly, there are input files depicted tuples relations, named as '*inputtuples*'.

Following this design we generate on our own the needed files: domains.pa, flowsTo.tuples and nodes.map. These files hold information of the number of nodes of the graph (domain), and the edges of the points-to graph (constraints). The flowsTo.tuples file represent the 'inputtuples' relations, and the nodes.map file lists all the nodes of the graph (elements of a domain). Every tuple of the flowsTo.tuples file is represented with numbers. Each number in the tuple displays an element in the domain and indicates what *line* in the nodes.map file to look at.

This way, bddbddb has all the required information to solve the constraints (edges) and answer the query (rule) for finding all the possible paths between the edges relations. So that, if there is an edge a $\rightarrow$ b and an edge b $\rightarrow$ c, bddbddb finds the path a $\rightarrow$ c. At the end of the bddbddb analysis, we show the total paths visited from the traversal of the graph, and bddbddb provides the running times it takes to answer every query given. As a result, we have the summary of the queries we need to answer in a quite efficient way.

5.4 Illustration of Our Results

In this section, we explain the meaning from all the figures with statistics gathered of all the experiments.

For a demonstration of the performance and precision statistics, consider Table 5.1, which summarizes the results of running our analysis on all of our benchmarks that vary in size, complexity and coding style. The first five benchmarks present a set of POSIX thread applications, the next four present SPEC 2000 benchmarks and the rest belong to the SPEC CPU 2006 benchmark suite. The first column of Table 5.1 gives the name of the benchmark, the following two show the number of lines on source code and the size of the bit-code file

Benchmark	SLOC	Bitcode File	Stats			Analysis Time (s)		
			$\|\Gamma\|$	$\|L\|$	$\|Wrn\|$	Con	Solve	Total
aget	1914	256KB	1864	1165	65	0.05	8.61	8.66
ctrace	2212	298KB	1530	857	63	0.03	8.57	8.61
knot	1985	496KB	2882	2117	2524	0.17	16.94	17.12
pfscan	1948	274KB	1804	895	29	0.05	9.45	9.5
smtprc	8624	1.6MB	12168	5216	132	0.46	10.7	11.19
164.gzip	8.6K	128KB	8008	2414	154	0.12	9.99	10.11
176.gcc	230.5K	5.2MB	n/a	n/a	n/a	-	-	-
181.mcf	2.5K	44KB	2917	1907	57	1.78	13.14	14.93
256.bzip2	4.7K	10KB	6318	1490	91	0.10	6.18	6.28
400.perlbench	169.9K	53MB	n/a	n/a	n/a	-	-	-
401.bzip2	8.3K	284KB	20415	6011	6005	0.4	13.66	14.06
429.mcf	2.7K	45KB	3178	2007	642	2.17	15.39	17.56
433.milc	15.0K	3.8MB	28882	14502	14822	1.17	166.77	167.94
445.gobmk	197.2K	45MB	206472	99483	49391	14.86	536.59	551.46
464.h264ref	51.6K	1.9MB	152941	47421	134796	10.36	260.27	270.63
470.lbm	1.2K	51KB	6031	1163	959	0.06	8.69	8.75
482.sphinx3	25.1K	5.5MB	38567	23489	38579	4.99	933.64	938.63

Table 5.1: Performance and precision metrics for all of our benchmarks over the pointer analysis.

(.ll) produced by LLVM, respectively. The next three columns show measurements regarding analysis precision. The fourth column displays the number of variables ($|\Gamma|$) stored in Gamma (Γ), the fifth column the number of labels ($|L|$) on the points-to graph and the sixth column the number of warnings ($|Wrn|$) that are raised for the construction of this graph. The next three columns refer to the times reporting the performance of the analysis. The seventh column presents the time needed for the construction (Con) of the graph. The eight column displays the time needed by bddbddb [50] to solve the generated graph and answer the query *'to which label this label points to?'* for *every* label of the graph. The benchmarks marked with '-' did not succeed in running the analysis after 2 hours.

Figures 5.1 to 5.12 display for every benchmark the cardinality of every pointer on the program. On X axis we display the size of points-to set that a label points to. On Y axis we display the number of labels on the graph. This way, we display the amount of labels that have the same points-to set size, giving us a good proxy of the precision of the analysis. The smaller the points-to set, the better the analysis.

Figures 5.13 to 5.15 help on the discussion of the experiments regarding how much related are the graph sizes, running times of analysis and size of Gamma with the size of the input program. On these 3 figures, we sort the benchmarks on X axis in increasing size of lines of code and associate with the metrics referred on Y axis.

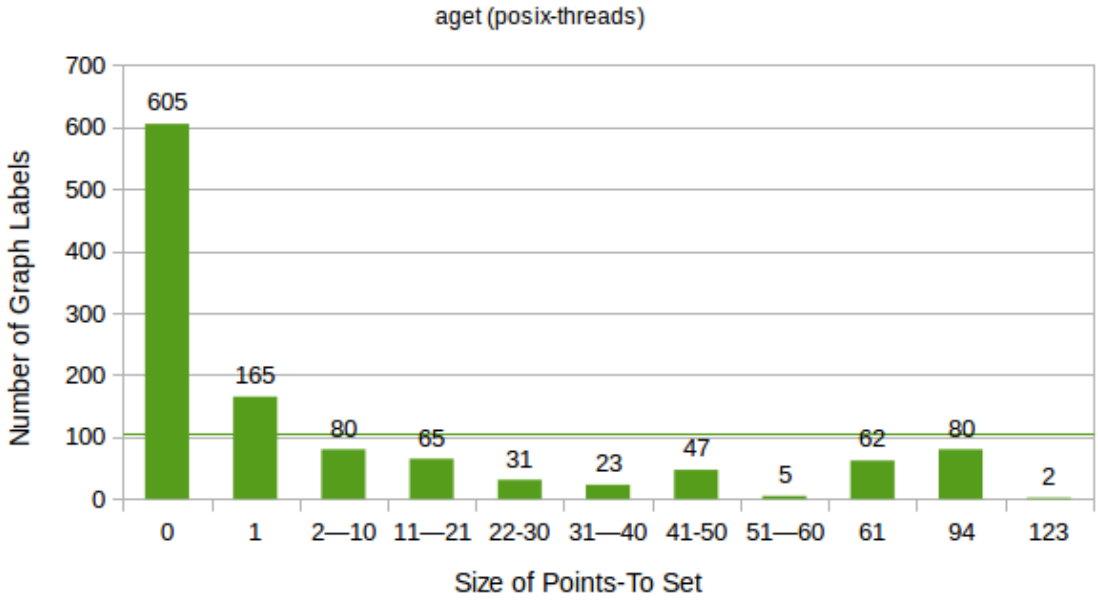

Figure 5.1: aget

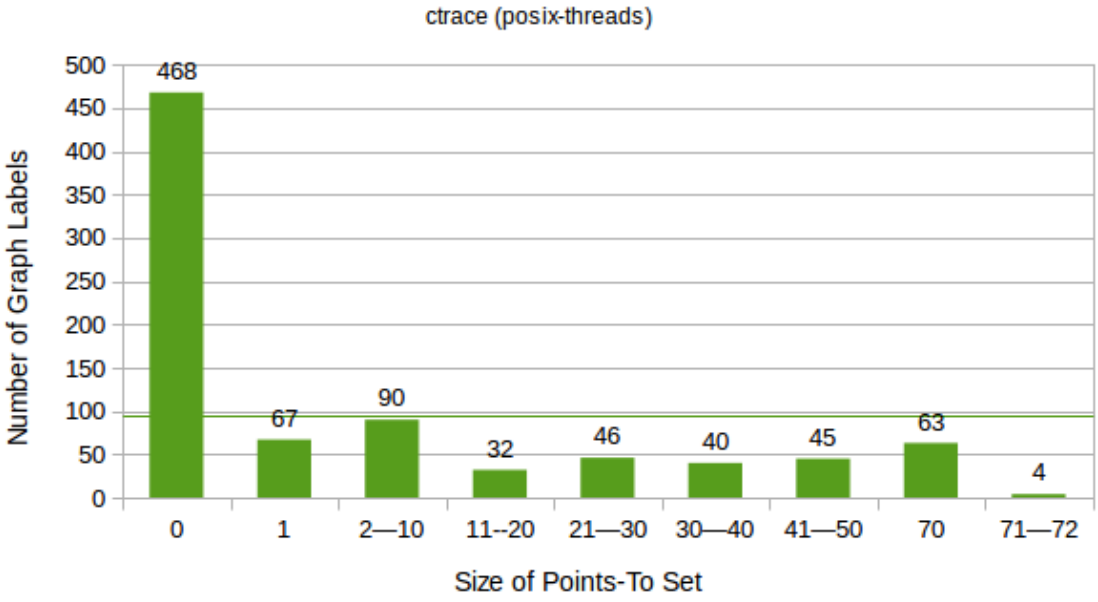

Figure 5.2: ctrace

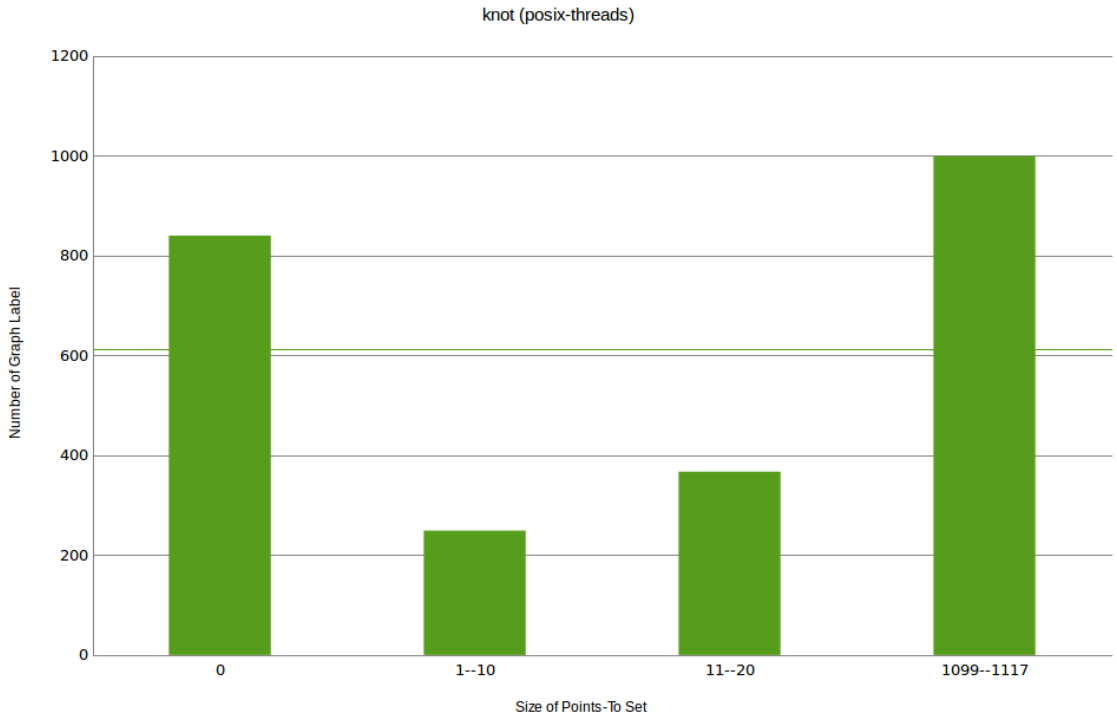

Figure 5.3: knot

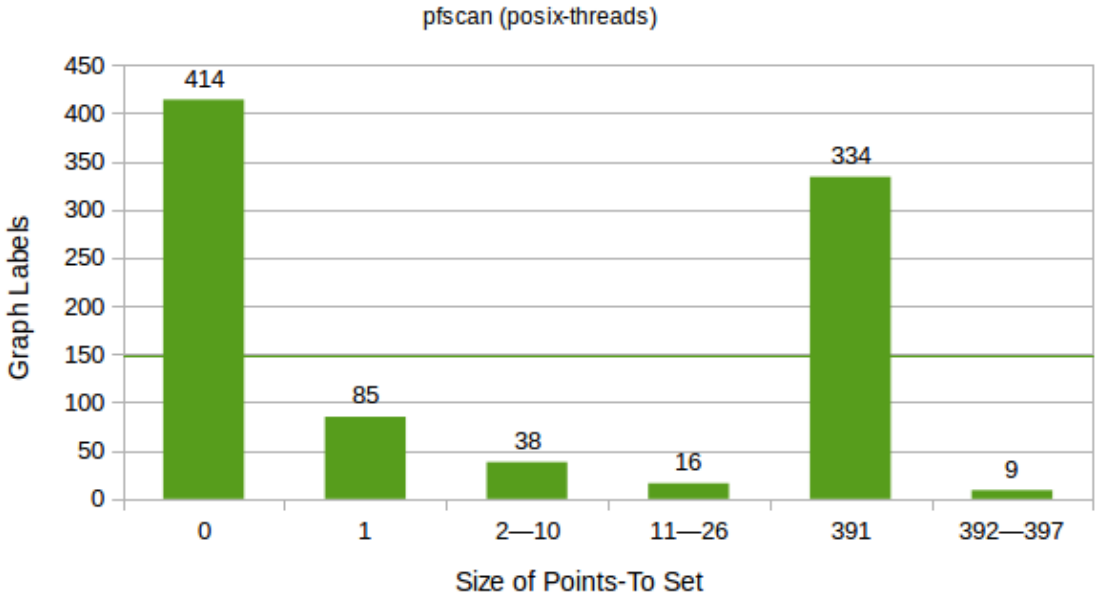

Figure 5.4: pfsc

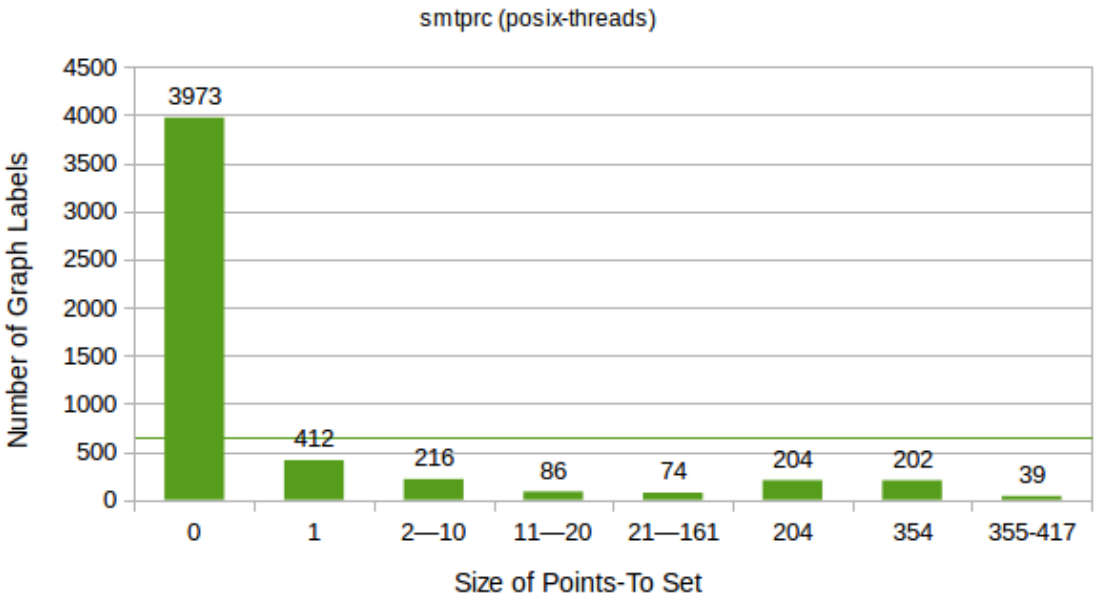

Figure 5.5: smtpr

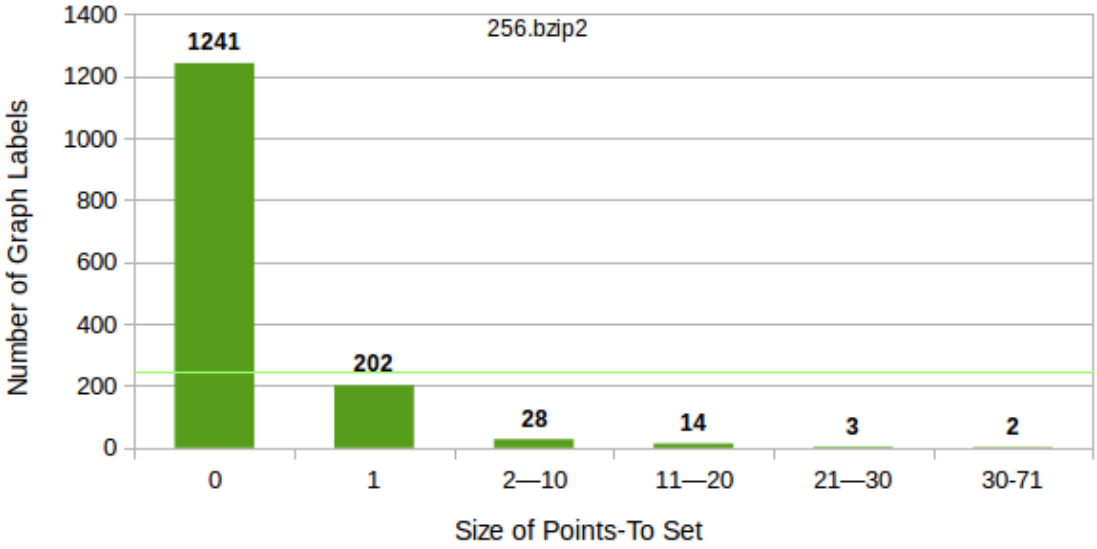

Figure 5.6: 256.bzip2

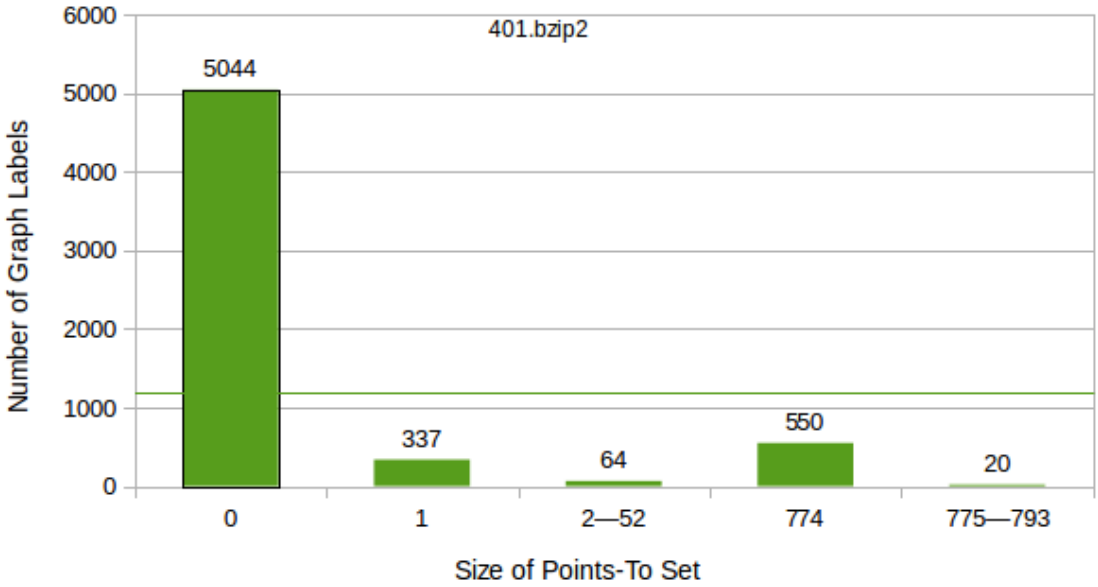

Figure 5.7: 401.bzip2

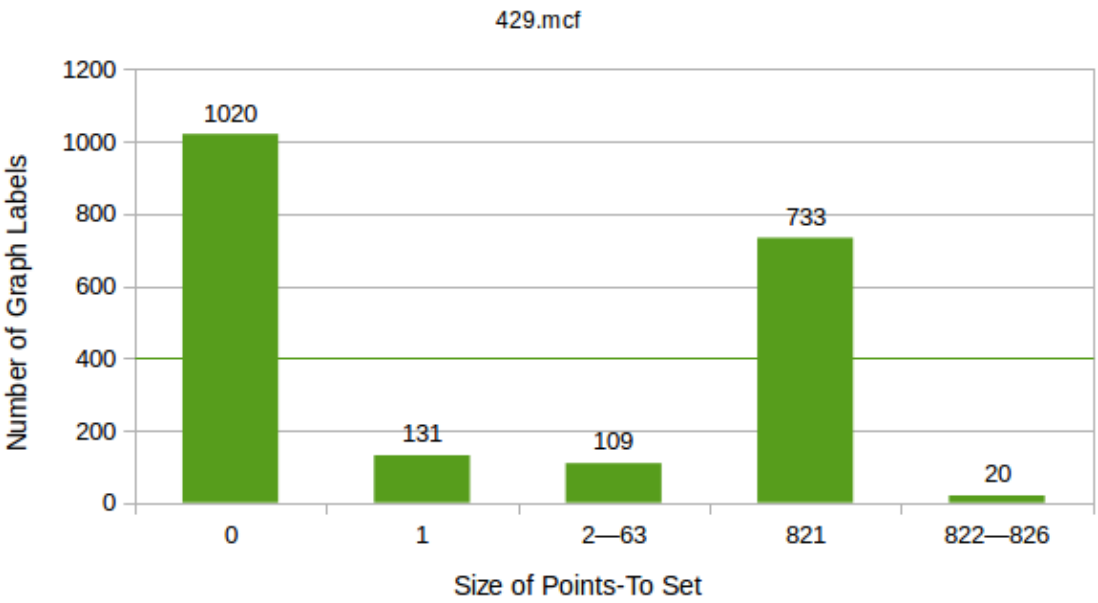

Figure 5.8: 429.mcf

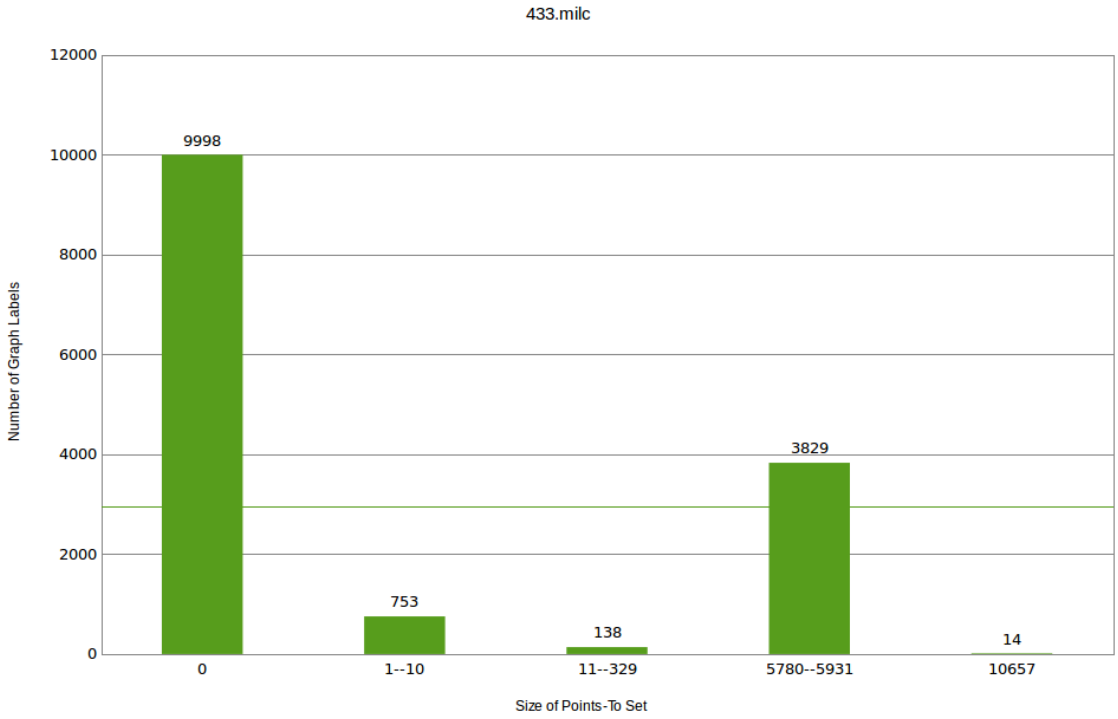

Figure 5.9: 433.milc

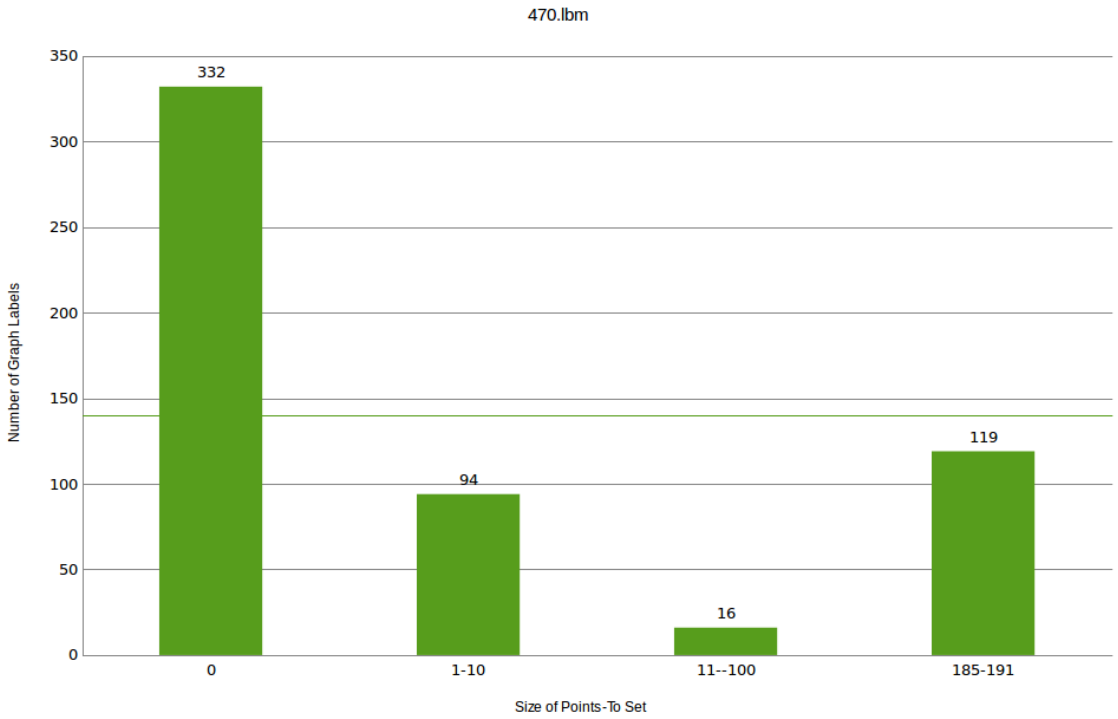

Figure 5.10: 470.lbm

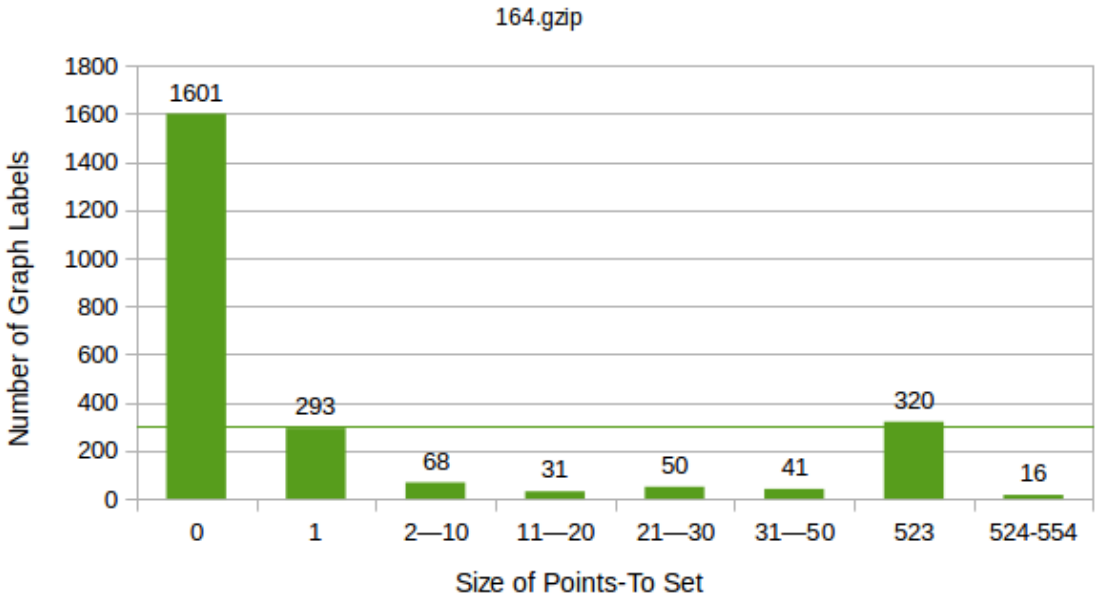

Figure 5.11: gzip2

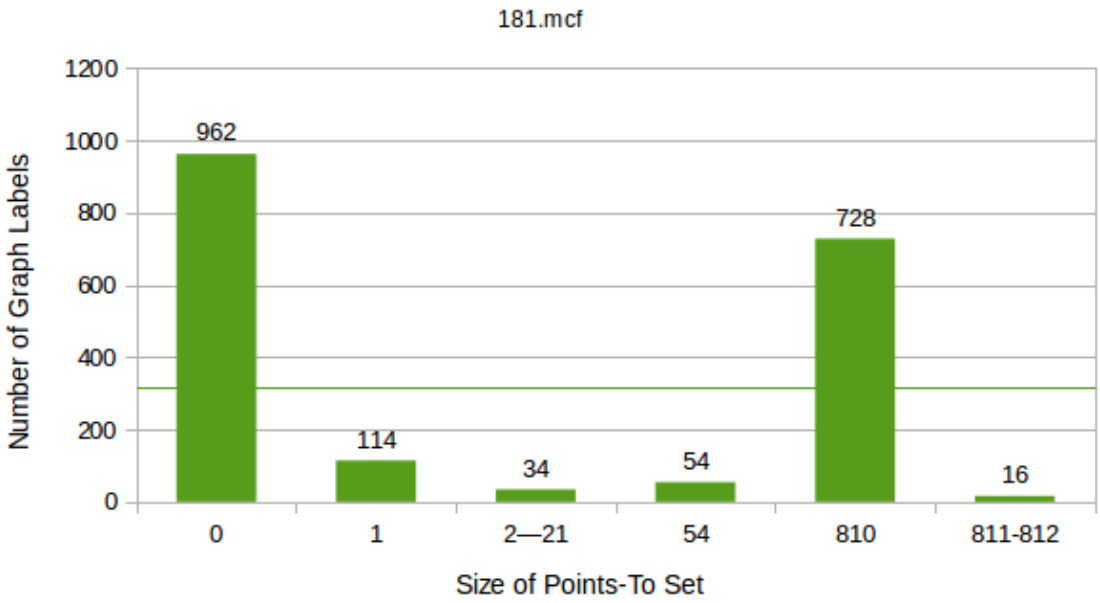

Figure 5.12: mcf2

5.5 Discussion

This section contributes to the discussion of the experiments results run above the presented pointer analysis method.

We notice that our approach method on pointer analysis over LLVM IR can efficiently and precisely work for medium-to-large scale programs that can even exceed the 150K lines of code. The running times of the analysis range from 0.03 seconds to 14.86 seconds.

Figure 5.13 gathers all the running times of analysis performed on every benchmark. The benchmarks on X axis are sorted by increasing number of lines of source code. We observe that may larger in size benchmarks have greater running times than the smaller ones, but this does not prove that lines of code (LOC) is a factor for the performance. Two of the large size benchmarks (*176.gcc, 400.perlbench*) do not complete the analysis within 2 hours and fail to construct a points-to graph, preventing further observations. As we observe, other benchmarks (445.gobmk) with similar amount of lines (around 200K), successfully complete the analysis and generate a points-to graph in reasonable time (nearly 14 seconds). For example, the smtprc benchmark from the POSIX-threads suite, has larger number of LOC than the benchmark 429.mcf. Though, the latter regardless its smaller size it needs more time to solve the constraints of the program and construct the graph. In addition to this, also 181.mcf, though it is smaller than 401.bzip2, has larger construction times. Therefore, the number of lines of code is not regarded as a performance factor. This behavior is a characteristic of the benchmarks themselves and their coding styles, rather than purely their size. High conflation happening due to recursive data structures (176.gcc) or several indirect calls (400.perlbench) is a greater indication indeed.

As shown in Table 5.1, the times needed by bddbddb to solve the generated points-to graph and find all the points-to sets for every label of the graph, are emphatically high, ranging from 6.18 seconds to 933.64 seconds. We presume that is due to the fact that the query is for all the graph and not a specific subset of it (as other algorithms focus on).

Similarly, Figure 5.14 gathers all the benchmarks sorted by increasing LOC along with the sizes of the generated points-to graphs. For each benchmark, we witness the correlation between the size of graph and the size of the program. As observed, although it is not a guarantee, greater in size programs produce larger points-to graphs. The only exception is the benchmark 401.bzip2 that generates a larger graph than other benchmarks with greater number of LOC. Thus, we come to the conclusion that more complex code odes to larger graphs.

Moreover, statistics on Table 5.1 prove the absence of any correlation between the graph sizes and the running times of the analysis. For instance, the graph produced for *181.mcf*

benchmark has 1907 labels, whereas the one for *433.milc* has 14502 labels. Even though, the latter has a graph 7 times the size of the former, it runs on 1.17 seconds in contrast to the former which runs on 1.78 seconds. Another proof is that *164.gzip* need 0.12 seconds to construct a graph of 2414 labels depending on a symbol table of 8008 pointers, whereas *181.mcf* needs 1.78 seconds to construct a graph of 1907 labels for a symbol table with size 2917. In other words, the size of the generated graph is not a clue of the time needed for its construction. Benchmarks with small graphs but many circular linked data structures and high conflation need more time for delivering the pointer analysis in contrast to simpler code programs.

Figure 5.15 refers to the relation between size of programs and size of the Γ environment. Similarly to the relation of graph sizes and programs, also here we observe that larger programs hold more information in the Γ. Specifically, the two diagrams of 5.14 and 5.15 seem identical. Therefore, we can conclude that programs with large Γ are expected to generate large graphs as well. Though, even here, we cannot safely declare that the size of the program is a factor for how large the produced Γ will be. Therefore, the complexity of code rather than its size seems to be of greater importance. Again here, the benchmark 401.bzip2 seems as the purest proof about this.

A general observation for all the experiments, is the large number of variable stored on the Γ (symbol table) and as an outcome, the large number of labels on our graph. This is greatly due to the LLVM technique of separating top-level and address-taken variable. For every address-taken variable `a` in the original code, LLVM creates a top-level pointer `p` pointing only to `a`, and all writes to `a` are transformed into stores through `p`. Since the points-to set of `p` is a singleton, the analysis can perform strong updates on a as if it were a top-level variable whose address had not been taken (except, of course, at program points where a actually is modified through other pointers). In contrast, the original flow-insensitive analysis is forced to model all such accesses to address-taken variables imprecisely as weak updates.

Summarizing all the statistics, we can conclude that the running times of the pointer analysis do not depend either on the lines of source code (Figure 5.13) or the size of the graph (Figure 5.14) or the Γ (Figure 5.15). As a result, we can emphatically state that there is no basic correlation between these statistics and the performance times.

Instead, we expect that coding style strongly matters in our method. As we explain in the implementation, recursive functions and data structures annotate every of their labels and this provokes many extra entries to both Gamma environment and to the graph. Thus, we expect programs with high conflation to face with scalability issues. A precision metric to depict this in a better way compared to the ones already referred is the cardinality of the

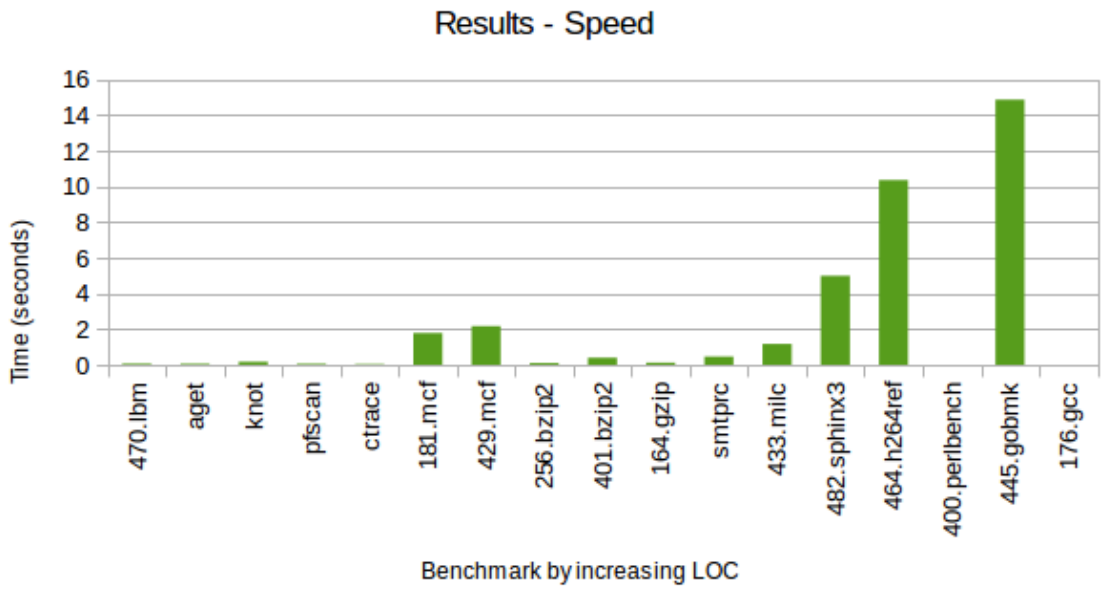

Figure 5.13: result-speed

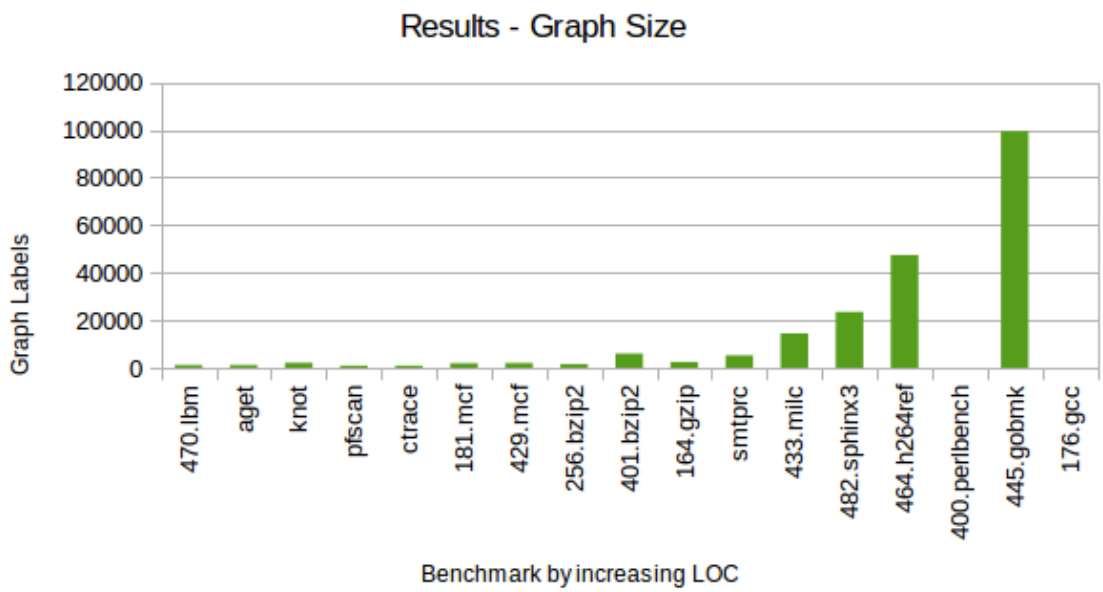

Figure 5.14: result-graph-size

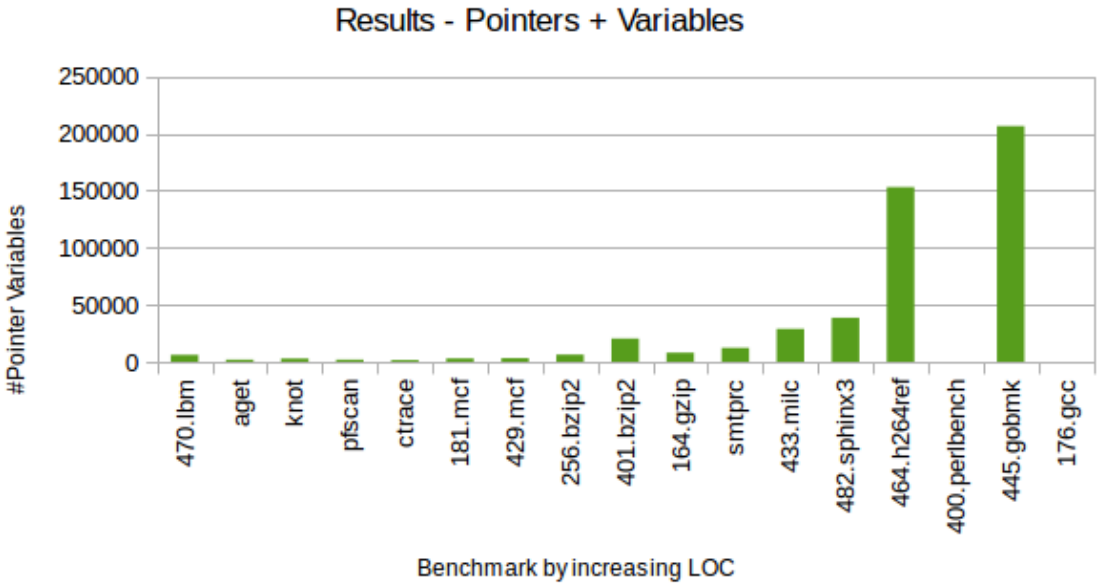

Figure 5.15: result-pointers

graph. Cardinality can prove how much complex is the graph, thus how much complexity the source code introduces.

All the Figures 5.1 to 5.12, show the cardinality of the nodes of the produced graphs. Observing many pointers with the same size of points-to set indicates the occurrence of high conflation on the code, potentially due to recursive data structures, function calls and pointer dereferences. Figures prove that we hold a quite large size of pointers that have no relation with any other pointer (observe the first columns on figures which show the points-to sets with zero elements). For example, on Figure 5.6, we observe that 1241 nodes have a zero points-to set, whereas 202 point only to one node. High conflation is also observed in benchmarks 176.gcc (many recursive data structures), 400.perlbench (several indirect calls).

Therefore, we can come to the conclusion that an optimization on our algorithm for erasing the pointers that are not address-take (have zero points-to set), would make our algorithm provide an even smaller symbol table and graph, making it even more efficient in terms of memory and performance.

5.6 Comparison With Other Analyses

In this section, we compare our method with other already implemented analysis.

Our conclusions from the evaluation, unfortunately, have to be restricted and be based on on the quality attributes of the analyses (see Chapter 2). This happens due to the fact that it is too hard to compare accurately the precision of different points-to analysis approaches.

There are many differences that stand as tough impediments on the comparisons, such as different abstract representations, benchmarks or metrics of precision and performance. Moreover, usually analyses often support different programming languages, as some are based on source code, while others on intermediate code. Different compilation or linkage of the same benchmarks may lead to different results, as well. Another drawback is that researchers use various evaluation methods and benchmarks, complicating even more the comparison, which often makes it impossible to reliably tell which approach describes the more accurate or efficient points-to analysis. In addition to, not all researchers make their implementations publicly available.

Though, for the best illustration of our results we also provide quantitative comparison with other already implemented analyses with respect to accuracy and efficiency. Note though that in most cases even the same metrics are measured with alternative techniques giving different meanings and making the overall comparison invalid.

On the following sections, we present the observations of the evaluation with analyses that either hold the same approach with ours or are also built on top of the LLVM platform. Specifically, we compared with: the flow-sensitive analysis of Locksmith [38], the LLVM-based implementations of Lhotak [29], Li et al. [30] and Yu [55].

5.6.1 Comparison to Locksmith's analysis

To our knowledge, Locksmith [38] provides the closest points-to analysis approach to our implementation, thus it consists our main comparable technique. Though, there are many differences that make conclusions of comparison not so valid or accurate.

Locksmith is a data race detection tool for C programs, with multiple performed analyses. Initially, Locksmith executes an *inclusion-based, context-sensitive, field-sensitive points-to analysis* based on C source code, to identify the locks and locations in the program. It enforces field-sensitivity by conflating all the elements of recursive data structures, and applies optimizations like lazy field propagation to treat them separately from each other. Locksmith (default phase) does not conflate under void*. Thus, Locksmith is based on C source code and provides context-sensitivity, in contrast to our method which works directly on IR code and is insensitive to context. Moreover, Locksmith forces further optimizations on its technique that help to minimize the use of labels on data structures.

Benchmark	LOC C	Bitcode File	Our Approach						Locksmith FS				
			$\|\Gamma\|$	$\|L\|$	$\|Wrn\|$	Con Tm (s)	Solve Tm (s)	Total Tm (s)	$\|\Gamma\|$	$\|L\|$	$\|Wrn\|$	CGen Tm (s)	Total Tm(s)
aget	1914	265K	1864	1165	65	0.05	8.61	8.66	1411	5634	62	0.55	0.85
ctrace	2212	298K	1530	857	63	0.03	8.57	8.61	1089	4351	10	0.4	0.59
knot	1985	496K	2882	2117	2524	0.17	16.94	17.12	1238	4752	12	0.55	0.78
pfscan	1948	274K	1804	895	29	0.05	9.45	9.5	987	4143	6	0.36	0.46
smtprc	8624	1.6M	12168	5216	132	0.46	10.72	11.19	4275	14815	46	3.09	5.37

Table 5.2: POSIX thread benchmarks: characteristics, analysis running times of our approach, Locksmith's default analysis.

Table 5.2 gives the overall comparison with Locksmith's analysis. The first column gives the name of the benchmarks, the following two show the number of lines of source code and the size of the bit-code (.ll) file generated by the LLVM (version 3.6) front-end, respectively. The next six columns show measurements of our analysis, both statistics and analysis times, and the next six columns are following with the measurements of Locksmith's analysis.

Note that the times provided on Locksmith's paper are not only for its points-to analysis, but for the running of the whole package of analyses it performs. Thus, the numeric comparison is not so accurate. Similarly, note the different meanings of every of the metrics:

Our statistics include the number of variables of stored on the Γ, the number of labels ($|L|$) in the points-to graph and the number of warnings ($|Wrn|$) raised for constructing this graph. Constructing time (Con Tm) corresponds to the time needed for the construction of the graph, based on LLVM platform. Solving time corresponds to the time bddbddb needs to solve the generated graph and answer the query *'to which label this label shows to?'* for every label of the graph.

For Locksmith's field sensitive (FS) analysis, the statistics include: the total number of pointers and variables ($|\mathsf{G}|$) of the sharing analysis, the number of generated labels ($|L|$) (locations and locks) and the number of warnings ($|Wrn|$) that are reported for the shared locations found that are not protected by any lock (warnings for data races). The analysis times correspond to the time for constraint generation (CGen Tm) (including annotating types with fresh labels, i.e., abstract locations and locks) and the total analysis time, presented on the last column.

As the data collected has contrasting meaning, the comparison can not be so accurate between the two different analyses and no safe conclusion can be given. Though in a try to demonstrate a closer comparison, we gather on Table 5.3 only the size of graphs and the running times of the two analyses.

A closer look at the results indicates that the construction time of our points-to graph is comparable to the corresponding one for the constraint generation of Locksmith, and confirm our assumes not only in theory but in practice as well. In all the experiments, our constructing graph times are much less than those of Locksmith.

Notably, the size of our graph is significantly smaller than the one generated from Locksmith. This is the most precise evidence to point out that analyses on the produced LLVM IR code have less simplicity than on the source code.

The findings of the research (Table 5.1) show that the solving of the graph from the bddbddb tool is more complex than previously assumed and takes more time than the solving time of Locksmith. It is important to note, however, that bddbddb solves all the graph and finds the points-to set of *every* label of the graph. Locksmith, on the contrary, focuses only

Benchmark	Our Approach		Locksmith	
	$\|L\|$	Con Tm (s)	$\|L\|$	CGen Tm (s)
aget	1165	0.05	5634	0.55
ctrace	857	0.03	4351	0.4
knot	2117	0.17	4752	0.55
pfscan	895	0.05	4143	0.36
smtprc	5216	0.46	14815	3.09

Table 5.3: Comparison with Locksmith's default analysis. Compare with number of labels of both graphs and construction running times of analyses.

on the points-to set that exist on the path of a possible data race.

At the same time, following our expectations, the number of variables (both stack and heap) we store in Γ environment is greater than the total number of pointers of the sharing analysis of Locksmith in all the experiments. As observed previously, the fact that we do not erase or concatenate any pointer but store even the ones which do not point to any other (observe the large points-to sets with 0 size in figures 5.1- 5.5) makes the symbol table to hold information for myriad pointers that may be not so interesting for the analysis results. In literature there are already many ways to deal with such issues (See Chapter 6), proving there is much space for further optimizations on our approach.

5.6.2 Comparison to Lhotak's analysis

From the most well-know pointer analyses built upon the LLVM platform and closer to our work is the one introduced by Lhotak and Chung in [29].

They prove the benefit of strong updates of the points-to information on the flow-sensitive algorithms by presenting a points-to analysis that combines the advantages of flow-insensitive and flow-sensitive analyses. It enables strong updates (a good proof of precision) as a proper flow-sensitive analysis does, but in times of a flow-insensitive analysis.

Lhotak's technique is also an adaptation of the Andersen-style analysis implementation from the LLVM compiler infrastructure. In pointer analysis terms, it is a flow-insensitive, context-insensitive, field-sensitive points-to algorithm. Their approach adds flow sensitivity only for singleton points-to sets.

Like our implementation, Lhotak's analysis uses the LLVM IR and analyzes C programs. In contrary to ours, Lhotak enforces many optimizations, such as an off-line optimization on the graph. This optimization includes off-line variable substitution algorithms intended to compute 'pointer and location equivalences' (i.e. pointers that have the same points-to sets or variables that appear together in points-to sets). Thus, this optimization manages to exclude the same set of objects from strong updates. It also includes an off-line cycle

Benchmark	SLOC	Bitcode File	#Variables		Graphs	
			Our $\|\Gamma\|$	LH $\|TOP - L\|$	Our $\|PTG\|$	LH $\|SPG\|$
164.gzip	8.6K	1.1MB	8008	1740	2414	2818
176.gcc	230.5K	50.3MB	n/a	104663	n/a	117121
181.mcf	2.5K	381.8KB	2917	1092	1907	821
256.bzip2	4.7K	828.8KB	6318	951	1490	1590
400.perlbench	169.9K	57.6M	n/a	89661	n/a	93792
401.bzip2	8.3K	2.8MB	20415	3265	6011	3243
429.mcf	2.7K	468.7KB	3178	1096	2007	823
433.milc	15.0K	3.9M	28882	5269	14502	5954
445.gobmk	197.2K	46.8M	206472	54022	99483	41769
464.h264ref	51.6K	17.1MB	152941	26884	47421	22951
470.lbm	1.2K	458KB	6031	337	1163	322
482.sphinx3	25.1K	5.7M	38567	12410	23489	10332

Table 5.4: SPEC 2000 and SPEC CPU 2006 benchmarks: characteristics, stats of ours and Lhotak's analyses.

detection algorithm that eliminates cycles. The authors claim that the precise points-to sets are the main cause of efficiency benefits and the high precision.

Unfortunately, also here, we cannot provide a solid contrast regarding analysis times. We encounter difficulties of running Lhotak's open-source analysis due to the outdated version of LLVM (2.6) it is based on. Lhotak, in his paper, illustrate a comparison with LLVM's built-in analyses, but these have been removed from later versions due to their low maintainability and the several bugs, as LLVM community claims. Therefore, we are not able to come to an accurate conclusion regarding performance of our analysis in contrast to the referred approaches. Instead, we demonstrate a comparison regarding precision.

Table 5.4 summarizes the statistics gathered by running our analysis on SPEC 2000 and SPEC CPU 2006 benchmarks. Lhotak's statistics are gathered from his paper. Though, underline that due to the important meaning differences of the collected stats, as happened also with the comparison with Locksmith, it is impossible to directly compare the analyses output in an accurate way.

The first three columns of Table 5.4 shows the name of benchmark, the number of lines on source code, and the size of the bitcode file generated by LLVM front-end. The next two sections gather statistics for both our analysis and Lhotak's. First section illustrates the number of variables stored and the second one the size of graphs generated by the two analyses. For our analysis, we demonstrate the number of entries on Γ (Our $|\Gamma|$) including both heap and stack variables and the size of the points-to graph (number of nodes) (Our $|PTG|$). For Lhotak's implementation, we show only the number of top-level variables (LH $|TOP - L|$) and the size of its sparse graph (LH $|SPG|$).

Despite the lack of optimizations on our analysis (compared to Lhotak's several off-line

Benchmark	# Pointers		Points-To Set			
			Our		Li et al.	
	Our Γ	Li et al.**	Avg**	Max**	Avg***	Max***
400.perlbench	n/a	920272	n/a	n/a	2.10	83
401.bzip2	20415	7522	73.4	793	1.02	2
429.mcf	3178	2658	309.8	826	1.42	8
445.gobmk	206472	261168	40.6	191	1.02	5

Table 5.5: Statistics gathered from ours and Li et al. [30] approaches.

optimizations), results show that in many experiments our analysis hold similar amount of pointers and labels. In two benchmarks, our analysis generates a smaller graph than Lhotak's. This proves the space for optimizations and further improvements on our algorithm. Following the example of Lhotak's analysis that applies further optimizations to exclude much pointer information from the sparse graph. In contrast to the flow-insensitive analysis of LLVM (for numbers refer to Lhotak's paper), we outperform in 4 out of 12 benchmarks. Therefore, our analysis performs better than the flow-sensitive analysis of LLVM in all the benchmarks compared.

5.6.3 Comparison to Li's analysis

Another LLVM-based pointer analysis is introduced by Li et al. [30] which contributes to the improvement of precision by tracking value flows in pointer analysis. This algorithm is a context-sensitive (to both pointer variables and heap objects), flow-sensitive and field-sensitive approach and scales well to large applications. Qualitatively comparing it to our method, the two analyses are common only on their field-sensitivity approach (apart from the LLVM dependency).

Table 5.5 summarizes the comparison between our analysis and Li's (without heap cloning) regarding the total size of pointers and the size of points-to sets. It is deemed that these two measures can show the precision and accuracy of a pointer analysis. However regarding the demonstrated data, we have to outline the distinct meaning among the metrics and the different way of their calculation:

For the number of pointers (#Pointers), on our analysis we demonstrate the number of entries on Γ, meaning the total number of variables both on the stack and on the heap. Retrieving the stats from Li's paper, their measurements show the total number of pointer variables, both the auxiliary that needed on the analysis and the original program pointer variables.

For the calculation of the size of points-to sets, for our analysis we demonstrate the average and maximum values from the *whole* graph. These stats (average and maximum

sizes) are based on the results of bddbddb, that estimate the size of points-to sets for *every* node of the graph. Though, Li et al. compute these values by 'compactly representing pointers with respect to function inputs'. That means that they regard *only* the flow inside the functions without computing the external flow from the call-sites. Thus, their average and maximum sizes are very small and can not support an accurate comparison with our metrics.

Taking into consideration both implementation and evaluation differences, we outperform Li's analysis on one benchmark (445.gobmk) and we have close total number of pointers on another one (429.mcf). This is another proof of the space of great optimizations on our algorithm, regarding pointers and variables that are never been used or pointers with the same points-to sets.

5.6.4 Comparison to 'Level by Level' analysis

A LLVM-based pointer analysis is introduced by Yu et al. in [55]. Yu's algorithm is a summary-based, context-, flow- sensitive, field-insensitive (FSCS) pointer analysis that scales well to large applications.

Yu displays experiments above various different benchmarks. Having only one in common with ours (445.gobmk) and given the fact that we are speaking about a FSCS pointer analysis, run in a different system to ours, we cannot provide a solid comparison. Indicatively, according to the numbers in the paper, Yu's analysis for 445.gobmk runs on 21.37 seconds in contrast to ours that runs on 14.86 seconds. Without being able to quantitatively compare our method with Yu's, we remain on a high-level qualitative comparison.

Yu's main contribution is on computing function summaries which are instantiated at their call-sites to create respective summaries for their callers. Specifically, it separates the program variables into different points-to levels by using an insensitive analysis. Then, contrary to our technique, it applies a flow-, context- sensitive analysis by processing variables 'level by level', from the highest to the lowest. The variables in the lower level use the analyzed results of higher level variables to compute the context- and flow- sensitive pointer information. Although, it does not distinguish the struct field accesses, as our method does.

However, this function summaries technique, may lose pointer information in the whole summary. For instance, it is quite difficult to answer the query "are these two pointers aliases on a specific call path?" in a precise way, as the points-to sets have no extra context information. For this reason, our method provides more precision on the results.

Chapter 6

Future Work on Pointer Analysis

In this chapter we make suggestions for future work, covering both extensions to the current work as well as further optimizations on the existed algorithmic approach. While practical experiments in Chapter5 expose the feasibility of the analysis, and the promising results, even though, there are still several areas for future work remain to be investigated.

6.1 Extensions to our Method

There are numerous and immediate improvements which could be made to the current method in order to make it deliver both better results and make it stand as a complete static analysis tool. We shortly elaborate on each now.

6.1.1 Client of analysis

As indicated from our evaluation, the best way to illustrate analysis results is the use of a client framework where points-to information is applied to all compiler analyses and optimizations. This way we would be able to measure directly the benefit from it (e.g., accuracy, speed-up). This would help for a better quantitative evaluation of the metrics results of the analysis, but would also enforce the proof its correctness. Apart from these, a support by a client would deliver a full static analysis tool and would be easier to be used by developers with no static analysis background.

6.1.2 Context-Sensitivity

A context-sensitivity would definitely boost our analysis results. We present a context-insensitive analysis which, as expected from theory (Chapter 2), lacks in precision in contrast to a context sensitive one. Our technique suffers from incorrect paths and it can falsely conclude that a call-site can return a value, as it merges information from all call-sites. Enriching our technique with a context-sensitive algorithm, would advance the existed method

and improve its precision.

In specific, context-sensitivity via context-free language (CFL) reachability [56] boosts the analysis efficiency. CFL is used to compute the flow of data through the programs by using labeled edges to distinguish different call sites. Thus, instead of conflating the functions to all their calls, as we do, we analyze different calls to the same function and reduce the problem of tracking flow through calls. This way, we gain more precision in our analysis and improve dramatically the current method.

6.1.3 Parallelize Analysis

Instead of running the analysis all together on a single processor, we can analyze and summarize the behavior of parts of the program in parallel and gain greater efficiency in speed. There has been already several attempts on parallelizing pointer analysis [32]. In particular, analyzing functions separately and compute summaries that hold the behavior of a function for any calling context enables a very easy parallelization [22, 32] that boosts greatly the scalability of the analysis.

6.1.4 Parallel Graph Algorithm

Parallelizing the construction of the points-to graph, could be essential for our method, as we construct and solve the points-to graph for the whole program together, and run it on a single processor.

Algorithms such as breadth-first search (BFS) [3, 19, 20], PageRank [19, 20] have been transformed also in parallelized versions and instrumented to divide heavy tasks. Although, we have to note that Andersen's analysis does not suffer so much for the construction of the graph, but for the series of modifications during the analysis.

6.1.5 Symbolic Range Analysis

An effective way to achieve more precision and to handle sufficiently the problem of pointer arithmetic in C would be to determine fields within arrays and structs by integrating alias analysis with symbolic range analysis of pointers as in [34].

6.2 Optimizations on our Method

Furthermore, we briefly outline some directions for future work explicitly on optimizations of our method that would help to limit the problems outlined from the experiments (size of the graph and number of pointer variables modeled).

6.2.1 Modeling types

Modeling struct types As observed in our experiments a large percentage of our speed and memory is devoted for the analysis of struct types. Our field-sensitive technique of fully annotating all field types of all C struct instances, although it significantly improves precision, it is obviously expensive and the main cause of the scalability issues we face. We waste memory and time by generating constraints and assigning abstract locations even to rarely or never used fields.

A possible optimization for this could be an implementation of a lazy annotation field sensitive algorithm [18]. By lazily annotating the fields of struct types, we do not create a label for every field of the struct automatically, but only when we encounter a field access, we add it to the corresponding struct type. Then, if there is any label flow between two structs, we conflate the labels of their fields.

This way, lazy annotation can ode to a deduction of label annotations and contribute significant savings. As an outcome, we will have a lower consumption of memory and time, without losing any precision.

Modeling pointer types Similarly to the lazy annotation of fields, we could lazily model all the pointer types, as well. In experiments results, we observe really large number of pointers with zero points-to set, meaning they are not re-used in the program, they not contribute to the analysis and only waste memory in our representation. By lazily annotating them, we gain a huge boost with savings from pointers that are never used.

6.2.2 On-line Cycle Detection

On-line cycle detection would hugely boost points-to graph efficiency, as it suffers from cycles among nodes specially in cases of conflating between structs and recursive pointers.

A helpful observation states that every variable that exists in a cycle in the points-to graph has an identical points-to set. This means that all these identical variables can be collapsed together into a single node and afford space in the graph without losing precision. This technique is named as on-line cycle detection [35] and breaks the dependence cycles amongst pointer variables on the fly.

Lazy Cycle Detection The technique of lazy cycle detection tries to detect cycles once they are created, by checking to see if the source and destination edges already have equal points-to sets. In case they do, it checks for a possible cycle by using depth-first search algorithm. If there is a cycle, all the involved nodes are collapsed into a single node.

Hybrid Cycle Detection An alternative hybrid technique, suggests the combination of lazy cycle detection and off-line one, by detecting cycles that are created on-line during the analysis, but without traversing the whole graph each time.

6.2.3 Off-line Optimizations

A profound technique in optimizing pointer analyses are the off-line optimizations. By off-line means it is performed before a standard points-to analysis.

Pointer Equivalence The large number of points-to sets shown in the experiments, indicate that the proposed technique would greatly benefit from removing variables and instructions in case they are found as equivalent. Two pointers are pointer-equivalent, if their points-to sets are equal.

A technique suggested by Smaragdakis et al. [41] intends to detect pointer equivalent variables. Before reaching the points-to level and evaluate the constraints, still at a pre-processing step, it computes constraints, finds all the pointer equivalent variables and then rewrites their constraints.

Variable substitution Our method could get more efficient with an off-line variable substitution transformation, such as [39], which reduces the input size of the program. This is achieved by replacing a set of variables in a program with a unique variable, so the whole number of pointers tracked is reduced. This would benefit the SSA format that IR follows, as it reduces the left value registers.

Chapter 7

Conclusions

We contribute a variation on Andersen-style (context-, flow- insensitive, field-sensitive) pointer analysis on top of LLVM IR code that provides comparable speed and quality characteristics as in high-level analysis. We cover the lack of the well-known Andersen algorithm from the platform of LLVM, and deliver a type-based, precise enough pointer analysis that can be used directly from LLVM optimizer and allow aggressive optimizations as redundant store elimination or load/store reordering. The results of the analysis are presented by a directed graph that depicts all the points-to relations in the entire program.

We benefit of performing an analysis on low-level code due to its language independence, simplicity, and light-weight design. On the IR highlights, we can include the treatment of strongly connected components of function calls, the abstract addresses, as well as the support of SSA-form without losing precision. Although, these benefits are clear, we cannot but underline the loss of accuracy that IR provides mainly due to its lack of rich type information. We deal with implementation problems such as the understanding where the pointers are "by look like" behavior, the infinite paths of pointers and the loss of information like the array boundaries on the stack.

While our goal has certainly been achieved, the lack of fundamental comparison techniques make our results slightly discouraging, as there is no strong proof of the accuracy and performance of the analysis. We evaluate the precision of the points-to analysis with respect to the cardinality of the graph nodes, the size of Gamma and graph in 17 realistic large C programs. The results reveal the large space for further optimizations, with focus on the size of the graph and the rarely used pointer variables.

Nevertheless, much has been achieved and, in particular, we feel that the analysis with some additional optimizations has an exciting future. We also consider that there remains work to be done on client analyses that intensively use the results of pointer analysis.

Finally, we thank *you* reader for your proved attention, as you are still reading these last lines, and hope you have found this work stimulating and enjoyable.

References

[1] bddbddb open-source code - bddbddb.sourceforge.net.

[2] The often misunderstood gep instruction http://llvm.org/releases/2.6/docs/getelementptr.html.

[3] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[4] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, New York, NY, USA, 1993. ACM.

[5] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[6] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 196–207, New York, NY, USA, 2003. ACM.

[8] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 46–55, New York, NY, USA, 1993. ACM.

[9] Silvian Calman and Jianwen Zhu. Increasing the scope and resolution of interprocedural static single assignment. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 154–170, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] Luca Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[11] Ramkrishna Chatterjee, Barbara G. Ryder, and William 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.

[12] Ben-Chung Cheng and Wen-Mei 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.

[13] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, May 2003.

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

[15] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander 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.

[16] Manuel Fähndrich, Jakob Rehof, and Manuvir 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.

[17] Cormac Flanagan. Effective static debugging via componential set-based analysis, 1997.

[18] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, November 2006.

[19] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM.

[20] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 851–862, Washington, DC, USA, 2013. IEEE Computer Society.

[21] Patrice Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, UK, 1997. Springer-Verlag.

[22] Brian Hackett and Alex 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.

[23] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Nevin Heintze. Aliasing analysis for a million lines of c. In *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, ASIA-PEPM '02, pages 47–49, New York, NY, USA, 2002. ACM.

[25] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[26] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[27] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[28] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[29] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.

[30] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. *SIGPLAN Not.*, 48(11):85–96, June 2013.

[31] Raymond Waiman Lo. *Static Analysis of Programs with Application to Malicious Core Detection.* PhD thesis, Davis, CA, USA, 1992. UMI Order No. GAX93-02610.

[32] Mario Méndez-Lojo, Augustine Mathew, and Keshav 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.

[33] Rupesh Nasre. Time- and space-efficient flow-sensitive points-to analysis. *ACM Trans. Archit. Code Optim.*, 10(4):39:1–39:27, December 2013.

[34] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Symbolic range analysis of pointers. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 171–181, New York, NY, USA, 2016. ACM.

[35] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, December 2004.

[36] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[37] Fernando Magno Quintao Pereira and Daniel 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.

[38] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41(6):320–331, June 2006.

[39] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. *SIGPLAN Not.*, 35(5):47–56, May 2000.

[40] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 10–16, New York, NY, USA, 1998. ACM.

[41] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. Set-based preprocessing for points-to analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages; Applications*, OOPSLA '13, pages 253–270, New York, NY, USA, 2013. ACM.

[42] Bjarne 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.

[43] Yulei Sui, Peng Di, and Jingling Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, pages 160–170, New York, NY, USA, 2016. ACM.

[44] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[45] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Softw. Eng.*, 40(2):107–122, February 2014.

[46] Paolo Tonella and Alessandra Potrich. Reverse engineering of the uml class diagram from c++ code in presence of weakly typed containers. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 376–, Washington, DC, USA, 2001. IEEE Computer Society.

[47] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[48] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[49] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[50] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, June 2004.

[51] Robert P. Wilson and Monica 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.

[52] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 279–289, New York, NY, USA, 2013. ACM.

[53] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.

[54] Suan Hsi Yong, Susan Horwitz, and Thomas 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.

[55] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing 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.

[56] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 829–845, New York, NY, USA, 2014. ACM.

[57] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. *SIGPLAN Not.*, 39(6):145–157, June 2004.

Appendix A

Brief User Guide

These are the steps to correctly compile and use the pointer analysis.

You will first need to build LLVM:

1. Download the open-source code on LLVM's site.

2. Create a folder 'build' on LLVM folder path.

3. Configure LLVM by:

```
$<LLVM_BUILD_PATH> ../$(LLVM_SRC_PATH)/configure
 --enable-cxx11 -enable-profiling --enable-clang-plugin-support --e
```

4. Compile LLVM by running make in the project root folder

```
make -j ENABLE_PROFILING=1
```

5. (Optional) Install the LLVM executable with:

```
sudo make install
```

To build the pointer analysis on LLVM:

1. Create folder 'PointerAnalysis' into

```
$(LLVM_BUILD_PATH)/lib/Transforms/
```

2. The pointer analysis is in the folder lib/Transforms/PointerAnalysis/. It is compiled separately and dynamically linked with the LLVM optimizer.

3. Run make in the analysis build folder.

4. To run a pass over an IR file, run the following command:

```
$<PROJECT_FOLDER>/Debug+Asserts/bin/opt -load PointerAnalysis.so -load
$<PROJECT_FOLDER>/Debug+Asserts/lib/PointerAnalysis.so
        -PointerAnalysis <InputProgramFile.s>
```

Appendix B

How To Run bddbddb

bddbddb is a free, open-source project that can be acquired from SourceForge [1].

To run bddbddb we just have to put all the needed libraries in the classpath and use net.sf.bddbddb.Solver as the main class for the Java VM. We give a Datalog query program as an input file. The bddbddb script provided simplifies setting the CLASSPATH and invoking the correct bddbddb classes. To use this script on a specific Datalog program, run the following command:

```
$bddbddb $<datalog\_program>
```

For help with running the bddbddb command, run:

```
$bddbddb -h
```