Efficient Flow-Sensitive Pointer Analysis on Full-Sparse Memory SSA

by

Sen Ye

A THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF NEW SOUTH WALES

SYDNEY • AUSTRALIA

School of Computer Science and Engineering Faculty of Engineering

October 28, 2015
### Abstract 350 words maximum: (PLEASE TYPE)

Pointer analysis is a fundamental research topic in computer science. It statically determines the potential runtime targets of pointers. Many clients benefit from this information, including compiler optimization, bug detection, security analysis and change impact analysis, etc. As a key dimension in pointer analysis, flow-sensitivity improves its precision by considering program execution order.

Ideally, flow-sensitive pointer analysis should be performed by analyzing each program path independently. However, even ignoring the branch conditions, such solution remains intractable and extremely expensive for whole program analysis due to potentially unbounded program paths. Finding the right balance between precision and efficiency in flow-sensitive pointer analysis lies at the heart of pointer analysis.

In this thesis, we first introduce an efficient inter-procedural full-sparse memory SSA construction algorithm. Then we improve flow-sensitive pointer analysis based on the memory SSA with following two contributions:

First, SELFS, a region-based selective flow-sensitive pointer analysis, is proposed to allow precision and efficiency trade-offs to be made according to region partitioning. By maintaining flow-sensitivity between regions instead of statements, SELFS is faster than the state-of-the-art full-sparse flow-sensitive analysis while achieving the same precision when used for alias queries.

Second, SPAS explores the intra-procedural path correlations on top of sparse flow-sensitive and context-sensitive pointer analysis. By using binary decision diagrams to represent the compact path conditions, SPAS improves the precision of pointer analysis while only introducing a small overhead.
COPYRIGHT STATEMENT

'I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or hereafter known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only).

I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.'

Signed ..............................................................

Date ..............................................................

17/04/2015

AUTHENTICITY STATEMENT

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed ..............................................................

Date ..............................................................

17/04/2015
ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ........................................

Date ..................17/04/2015......................
Abstract

Pointer analysis is a fundamental research topic in computer science. It statically determines the potential runtime targets of pointers. Many clients benefit from this information, including compiler optimization, bug detection, security analysis and change impact analysis, etc. As a key dimension in pointer analysis, flow-sensitivity improves its precision by considering program execution order. Ideally, flow-sensitive pointer analysis should be performed by analyzing each program path independently. However, even ignoring the branch conditions, such solution remains intractable and extremely expensive for whole program analysis due to potentially unbounded program paths. Finding the right balance between precision and efficiency in flow-sensitive pointer analysis lies at the heart of pointer analysis.

In this thesis, we first introduce an efficient inter-procedural full-sparse memory SSA construction algorithm. Then we improve flow-sensitive pointer analysis based on the memory SSA with the following two contributions:

First, SELFS, a region-based selective flow-sensitive pointer analysis, is proposed to allow precision and efficiency trade-offs to be made according to region partitioning. By maintaining flow-sensitivity between regions instead of statements, SELFS is faster than the state-of-the-art full-sparse flow-sensitive analysis while achieving the same precision when used for alias queries. Second, SPAS explores the intra-procedural path correlations on top of sparse flow-sensitive and context-sensitive
pointer analysis. By using binary decision diagrams to represent the compact path conditions, SPAS improves the precision of pointer analysis while only introducing a small overhead.
Acknowledgements

Foremost, I would like to express my great thanks to my supervisor, Prof. Jingling Xue, for his kindness and continuous support during the past four years. His patience, encouragement and immense knowledge pave the way for my success. His guidance helped me in all the time of research and writing this thesis. It would be much more difficult for me to complete my Ph.D study without his help. He showed us what a top researcher looked like and how to think, prove and verify problems in a thorough way. He is also the best supervisor with many brilliant ideas I have met.

I am also grateful to Yulei Sui, the postdoctoral research associate in CORG, UNSW. We have worked together for the past four years and he contributed a great deal to my research work in pointer analysis.

It is a great experience to work with all the other members from CORG group, Peng Di, Xinwei Xie, Lei Shang, Qing Wan, Ding Ye, Yu Su, Yue Li, Hao Zhou, Xiaokang Fan, Tian Tan, Hua Yan and Feng Zhang. We work, study and live together just as a big family. Life becomes colorful with all these friends around.

I would like to thank the Graduate Research School of UNSW and the Ministry of Education of China for their generous financial support which made my living and studies much easier.

Finally, I thank all my family members who love and support me. It is difficult
to live away from them. They are the spirit of my courage and hope in the lonely
days. I hope them are all healthy and live happily.
# Table of Contents

Abstract ........................................... i

Acknowledgements .................................. iii

List of Figures ..................................... viii

List of Tables ...................................... ix

List of Algorithms ................................. x

1 Introduction ...................................... 1
   1.1 Challenges .................................. 3
   1.2 Contributions ................................ 4
   1.3 Thesis Organization .......................... 5

2 Background ....................................... 7
   2.1 Flow-Sensitivity .............................. 7
   2.2 Field-Sensitivity ............................. 10
   2.3 Full-Sparse Memory SSA ..................... 12
       2.3.1 Memory SSA form ....................... 12
       2.3.2 Memory Region Partition ............... 14
       2.3.3 Memory SSA Construction Algorithm .. 16
2.4 Summary ......................................................... 21

3 Region-based Selective Flow-Sensitive Pointer Analysis 22

3.1 Overview ......................................................... 22

3.2 The SELFS Analysis Framework ............................... 26

3.2.1 Canonical Representation ................................ 27

3.2.2 Region Graph ................................................. 27

3.2.3 Region-Based Flow-Sensitivity ............................... 29

3.3 Instantiating the SELFS Analysis ............................... 32

3.3.1 Load-Precision-Preserving Partitioning ................... 33

3.3.2 Unification ...................................................... 34

3.4 Evaluation ....................................................... 39

3.4.1 Methodology .................................................... 39

3.4.2 Implementation ............................................... 42

3.4.3 Results and Analysis ........................................ 42

3.5 Related Work ..................................................... 44

3.6 Summary ......................................................... 46

4 SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA 47

4.1 Overview ......................................................... 47

4.2 The Basic Idea ................................................... 50

4.3 A Motivating Example .......................................... 53

4.4 The SPAS Framework ............................................ 57

4.4.1 Encoding Program Paths as BDDs ......................... 57

4.4.2 Extended SSA Form ......................................... 58

4.4.3 Capturing Path Correlation ................................ 58
4.4.4 Supporting Strong Updates ........................................ 65
4.5 Experimental Evaluation ............................................. 68
    4.5.1 Analysis Overhead ............................................ 68
    4.5.2 Path-Sensitive Precision ...................................... 70
4.6 Summary ............................................................. 72

5 Conclusions and Future Work ........................................ 74
    5.1 Conclusions ...................................................... 74
    5.2 Future Work ..................................................... 75

Bibliography .............................................................. 77
List of Figures

2.1 Flow-insensitive and -sensitive points-to results for the same code. . 8
2.2 A C program and its corresponding SSA form. . . . . . . . . . . . . 14
3.1 Rules for two traditional pointer analyses, FI and FS, for C pro-
grams. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.2 An illustration of SELFS performed on regions $\gamma_1$ and $\gamma_2$ . . . . . . 26
3.3 The region graph in Figure 3.2 redrawn with all indirect def-use
edges made explicit. . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.4 Inference rules for SELFS (with top-level variables in SSA). . . . . . 30
3.5 Some possible unification sequences illustrated for a part of a region
graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.6 Percentage distributions of SELFS' analysis time for all benchmarks 43
4.1 Effects of path-sensitivity on indirect updates at a store $*x=q$ in
FSCS analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.2 Level-wise SSA construction and path-sensitive pointer analysis for
Figure 4.1(a). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.3 Bottom-up and top-down analysis of method $m$ at level $\ell$. . . . . . . 61
List of Tables

3.1 Program characteristics. ................................................. 40
3.2 Comparing SFS and SELFs. ............................................. 41
3.3 Propagation times of SFS and SELFs for analysing the address-taken
variables across their indirect def-use chains. ...................... 44
4.1 Rules for resolving points-to maps $\text{PtrMap}(x) = \{\text{Loc}(x), \text{Dep}(x)\}$ in
method $m$ for level $\ell$. .................................................... 63
4.2 Percentage increases of analysis overhead under SPAS w.r.t. LevPA. 69
4.3 Percentages of variables at $\chi$’s and $\mu$’s with more accurate points-to
sets. ................................................................. 71
List of Algorithms

1. Memory SSA Construction ........................................ 17
2. Create Mu and Chi Functions ................................. 18
3. Place PHI Function for Memory Regions ..................... 19
4. SSA Rename .......................................................... 20
Chapter 1

Introduction

Software has reached every corner in modern society by providing various features, and yet at the same time it also becomes more complex with the increasing size. For example, the number of LOC (lines of code) of linux kernal has exceeded 15 million [47]. This complexity makes it difficult to completely understand, optimize and test such large software. Dynamic analysis is an effective approach in debugging and finding true errors in software, but it suffers from the runtime overhead introduced by the instrumentation. It also cannot find all possible defects in the program because the program parts being analyzed depend on the specified runtime input.

Static analysis is the analysis of software that is performed without actually executing programs built from that software. Compared with dynamic analysis, it has full code coverage and does not depend on the environment where the software will be executed. This allows us to be more expressive and work for large classes of systems. Static analysis has already been widely used in various areas, such as compiler optimizations [20, 26], security analysis [3, 10, 22, 73], change impact analysis [1] and hardware synthesis [100].

Much progress has been made on static analysis, but it still remains a difficult
program to produce both a sound and precise result when considering the *indirection* introduced by *pointers*. As an important feature in many modern programming languages (e.g., C, C++, Java), pointers are used in almost every program for constructing a program efficiently. Pointers are used for indirect memory location accessing and looking up function targets during runtime. It is crucial to solve the targets of pointers as precisely as possible, i.e., the points-to set of pointers should be as small as possible.

Pointer analysis is a fundamental analysis in static program analysis. It tries to determine all possible runtime targets of pointers statically. It may also be referred to as *alias analysis* if the points-to information computed is used to determine the aliases in the programs. Like most static analyses, pointer analysis is undecidable [43, 48, 49, 65] in the presence of dynamically allocated memory and recursive data structures. All current research can only give approximation algorithms by ignoring certain parts of information in programs. Pointer analyses can be categorized subject to the following dimensions:

**Flow-Sensitivity** A pointer analysis is *flow-sensitive* if it respects the control flow in the programs and *flow-insensitive* otherwise. By considering the program execution order, flow-sensitive analysis computes and maintains a points-to solution for each program point, whereas the insensitive one only computes a single solution for the whole program. The former one is more precise while the latter one is more efficient.

**Context-Sensitivity** A pointer analysis is *context-sensitive* if it distinguishes different calling contexts and *context-insensitive* otherwise. By analyzing each procedure independently for each calling context, context-sensitive analysis avoids spurious value flows, thereby improving the precision of pointer analysis as opposed
to its insensitive counterpart which merges all calling contexts of the procedure and analyzes them together.

**Path-Sensitivity** A pointer analysis is *path-sensitive* if it considers the path correlations in programs and *path-insensitive* otherwise. By encoding program paths along with the points-to information, spurious results can be eliminated in path-sensitive analysis compared with the insensitive analysis.

**Field-Sensitivity** A pointer analysis is *field-sensitive* if it models each field of every aggregate instance with a unique variable and *field-insensitive* otherwise. By differentiating fields in different aggregate instances, field-sensitive analysis is more expressive and precise than the insensitive one when analyzing programs with many aggregate data types, such as structs, arrays and heaps.

Other dimensions, such as heap sensitivity [59, 91], demand-driven analysis [37, 99], etc., also affect the efficiency and precision trade-offs of pointer analysis.

### 1.1 Challenges

Pointer analysis is an active research topic in program analysis and many techniques have been developed in recent years [2, 34, 53, 54, 86]. Although many applications already have been carried out with the help of information provided by pointer analysis, it is noticeable that advanced techniques in pointer analysis are still missing in industrial products. For example, modern compilers, like GCC, Open64 and LLVM, are still using imprecise pointer analyses which are both *flow- and context-insensitive* to generate points-to solutions for succeeding optimizations.

As a simulation of program runtime behaviours, pointer analysis may need to consider all possible situations which may happen during the execution. Fully
sensitive analyses are attractable as they provide accurate points-to solutions which may further improve the precision of other applications. But these analyses are extremely expensive when analyzing programs with millions of lines code due to the infinite program paths and calling contexts [18, 60, 81, 85]. Blindly applying precise analysis across the entire program may even result in huge analysis overhead with little precision improvement [30, 77, 81]. Their counterparts, the insensitive analyses, are popular and adopted by many applications for their scalability [78,94]. By ignoring some program information, like control flows, calling contexts etc., these analyses are cheap and scale well to huge programs. However they may suffer from poor precision due to the lost information and make applications complain about the false alarms reported. How to find the right balance between efficiency and precision lies in the heart of designing and implementing a practical pointer analysis.

A program usually exhibits diverse characteristics in its different parts, which should be handled with various efficiency and precision trade-offs. It is crucial to make the right decisions for different parts of a program to avoid over- or under-analysing. Furthermore, different applications require different levels of precision. Pointer analysis is supposed to be flexible enough for precision adjustment and to be aware of the bottleneck of the overhead. It would be helpful if the pointer analysis could understand when and where to put more efforts, and gradually adapt the precision to meet the applications’ needs.

1.2 Contributions

This thesis makes the following contributions:
Chapter 1. Introduction

Region-Based Selective Flow-Sensitive Pointer Analysis

We introduce a new region-based SElective Flow-Sensitive (SElfS) approach to inter-procedural pointer analysis for programs written in C language. It operates on the regions partitioned from that program and maintains flow-sensitivity between regions instead of program statements. This allows different efficiency and precision trade-offs to be made subject to different region partitioning strategies used. By using a unification-based region partitioning approach, SElfS can accelerate the state-of-the-art flow-sensitive by 2.13X on average while maintaining almost the same precision for all alias queries. In addition, the best speedups observed are 7.45X and 6.08X for some benchmarks.

Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA

SPAS is a Scalable PAth-Sensitive framework which resolves points-to sets in C programs that exploits recent advances in pointer analysis. It enables intra-procedural path-sensitivity to be obtained in flow-sensitive and context-sensitive techniques (FSCS) scalably, by using BDDs to manipulate program paths and by performing pointer analysis level-by-level on a full-sparse memory SSA (Static Single Assignment) representation similarly as the state-of-the-art LevPA [96] (the FSCS version of SPAS). Compared with LevPA using all 27 C benchmarks in SPEC CPU2000 and CPU2006, SPAS incurs 18.42% increase in analysis time and 10.97% increase in memory usage on average, while guaranteeing that all points-to sets are obtained with non-decreasing precision.

1.3 Thesis Organization

The remainder of this thesis is organised as follows:

Chapter 2 provides some background information about flow-sensitivity and
field-sensitivity in pointer analysis. It also introduces the full-sparse memory SSA form which is used as a foundation of flow-sensitive pointer analysis.

Chapter 3 describes our region-based selective flow-sensitive inter-procedural pointer analysis (SEFS). By maintaining flow-sensitivity between regions instead of statements, SEFS is more flexible in making different efficiency and precision trade-offs. A unification-based region partitioning approach is also included.

Chapter 4 discusses incorporating path-sensitivity upon a flow- and context-sensitive pointer analysis. By capturing path correlations in the program, more precision is obtained with only a small analysis overhead.

Chapter 5 concludes this thesis and highlights some opportunities for future research not only for pointer analysis itself, but also combined with other application clients.
Chapter 2

Background

2.1 Flow-Sensitivity

Flow-sensitivity is an important dimension in pointer analysis. A flow-insensitive pointer analysis ignores the program execution order and produces a single conservative solution for the whole program [5, 32, 33, 38, 62]. In contrast, a flow-sensitive solution respects the control flow of the program and computes points-to solutions for each program point [11, 34, 82, 96].

Figure 2.1 shows the different points-to solutions produced by flow-insensitive and -sensitive analyses. $pt$ is the set of points-to relations computed by an Andersen-style flow-insensitive analysis. $pt[\ell_i]/pt[\ell_j]$ is the set of all points-to relations held before/after statement $\ell_i$ under flow-sensitive analysis. The points-to relations highlighted in red are spurious results produced by flow-insensitive analysis compared with the flow-sensitive analysis. The points-to relation $a \rightarrow b$ is valid only before statement $\ell_6$, and $c \rightarrow d$ is valid only before $\ell_7$. Any usage of those two points-to relations after the two statements may produce spurious results. These differences between flow-insensitive and -sensitive analyses are caused by the dif-
ferent treatments of statements $\ell_6$ and $\ell_7$. For the sensitive analysis, the points-to relations $a \rightarrow b$ and $c \rightarrow d$ are killed at $\ell_6$ and $\ell_7$ respectively due to the strong updates. On the other hand, the flow-insensitive analysis never removes any points-to relations, thereby keeping those two relations as spurious results.

<table>
<thead>
<tr>
<th>$\ell_1$: $p = &amp;a;$</th>
<th>$\ell_2$: $q = &amp;c;$</th>
<th>$\ell_3$: $a = &amp;b;$</th>
<th>$\ell_4$: $c = &amp;d;$</th>
<th>$\ell_5$: $t = *p;$</th>
<th>$\ell_6$: $*p = *q;$</th>
<th>$\ell_7$: $*q = t;$</th>
<th>$pt = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d, t \rightarrow b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p[t_1] = \emptyset$</td>
<td>$p[t_1] = {p \rightarrow a}$</td>
<td>$p[t_2] = p[t_1]$</td>
<td>$p[t_2] = {p \rightarrow a, q \rightarrow c}$</td>
<td>$p[t_3] = p[t_2]$</td>
<td>$p[t_3] = {p \rightarrow a, q \rightarrow c, a \rightarrow b}$</td>
<td>$p[t_4] = p[t_3]$</td>
<td>$p[t_4] = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d}$</td>
</tr>
<tr>
<td>$p[t_5] = p[t_4]$</td>
<td>$p[t_5] = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d}$</td>
<td>$p[t_6] = p[t_5]$</td>
<td>$p[t_6] = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d, t \rightarrow b}$</td>
<td>$p[t_7] = p[t_6]$</td>
<td>$p[t_7] = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d, t \rightarrow b}$</td>
<td>$p[t_7] = p[t_6]$</td>
<td>$p[t_7] = {p \rightarrow a, q \rightarrow c, a \rightarrow b, c \rightarrow d, t \rightarrow b}$</td>
</tr>
</tbody>
</table>

(a) C code (b) Flow-insensitive results (c) Flow-sensitive results

Figure 2.1: Flow-insensitive and -sensitive points-to results for the same code. Points-to relations highlighted in red are spurious results computed by the flow-insensitive pointer analysis compared with the flow-sensitive analysis.

By respecting the control flow in the program, flow-sensitive analysis greatly improves the precision of pointer analysis compared to flow-insensitive analysis. However, it suffers from the scalability problem. The traditional flow-sensitive algorithm is built upon an iterative data-flow analysis framework. It is inefficient for analyzing large programs for two reasons: First, the points-to information is propagated blindly from the birth points to all reachable program points. Second, every program point keeps two sets of points-to information, one for the incoming results received from all of its predecessors and another for the outgoing information which will be propagated to all of its successors. Some points-to relations reside in both sets at a program point but are never used there. For example, $p \rightarrow a$ exists in all points-to solutions in Figure 2.1(c), but it is generated at program point $\ell_1$ and is only used by $\ell_5$ and $\ell_6$. It is a waste of time and memory to store this points-to relation in other program points or to propagate between them.
The Singleton Strong Updates approach [52] sacrifices the precision of flow-sensitive analysis to gain efficiency. Strong updates are the main benefit of a flow-sensitive analysis compared with a flow-insensitive analysis and they can be performed only at store statements which access a single concrete memory location. So this approach focuses on store statements where flow-sensitive singleton points-to sets are available and perform strong updates for them to produce the precise points-to solutions. It falls back to the flow-insensitive points-to information for other store statements.

Sparse flow-sensitive pointer analysis [35, 55, 60] is the state-of-the-art flow-sensitive pointer analysis, which can improve the performance of the traditional flow-sensitive analysis while maintaining the same precision. Instead of propagating information along every statement in a program, it propagates points-to solutions according to a pre-computed value-flow graph of the program. This graph is built by converting the program into the SSA (Static Single Assignment) form [15] and then connecting the variables’ definition sites to their corresponding use sites. The propagation through this graph is sparser than the traditional flow-sensitive pointer analysis since the points-to information can be fed into their use sites directly from the program points where they are generated. This sparse method significantly reduces both the points-to information maintained in analysis and the propagation overhead, thereby making flow-sensitive pointer analysis scalable for large programs.

LevPA [96] achieves the sparsity in a different way. It builds the full-sparse memory SSA form incrementally. In LevPA, all pointers in a program are divided into different groups according their points-to levels computed by the Steensgaard pointer analysis [75]. Then the points-to resolution is carried out in the decreasing order of the points-to levels. The results of pointers at a certain level are computed
based on the SSA representation created from a higher level. When the points-to information of pointers at this level is completely resolved, it is safe to perform SSA conversion for pointers at a lower level and prepare them for the points-to resolution on that level afterwards.

2.2 Field-Sensitivity

Aggregate data structures which contain multiple fields, e.g., structs and arrays, are heavily used in C programs. Unlike Java, which is a strongly-typed language, pointer analysis for C programs has to develop its own semantic assumptions. The abstract representations of aggregate data structures greatly affect the precision and scalability of pointer analysis.

For efficiency concerns, some previous work ignored the field-information by treating aggregate data structure as a single variable and produced field-insensitive points-to results [21, 23, 36, 41]. Others adopted the field-based approach by using one variable to model all instances of a certain field in the aggregate data structure [2, 27, 38]. The precision of both methods highly depends on the programs being analyzed: they work well for programs with a small number of aggregate data structures, while producing imprecise results for programs which use a large collection of aggregates.

To provide more precise points-to information, it is important to distinguish different fields in the same aggregate data structure in pointer analysis. However, due to the lack of type safety guarantees in the C language, it can be challenging to develop a field-sensitive pointer analysis algorithm for the C language by modeling the aggregate data structures precisely. Currently there are two major modelling strategies to obtain field-sensitivity.
Field-Index-Based  Pearce et al. [61] were the first to propose and implement an efficient *field-sensitive* pointer analysis for C programs. Each field of an aggregate in the program is labelled with a unique index. This simplifies storing and processing fields as they can be handled using bit vectors. The work can also handle pointer arithmetic by adding the offset directly onto the index of the variable since variables within the same aggregate are labelled with continuous indices. They give a thorough discussion about the existence of cycles in pointer arithmetic. The modeling of the arrow operator is also mentioned and provided as the implicit pointer arithmetic operations.

Offset-Based  Wilson et al. [87] proposed an offset-based approach to perform field-sensitivity. Unlike the field-index-based approach, each aggregate data represents a memory block and different variables that reside in the same aggregate are distinguished by their offsets to the base of the memory block. *Location set*, a tuple containing an offset and a stride, is used to record the position of each variable in a memory block. If the stride is not zero, the location set represents multiple memory locations within the memory block, so it is more expressive than the field-index-based approach as it can represent multiple fields in an array variable. This work not only discussed arithmetic operations on location set, but also included memory location intersections and pointer alignments which are common in compiler techniques.

In this thesis, we use the offset-based approach to obtain the field-sensitivity in pointer analysis with the following restrictions:

- Casting. As there are no type safety guarantees in C programs, castings between pointers are be treated the same way as assignments, while castings between pointers and non-pointers are not be considered since the destination
targets are unknown.

- Pointer Arithmetic. Pointer arithmetic is handled by adding offsets together and pointer alignments are applied where necessary.

- Structure to Structure Copy. Assignments are created between corresponding fields from the source and destination structures.

- Heap. Heap objects are represented by their allocation sites without heap cloning. If the size of a heap object is not clear during the analysis, a conservative upper bound size is used to limit the possible objects created in the analysis.

- Arrays. Arrays are treated monolithically, i.e., only the first element in the array is modelled and it represents all other elements in the same array variable.

2.3 Full-Sparse Memory SSA

2.3.1 Memory SSA form

The Static Single Assignment (SSA) form is an intermediate representation of a program in which every variable can only be defined exactly once. Variables with multiple definitions are replaced with different instances, one for each definition point. \( \phi \) function will be added at the joint points of control flows for a variable which has multiple instances live there. The \( \phi \) function takes all the live instances of the variable as the operands and generates a new instance to be used afterwards.

Programs may contain indirect operations that access variables implicitly, e.g. load and store statements. The exact variables defined or used through such indirect operations cannot be determined during static analysis, thereby making the
SSA conversion more complicated. As pointer analysis only gives conservative approximations of the points-to results, store or load statements can only be considered as possible definition or use sites respectively for variables implicitly accessed there. When targets for pointers are obtained, the SSA conversion of indirect accessed variables at load and store statements can be completed by adding $\mu$ and $\chi$ functions introduced in [14]. If a store statement implicitly defines a variable $v$, it is annotated with a function $v = \chi(v)$ to represent the possible new definition of $v$. Similarly, if a load statement implicitly accesses a variable $v$, it is annotated with a function $\mu(v)$ to represent the use of $v$ at that program point. After $\mu$ and $\chi$ functions are added for all variables accessed indirectly, they can be converted to SSA form using the standard algorithm proposed for the explicitly accessed variables [4, 6, 7, 15]. Figure 2.2 shows the SSA form of a C program. Variables with numeric footnotes are different instances of variables representing the multiple definitions in the original program. Variables $a$ and $b$ are considered as points-to targets of $p_3$, so $\mu$ and $\chi$ functions are added at $\ell_4$ and $\ell_5$ for them respectively.

With the help of the SSA form, the value-flow graph can be easily constructed by connecting the use sites of variables to their corresponding definition sites. Then flow-sensitive analysis can be carried out on the value-flow graph by propagating points-to relations sparsely, i.e. along the value-flow edges only. This significantly reduces both time and memory overheads of the analysis while keeping the same precision as the traditional flow-sensitive analysis. It can also be observed that the efficiency of this sparse flow-sensitive analysis depends highly on the sparsity of the value-flow graph constructed. The sparser the graph is, the more efficient the pointer analysis becomes.

**Staged Analysis** In order to perform a full-sparse flow-sensitive pointer analysis, a value-flow graph should be built for both explicitly and implicitly accessed
variables in the program. However, building such a graph requires the information about which variables would be accessed implicitly through those indirect operations, thus requiring points-to information of the program to be computed first. Hardekopf et al. [35] proposed the staged analysis to break this cyclic dependence. First, a less precise but cheaper pointer analysis is conducted over the target program. Then the SSA form and the value-flow graph can be built based on the information obtained from the pre-analysis. In the final stage, a full-sparse flow-sensitive analysis is carried out over the value-flow graph to produce a more precise points-to solution that can be used for applications.

### 2.3.2 Memory Region Partition

It is crucial to build a compact graph to minimize the propagation overhead for the analysis. Since the value-flow graph is built upon the points-to information provided by the pre-analysis, it is obvious that the more precise the pre-analysis

---

| \( \ell_1 \): | \( p = \&a \); |
| \( \ell_2 \): | \( q = p \); |
| \( \ell_3 \): | \( if(*) p = \&b \) |
| \( \ell_4 \): | \( *p = r \); |
| \( \ell_5 \): | \( q = *p \); |

(a) Original program

| \( \ell_1 \): | \( p_1 = \&a \); |
| \( \ell_2 \): | \( q_1 = p_1 \); |
| \( \ell_3 \): | \( if(*) p_2 = \&b \) |
| \( \ell_4 \): | \( *p_3 = r_1 \); |
| \( \ell_5 \): | \( q_2 = *p_3 \); |

\( p_3 = \phi(p_1, p_2) \)

\( a_1 = \chi(a_0) \)

\( b_1 = \chi(b_0) \)

\( \mu(a_1) \)

\( \mu(b_1) \)

(b) SSA form

---

Figure 2.2: A C program and its corresponding SSA form.
is, the sparser the value-flow graph will be. However in a staged analysis, it is meaningless to use a pointer analysis that is more precise than the main analysis in the pre-analysis stage. For flow-sensitive analysis, it is sufficient to use an inclusion-based analysis as the pre-analysis to construct the value-flow graph. Empirical studies \cite{39,41,42} suggested that flow-insensitive pointer analysis already provides points-to solution that has comparable precision as the one computed by a flow-sensitive analysis without considering context-sensitivity.

Another solution is to group variables into different memory regions, and then perform the SSA conversion and value-flow graph construction for those regions instead of the variables. Access equivalence \cite{35} is one of the memory region partitioning strategies for obtaining such a compact value-flow graph. Two variables are access equivalent if they are always accessed by the same indirect operations, i.e. load and store statements. Because the SSA algorithm always computes identical def-use chains for all access equivalent variables, points-to information related to those variables are always propagated along the same program paths. Instead of creating $\mu$ and $\chi$ functions, performing the SSA conversion and connecting def-use edges for each of the indirect accessed variables, it would be much simpler to perform these operations upon memory regions that contain access equivalent variables. Hardekopf et al. \cite{35} give detailed information about how to perform the SSA conversion and value-flow graph construction with access equivalence. An inter-procedural control-flow graph (ICFG), with indirect call sites connected to their potential function targets resolved by the pre-analysis, is used to facilitate the building process.

Although access equivalence reduces the complexity of the value-flow graph, determining the access equivalent variables requires extra effort as it checks all indirect operations in the program. This process takes $O(I \cdot V)$ time, where $I$
is the number of load and store statements in the program and \( V \) is the number of variables that may be indirectly accessed. For large programs with massive load and store statements, finding access equivalent variables may be extremely expensive since variables are checked pair-wise through all indirect instructions in the program.

The efficiency of finding access equivalent variables can be improved by performing those operations in an *intra*-procedural manner instead of checking across the entire program. If two variables are access equivalent *within* a method, i.e. they are accessed at the same load and store statements in this method, we group them into the same memory region even if they may not be access equivalent in other methods. Due to the program locality, most variables are defined and used locally, so it would be much easier to find access equivalent variables within the same method.

One problem of performing this intra-procedural access equivalence checking is that each method has its own set of memory regions that may be different from others, thereby complicating the connecting of def-use edges which may go across procedure boundaries. It may not be possible to find two regions that are exactly the same and reside in caller and callee methods respectively. When this happens, def-use edges are added for any two memory regions that are aliased, i.e. they have overlapped access equivalent variables.

### 2.3.3 Memory SSA Construction Algorithm

The algorithm for our memory SSA construction is highlighted in Algorithm 1. It is a derivation of the algorithm illustrated in [83]. There are three phases: (1) Creating \( \mu \) and \( \chi \) functions for variables at their use and definition sites; (2) Adding \( \phi \) functions for multiple definitions of the same variable that live at joint
points of control flows; (3) Performing SSA conversion to rename all the instances of variables. After these three phases, the final value-flow graph can be easily constructed by connecting the variables’ definition sites to their corresponding use sites.

Algorithms 2, 3 and 4 are almost the same as in [83], except that when annotating $\mu$ and $\chi$ functions to load and store statements in Algorithm 2, the `getMemRegion()` method is used as a general interface for memory region partitioning. It accepts a points-to set computed by the pre-analysis and returns a memory region that contains access equivalent variables. Then $\mu$ or $\chi$ functions are added for this memory region instead of those variables.

Algorithm 1 Memory SSA Construction

**Procedure** `ConstructMemSSA()`

```plaintext
def Procedure ConstructMemSSA()
begin
  foreach proc in CallGraph do
    CREATEMUCHI(proc)
    INSERTPHI(proc)
    SSARENAME(proc)
end
```
Algorithm 2 Create Mu and Chi Functions

Procedure CREATEMUCHI(proc)

begin

1. foreach bb ∈ proc do

2.   foreach ℓ : p=*q ∈ bb do

3.     mr = getMemRegion(pt(q))

4.     add µ(mr) before ℓ

5.   foreach ℓ : *p=q ∈ bb do

6.     mr = getMemRegion(pt(p))

7.     add mr = χ(mr) after ℓ
Algorithm 3 Place PHI Function for Memory Regions

Procedure INSERTPHI(proc)

begin
1 \text{FINDGLOBALNAMES(proc)}
2 let DF(bb) be the dominant frontiers of bb
3 \textbf{foreach} mr ∈ Globals \textbf{do}
4 \hspace{1em} W ← Blocks(mr)
5 \hspace{1em} \textbf{while} W ≠ ∅ \textbf{do}
6 \hspace{2em} bb ← W.pop()
7 \hspace{2em} \textbf{foreach} bb' ∈ DF(bb) \textbf{do}
8 \hspace{3em} \text{inset } φ\text{-function for } x \text{ in } bb'
9 \hspace{3em} W ← W ∪ bb'

Procedure FINDGLOBALNAMES(proc)

begin
10 \hspace{1em} Globals ← ∅
11 \textbf{foreach} bb ∈ proc \textbf{do}
12 \hspace{2em} VarKill ← ∅
13 \hspace{2em} \textbf{foreach} µ(mr) in bb \textbf{do}
14 \hspace{3em} \textbf{if} mr \notin VarKill \textbf{then}
15 \hspace{4em} Globals ← Globals ∪ mr
16 \hspace{2em} \textbf{foreach} mr = χ(mr) in bb \textbf{do}
17 \hspace{3em} VarKill ← VarKill ∪ mr
18 \hspace{3em} Blocks(mr) ← Blocks(mr) ∪ bb
Algorithm 4 SSA Rename

Procedure SSARename(proc)

begin
1. let $bb_{entry}$ be the basic block at the entry of procedure $proc$
2. $\text{renameBB}(bb_{entry})$

Procedure $\text{renameBB}(bb)$

begin
3. let $DF(bb)$ be the dominant frontiers of $bb$
4. foreach $(mr = \phi(...)) \in bb$ do
   5. rename $mr$ as $\text{NEWSSANAME}(mr)$
6. foreach $(\mu(mr)) \in bb$ do
   7. rewrite $mr$ as top(stack[$mr$])
8. foreach $(mr = \chi(mr')) \in bb$ do
   9. rewrite $mr'$ as top(stack[$mr$])
   10. rewrite $mr'$ as $\text{NEWSSANAME}(mr)$
9. foreach successor in the CFG do
   10. fill in $\phi$-function parameters
11. foreach successor $s$ in the dominator tree do
   12. $\text{renameBB}(s)$
13. foreach $(mr = \chi(...))and(mr = \phi(...)) \in bb$ do
   14. $\text{Pop}(stack[mr])$

Procedure $\text{NEWSSANAME}(bb)$

begin
15. $i \leftarrow \text{counter}[n]$
16. $\text{counter}[n] \leftarrow \text{counter}[n] + 1$
17. push $n_i$ onto stack[$n$]
18. return $n_i$
2.4 Summary

In this chapter, we introduced some background information about flow-sensitivity and field-sensitivity which affect the precision and scalability of pointer analysis. We also discussed the construction and efficiency of full-sparse memory SSA form which is already used by many applications [35, 78, 95]. The efficiency of the memory SSA form depends on several factors, such as the precision of the pre-analysis and the memory region partition strategy.
Chapter 3

Region-based Selective

Flow-Sensitive Pointer Analysis

3.1 Overview

In this chapter we present SELFS, a new region-based selective flow-sensitive pointer analysis. In SELFS, flow-sensitivity is maintained between regions instead of statements, thereby allowing different efficiency and precision trade-offs to be made subject to different region partitioning strategies used. With a unification-based region partitioning approach, SELFS can accelerate the state-of-the-art sparse flow-sensitive pointer analysis and yet achieve nearly the same precision for almost all practical purposes, such as answering the alias queries.

Finding a right balance between efficiency and precision lies at the core of pointer analysis. A flow-insensitive analysis (FI), as formulated for C using Andersen’s algorithm [2] in Figure 3.1(a), is fast but imprecise, because it ignores control flow and thus computes a single solution $pt$ to the entire program. Here, $pt(v)$ gives the points-to set of a variable $v$. In contrast, a flow-sensitive analysis (FS),
as formulated by solving a data-flow problem in Figure 3.1(b), makes the opposite trade-off. By respecting control flow ([S-OUTIN], [S-INOUT1] and [S-INOUT2]), separate solutions $pt[\bar{\ell}]$ and $pt[\ell]$ at distinct program points $\bar{\ell}$ and $\ell$ (the ones immediately before and after each $\ell$-labeled statement) are computed and maintained. Preserving flow-sensitivity this way has two precision benefits. One is to track the values read at a location through the control flow. The other is to enable strong updates: if a location is definitely updated by an assignment, then the previous values at the location can be killed. In [S-ADDRDF], [S-COPY] and [S-LOAD], $p$ at $\ell$ is strongly updated since $pt[\bar{\ell}](p)$ is killed. For any $q \neq p$, its points-to information is preserved ([S-INOUT1]). In [S-STORE], $o$ at $\ell$ is strongly updated if $o$ is a singleton, i.e., a concrete location uniquely pointed to by $p$ ([S-STORESU]) and weakly updated otherwise ([S-STOREWU]). For a target $o'$ not pointed by $p$, its points-to information remains unchanged ([S-INOUT2]).

Flow-sensitivity is beneficial for a wide range of clients such as bug detection [58, 78, 79, 94], program verification [18, 22] and change impact analysis [1, 9]. As the size and complexity of software increases, how to achieve flow-sensitivity exactly or approximately with desired efficiency and precision trade-offs becomes attractive. As mentioned in Section 2.1, the sparse approach [35, 60, 96] aims to achieve the same precision as FS by performing the points-to facts propagation sparsely along the pre-computed def-use chains instead of across all program points as FS does. Alternatively, the singleton strong updates approach [52] sacrifices the precision of FS in order to gain better efficiency by performing strong updates only at store statements where flow-sensitive singleton points-to sets are available. Despite these recent advances on flow-sensitive analysis, balancing efficiency and precision remains challenging, partly due to the difficulty in orchestrating various algorithms used during the analysis and partly due to the desire to meet different
The image contains a table and some text. The table outlines rules for two traditional pointer analyses, FI and FS, for C programs.

(a) FI: constraints for Andersen’s algorithm (flow-insensitive)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>[I-ADDROF]</td>
<td>( p = &amp; o ) {o} \subseteq pt(p)</td>
</tr>
<tr>
<td>[I-COPY]</td>
<td>( p = q ) \quad pt(q) \subseteq pt(p)</td>
</tr>
<tr>
<td>[I-STORE]</td>
<td>( \ast p = q ) \quad o \in pt(p) \qquad pt(q) \subseteq pt(o)</td>
</tr>
<tr>
<td>[I-LOAD]</td>
<td>( p = \ast q ) \quad o \in pt(q) \qquad pt(o) \subseteq pt(p)</td>
</tr>
</tbody>
</table>

(b) FS: constraints for data-flow (flow-sensitive)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>[S-ADDROF]</td>
<td>( \ell; \quad p = &amp; o ) {o} \subseteq pt<a href="p">\ell</a></td>
</tr>
<tr>
<td>[S-COPY]</td>
<td>( \ell; \quad p = q ) \quad pt<a href="q">\ell</a> \subseteq pt<a href="p">\ell</a></td>
</tr>
<tr>
<td>[S-STORE]</td>
<td>( \ell; \quad \ast p = q ) \quad o \in pt<a href="p">\ell</a> \qquad pt<a href="q">\ell</a> \subseteq pt<a href="o">\ell</a></td>
</tr>
<tr>
<td>[S-LOAD]</td>
<td>( \ell; \quad p = \ast q ) \quad o \in pt<a href="q">\ell</a> \qquad pt<a href="o">\ell</a> \subseteq pt<a href="p">\ell</a></td>
</tr>
<tr>
<td>[S-OUTIN]</td>
<td>( \ell; \quad v \in V \quad \ell' \in succ(\ell) \qquad pt<a href="v">\ell</a> \subseteq pt<a href="v">\ell'</a></td>
</tr>
<tr>
<td>[S-INOUT1]</td>
<td>( \ell; \quad p = \ast q \quad q \neq p ) \quad pt<a href="q">\ell</a> \subseteq pt<a href="p">\ell</a></td>
</tr>
<tr>
<td>[S-INOUT2]</td>
<td>( \ell; \quad \ast p = q \quad o' \notin pt<a href="p">\ell</a> \qquad pt<a href="o'">\ell</a> \subseteq pt<a href="o'">\ell</a></td>
</tr>
</tbody>
</table>

\( V \): set of variables \quad succ: mapping statements to control flow successors

Figure 3.1: Rules for two traditional pointer analyses, FI and FS, for C programs.

Clients’ needs. A program usually exhibits diverse characteristics in its different code regions, which should be handled with different efficiency and precision trade-offs (to avoid under- or over-analysing).

SEILS operates on the regions partitioned from a program rather than individual statements as in [35, 96]. Top-level pointers can be put in SSA form [15] without requiring pointer analysis. To track the value-flows of address-taken variables effectively, we will perform a pre-analysis to enable their sparse analysis as in [35, 60, 96], but on a region graph with its regions containing loads and stores. Each region is analysed flow-insensitively but flow-sensitivity is maintained across...
the regions.

Consider Figure 3.2, where the points-to relations in *init* are known before the code is analysed. Figure 3.2(a) depicts the points-to relations obtained by applying FS to the code. Note that a strong update on $m$ (assumed to be a singleton) is performed at $\ell_3$. Figure 3.2(b) gives the solution obtained by SELFS on a region graph (with two regions $\gamma_1$ and $\gamma_2$) to achieve more efficiently the same precision for reads from (but not necessarily for writes into) each variable. As $p$ points to $m$ and $n$, no strong update is possible at $\ell_1$. Instead of flow-sensitively propagating the points-to relations from $\ell_1$ to $\ell_2$, both can be analysed flow-insensitively in region $\gamma_1$ without any precision loss for the reads from $m$ and $n$ at $\ell_2$. Interestingly, even if a strong update is performed for $m$ at $\ell_3$, the points-to relations from $\ell_2$ and $\ell_3$ are merged on entry of $\ell_4$, making any read from $m$ at $\ell_4$ (if any) as precise as if $\ell_3$ and $\ell_4$ are analysed together in $\gamma_2$ flow-insensitively. Note that SELFS has over-approximated the potential target of $m$ at $\ell_3$: $m \rightarrow y$ found by FS in Figure 3.2(a) with $m \rightarrow x$, $m \rightarrow y$ and $m \rightarrow z$ given in Figure 3.2(b). As argued in Section 3.3, preserving the precision for reads from all variables always preserves the alias information (among others). By operating at the granularity of regions rather than statements while maintaining flow-sensitivity across their edges (illustrated further in Figure 3.3), SELFS is expected to run faster.

SELFS analysis is also advantageous in that region partitioning is separated as an independent concern from the rest of the analysis. Different region partitions may lead to different degrees of flow-sensitivity, resulting in different efficiency and precision trade-offs being made. As discussed in Section 3.3, the two traditional analyses, FI and FS, given in Figure 3.1 and some recent sparse flow-sensitive analyses [35, 60, 96] are all special instances of SELFS. As a result, SELFS provides a general framework for designers to develop and evaluate different flow-sensitive
Points-to relations resolved before $\ell_1$: \( \text{init} = \{ p \rightarrow m, p \rightarrow n, q \rightarrow m, r \rightarrow x, s \rightarrow y, t \rightarrow z \} \)

(a) Flow-sensitive analysis

(b) The SELFS analysis

Figure 3.2: An illustration of SELFS performed on regions $\gamma_1$ and $\gamma_2$ by preserving the precision of FS with respect to the reads from variables (with further details given in Figure 3.3 and Examples 1 – 3). The focus is on analysing the points-to relations for the top-level variable $v$ and the two address-taken variables $m$ (a singleton) and $n$, by assuming that the points-to relations in init are given.

variations by reusing existing pointer resolution algorithms.

### 3.2 The SELFS Analysis Framework

In this section, we present our SELFS analysis on a given region graph created from a program. How to partition a program into multiple regions will be discussed later in Section 3.3. Section 3.2.1 makes precise the canonical representation used for analysing C programs. Section 3.2.2 defines the region graphs operated on by SELFS. Section 3.2.3 formalises our region-based flow-sensitive pointer analysis.
3.2.1 Canonical Representation

In the pointer analysis literature, a C program is represented by a CFG (Control-Flow Graph) containing the four types of pointer-manipulating statements shown in Figure 3.1: $p = \&o$ (ADDROF), $p = q$ (COPY), $p = *q$ (LOAD) and $*p = q$ (STORE). More complex statements are decomposed into these basic ones. Passing arguments into and returning results from functions are modeled by copies. For a given ADDROF statement $p = \&o$, $o$ is either a stack variable with its address taken or an abstract object dynamically created at an allocation site.

For simplicity, we adopt the convention of LLVM by separating the set $V$ of all variables into two subsets, (1) $A$ containing all possible targets, i.e., address-taken variables of a pointer and (2) $T$ containing all top-level variables, where $V = T \cup A$.

For the four types of statements given, we have $p, q \in T$ and $o \in A$.

3.2.2 Region Graph

Our SELF analysis operates on a region graph created from a program being analysed. Leveraging recent progress on sparse flow-sensitive analysis [35, 60, 96], we will perform a pre-analysis to both guide region partitioning and enable sparse analysis at the granularity of regions rather than individual statements.

To obtain a region graph from a program, top-level and address-taken variables are treated differently. In our SELF framework, top-level variables are always analysed sparsely since they can be put in SSA without requiring pointer analysis. A top-level pointer that is defined multiple times is split into distinct versions after SSA conversion. All versions of a variable, say $q_1, \ldots, q_n$, that reach a joint point at the CFG are combined by introducing a new PHI statement, $q_j = \phi(q_{i_1}, \ldots, q_{i_n})$, where $q_j, q_{i_1}, \ldots, q_{i_n} \in T$, so that every version is defined once (statically). After SSA conversion, the (direct) def-use chains for all top-level variables are readily
available. As a result, their points-to sets can be simply obtained flow-sensitively by performing a flow-insensitive analysis.

Unlike top-level variables, address-taken variables are read/written indirectly via top-level pointers at loads/stores and thus harder to analyse sparsely. Sparsity requires points-to information to be propagated along def-use chains but the (indirect) def-use chains for address-taken variables can only be computed using points-to information. To break the cycle, we perform a pre-analysis as in [14,35,60,78] to first over-approximate indirect def-use chains and then refine them by performing a data-flow analysis sparsely along such pre-computed def-use chains.

Note that an address-taken variable \( o \) accessed at a store represents a potential use of \( o \) if a weak update is performed (\([S\text{-}STORE}^\text{WU}\]) in Figure 3.1(b)). Due to space limitation, we refer to [35] on how to over-approximate indirect def-use chains (via a pre-analysis named Aux). The basic idea is to annotate a load \( p = \ast q \) with a potential use of \( o \) for every \( o \) pointed to by \( q \) and a store \( \ast p = q \) with a potential use and def of \( o \) for every \( o \) pointed to by \( p \). Then indirect def-use chains can be built by putting all address-taken variables in SSA.

Therefore, SELFS keeps track of value flows for top-level variables in SSA explicitly along their (direct) def-use chains and refines value flows for address-taken variables along their pre-computed (indirect) def-chains in a region graph.

**Definition 1** (Region Graph). A region graph \( \mathcal{G}_{\text{rg}} = (\mathcal{N}_{\text{rg}}, \mathcal{E}_{\text{rg}}) \) for a program is a multi-edged directed graph. \( \mathcal{N}_{\text{rg}} \) is a partition of the set of all loads and stores in the program. \( \mathcal{E}_{\text{rg}} \) contains an edge \( \gamma_1 \xrightarrow{\omega} \gamma_2 \) labeled by an address-taken variable \( o \in \mathcal{A} \), where \( \gamma_1 \) and \( \gamma_2 \) may be identical, if there is an indirect def-use chain for \( o \) originating from \( \gamma_1 \) and ending at \( \gamma_2 \) computed by the pre-analysis.

**Example 1.** Consider our example again in Figure 3.3. Figure 3.3(b) duplicates the region graph from Figure 3.2(b) except that its edges are now annotated ex-
**Chapter 3. Region-based Selective Flow-Sensitive Pointer Analysis**

![Diagram](image)

(a) Indirect def-use chains

(b) Region graph

Figure 3.3: The region graph in Figure 3.2(b) redrawn with all indirect def-use edges made explicit. The pre-analysis yields \( p \rightarrow m, p \rightarrow n \) and \( q \rightarrow m \) (included in **init** in Figure 3.2).

Explicitly with address-taken variables indicating their value flows. By Definition 1, these edges are added based on the statement-level indirect def-use chains obtained by pre-analysis, given in Figure 3.3(a). The presence of self-loop edge(s) in a region allows naturally the loads/stores inside to be analysed flow-insensitively. □

### 3.2.3 Region-Based Flow-Sensitivity

Figure 3.4 gives the inference rules used in our SELFS framework. Top-level variables are analysed as before except that they are now in SSA ([R-PHI]). Therefore, analysing the top-level variables in SSA flow-insensitively ([R-ALLOC] and [R-COPY]) as in FI gives rise to the flow-sensitive precision obtained as in FS. As a result, the points-to sets \( pt(p) \) and \( pt(q) \) of top-level pointers \( p \) and \( q \) are directly read off at a load ([R-LOAD]), a store ([R-STORE]), and in [R-INOUT].

SELFS computes and maintains the points-to relations for address-taken variables sparsely in a region graph. Flow-sensitivity is maintained across the regions (along their indirect region-level def-use edges) but not inside. This implies that all statements in a region \( \gamma \) are handled flow-insensitively if \( |\gamma| > 1 \) and flow-sensitively
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>[R-ADDR]</strong></td>
<td>$\ell: p = &amp; o \quad { o } \subseteq pt(p)$</td>
</tr>
<tr>
<td><strong>[R-COPY]</strong></td>
<td>$\ell: p = q \quad pt(q) \subseteq pt(p)$</td>
</tr>
<tr>
<td><strong>[R-PHI]</strong></td>
<td>$\ell: p = \phi(o, q, \cdot) \quad pt(q) \subseteq pt(p)$</td>
</tr>
<tr>
<td><strong>[R-STORE]</strong></td>
<td>$\ell: \ast p = q \quad o \in pt(p) \quad \gamma = selR(\ell) \quad pt<a href="o">\gamma</a> \subseteq pt<a href="o">\gamma'</a>$</td>
</tr>
<tr>
<td><strong>[R-LOAD]</strong></td>
<td>$\ell: p = \ast q \quad o \in pt(q) \quad \gamma = selR(\ell) \quad pt<a href="o">\gamma</a> \subseteq pt<a href="o">\gamma'</a>$</td>
</tr>
<tr>
<td><strong>[R-DU]</strong></td>
<td>$\gamma \in Nrg \quad o' \notin { o \in pt(p) \mid (\ast p = q) \in \gamma } \quad pt<a href="o'">\gamma</a> \subseteq pt<a href="o'">\gamma'</a>$</td>
</tr>
</tbody>
</table>

Figure 3.4: Inference rules for SELFS (with top-level variables in SSA).

otherwise (i.e., if $|\gamma| = 1$). As SELFS operates at the granularity of regions, the notation $pt[\gamma]$ (pt[\ell]) for a region $\gamma$ is an analogue of $pt[\ell]$ (pt[\ell]) for a statement $\ell$, as already illustrated in Figure 3.2(b). For a region $\gamma$ containing multiple statements, $pt[\gamma]$ is the solution for all program points inside the region.

Below we explain the four rules, [R-DU], [R-INOUT], [R-LOAD] and [R-STORE], used to compute the points-to relations for address-taken variables.

Together with [R-INOUT], [R-DU] represents the sparse propagation of points-to relations for address-taken variables $o \in A$ across their pre-computed def-use chains at the granularity of regions. In contrast, FS propagates such points-to relations blindly across the control flow ([S-OUTIN], [S-INOUT1] and [S-INOUT2]).

In [R-LOAD], $\gamma = selR(\ell)$ is the region where the load at $\ell$ resides. No strong update is possible even if $\gamma$ contains a store since $|\gamma| > 1$ will then hold. Regardless of how many statements that $\gamma$ contains, the points-to set of $o$ at the entry of $\gamma$ is propagated into the points-to set of $p$: $pt[\gamma](o) \subseteq pt(p)$, where $pt[\gamma](o)$ contains the points-to relations that are (1) either received from its predecessors ([R-DU]) or (2) generated inside $\gamma$ (due to a self-loop edge labeled by $o$ when $|\gamma| > 1$), in which case, all statements inside $\gamma$ are analysed flow-insensitively.
[R-STORE] is similar except that the points-to set of o indirectly accessed at a store is updated at the end of the region γ that contains the store. [R-STORESU], which is explained earlier in Section 3.1, comes into play only when |γ| = 1. Recall that SELFS only analyses single-statement regions flow-sensitively.

Example 2. Let us apply our inference rules to the region graph in Figure 3.3(b) to obtain the points-to relations given in Figure 3.2(b). As |γ1| = |γ2| = 2, [R-STORESU] cannot be applied. When γ1 is processed, applying [R-STOREWU] to ℓ1 adds m→x and n→x to pt[γ1] and applying [R-LOAD] to ℓ2 yields pt(v) = x. Applying [R-DU] to the two self-loop edges [m] and [n] around γ1 gives rise to pt[γ1] = pt[γ1]. Applying [R-DU] to the two edges [m] and [n] from γ1 to γ2, we obtain pt[γ2] = {m→x, n→x}.

When γ2 is analysed, [R-STOREWU] is applied to each store. So the points-to relations in pt[γ2] are preserved in pt[γ2]. Then we add m→y generated at ℓ3 and m→z and n→z at ℓ4 to pt[γ2]. Finally, applying [R-DU] to the self-loop [m] on γ2 causes m→y and m→z to be added to pt[γ2]. Finally, we obtain pt[γ2] = {m→x, m→y, m→z, n→x} and pt[γ2] = pt[γ2] ∪ {n→z}.

□

Theorem 1 (Soundness). SELFS is sound if the pre-analysis used is.

Proof Sketch. A sound pre-analysis over-approximates the indirect def-use chains used for constructing the edges in a region graph Grg. Essentially, SELFS combines FI and FS to refine such pre-computed def-use chains flow-sensitively ([R-DU] and [R-INOUT]) by performing strong updates ([R-STORESU]).

Theorem 2 (Precision). Suppose FI is used as the pre-analysis, Then SELFS lies between FI and FS in terms of precision.

Proof Sketch. We can show that SELFS is no more precise than FS (with respect to each variable’s points-to set) by observing the following facts: (1) performing
the pre-analysis with FI gives rise to over-approximated indirect def-use chains (Theorem 1), (2) both analyses handle top-level pointers in exactly the same way except that SELFS does it sparsely in SSA ([R-ADDOF], [R-COPY] and [R-PHI]) and FS takes a data-flow approach ([S-ADDOF] and [S-COPY]), and (3) SELFS applies FS only to a region that contains one load or one store and FI to handle the remaining regions flow-insensitively. Thus, for every variable, the points-to set obtained by SELFS is no smaller than that obtained by FS.

To see that SELFS is no less precise than FI, we note that SELFS works by refining the points-to sets produced by FI (as the pre-analysis) through performing strong updates and maintaining inter-region flow-sensitivity.

Finally, some prior representative analyses are special instances of SELFS, with the following changes made to SELFS (mainly to region partitioning):

**FI in Figure 3.1(a):** All loads and stores are in one region (and top-level variables are not in SSA if they are to be analysed also flow-insensitively).

[35]: Each region contains one load or one store (same precision as FS).

**FS in Figure 3.1(b):** Each region contains one load or one store and each inter-region edge represents control flow, labeled by all address-taken variables.

[52]: Each store is in its own region if it can be strongly updated and all the other stores and all the loads are in another region (less precise than FS).

### 3.3 Instantiating the Selfs Analysis

SELFS is sound (Theorem 1) and can easily achieve a precision between FI and FS on an arbitrary region graph $\mathcal{G}_{rg}$ (Theorem 4). Ideally, we should use a region graph $\mathcal{G}_{rg}$ that allows SELFS to attach the precision of FS at the efficiency of FI.
In this section, we introduce a new unification-based approach that allows SELFS to preserve the precision of FS with respect to the reads from all variables, thus making it nearly as precise as FS in practice. We also discuss how to relax this so-called strict load-precision-preserving approach to tolerate some precision loss in future work in Section 3.6. Our focus is on demonstrating the generality and flexibility of SELFS in allowing efficiency and precision trade-offs to be made subject to region partitioning strategies used.

3.3.1 Load-Precision-Preserving Partitioning

As discussed in Section 3.2, SELFS degenerates into the sparse analysis [35] if SELFS operates on a region graph, denoted $G_{one}$, such that each of its regions contains one load or one store. In this important special case, SELFS is significantly faster than FS while achieving the same precision as FS, but can still be costly for large programs, especially when field-sensitivity is considered. By merging small regions into larger ones, SELFS can run faster at some possible precision loss.

We introduce a partitioning strategy that works by unifying two regions into a larger one successively, starting from any given region graph, say $G_{one}$. Some unification steps are applied online if they require the knowledge about whether a strong or weak update is performed at a store and some can otherwise be applied offline. Our rules are load-precision-preserving in the sense that every load behaves identically before and after each unification. Thus, for every load $\cdots = *q$, the points-to set of every target $a$ pointed to by $q$ remains unchanged.

For almost all practical purposes, making all the loads precise is as good as making SELFS as precise as FS. First of all, the points-to set of every top-level pointer remains unchanged (since it is overwritten from either an ADDR OF, a LOAD statement, or possibly via a sequence of COPY or PHI assignments). Thus,
the precision for reads of $q$ in $\cdots = q$ and $\cdots = *q$ is preserved. In addition, the alias information remains unchanged. This is because in our LLVM-like canonical representation, all aliases must be tested between top-level pointers. Similarly, all function pointers are reserved in the same way as they are all top-level.

However, some stores can be imprecise, but only when they are not read from later, as is the case of $*q = s$ illustrated in Figure 3.2(b) and revisited later in Example 3. Such stores are dead code and can thus be eliminated.

### 3.3.2 Unification

The following lemma gives a sufficient condition to make SELFS load-precision-preserving and motivates the development of our unification approach.

**Lemma 1.** Let $G_{rg}$ be a region graph with its two regions identified by $\gamma_i$ and $\gamma_j$. Let $G'_{rg}$ the resulting graph after $\gamma_i$ and $\gamma_j$ are unified (i.e., merged) into a new region $\gamma_{i,j}$. Let $L$ be the set of all regions in $G_{rg}$ such that each contains at least one load. Let $L'$ be similarly defined for $G'_{rg}$. Let $\pi : L \mapsto L'$ be defined such that $\pi(\gamma) = \gamma$ if $\gamma \notin \{\gamma_i, \gamma_j\}$ and $\pi(\gamma) = \gamma_{i,j}$ otherwise. If $\forall \gamma \in L : \text{pt}[\gamma] = \text{pt}[\pi(\gamma)]$ before and after the unification, then SELFS is load-precision-preserving.

**Proof.** No strong update can be performed in a region $\gamma$ that contains a load since that would imply $|\gamma| > 1$. For every region $\gamma \in L$, if $\text{pt}[\gamma] = \text{pt}[\pi(\gamma)]$, then $\text{pt}[\gamma][o] = \text{pt}[\pi(\gamma)][o]$ for every load $p = *q$ that appears in both $\gamma$ and $\pi(\gamma)$, where $o \in \text{pt}(q)$ ([R-LOAD]). So SELFS is load-precision-preserving.

This lemma is expensive to apply during the analysis. Guided by the basic idea behind, we have developed a conservative but simple unification approach. Each unification step operates on a small neighbourhood of the two regions being
unified. Our approach is promising as it can be relaxed to allow different efficiency and precision trade-offs to be made, as discussed in Section 3.6.

**Definition 2 (Region Types).** Given a region $\gamma$, $\tau(\gamma) = S$ if a strong update can be performed inside (implying that $\gamma$ contains a single store), $\tau(\gamma) = W$ if a weak update can be performed inside, and $\tau(\gamma) = L$ if $\gamma$ contains loads only.

When unifying a region $\gamma$ with another region in a region graph $G_{rg} = (N_{rg}, E_{rg})$, we can identify the potential points-to relations generated by $\gamma$ and the potential uses for the points-to relations generated by the two regions being unified directly from $G_{rg}$. Below we write $rsucc$ ($rpred$) to relate a region to its set of successors (predecessors) in a region graph.

- $GEN(\gamma) = \{ o \mid \gamma \xrightarrow{o} \gamma', \gamma' \in N_{rg} \}$ contains the address-taken variables potentially defined in $\gamma$ ([R-DU]), which implies $GEN(\gamma) = \emptyset$ if $\tau(\gamma) = L$.

- $USE(\gamma) = \{ \gamma' \mid \gamma' \in rsucc(\gamma) \} \cup \{ \gamma \mid \gamma \text{ contains a load} \}$ gives the set of potential uses for the points-to relations generated by $\gamma$ and the region to be unified together. Note that $rsucc(\gamma) \supseteq \{ \gamma \}$ if $\gamma$ contains a store that produces values used by some other stores or some loads also contained in $\gamma$ (Figure 3.3).

- $PRD(\gamma) = rpred(\gamma)$ gives the set of all potential defs for the points-to relations used in $\gamma$.

When merging two regions, we need to reason about the value flows for the address-taken variables potentially defined inside these two regions.

**Definition 3 (Value-Flow Reachability).** Let $\gamma_i$ and $\gamma_j$ be two regions in $G_{rg} = (N_{rg}, E_{rg})$. We say that $\gamma_j$ is $o$-reachable from $\gamma_i$, where $o \in A$, and write $\gamma_i \xrightarrow{o} \gamma_j$ if there is either (1) an edge $\gamma_i \xrightarrow{o} \gamma_j \in E_{rg}$ (directly reachable) or (2) a path $\gamma_i \xrightarrow{o}$
\( \gamma_1, \ldots, \gamma_n \xrightarrow{o} \gamma_j \) in \( G_{rg} \) (indirectly reachable) via one or more regions, \( \gamma_1, \ldots, \gamma_n \), where \( \tau(\gamma_k) = W \), for weakly updating \( o \).

When two regions \( \gamma_i \) and \( \gamma_j \) are unified, all loads and stores in the resulting region are resolved flow-insensitively (since \( |\gamma_i \cup \gamma_j| > 1 \)), even though a strong update is possible in either region before. The points-to relations flowing into both \( \gamma_i \) and \( \gamma_j \) from \( PRD(\gamma_i) \) and \( PRD(\gamma_j) \) are merged, preserved and propagated together with the points-to relations generated inside \( \gamma_i \) and \( \gamma_j \) to their uses in \( USE(\gamma_i) \) and \( USE(\gamma_j) \). In order to preserve the precision for loads, we can ensure conservatively that the same propagation happens before and after each unification (for loads). The presence of a strong update in \( \gamma_i \) or \( \gamma_j \) can be unification-preventing only when the killed values in \( \gamma_i \) or \( \gamma_j \) cannot already reach their uses in \( USE(\gamma_i) \) and \( USE(\gamma_j) \) before the unification.

Let us introduce some notational shorthand, where \( R, R', O \subseteq N_{rg} \) and \( O \subseteq A \):

\[ R \xrightarrow{O} R' \xrightarrow{=} \forall o \in O : \forall \gamma \times \gamma' \in R \times R' : \gamma \xrightarrow{o} \gamma' \]

**Theorem 3** (Load-Precision-Preserving Unification). Unifying \( \gamma_i \) and \( \gamma_j \) will make SELFS load-precision-preserving if all the following conditions hold:

**C1** \{\( \gamma_i \)\} \xrightarrow{\text{GEN}(\gamma_i)} \text{USE}(\gamma_j);

**C2** \( \text{PRD}(\gamma_i) \xrightarrow{\text{GEN}(\gamma_i)} \text{USE}(\gamma_i) \cup \text{USE}(\gamma_j) \);

**C3** \{\( \gamma_j \)\} \xrightarrow{\text{GEN}(\gamma_j)} \text{USE}(\gamma_i); and

**C4** \( \text{PRD}(\gamma_j) \xrightarrow{\text{GEN}(\gamma_j)} \text{USE}(\gamma_i) \cup \text{USE}(\gamma_j) \).

**Proof Sketch.** By unifying \( \gamma_i \) and \( \gamma_j \), only the points-to relations reaching the regions in \( \text{USE}(\gamma_i) \cup \text{USE}(\gamma_j) \) are affected. As is standard, SELFS is monotonic, implying that no strong update at a store is possible after the store has been
weakly updated. Therefore, any indirect reachable path (Definition 3), once established, will remain unchanged. For reasons of symmetry, let us consider C1 and C2 only. C1 says that whatever $\gamma_i$ generates (along its def-use edges) must be used not only by $USE(\gamma_i)$ (by construction) but also by $USE(\gamma_j)$. C2 says that even if some values are killed in $\gamma_i$ due to a strong update, the killed values will still reach both $USE(\gamma_i)$ and $USE(\gamma_j)$ via a different path (without going through $\gamma_i$), rendering the values non-killable (effectively).

Let $\pi$ be defined in Lemma 1. If C1 – C4 hold, then $\forall \gamma \in USE(\gamma_i) \cup USE(\gamma_j)$ : $pt[\pi(\gamma)] = pt[\pi(\gamma)]$. By Lemma 1, SELFS is load-precision-preserving.

**Example 3.** Let us apply Theorem 3 to the example given in Figure 3.3, assuming initially that each statement is in its own region: $\gamma_1 = \{\ell_1\}$, $\gamma_2 = \{\ell_2\}$, $\gamma_3 = \{\ell_3\}$ and $\gamma_4 = \{\ell_4\}$. Let us try to unify $\gamma_1$ and $\gamma_2$. According to the region graph given in Figure 3.3(a), we have $GEN(\gamma_1) = \{m,n\}$, $GEN(\gamma_2) = \emptyset$. $USE(\gamma_1) = \{\gamma_2, \gamma_3, \gamma_4\}$, $USE(\gamma_2) = \{\gamma_2\}$, $PRD(\gamma_1) = \emptyset$ and $PRD(\gamma_2) = \{\gamma_1\}$. By Theorem 3, C1 – C4 are satisfied. So $\gamma_1$ and $\gamma_2$ are unifiable. We can also choose to unify $\gamma_3$ and $\gamma_4$ instead. Then $GEN(\gamma_3) = \{m\}$, $GEN(\gamma_4) = \emptyset$. $USE(\gamma_3) = \{\gamma_4\}$, $USE(\gamma_4) = \emptyset$, $PRD(\gamma_3) = \{\gamma_1\}$ and $PRD(\gamma_4) = \{\gamma_1, \gamma_3\}$. Note that $GEN(\gamma_4) = \emptyset$ because there are no outgoing def-use chains from $\ell_4$. Again, C1 – C4 are satisfied, making $\gamma_3$ and $\gamma_4$ unifiable. By proceeding in either order, we will obtain the region graph shown in Figure 3.3(b).

Note that unifying $\gamma_3$ and $\gamma_4$ makes SELFS lose the precision at $\ell_3$ as explained earlier. However, in this example, both $\ell_3$ and $\ell_4$ are dead code. If we add a load $\ell_5 : w = *q$ immediately after $\ell_3$, which is in a new region $\gamma_5 = \{\ell_5\}$, there will be a new indirect def-use $\ell_3 \xrightarrow{m} \ell_5$ in Figure 3.3(a). In this case, $\gamma_3$ and $\gamma_4$ are no longer unifiable since $USE(\gamma_3) = \{\gamma_4, \gamma_5\}$. By treating $\gamma_3$ as $\gamma_i$ in Theorem 3, C2 is violated since there is only one path from $\gamma_1$ to $\gamma_5$: $\gamma_1 \xrightarrow{m} \gamma_3 \xrightarrow{m} \gamma_5$, where a
Figure 3.5: Some possible unification sequences illustrated for a part of a region graph, by assuming that all indirect-use edges are related to one single address-taken variable, $o$. The type of a region is identified by $S$, $W$ or $L$ (Definition 2).

A strong update is performed on $m$ in $\gamma_3$. So $\gamma_5$ is not $m$-reachable from $\gamma_1$. In fact, merging $\gamma_3$ and $\gamma_4$ will cause the load $\ell_5: w = *q$ to lose precision since the store at $\ell_3$ will only be weakly updated afterwards. $\square$

Figure 3.5 illustrates our unification approach further, by assuming that all indirect def-use chains are related to one address-taken variable, $o$. In Figure 3.5(d), if $W_3$ is $S_3$ (with a strong update to $o$ inside), then the unification cannot be performed as $L_{45}$ is not $o$-reachable from $S_1$ (the predecessor of $S_2$ and $S_3$). Otherwise, $L_{45}$ may receive spurious points-to relations propagated from $S_1$. In Figure 3.5(h), $S_1$ and $W_{2345}$ cannot be unified further because the predecessors of $S_1$ reach $W_{2345}$ via only $S_1$ (where a strong update to $o$ is performed).
3.4 Evaluation

We demonstrate the effectiveness of SELFS under our unification-based region partitioning strategy. The baseline is a state-of-the-art sparse yet precision-preserving version [35], denoted SFS, of FS given in Figure 3.1(b). We have selected 14 large C programs (totalling 672 KLOC) from SPEC CPU2000 and CPU2006, with their characteristics given in Table 3.1. Our platform is a 2.00GHz Intel Xeon 32-core CPU running Ubuntu Linux with 64GB memory.

3.4.1 Methodology

As discussed in Section 3.2, SFS works on a region graph such that each region contains one single load or store. To apply our load-precision-preserving partitioning in SELFS, we start with a region graph such that each load or store is in its own region. We then apply our unification-based approach to form larger regions. As a result, SELFS is load-precision-preserving (Theorem 3), resulting in the same precision for alias queries as SFS (among others). We repeat a process of picking a region randomly and trying to unify it with one of its predecessors, successors, and siblings in that order until no more unification is possible.

For efficiency reasons, we verify the four conditions in Theorem 3 during the SELFS analysis by restricting ourselves to the o-reachable paths (Definition 3) such that each of its intermediate nodes is one of the two regions to be unified. As a result, starting with Figure 3.5(a), we will accept Figures 3.5(c) and (d) but reject Figure 3.5(b). Finally, some unification steps are performed offline rather than online during the analysis if they do not require the knowledge about whether a store in a region (containing that store only) can be strongly updated or not.
## Chapter 3. Region-based Selective Flow-Sensitive Pointer Analysis

### Table 3.1: Program characteristics.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>#Statement</th>
<th>#Ptrs</th>
<th>#Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KLOC</td>
<td>AddrOf</td>
<td>Copy</td>
<td>Load</td>
</tr>
<tr>
<td>ammp</td>
<td>13.4</td>
<td>702</td>
<td>6875</td>
<td>925</td>
</tr>
<tr>
<td>crafty</td>
<td>21.2</td>
<td>1632</td>
<td>9603</td>
<td>1011</td>
</tr>
<tr>
<td>gcc</td>
<td>230.4</td>
<td>8934</td>
<td>135332</td>
<td>32498</td>
</tr>
<tr>
<td>h264ref</td>
<td>51.6</td>
<td>1829</td>
<td>27845</td>
<td>8324</td>
</tr>
<tr>
<td>hmmer</td>
<td>36</td>
<td>1195</td>
<td>7635</td>
<td>2083</td>
</tr>
<tr>
<td>mesa</td>
<td>61.3</td>
<td>2691</td>
<td>45447</td>
<td>6112</td>
</tr>
<tr>
<td>mile</td>
<td>15</td>
<td>1104</td>
<td>8591</td>
<td>1138</td>
</tr>
<tr>
<td>parser</td>
<td>11.4</td>
<td>1417</td>
<td>6045</td>
<td>1626</td>
</tr>
<tr>
<td>perlmbk</td>
<td>87.1</td>
<td>4366</td>
<td>39602</td>
<td>17096</td>
</tr>
<tr>
<td>sjeng</td>
<td>13.9</td>
<td>926</td>
<td>5579</td>
<td>848</td>
</tr>
<tr>
<td>sphinx3</td>
<td>25.1</td>
<td>1898</td>
<td>10169</td>
<td>2482</td>
</tr>
<tr>
<td>twolf</td>
<td>20.5</td>
<td>1371</td>
<td>14390</td>
<td>7526</td>
</tr>
<tr>
<td>vortex</td>
<td>67.3</td>
<td>6636</td>
<td>20408</td>
<td>6577</td>
</tr>
<tr>
<td>vpr</td>
<td>17.8</td>
<td>1195</td>
<td>5703</td>
<td>2222</td>
</tr>
</tbody>
</table>

Program Size #Statement #Ptrs #Object
KLOC AddrOf Copy Load Store Total #Ptrs #Object
Glob. Heap Stk Func Total
ammp 13.4 702 6875 ... 67.3 6636 20408 6577 3185 36806 104218 739 29 1864 961 3593
vpr 17.8 1195 5703 ... 17.8 1195 5703 2222 310 9430 28405 101 6 101 303 511

Table 3.1: Program characteristics.
Table 3.2: Comparing SFS and SELFS.
3.4.2 Implementation

We have implemented SELFS in LLVM (version 3.3). The source files of each benchmark are compiled into bit-code files using clang and then linked together using llvm-link, with mem2reg being applied to promote memory into registers. We use FI, i.e., Andersen’s analysis (using the constraint resolution techniques from [63,68]) as pre-analysis for building indirect def-use chains [35,78,79].

SELFS is field-sensitive with the offset-based modelling illustrated in Section 2.2: each field of a struct is treated as a separate object, but arrays are considered monolithic. Positive weight cycles (PWCs) that arise from processing fields of struct objects are detected and collapsed [61]. Distinct allocation sites are modeled by distinct abstract objects as in [35,78,79].

We have implemented SFS differently from that in [35] by building program’s call graph on the fly and storing the points-to sets using sparse bit vectors, for both SFS and SELFS, which are implemented in LLVM 3.3. In [35], implemented in LLVM 2.5, a program’s call graph is pre-computed and points-to sets are represented using binary decision diagrams (BDDs).

3.4.3 Results and Analysis

As shown in Table 3.2, SELFS is 2.13X faster than SFS on average under our load-precision-preserving partitioning strategy while maintaining the same precision for reads, i.e., for all alias queries in all the functions from a benchmark. The best speedups are achieved at h264ref (7.45X) and mesa (6.08X). Note that SELFS can go faster, approaching eventually the efficiency of FI, if increasingly larger regions are used. The analysis time of a benchmark, which excludes the time spent on pre-analysis, is the average of three runs.

Let us look at the results of the two analyses in more detail. SFS spends 2465.39
Figure 3.6: Percentage distributions of SELF's analysis time in a benchmark over “FSAnal” (the time for its flow-sensitive analysis), “FIAnal” (the time for its flow-insensitive analysis), “Partitioning” (the time on region partitioning), and “Propagation” (the time for propagating points-to facts across the indirect def-use chains in the region graph).

seconds to analyse all the benchmarks but SELF finishes in 782.37 seconds. In Columns 6 – 10, we see the number of regions of each type as well as the average and maximum region sizes. The average region sizes range from 1.40 (mesa) to 3.51 (twolf). In gcc, perlbench and vortex, each largest region ends up with 150+ loads or stores being resolved flow-insensitively, with the precision for all reads being preserved. This shows the great potential promised by SELF in achieving load-precision-preserving flow-sensitive analysis in a region-based manner. With better tuned unification rules, better speedups are expected.

For relatively small program, such as ammp, hmer, milc and sjeng, SELF yields little or no performance benefits due to the overhead on region partitioning, as illustrated in Figure 3.6. However, for relatively larger ones, which contain more objects and more dense def-use chains to be dealt with flow-sensitively (Table 3.1), such as gcc, mesa, perlbench and vortex, SELF is beneficial. The best speedups are
<table>
<thead>
<tr>
<th>Program</th>
<th>Propagation Times (Secs)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SFS</td>
<td>SELF$^*$</td>
</tr>
<tr>
<td>ammp</td>
<td>0.13</td>
<td>0.05</td>
</tr>
<tr>
<td>crafty</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>gcc</td>
<td>741.42</td>
<td>270.32</td>
</tr>
<tr>
<td>h264ref</td>
<td>25.20</td>
<td>1.13</td>
</tr>
<tr>
<td>hmmer</td>
<td>0.11</td>
<td>0.09</td>
</tr>
<tr>
<td>mesa</td>
<td>719.97</td>
<td>70.31</td>
</tr>
<tr>
<td>milc</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>parser</td>
<td>2.65</td>
<td>0.43</td>
</tr>
<tr>
<td>perlbmk</td>
<td>285.90</td>
<td>68.96</td>
</tr>
<tr>
<td>sjeng</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>sphinx3</td>
<td>0.53</td>
<td>0.08</td>
</tr>
<tr>
<td>twolf</td>
<td>0.42</td>
<td>0.23</td>
</tr>
<tr>
<td>vortex</td>
<td>77.33</td>
<td>18.47</td>
</tr>
<tr>
<td>vpr</td>
<td>0.05</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 3.3: Propagation times of SFS and SELF$^*$ for analysing the address-taken variables across their indirect def-use chains.

observed at h264ref (7.45X) and mesa (6.08X), because the times for propagating points-to facts across indirect def-use chains have been significantly reduced by 22.3X and 10.24X, respectively (Table 3.3).

### 3.5 Related Work

**Sparse Pointer Analysis**  Sparse analysis, a recent improvement over the classic iterative data-flow approach, can achieve flow-sensitivity more efficiently by propagating points-to facts sparsely across pre-computed def-use chains [34,35,60,80,96]. Initially, sparsity was experimented with in [39,40] on a Sparse Evaluation Graph [12,66], a refined CFG with irrelevant nodes being removed. On various SSA
form representations (e.g., factored SSA [13], HSSA [14] and partial SSA [50]), further progress was made later. The def-use chains for top-level pointers, once put in SSA, can be explicitly and precisely identified, giving rise to a so-called semi-sparse flow-sensitive analysis [34]. Recently, the idea of staged analysis [22,35] that uses pre-computed points-to information to bootstrap a later more precise analysis has been leveraged to make pointer analysis full-sparse for both top-level and address-taken variables [35,60,96].

**Hybrid Analysis**  The aim of hybrid-sensitive pointer analysis is to find a right balance between efficiency and precision. As a well-known example for mixing different flow-insensitive analyses, one-level approach [17] lies between Steensgaard’s and Andersen’s analyses (in terms of precision) by not applying its unification process to top-level pointers. For context-sensitivity, hybrid analysis has been used in Java to pick up the benefits of both call-site sensitivity and object sensitivity [46]. In [52], strong updates are performed for only singleton objects on top of a flow- and field-insensitive Andersen’s analysis. Earlier [29,77], how to adjust the analysis precision according to clients’ needs is discussed.

**Region-based Analysis**  Region-based analysis, which partitions a program into different compilation units, was commonly used to explore locality and gain more opportunities for compiler optimisations, such as inlining [31,84], partial dead code elimination [92], and just-in-time optimisation [76]. In [97,98], programs are decomposed into different regions according to the alias relations and each region is solved independently. Same partition strategy was also adopted by [45] to speed up a flow- and context-sensitive pointer analysis.
3.6 Summary

In this chapter, we introduced SELFS, which is flexible for making efficiency and precision trade-offs subject to region partitioning strategies used. We have implemented SELFS in LLVM and demonstrated its effectiveness with a unification-based region partitioning strategy, by comparing it with a state-of-the-art flow-sensitive analysis. In addition, our unification-based approach is interesting in its own right as it leads to a particular analysis that is as precise as FS for almost all practical purposes.

In future work, we will develop a range of partitioning strategies to relax our unification-based approach. There is a lot of freedom in performing a precision-loss partitioning (Theorems 1 and 4). In order to be tunable and client-specific, a relaxed strategy can be designed along the following directions (among others). First, the user can identify parts of a program that require flow-sensitive analysis based on client analyses' needs (e.g., hot functions and major changes made during software development). Second, the user may request customised flow-sensitivity for some selected variables. Third, some stores can always be weakly updated (to enable more offline unification steps, for example). Finally, our unification approach can be relaxed to enable more regions to be merged without having to preserve the precision for all the loads.
Chapter 4

SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

4.1 Overview

As reported recently [35,45,96], insensitive analysis can be leveraged to bootstrap sensitive analysis, thereby leading to significant improvements in scalability and precision. In particular, it is shown by LevPA [96] that flow-sensitive and context-sensitive (FSCS) analysis becomes substantially more scalable when performed on full-sparse memory SSA, level by level (in order of their decreasing points-to levels), with each level being analyzed with an inclusion-based flow-insensitive algorithm. However, little progress [58,89] has been made when path-sensitivity is also considered. Exploiting recent advances in FSCS pointer analysis, we describe a SPAS (Scalable Path-Sensitive) framework that explores intra-procedural path correlations for C programs on top of the state-of-the-art LevPA (the FSCS version
of SPAS). SPAS obtains all points-to sets with non-decreasing precision by adding small analysis overhead in both time and space, as validated using SPEC CPU2000 and SPEC2006.

Equipped with path-sensitivity, a FSCS pointer analysis is equally as or more precise, as illustrated in Figure 4.1. With such path-sensitive precision, the quality of many software tools and techniques in program optimization, analysis and verification can be significantly improved. Examples include bug hunting [51], memory leak analysis [89] and software vulnerability detection [58,90].

A major hindrance to path-sensitive pointer analysis is the lack of scalability. We tackle it by tracking program paths using Binary Decision Diagrams (BDDs). Berndl et al. [5] proposed to use BDDs to encode points-to relations and also studied the impact of BDD variable ordering on analysis performance. Later, BDDs were also used to encode transfer functions [101] and contexts [85,96] in context-sensitive analysis. In [90], Xie and Aiken discussed to use BDDs to represent program paths and simplify paths heuristically in SATURN, a SAT-based bug detection tool. In our Spas pointer analyzer, BDDs are used to encode paths (and contexts) on full-sparse SSA using a standard BDD library. This enables the pointers to be resolved using a guarded inclusion-based insensitive analysis on full-sparse memory SSA, where the guards are contexts and program paths.

SPAS is the first pointer analyzer that support path-sensitivity on full-sparse memory SSA. In some bug-hunting tools like Prefix [8], path-sensitivity was exploited to look for bugs in some selected paths. A recent sound and complete generalization for SATURN [19] can compute various program properties using a SAT solver with even interprocedural path-sensitivity. However, it is not suited for whole-program points-to resolution. SPAS and bug analysis aim to achieve different goals. Some bug detection tools rule out some infeasible paths based on branch
Figure 4.1: Effects of path-sensitivity on indirect updates at a store *x=q in FSCS analysis.
conditions to reduce false positives but SPAS presently does not. However, SPAS can already provide more precise points-to information to make these tools more effective. Finally, Gutzmann et al. [28] discussed how to filter out spurious points-to relations flowing out of a branch node but their approach neither captures path correlation as SPAS does nor rules out infeasible paths.

4.2 The Basic Idea

SPAS analyzes all features of C by considering four types of assignments: \( x = y \) (copy), \( x = \&y \) (address), \( \ast x = y \) (store) and \( x = \ast y \) (load). SPAS is field-insensitive for arrays (by not distinguishing array elements) but field-sensitive for structs (by flattening and replacing them with separate variables, one for each field). SPAS names abstraction heap objects by their allocation sites.

The following sets and functions are used in some definitions given below.

- \( \mathcal{O} \): set of abstract memory locations representing the variables in a program.
- \( \mathcal{L} : \mathcal{O} \rightarrow \{0, \ldots, L\} \): a level map giving each variable a points-to level. If \( q \) may be modified by operations on \( p \) (possibly indirectly), then \( \mathcal{L}(p) \geq \mathcal{L}(q) \).
- \( C \): set of contexts represented as Boolean expressions over the set \( \mathcal{O} \times \mathcal{O} \) of points-to relations. The notation \( p \rightarrow o \) means that \( p \) may point to \( o \).
- \( \mathcal{D} \): set of paths represented as Boolean conditions over the set of decision variables (encoding the branches in a CFG to be introduced in Section 4.4.1).
- \( \mathcal{C} : C \times \mathcal{D} \): set of combined context and path conditions used to specify under which condition in \( C \) a pointer may point to a memory object in \( \mathcal{O} \).

**Definition 4** (Formal-Ins). Given a method \( m \), \( \mathcal{F}_m \subseteq \mathcal{O} \) denotes the set of its formal-ins, i.e., its formal parameters and non-local variables accessed in \( m \).
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

**Definition 5** (Formal-Outs). Given a method $m$, $F^m_{\text{out}} \subseteq O$ denotes the set of its formal-outs, i.e., its return parameter (a special local variable of $m$ containing its return value) and non-local variables that may be modified in $m$.

In flow-sensitive analysis, we speak of the points-to sets of a pointer $p$ at program points, which are identified by the versions of $p$ at those points in SSA.

**Definition 6** (Points-to Sets). The points-to set of a pointer $p$ at a program point, $\text{PtrSet}(p) \subseteq O$, is a set of locations in $O$ possibly pointed to by $p$.

SPAS achieves context-sensitivity by traversing the call graph of a program bidirectionally. During a bottom-up traversal, the points-to sets of a pointer $p$ at a point may be related to those of some formal-ins in terms of points-to maps (Definition 7). During a top-down traversal, $p$ is resolved once the points-to sets of all its dependent formal-ins have been found (Definition 6).

**Definition 7** (Points-to Maps). The points-to map of a pointer $p$ at a program point in a method $m$ is given by:

$$\text{PtrMap}(p) = (\text{Loc}(p), \text{Dep}(p))$$

(4.1)

where $\text{Loc}(p) \subseteq O \times C$ contains each tuple $(v, C_v, P_v)$ such that $p$ may point to $v$ in context $C_v$ along path $P_v$ within method $m$ and $\text{Dep}(p) \subseteq F^m_{\text{in}} \times C$ contains each tuple $(f, C_f, P_f)$ such that $p$ may point to what formal-in $f$ points to in context $C_f$ along path $P_f$ within method $m$.

SPAS is intraprocedurally path-sensitive. Thus, the transfer functions ($\text{MOD}$ and $\text{USE}$) of a formal-out in a method $m$ are predicated by $m$’s calling contexts (without $m$’s path conditions). Similarly, the path conditions in their specified side effects are also ignored (and hence, the $^*$’s). To support strong updates, we distinguish MAY-DEFs and MUST-DEFs at stores.
Definition 8 (MOD). The transfer function $\text{MOD}$ of a formal-out $f_{\text{out}} \in F_{\text{out}}^m$ in a method $m$ describes its interprocedural modification side effect:

$$\text{MOD}(m, f_{\text{out}}) = (\text{Loc}(f_{\text{out}}), \text{Dep}(f_{\text{out}}), C_{f_{\text{out}}}^{\text{may}}, C_{f_{\text{out}}}^{\text{must}})$$ (4.2)

indicating that $f_{\text{out}}$ may be modified, i.e., MAY-DEF’ed in context $C_{f_{\text{out}}}^{\text{may}} \in C$ to point to either (a) $v$ for each $(v, C_v, \ast) \in \text{Loc}(f_{\text{out}})$ when $C_v$ holds or (b) whatever $f$ points to for each $(f, C_f, \ast) \in \text{Dep}(f_{\text{out}})$ when $C_f$ holds. If $C_{f_{\text{out}}}^{\text{must}} \in C$ also holds, then the MAY-DEF is actually a MUST-DEF.

Definition 9 (USE). The transfer function $\text{USE}$ of a formal-in $f_{\text{in}} \in F_{\text{in}}^m$ in a method $m$ describes its MAY-USE, i.e., interprocedural read side effect:

$$\text{USE}(m, f_{\text{in}}) = (\text{PtrSet}(f_{\text{in}}), C_{f_{\text{in}}})$$ (4.3)

indicating that what is pointed to by $f_{\text{in}}$ may be read in context $C_{f_{\text{in}}} \in C$. (If $m$ is a formal parameter, it does not need a USE function as $m$ is local.)

The basic idea behind SPAS is simple. The pointers are resolved in order of their decreasing points-to levels by maintaining the invariant stated below.

Property 1 (Level-Wise Invariant). Just before level $\ell$ is analyzed, (a) all (direct and indirect) accesses to the pointers at higher levels are in SSA, with the indirect accesses via pointer dereferencing and calls being expressed using $\mu$ (MAY-USE) and $\chi$ (MAY/MUST-DEF) operations [14], (b) all pointers at higher levels have been soundly resolved, and (c) all indirect accesses made by dereferencing the pointers at level $\ell$ are exposed using $\mu$ and $\chi$ operations.

The analysis performed at level $\ell$ is to ensure that this invariant holds at the beginning of $\ell - 1$. This is done by traversing the call graph of a program first bottom-up and then top-down iteratively. During bottom-up analysis, SPAS analyzes each method $m$ by (B1) building its SSA (doable as Property 1(c) holds for
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

\( \ell \) and (B2) computing the points-to maps for its pointers at \( \ell \) (Definition 7). Prior to (B1), SPAS inserts \( \mu \) and \( \chi \) operations for each of its call sites, \( c \), to expose the MAY-USEs and MAY/MUST-DEFs made by \( c \)'s callees to every pointer at \( \ell \). This is done by applying the callees’ transfer functions at call site \( c \). After (B2) is done, the transfer functions of method \( m \) are computed. During top-down analysis, SPAS (T1) resolves the points-to sets of the pointers at \( \ell \) by propagating the dependent points-to sets to formal-ins (Definition 6) and (T2) annotates the dereferences to these pointers with \( \mu \) and \( \chi \) operations.

4.3 A Motivating Example

We describe how SPAS improves FSCS pointer analysis by refining Figure 4.1(d) into Figure 4.1(c). This program may look complex but it appears to be one of the smallest examples that we can come up with in order to illustrate all key aspects of SPAS. The variables are partitioned into three levels: \( \{a, x\} \) at level 2, \( \{q, f, g\} \) at level 1 and \( \{v, w, z\} \) at level 0. We examine the top two levels only.

Level 2 The pointers \( a \) and \( x \) are considered. The input is Figure 4.1(a), for which Property 1 holds trivially at this level. The output is given in Figure 4.2(a).

- **Bottom-Up** When \( foo \) is analyzed, all accesses to \( x \) are first put in SSA. During points-to resolution, the points-to maps for its three definitions are found. In particular, \( x_0 \) is recorded to point to what formal-in \( x \) points to. By analyzing the \( \phi \) node for \( x \) path-sensitively, \( x_2 \) is found to point to \( g \) in context true (i.e., any context) along path \( P_1 \) and what \( x \) points to in context true along \( \neg P_1 \). When \( main \) is analyzed, the points-to map of \( a_0 \) is found.

- **Top-Down** In \( main \), \( a_0 \) has been resolved locally. Binding \( a_0 \) with formal-in
```
int *q, v, w, z;
void main() { 
    int **a, *f = &z;
    a[0] = &f; q = &v;
    foo(a[0]);
}

void foo(int **x) {
    x[0] = x; // ... :P1); (w; true; P1)g; ∅)
    PtrSet(f[0]) = fzg
    PtrSet(q[0]) = fvg
}
```

(a) After level 2 is analyzed

(b) After level 1 is analyzed

Figure 4.2: Level-wise SSA construction and path-sensitive pointer analysis for Figure 4.1(a).
x at the call site to foo reveals that $\text{PtrSet}(x_0) = \{ f \}$. When foo is processed, the MAY-DEF to f (g) via $*x_2$ is exposed by a $\chi$ operation, where context condition $x \rightarrow f$ (true) indicates that the MAY-DEF occurs when formal parameter $x$ points to f (in any context). Like LevPA, SPAS uses points-to relations holding at a call site to represent and distinguish calling contexts.

**Level 1** The pointers q, f and g are considered. The input is Figure 4.2(a), for which Property 1 holds at this level. The output is given in Figure 4.2(b).

- **Bottom-Up** When foo is analyzed, all accesses to the three pointers (including the two MAY-DEFs) are first put in SSA. Like x, $q_0 = q$ and $f_0 = f$ are inserted for the formal-ins q and f. Here, q is a global and f is an invisible [49] accessed via pointer dereferencing. The points-to resolution for q is done similarly as x with the points-to maps obtained as shown. We now consider how store $*x_2 = q_2$ is analyzed together with its two MAY-DEFs to resolve $f_1$ and $g_1$. By capturing path correlation, SPAS deduces that $f_1$ points to whatever formal-in q does along path $\neg P_1$ and whatever formal-in f does along $P_1$ and that $g_1$ points to z along $\neg P_1$ and w along $P_1$. The transfer functions relevant for computing the side effects of the call to foo are:

$$
\begin{align*}
\text{MOD}(\text{foo}, f) &= (\{ \text{Loc}(f_1), \text{Dep}(f_1), x \rightarrow f, \text{false} \}) \\
\text{USE}(\text{foo}, q) &= (\text{PtrSet}(q_0), \text{true})
\end{align*}
$$

When main is analyzed, a MAY-DEF for f is added since context $C_f^{\text{may}} = x \rightarrow f$ in $\text{MOD}(\text{foo}, f)$, once mapped to $a \rightarrow f$ at the call site, holds. However, this is not a MUST-DEF since $C_f^{\text{must}} = \text{false}$. In addition, a MAY-USE for q is added. Then the SSA form is built. Finally, by performing the points-to resolution for f and q with the modification side effects of foo on f being accounted for, we obtain their points-to maps as shown.
• **Top-Down** When `main` is analyzed, its pointers at this level are already resolved. Propagating the points-to sets of \( q_1 \) and \( f_0 \) at the call site to `foo`, we find \( \text{PtrSet}(f_0) = \{ z \} \) and \( \text{PtrSet}(q_0) = \{ v \} \) in `foo`. When `foo` is processed next, all its pointers at this level can be resolved by Definition 7. As a result, the points-to relations for \( f_1 \) and \( g_1 \) are obtained as shown in Figure 4.1(c).

SPAS obtains such improved precision with a slight increase in analysis overhead. There are several reasons for SPAS to achieve this level of scalability:

**Program Paths Manipulated as BDDs** Like contexts [96], program paths are also represented and operated on using BDDs in a compact and canonical fashion, resulting in fast operations on program paths.

**Full-Sparse SSA (at All Points-to Levels)** As in LevPA [96], pointers are resolved, level by level, in order of their decreasing points-to levels. At the same time, the full-sparse SSA form is being built incrementally. The points-to relations of a pointer at a particular level cannot be propagated to lower-level pointers unless it has fully been resolved. Thus, the number of repropagations is reduced, leading to faster convergence for the points-to resolution.

**Flow-Insensitive Points-to Resolution** SSA is ideal for enabling sparse analysis because it makes def-use information explicit. Like contexts [96], program paths are also used to guard what points-to information can be propagated across an pointer assignment. Thus, our path-sensitive pointer analysis is sped up with a guarded inclusion-based flow-insensitive pointer analysis.
4.4 The Spas Framework

SPAS is a summary-based FSCS pointer analyzer with intraprocedural path-sensitivity being supported. In particular, SPAS builds $MOD$ and $USE$ functions for each method and applies them to all its calling contexts.

Section 4.4.1 discusses how to encode program paths using BDDs. Section 4.4.2 examines the $\chi$ and $\mu$ operators added to the classic SSA form. To ease understanding, we introduce SPAS in two stages. In Section 4.4.3, we focus on capturing path correlation without performing strong updates. In Section 4.4.4, we discuss briefly but precisely how to extend it to perform path-sensitive strong updates.

4.4.1 Encoding Program Paths as BDDs

SPAS does not presently distinguish the paths inside a loop-induced cycle but can analyze the first few iterations of a loop path-sensitively via loop peeling.

All branch nodes are assumed to be binary. We use decision variables to encode branch nodes to express program paths. The edges and blocks in a CFG (with cycles collapsed) are associated with paths as follows. The path for the incoming edge of the entry block is initialized to $\mathbf{true}$ (representing the set of all paths). Let $B$ be a block with $n$ incoming edges associated with paths $P_1, \ldots, P_n$. The path for $B$ is $P_1 \lor \cdots \lor P_n$. If $B$ is a branch node encoded with decision variable $Q$, the paths for its two outgoing edges are $(P_1 \lor \cdots \lor P_n) \land Q$ and $(P_1 \lor \cdots \lor P_n) \land \neg Q$, respectively. Otherwise, its unique outgoing edge is $P_1 \lor \cdots \lor P_n$.

Our BDD encoding has three advantages. First, the number of BDD variables used is kept to a minimum. Second, it plays up the strengths of BDDs by exposing opportunities for path redundancy elimination. Third, the paths combined at a join node are effectively simplified (e.g., with $P_1 \lor \neg P_1$ being reduced into $\mathbf{true}$),
resulting in fast propagation of path conditions during points-to resolution.

4.4.2 Extended SSA Form

The classic SSA representation [15] is mainly useful for scalars without aliases. Following [96], we extend the classic SSA form by using the $\mu$ and $\chi$ operators [14] to make explicit all potential uses and definitions at loads/stores and call sites, as shown in Figure 4.2. A load or call site is annotated with a $\mu(v, C_v, P_v)$ operation to indicate a MAY-USE of $v$ in context $C_v$ along path $P_v$. A store or call site is annotated with a $v = \chi(v, C_v, P_v, M_v)$ operation to indicate a MAY-DEF (MUST-DEF) of $v$ in context $C_v$ along path $P_v$ if $M_v$ is MAY (MUST).

4.4.3 Capturing Path Correlation

This section focuses on capturing path correlation without strong updates. We consider functions with return statements in Section 4.4.4. As we do not distinguish MAY-DEFs and MUST-DEFs for now, the last entry $C_{\text{must}}$ in a MOD function (Definition 8) is ignored and the last entry in a $\chi$ operation is always a MAY. For Figure 4.2, all points-to maps are unchanged except for $f_1$ and $g_1$ in $\text{foo}$:

\[
\text{PtrMap}(f_1) = (\emptyset, \{(q, x \rightarrow f, \neg P_1), (f, x \rightarrow f, \text{true})\})
\]
\[
\text{PtrMap}(g_1) = (\{(z, \text{true}, \text{true}), (w, \text{true}, P_1)\}, \emptyset)
\]

Without path-sensitive strong updates, the old points-to sets (i.e., $\text{PtrMap}(f_0)$ for $f_1$ and $\text{PtrMap}(g_0)$ for $g_1$ in Figure 4.2(b)) must be preserved along path “true” as above and cannot be killed path-sensitively as in Figure 4.2(b).

To account for the read and modification side effects at a call site, the binding between the actual and formal parameters is performed in the standard manner.

**Definition 10** (Mappings of Formal-Ins and Formal-Outs). *For a formal-in $f_{\text{in}} \in \text{MOD}(...)*
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on
Full-Sparse Memory SSA

\( F^n_{in} \) of method \( n \) invoked at call site \( c \), \( \text{Callee2Caller}_{in}(c,n,f_{in}) \) denotes the corresponding actual parameter of \( f_{in} \) at \( c \) if \( f_{in} \) is a formal parameter of \( n \) and \( f_{in} \) itself otherwise (i.e., if \( f_{in} \) is a nonlocal). For a formal-out \( f_{out} \in F^n_{out} \) of \( n \) invoked at \( c \), \( \text{Callee2Caller}_{out}(c,n,f_{out}) \) denotes the variable at \( c \) that is assigned from \( f_{out} \) if \( f_{out} \) is a return parameter of \( n \) and \( f_{out} \) itself otherwise.

In Figure 4.2, \( \text{Callee2Caller}_{in}(c,foo,x)=a \), \( \text{Callee2Caller}_{in}(c,foo,q)=q \), and \( \text{Callee2Caller}_{in}(c,foo,f)=\text{Callee2Caller}_{out}(c,foo,f)=f \), where \( c \) is the call to \( foo \).

Conceptually, SPAS proceeds in the following two sequential stages:

**Stage 1.** \( L = \text{Partition}(\emptyset) \) We compute \( L \) by using some fast flow-insensitive pointer analysis for the pointers in \( \emptyset \). For example, we can apply Steensgaard’s algorithm [75] to obtain a points-to graph, merge all predecessors of each node, and finally, make the points-to graph acyclic by collapsing SCCs, as in [96]. The points-to level of a variable is its longest length over \( \{0, \ldots, L\} \) to a sink node.

**Stage 2.** \( \Delta_{-1} = \text{Analyze}(L,\Delta_L,G) \) We build SSA and resolve pointers, level by level, from \( L \) to 0. \( \Delta_L \) is the initial SSA that satisfies vacuously Property 1 for \( L \) and \( G \) is the initial call graph constructed when function pointers are not yet resolved. \text{Analyze} is restarted whenever new points-to relations are discovered for a function pointer. \( G \) is always a directed acyclic graph. In the presence of recursive calls, \( G \) is made acyclic by collapsing all SCCs. The analysis within each SCC is performed iteratively until a fixed-point is reached to obtain full context sensitivity for all the methods in the SCC. Once \text{Analyze} has run to completion, \( \Delta_{-1} \) is the full-sparse SSA obtained that satisfies Property 1 for level \(-1\) (excluding its Part(c)) and all pointers have been fully resolved.

When analyzing level \( \ell \), \text{Analyze} starts with \( \Delta_{\ell} \), i.e., the SSA form that satisfies Property 1 for \( \ell \) and ends with producing \( \Delta_{\ell-1} \), i.e., the SSA form that satisfies
Property 1 for $\ell - 1$. The call graph $G$ is traversed twice, first bottom-up (reversal topologically) and then top-down (topologically). When points-to cycles are detected, level $\ell$ is re-reanalyzed until $\Delta_{\ell-1}$ is completely built. Thus, the contexts in a transfer function may comprise the points-to relations of some formal-ins discovered earlier at higher levels and the current level $\ell$.

A context used in a callee is mapped to a caller in the standard manner by applying the context mapping introduced in Definition 11 below.

**Definition 11 (Context Mapping).** Let $C$ be a context used in a callee $n$ invoked at a call site $c$ in a method $m$. $\text{Callee2Caller}_{ctx}(c, n, C)$ denotes the mapping of $C$ from callee $n$ to call site $c$ by performing a formal-to-actual parameter mapping. It is understood that every points-to relation in $\text{Callee2Caller}_{ctx}(c, n, C)$ that is not dependent on any of $m$’s contexts is fully evaluated (to true or false).

Figure 4.3 gives our algorithm for analyzing a method $m$ at level $\ell$. We describe the bottom-up phase first but both phases may have to be understood together.

To soundly capture path correlation, the path assigned to a variable at any of its definition site must not under-approximate the scope of its definition.

Due to Property 1(c), SPAS proceeds to expose the MAY-USEs and MAY-DEFs for each pointer at level $\ell$ that is accessed at a call site $c$. This is done by simply examining the context condition $C_{f_{\text{out}}}$ of $\text{MOD}(n, f_{\text{out}})$ (Definition 8) and the context condition $C_{f_{\text{in}}}$ of $\text{USE}(n, f_{\text{in}})$ (Definition 9) of each callee $n$ invoked at $c$, which were computed earlier during the same bottom-up phase. In line 13, the path for a $\chi$ operation is safely over-approximated as $P_c$, i.e., the path of call site $c$, where $f_{\text{out}}$ may be defined. As SPAS tracks path-sensitivity intraprocedurally, the path condition for a $\mu$ operation at a call site is irrelevant and thus marked with a ‘*’.

Let us see how the MAY-USE and MAY-DEF are added for the call site $c_{\text{foo}}$ to $\text{foo}$ in main in Figure 4.2(b), given the transfer functions of $\text{foo}$ in (4.4). In
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

1 BottomUp (Method: m, Level ℓ)
2 Step 1: Addµ,χCallsites(m, ℓ)
3 Step 2: BuildSSA(m, ℓ)
4 Step 3: PointerInference(m, ℓ)
5 Step 4: Comp_MOD_USE_Funs (m, ℓ)

6 Addµ,χCallsites (Method: m, Level ℓ)
7 for each call site c in method m
8 Let P_c be the path allocated to call site c
9 for each callee n invoked at call site c
10 for each formal-out f_out ∈ F_out of n,
11 \( \mathcal{L}(f_{out}) = \ell \), that is not a return parameter
12 if \( (C_{f_{out}}^{\text{call}}(c, n, C_{f_{out}}^{\text{may}}))^\neq \text{false} \)
13 Add \( f_{out} = \chi(f_{out}, C_{f_{out}}^{\text{may}}, P_c, \text{MAY}) \) for c
14 for each formal-in f_in ∈ F_in of n,
15 \( \mathcal{L}(f_{in}) = \ell \), that is not a formal parameter
16 if \( (C_{f_{in}}^{\text{call}}(c, n, C_{f_{in}}^{\text{may}}))^\neq \text{false} \)
17 Add \( \mu(f_{in}, C_{f_{in}}^{\text{may}}, v) \) for c

18 BuildSSA (Method: m, Level ℓ)
19 Apply the SSA construction algorithm [15]

20 PointerInference (Method: m, Level ℓ)
21 Perform a guarded inclusion-based flow-insensitive pointer analysis using the rules in Table 4.1

22 Comp_MOD_USE_Funs (Method: m, Level ℓ)
23 See text (Section 4.4.3)

24 TopDown (Method: m, Level ℓ)
25 Step 1: Resolve_PointToSets(m, ℓ)
26 Step 2: Addµ,χDerefs(m, ℓ)
27 Resolve_PointToSets (Method: m, Level: ℓ)
28 for each call site c in method m
29 for each callee n invoked at call site c
30 for each variable version p_i in m, \( \mathcal{L}(p) = \ell \), such that p_i reaches call site c and p = Callee2Caller(c, n, f_in), where f_in ∈ F_in
31 for each (v, C_f, P_f) ∈ Dep(p_i)
32 PtrSet(f_in) ⊇ \{v\}
33 for each (f, C_f, P_f) ∈ Dep(p_i)
34 PtrSet(f_in) ⊇ PtrSet(f)

35 Addµ,χDerefs (Method m, Level ℓ)
36 for each store “sp_i = ...” in m, \( \mathcal{L}(p) = \ell \)
37 Let PtrMap(p_i) = (Loc(p_i), Dep(p_i))
38 for each (v, C_v, P_v) ∈ Loc(p_i)
39 Add v = \( \chi(v, C_v, P_v, \text{MAY}) \)
40 for each (f, C_f, P_f) ∈ Dep(p_i)
41 for each v ∈ PtrSet(f_in)
42 Add \( v = \chi(v, C_{f_{in}} \&, f_{in} \rightarrow v, P_{f_{in}}, \text{MAY}) \)
43 for each load “... sp_i” in m, \( \mathcal{L}(p) = \ell \)
44 Let PtrMap(p_i) = (Loc(p_i), Dep(p_i))
45 for each (v, C_v, P_v) ∈ Loc(p_i)
46 Add \( v = \chi(v, C_v, P_v) \)
47 for each (f, C_f, P_f) ∈ Dep(p_i)
48 for each v ∈ PtrSet(f_in)
49 Add \( v = \chi(v, C_{f_{in}} \&, f_{in} \rightarrow v, P_{f_{in}}, \text{MAY}) \)

Figure 4.3: Bottom-up and top-down analysis of method m at level ℓ.
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

line 12, \( C_f^{\text{may}} = \text{Callee2Caller}_\text{ctx}(c_{\text{foo}}, \text{foo}, C_f^{\text{may}}) = \text{true} \) since by Definition 11, \( C_f^{\text{may}} = x \to f \) is mapped to \( a \to f \) at the call site, which is generated locally in \text{main}. So the MAY-DEF, \( f = \chi(f, \text{true}, \text{true}, \text{MAY}) \), is added. The MAY-USE, \( \mu(q, \text{true}, *) \), is added since \( C_q = \text{true} \) in \text{USE}(\text{foo}, q).

all MAY-USES and MAY-DEFS are exposed for the pointers at level \( \ell \) accessed, they can be put in SSA by applying the classic SSA construction algorithm [15], as illustrated in Figure 4.2.

Table 4.1 lists the seven rules for resolving the points-to maps for the pointers at level \( \ell \) in a method \( m \). The first six rules are illustrated in Figure 4.2 and the last partially when \text{Add}_{\mu, \chi}_\text{Callsites} \) is discussed.

The propagation of points-to information across an assignment may be guarded by both a context condition and a path condition. We define \( P(x) \times C_x \times P_z = \{(v, C_v \wedge C_x, P_v \wedge P_z) \mid (v, C_v, P_v) \in P(x)\} \). \( P(x) \cup P(y) \) includes all and only elements in \( P(x) \) and \( P(y) \) such that if \((v, C_v^x, P_v^x) \in P(x)\) and \((v, C_v^y, P_v^y) \in P(y)\), then both are merged as \((v, C_v^x \vee C_v^y, P_v^x \vee P_v^y)\).

\text{Loc-Init} \) is self-explanatory. As SPAS is intraprocedurally path-sensitive, the path \( P_{p_0=\&a} \) is generated locally in method \( m \). \text{Dep-Init} \) is applied to a copy of the form \( p_0 = p \), where \( p \) is a formal-in of method \( m \). Such copies are added at the entry of \( m \) for all its formal-ins. The path condition is over-approximated as \text{true} \) since \( p_0 \) may point to whatever \( p \) point to on entry of the method considered.

\text{Assn} \) applies to every other copy assignment. The points-to relations at the RHS are propagated to the LHS, guarded by the path of the assignment.

Rules \text{Mu} \) and \text{Chi} \) are also easy to understand. The context and path conditions in a \( \chi \) or \( \mu \) operation serve as the guards to enforce context- and path-sensitivity. According to the second constraint for \text{Chi}, the old points-to relations of \( v \) are weakly updated, i.e., simply propagated from \( v_t \) to \( v_s \) unchanged.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
<th>Constraints</th>
<th>Inference Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc-Init</td>
<td>$p_i = &amp;a$ (on path $P_{p_i=&amp;a}$)</td>
<td>$p_i \supseteq {a}$</td>
<td>$\text{Loc}(p_i) = {(a, \text{true}, P_{p_i=&amp;a})}$ $\text{Dep}(p_i) = \emptyset$</td>
</tr>
<tr>
<td>Dep-Init</td>
<td>$p_0 = p$ (a formal-in)</td>
<td>$p_0 \supseteq p$</td>
<td>$\text{Loc}(p_0) = \emptyset$</td>
</tr>
<tr>
<td>Assn</td>
<td>$p_i = q_j$ (on path $P_{p_i=q_j}$)</td>
<td>$p_i \supseteq \text{true} \times P_{v_i=q_j} \times q_j$</td>
<td>$P(p_i) = P(q_j) \times \text{true} \times P_{p_i=q_j}$</td>
</tr>
<tr>
<td>Mu</td>
<td>$p_i = \phi(p_j, p_k)$</td>
<td>$p_i \supseteq C_{v_i} \times P_{v_i}$ $v_k$</td>
<td>$P(p_i) = P(v_k) \times C_{v_i} \times P_{v_i}$</td>
</tr>
<tr>
<td>Chi</td>
<td>$p_i = q_j$</td>
<td>$v_s = \chi(v_i, C_{v_i}, P_{v_i}, \text{MAY})$</td>
<td>$P(v_i) = P(q_j) \times C_{v_i} \times P_{v_i}$</td>
</tr>
<tr>
<td>Phi</td>
<td>$p_i = \phi(p_j, p_k)$</td>
<td>$P(p_j)$ $P(p_k)$</td>
<td>$P(p_i) = P(p_j) \times \text{true} \times P_{p_j}$</td>
</tr>
</tbody>
</table>

Table 4.1: Rules for resolving points-to maps $\text{PtrMap}(x) = \{\text{Loc}(x), \text{Dep}(x)\}$ in method $m$ for level $\ell$. Each of the last five is applied once for $P = \text{Loc}$ and once for $P = \text{Dep}$.
Let us consider Rule \textit{Phi}. For each operand, we use the path along which its value flows into the result as the guard to propagate its points-to relations into the result. In a FSCS pointer analyzer that does not consider path-sensitivity, the two unguarded constraints $p_i \supseteq p_j$ and $p_i \supseteq p_k$ are generated. Applying these two would yield the two spurious points-to relations shown in Figure 4.1(d).

Finally, Rule \textit{Call} is applied to a call site $c$ in the standard manner. In lines C4 and C7, constraints are generated to propagate the points-to relations created by a callee $n$ in $\text{Loc}(f_{\text{out}})$ and $\text{Dep}(f_{\text{out}})$ to $v_s$, guarded by the (mapped) context conditions $C_o$ and $C_{\text{fin}}$, respectively; but the paths created inside the callee are ignored (and hence, the $*$’s). In line C8, $v$ is weakly updated as in Rule \textit{Chi}.

For each formal-out $f_{\text{out}} \in F_{\text{out}}^m$ at level $\ell$, we write $f_{\text{out}}^{\text{max}}$ for its last SSA version in method $m$. Let $\text{PtrMap}(f_{\text{out}}^{\text{max}}) = (\text{Loc}(f_{\text{out}}^{\text{max}}), \text{Dep}(f_{\text{out}}^{\text{max}}))$, which is already available. Then $\text{MOD}(m, f_{\text{out}})$ is defined to be $(\text{Loc}(f_{\text{out}}^{\text{max}}), \text{Dep}(f_{\text{out}}^{\text{max}}), C_{\text{may}})$, where $C_{\text{may}}$ is set as \text{true} if $f_{\text{out}}$ is directly modified in method $m$ and set otherwise as a disjunction of the context conditions in all its MAY-DEF sites, i.e., all $\chi$ operations of $f_{\text{out}}^{\text{max}}$ in method $m$.

For a formal-in $f_{\text{in}} \in F_{\text{in}}^m$ at level $\ell$, we write $f_{\text{in}}^{0}$ for its first SSA version in $m$. Thus, $\text{USE}(m, f_{\text{in}}) = (\text{PtrSet}(f_{\text{in}}), C_{\text{fin}})$, where $C_{\text{fin}}$ is \text{true} if $f_{\text{in}}^{0}$ is directly used in method $m$ and otherwise as a disjunction of the context conditions at all MAY-USE sites, i.e., $\mu$ operations of $f_{\text{in}}^{0}$ in $m$. $\text{PtrSet}(f_{\text{in}})$ is computed later by $\text{Resolve\_Points\_To\_Sets}$ and subsequently used in lines 41 and 48 of $\text{Add\_Mu\_Chi\_Derefs}$.

In Figure 4.2(b), the $\text{MOD}$ and $\text{USE}$ functions of $\text{foo}$ are given in (4.4).

The points-to sets of the pointers at level $\ell$ in method $m$ can now be obtained by resolving all formal-ins (lines 32 and 34).
points-to relations in $\text{Loc}(p_i)$ are generated locally in method $m$ and handled straightforwardly. To deal with those generated by $m$’s callers in $\text{Dep}(p_i)$ in lines 42 and 49, new context conditions are generated. If $p_i$ points to $v$, because a formal-in $f_{in}$ does, then $C_{f_{in}}$ is strengthened to include $f_{in} \to v$ to indicate the context condition under which the MAY-USE/MAY-DEF occurs (Figure 4.2).

SPAS soundly tracks path correlation on top of a FSCS pointer analyser.

**Theorem 4.** $\text{PtrSet}(p)$ contains all possible targets for $p$ during any execution.

*Proof sketch.* In a FSCS pointer analyser without considering path-sensitivity, the path/scope for a variable definition is taken as true. SPAS refines but never under-approximates it at a call site (lines 8 and 13 in Figure 4.3) and in Rules $\text{Loc-Init}$, $\text{Dep-Init}$, $\text{Assn}$ and $\text{Phi}$ (Table 4.1). So the soundness of SPAS follows from that of the underlying FSCS analyser, which preserves Property 1 level-wise.

The following theorem states a well-known fact about path-sensitivity.

**Theorem 5.** Let $\text{PtrSet}^{\text{SPAS}}(p)$ ($\text{PtrSet}^{\text{LevPA}}(p)$) be the points-to set of $p$ found by SPAS (a FSCS pointer analyser like $\text{LevPA}$). Then $\text{PtrSet}^{\text{SPAS}}(p) \subseteq \text{PtrSet}^{\text{LevPA}}(p)$.

*Proof sketch.* Compared to $\text{LevPA}$, as argued in the proof of Theorem 4, the path condition at a variable definition site in SPAS is either the same or strengthened.

### 4.4.4 Supporting Strong Updates

In Table 4.1, a $\chi$ operation $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \text{MAY})$ represents a MAY-DEF, where $P_{v_s}$ safely overapproximates the scope where $v_s$ is defined (Theorem 4). In Rules $\text{Chi}$ and $\text{Call}$, $v_s \supseteq v_t$ is always used as only weak updates are allowed.

To support strong updates, we consider a $\chi$ operation $v_s = \chi(v_t, C_{v_s}, P_{v_s}, M_{v_s})$, associated with a store “*$p_i = \ldots$” residing on a path $P_{\chi}$ in method $m$. Let this $\chi$ operation be referred to as $\chi_{op}$. In $\chi_{op}$, $M_{v_s} \in \{\text{MAY}, \text{MAY}^+, \text{MUST}\}$. So $\text{MAY}^+$
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

is now identified as a special case of MAY introduced in Section 4.4.2. $M_{vs}$ is set as MUST when $\chi_{op}$ is a MUST-DEF. This is both context-sensitive and path-sensitive, meaning that $p_i$ must point to $v$ along path $P_{vs}$ in context $C_{vs}$. However, in the other contexts, $p_i \rightarrow v$ may not hold, i.e., $P_{vs}$ may not be exact. $M_{vs}$ is MUST when $\text{UniqueTarget}(m, p_i, v)$ is true, which is defined to hold when (a) $p_i$ points to $v$ uniquely whenever method $m$ is invoked at a call site such that $p_i$ points to $v$ and (b) $v$ is a concrete object in $\text{Singletons}$. Following [52], $\text{Singletons}$ is the subset of locations in $\mathbb{O}$ with arrays, heap objects and locals inside recursion cycles being removed. $M_{vs}$ is set as $\text{MAY}^+$ or MAY when $\chi_{op}$ is a MAY-DEF, in which case, dereferencing $p_i$ may yield more than one target. $M_{vs}$ is set as $\text{MAY}^+$ when $C_{vs} = \text{true}$ and $P_{vs}$ is exact, meaning that $p_i$ must point to $v$ whenever the program is executed along $P_{vs}$. Otherwise, $M_{vs}$ is set as MAY, in which case, $p_i$ may or may not point to $v$ as described in Section 4.4.3.

Now, constraint $v_s \supseteq v_t$ used in Table 4.1 is augmented with the guards:

$$v_s \supseteq_{C_{old} \times P_{old}} v_t, \text{ where } (C_{old}, P_{old}) = \begin{cases} (-C_{vs}, \text{true}) & \text{if } M_{vs} = \text{MUST} \\ (\text{true}, P_{\chi} \wedge \neg P_{vs}) & \text{if } M_{vs} = \text{MAY}^+(4.6) \\ (\text{true}, \text{true}) & \text{if } M_{vs} = \text{MAY} \end{cases}$$

The base version of our algorithm, shown in Figure 4.3, performs weak updates only as it treats MUST and $\text{MAY}^+$ conservatively as MAY. In our fully-fledged algorithm, SPAS obtains improved precision since all-path strong updates, much like Dead Code Elimination (DCE), are enabled when $M_{vs}$ is MUST. In this case, the old points-to relations of $v_t$ at all incoming paths of the store $p_i = \ldots$ are killed if they are in the same context $C_{vs}$. In addition, SPAS improves analysis precision further since some-path strong updates, much like Partial DCE [93], are also performed when $M_{vs}$ is $\text{MAY}^+$. In this case, the old points-to relations of $v_t$ in any context are killed along $P_{vs}$ but allowed to flow into $v_s$ along $P_{\chi} \wedge \neg P_{vs}$. 
We only need to make small changes to our algorithm in Figure 4.3. Only the path conditions for the points-to relations of \( p_i \) established intraprocedurally in \( \text{Loc}(p_i) \) may be considered as being exact conservatively.

**Line 8** Insert a MUST-DEF, \( r = \chi(r, \text{true}, P_c, \text{MUST}) \), between lines 8 and 9 to handle the assignment of a return parameter in a function invoked at the call site \( c \) to a locally-defined pointer \( r \) in method \( m \) (Definition 5).

**Line 11** Use \( \text{MOD}(m, f_{\text{out}}) \) given in Definition 8, where its fourth component \( C_{\text{out}}^{\text{must}} \) is a conjunction of context conditions, one condition \( C_P \) for every possible path \( P \) from the entry to the exit of method \( m \), such that \( C_P \) is true if \( f_{\text{out}} \) is directly modified on \( P \) and a disjunction of context conditions in all \( \chi \)'s representing MUST-DEFs on \( P \) otherwise.

**Line 13** \( M_{v_s} \) is MUST if \( \text{Callee2Caller}_{\text{ctx}}(c, n, C_{\text{must}}^{\text{out}}) \) holds for every callee \( n \) checked in line 9 and MAY otherwise.

**Line 39** \( M_{v_s} \) is MAY\(^+\) if \( v \in \text{Singletons} \) and \( p_i \) is not defined in a cycle in the CFG of method \( m \) (decided in \( \text{Pointer\_Inference} \)), and MAY otherwise.

**Line 42** \( M_{v_s} \) is MUST if \( \text{UniqueTarget}(m, p_i, v) \) holds and MAY otherwise. (This overwrites MAY\(^+\) set for \( v \) in line 39 if \( \text{UniqueTarget}(m, p_i, v) \) holds.)

The points-to maps for \( f_1 \) and \( g_1 \) are thus refined from (4.5) to those in Figure 4.2.

**Theorem 6.** With strong updates thus specified, Theorems 4 and 5 remain valid.

**Proof sketch.** Follows simply from the definitions of MUST, MAY\(^+\) and MAY.
4.5 Experimental Evaluation

We have implemented SPAS in the Open64 compiler (v4.2). We use the CUDD2.4.2 library for representing points-to relations, contexts and paths. As in [96], parameterised spaces are used to reduce analysis overhead and improve precision. We evaluate the scalability of SPAS in handling (intrprocedural) path-sensitivity by integrating it with a state-of-the-art FSCS pointer analyser, LevPA [96], which already performs all-path strong updates (for MUST-DEFs). Despite this, SPAS obtains points-to information with non-decreasing precision with improvements at stores/loads that are amenable to path-sensitive pointer analysis, at a small increase in analysis overhead. We have used all 27 C benchmarks from SPEC CPU2000 and CPU2006 and carried out our experiments on a 3.0GHz quad-core Intel Xeon system running Redhat Enterprise Linux 5 (kernel version is 2.6.18) with 16GB memory. Benchmarks 253.perlbmk and 403.gcc run out of memory under both analyzers and are thus excluded in further discussions.

4.5.1 Analysis Overhead

As shown in Table 4.2, SPAS uses 18.42% more time and 10.97% more memory than LevPA on average. To the best of our knowledge, SPAS is the fastest path-sensitive pointer analysis reported. Benchmarks 176.gcc and 400.perlbench are the most costly to analyze due to many iterations required for handling function pointers and recursion cycles. From the statistics in the last nine columns, we can see the extra analysis overhead incurred by SPAS. Column “D-Vars” gives the number of decision variables used to encode the paths in a program using BDDs, resulting in a slight increase in the total memory usage by SPAS for each benchmark (measured using the memory tracing tool available in Open64).
## Table 4.2: Percentage increases of analysis overhead under SPAS w.r.t. LevPA.
In the last eight columns, the analysis time for a benchmark is broken down into eight parts on computing points-to levels, generating paths (Section 4.4.1), and performing the six steps of SPAS in Figure 4.3. In 13 out of 25 benchmarks, Step 2 (Build_SSA) and Step 3 (Pointer_Inference) consume most of the analysis time in a benchmark. These are also the very steps where SPAS spends more analysis time than LevPA as it does extra work in handling program paths. For 186.crafty, 188.ammp, 401.bzip2 and 464.h264ref, the analysis times under LevPA are small. SPAS adds relatively high overheads mainly in Steps 2 and 3.

4.5.2 Path-Sensitive Precision

Table 4.3 shows that SPAS can obtain more precise points-to sets than LevPA at certain loads/stores in most benchmarks. We consider only the loads/stores that reside beyond the first branch in the CFG of a method after all its SCCs (strongly connected components) have been collapsed. The pointers accessed indirectly at their associated χ and μ operations (MAY/MUST-DEFs and MAY-USEs) are the ones whose points-to information may be improved by SPAS.

We measure the number of χ’s and μ’s with improved points-to information in two ways, indicated by their Traditional and Path-Sensitive columns. Note that by Theorem 5, \( \text{PtrSet}(p)^{\text{SPAS}} \subseteq \text{PtrSet}(p)^{\text{LevPA}} \) holds for any pointer \( p \). With the traditional metric, the points-to set of \( p \) is said to be more precise under SPAS if \( |\text{PtrSet}(p)^{\text{SPAS}}| < |\text{PtrSet}(p)^{\text{LevPA}}| \). With the path-sensitive metric, the path, i.e., the scope information governing each points-to target is also taken into account. For LevPA, the path guarding a χ or μ operation is always true. Thus, the points-to set of \( p \) is more precise under SPAS if either \( |\text{PtrSet}(p)^{\text{SPAS}}| < |\text{PtrSet}(p)^{\text{LevPA}}| \) or the guarding path for a χ or μ operation is not true (i.e., more restricted).

As shown in Table 4.3, SPAS has improved points-to information in most bench-
Table 4.3: Percentages of variables at χ’s and µ’s with more accurate points-to sets.
Chapter 4. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse Memory SSA

marks. Under “Traditional” for $\chi$’s, the percentage improvements range from 0% to 6.71% with an average of 2.61%. Under “Traditional” for $\mu$’s, the percentages are within 0 to 7.08% with an average of 2.49%. Under “Path-Sensitive”, the improvements are more significant with an average of 42.38% for $\chi$’s and 41.07% for $\mu$’s. These results should be understood with some caveats. First, what SPAS is compared with is a state-of-the-art FSCS pointer analyser that already performs all-path strong updates. Second, SPAS obtains such improved path-sensitive precision at small analysis overhead. Finally, such improvement can be critical for some client applications (e.g., bug detection).

Let us look at some benchmarks in detail. In the case of 164.gzip, 176.gcc, 197.parser, 400.perlbench, 429.mcf, 464.h264ref, 482.sphinx3, 458.sjeng the improvements are mostly over 3% for both $\chi$’s and $\mu$’s. Path-sensitive analysis provides little benefits for seven benchmarks: 179.art, 181.mcf, 183.equake, 256.bzip2, 433.milc, 462.libquantum and 470.lbm. They are small programs with few pointers but mostly scalar-to-array assignments. For other benchmarks on scientific computations, such as 177.mesa, and 255.vortex, the improvements are below 2%. In contrast, benchmarks such as 186.crafty, 445.gobmk and 456.hmmer exhibit much better precision improvements. They each have a relatively high number of decision variables, giving rise to more opportunities for capturing path correlation.

4.6 Summary

In this chapter, we presented SPAS, a pointer analysis that explores path correlations based on a recent flow- and context-sensitive pointer analysis LevPA. Our experimental evaluation shows that SPAS incurs reasonable analysis overhead over
LevPA (on average 18.42% increase in analysis time and 10.97% increase in memory usage) and computes more precise points-to information. It is expected that SPAS can provide some insights for developing client-driven pointer analysis techniques in the future.
Chapter 5

Conclusions and Future Work

5.1 Conclusions

Pointer analysis is a fundamental research in program analysis. It provides information for subsequent applications by solving possible runtime targets of pointers statically. The more precise information it produces, the more effective the subsequent applications can be. Due to the undecidability of pointer analysis, finding the right balance between efficiency and precision is crucial for making a practical pointer analysis for applications.

In this thesis, we first introduced some background information about pointer analysis, including flow-sensitivity and field-sensitivity. We also presented the construction of the full-sparse memory SSA form which is used in many pointer analyses. These built the foundation for our two pointer analysis techniques: have introduced two pointer analysis techniques: (1) SELFS, a flexible region-based interprocedural pointer analysis which allows efficiency and precision trade-offs to be made according to the region partitioning strategies used. By maintaining flow-sensitivity between regions and solving flow-insensitively inside regions, SELFS is
able to speed up the state-of-the-art sparse flow-sensitive analysis while obtaining almost the same precision. (2) SPAS, a full-sparse path-sensitive FSCS pointer analysis which obtains intra-procedural path-sensitivity by encoding program paths using BDDs. It has compatible performance to the state-of-the-art FSCS analyzer LevPA by adding a small overhead while ensuring non-decreasing precision of the points-to information.

5.2 Future Work

Demand-Driven Analysis

It will be more difficult to do the whole program analysis for software in the future due to the sheer size and complexity. Fully sensitive analyses are prohibitively expensive to be carried out over the entire program. However in practice, some applications may require only parts of the program to be analyzed while needing little done about the other parts. This would significantly reduce the size of the problem domain. Instead of evaluating all possible program paths blindly, pointer analysis can be more effective by focusing on the hot parts specified by applications. Although some demand-driven approaches have been proposed for pointer analysis [72, 74], all the existing methods are flow-insensitive. It is promising that SELFS can be used to facilitate the demand-driven analysis by applying different strategies to different regions.

Hybrid Analysis

There are many dimensions in pointer analysis, such as flow-sensitivity, context-sensitivity, path-sensitivity, field-sensitivity etc. Most pointer analyses try to solve problems by using one or more types of sensitivity over the program without noticing that different parts of the program may have different characteristics. For exam-
ple, hot functions which maybe invoked many times may need context-sensitivity to improve the precision, while a straight line call chain without any other callers in between can be analyzed context-insensitively and achieve the same precision as the sensitive one. It is challenging yet promising to apply different sensitivity strategies to different program parts to improve the precision of pointer analysis while reducing its overhead.

**Multi-Threaded Program Analysis**

As an important technique to improve the performance of programs, multi-core and multi-threading have been widely used in modern software. There are many hot topics in this area, such as compiler optimizations [16, 44], data race detection [64, 88], speculative parallelization [24, 25, 67] and atomicity violations [56, 57]. Due to the unpredictable execution order of the threads, it is difficult to examine and find all errors in multi-threaded programs with dynamic analysis. There are already several works trying to analyze multi-threaded programs with the help of pointer analysis [69–71]. However the open problem about how to reduce the false alarms reported and reduce the analysis overhead still remains.
Bibliography


