Fast Flow-Sensitive Pointer Analysis

A Thesis
Submitted For the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

by

Vaivaswatha N

Supercomputer Education and Research Centre
Indian Institute of Science
BANGALORE – 560 012
August 2014
TO

My Parents
Acknowledgements

I thank my advisor Prof. Govindarajan for guiding me throughout this research. If not for his support, patience and encouragement, this thesis would not have seen the light of day. I am grateful to him for giving me the freedom and time to explore different topics, even those that were not promising.

I am grateful to Prof. Matthew Jacob T, Prof. Uday Reddy and Prof. Vittal Rao, whose courses in my first semester helped me get a good start on my research problem. I am indebted to Mr. Prashantha Naduthota for introducing me to Compilers and encouraging me to pursue research at IISc.

I express my gratitude to Ben Hardekopf, Christian Borgelt and Xiao Xiao for making their implementations of the staged flow-sensitive pointer analysis, Eclat and Pestrie algorithms respectively, public. These tools form a critical part of this thesis. I would also like to thank Mario Méndez-Lojo and Rupesh Nasre for their comments on the parallelization work.

If not for my parents’ emphasis on a good education, I might never have gotten into IISc to write this thesis. I am grateful to them, my brother and my sister-in-law for their constant support throughout.

I thank Jayvant, Nagendra, Sreepathi, Mani, Prasanna, Shilpa and Madhura for all their help and for the interesting discussions we had at the lab. I also thank my friends Ravi, Ninad, Vidya and Aravind for the interesting and fun chats we had.
Publications from this Thesis

1. Vaivaswatha Nagaraj and R. Govindarajan; “Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting”. In the Proceedings of The 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), Edinburgh, Scotland, Sept 7-11 2013

Abstract

Pointer alias analysis is a well researched problem in the area of compilers and static program analysis. Many recent works in this area have focused on flow-sensitivity due to the additional precision it offers. Precise analyses allow clients to make more accurate and useful decisions. With higher precision, a compiler can perform more optimizations and a static analysis tool can report more useful information about a program.

Flow-sensitivity is a dimension of pointer analysis that determines whether the analysis takes the program’s control flow into account. A flow-sensitive pointer analysis computes points-to sets for pointers at each program point, and hence, is more precise than a flow-insensitive analysis. However, a flow-sensitive analysis is computationally expensive. This has limited its use in larger programs. Earlier works on flow-sensitive pointer analysis have dealt with scaling it to larger programs. In this work, we propose and study two approaches to further decrease the time taken by a flow-sensitive analysis. We propose a technique to parallelize flow-sensitive pointer analysis, thus allowing it to utilize multiple processor cores. We also introduce an approximation algorithm for flow-sensitive pointer analysis, enabling faster execution with a small precision loss.

In the first part of our work, we formulate the staged flow-sensitive pointer analysis as a graph-rewriting problem. Graph-rewriting has already been used for flow-insensitive analysis. However, formulating flow-sensitive pointer analysis as a graph-rewriting problem adds additional challenges due to the nature of flow-sensitivity. In particular, we identify two key challenges that flow-sensitivity adds to a graph-rewriting formulation of pointer analysis and provide solutions to handle them. We implement our parallel algorithm using Intel Threading Building Blocks and demonstrate considerable scaling (upto
4.05x) for 8 threads on a set of 10 benchmarks. Compared to the sequential implementation of staged flow-sensitive analysis, a single threaded execution of our implementation performs better in 9 out of the 10 benchmarks.

In the second part of our work, we observe that a number of object sets, consisting of tens to hundreds of objects appear together and frequently in many points-to sets. We observe that propagating these large points-to sets during the analysis is expensive. By approximating each of these object sets by a single object, we can speedup the computation of points-to sets. Although the proposed approach incurs a slight loss in precision, it is shown to be safe. We use a well-known data mining technique called frequent itemset mining to find these frequently occurring object sets. We compare our approximation to a fully flow-sensitive pointer analysis on a set of 10 benchmarks. We measure precision loss using two common client analysis queries and report an average precision loss of 0.25% on one measure and 1.40% on the other. The proposed approach results in a speedup of upto 12.9x (and an average speedup of 6.2x) in computing the points-to sets.
# Contents

Acknowledgements iii  
Publications from this Thesis v  
Abstract vii  
Keywords xv  

1 Introduction 1  
1.1 What Is Pointer Analysis? 2  
1.2 Dimensions of Pointer Analysis 3  
1.2.1 Context-Sensitivity 3  
1.2.2 Flow-Sensitivity 3  
1.2.3 Field-Sensitivity 4  
1.3 Why Pointer Analysis? 4  
1.4 Our Contributions 6  
1.5 Organization of this Thesis 7  

2 Background 8  
2.1 Static Single Assignment Form 8  
2.2 Pointer Analysis 10  
2.2.1 An Example: Andersen’s Analysis 11  
2.3 Flow-Sensitive Pointer Analysis 13  
2.3.1 A Formal Description 14  
2.4 Staged Flow-Sensitive Pointer analysis 16  

3 Parallelization 18  
3.1 Graph-Rewriting 19  
3.2 Challenges and Solutions 21  
3.2.1 Presence of Spurious Edges 21  
3.2.2 Strong and Weak Updates 22  
3.3 Algorithm 24  
3.3.1 Graph Structure 24
3.3.2 Rewrite Rules ................................................. 28
3.3.3 Putting It All Together ..................................... 28
3.3.4 Properties of Our Graph-rewriting System .......... 30

3.4 Implementation and Optimizations .......................... 32

3.5 Evaluation ....................................................... 33
3.5.1 Comparison with SFS ...................................... 35
3.5.2 Scaling ....................................................... 35

3.6 Chapter Summary ................................................ 37

4 An Approximation Algorithm ................................. 39
4.1 Motivation ....................................................... 40
4.2 Frequent Itemset Mining ..................................... 43
4.3 Frequent Object Set Merging (FOSM) .................... 43
4.3.1 Correctness ............................................... 45
4.3.2 Precision .................................................. 46
4.3.3 Implementation .......................................... 47

4.4 Evaluation ....................................................... 48
4.4.1 Performance ............................................. 48
4.4.2 Convergence ............................................. 49
4.4.3 Precision .................................................. 49
4.4.4 Choice of Objects to be Merged ....................... 51

4.5 Chapter Summary ................................................ 52

5 Related Work ....................................................... 53
5.1 Flow-Insensitive Analysis .................................. 53
5.1.1 Early Works ........................................... 53
5.1.2 Speeding Up Andersen’s Analysis .................. 54
5.1.3 Approximate Flow-Insensitive Analysis .......... 55

5.2 Flow-Sensitive Pointer Analysis ............................ 55
5.3 Complexity of Pointer Analysis ............................ 58

6 Conclusion and Future Work .................................. 60

References .......................................................... 63
List of Tables

2.1 Andersen’s analysis: an example ................................................. 12
3.1 Number of nodes of each type ............................................... 34
3.2 Average number of nodes in each worklist ............................ 35
4.1 Details of frequently occurring objects ................................. 41
4.2 Similarity between the frequent object sets of the flow-insensitive and the flow-sensitive pointer analyses ................................. 45
4.3 Speedup in execution time of the analysis and the fraction of worklist items processed after frequent object set merging ......................... 49
4.4 Percentage of pointer pairs that alias and the percentage of dependent load/stores in each function ........................................... 51
List of Figures

1.1 What does a pointer analysis do? ........................................... 2
1.2 Need for pointer analysis ..................................................... 5
2.1 (a) An example program and (b) its SSA form .......................... 9
2.2 Updating points-to sets in Andersen’s analysis ......................... 12
2.3 Difference in points-to sets of a flow-insensitive and a flow-sensitive pointer analysis ....................................................... 13
2.4 Example: Strong vs Weak Updates ....................................... 14
2.5 Simple example to illustrate the need for AUX ....................... 17
3.1 Rewrite rules for Andersen’s analysis. Dashed arrows indicate newly added edges. The edge types are: p(points-to), c(copy), l(load-to), s(store-from) 20
3.2 (a) Example points-to constraints, (b) Initial constraint graph, (c) The constraint graph of (b) after applying the copy rewrite rule. ............. 20
3.3 (a) A simple program. (b) Flow-insensitive constraint graph corresponding to the load constraints in nodes 3 and 5 of (a) .................. 22
3.4 The points-to set of variable $a_1$ in (a) is decided by the points-to set of $x_1$ as show in (b). ......................................................... 23
3.5 Initial graph for different types of constraints. In (a), $k_1$ is a Klique node. In (e), $v_1$, $v_2$, $v_3$ may be either top-level or address-taken. ............. 27
3.6 Rewrite rules for flow-sensitive pointer analysis ....................... 29
3.7 Comparison of 1-thread execution with SFS ............................. 36
3.8 Thread scaling ............................................................... 37
3.9 Scaling of number of conflicts ............................................. 38
4.1 Progress of the analysis vs (a) Growth of frequent object sets (b) Cumulative time: with and without optimizing frequent object sets. ........... 42
5.1 Simple example to illustrate imprecision of Hasti and Horwitz’s algorithm 57
Keywords

Flow-sensitive pointer analysis, parallelization, graph-rewriting, approximate pointer analysis, frequent itemset mining.
Chapter 1

Introduction

Data-flow analysis is a compiler analysis step that focuses on statically (at compile time) determining the dynamic (run-time) properties of a program. Data-flow analysis is extensively used by compilers to perform safe code optimization [2]. It is also used in program diagnostic tools such (such as lint [31]) to detect and warn about potential bugs in a program.

Pointer (or points-to) analysis is a data-flow analysis whose goal is to determine statically, the dynamic properties of pointers. Such an analysis provides useful information, such as possible aliasing between different variables, which in-turn can be used by compilers for performing safe code optimizations, or by program diagnostic tools to provide useful information about the program being analyzed.

Recent emphasis on parallelizing programs [63] and analysis of bugs [20] or security vulnerabilities [32] in programs have given rise to increased attention to static analysis in general and pointer analysis in particular. However, the pointer analysis problem is computationally expensive (analysis of large programs can take a few hours to complete [22]). Thus, scaling pointer analysis has attracted much research attention [23,35,38].

In this thesis, we propose two methods for improving the performance of a specific type of pointer analysis. To understand the contribution of this thesis, first, we give a brief introduction to the pointer analysis problem. We then explain and discuss different dimensions of the analysis. Later in this chapter, we outline our contributions in detail.
1.1 What Is Pointer Analysis?

Pointer analysis is a static program analysis technique that attempts to determine the possible memory locations that a pointer may point to during any run of the program being analyzed [29].

Consider the program shown in Figure 1.1. Suppose we wish to determine the possible locations to which pointer \( p \) may point to, say at program point 2 (i.e., after the execution of statement \( s_3 \)) in the program. A pointer analysis on this program may be able to determine that \( p \), at program point 2, can only point to \( a \). In other words, the analysis may compute the points-to set of \( p \) at program point 2 to be \( \{a\} \). Similarly, at program point 3 (i.e., after execution of statement \( s_4 \)), it may compute the points-to set of \( p \) to be \( \{b\} \), while at program point 4 it may compute \( \{a,b\} \).

Note that we say “analysis may compute” rather than “analysis will compute”. This is because there are many variations (also called different dimensions) of pointer analysis, each computing their results with different precision. We will discuss the various dimensions of pointer analysis in the next section and what is meant by precision of an analysis in Section 2.2.
1.2 Dimensions of Pointer Analysis

Since a pointer analysis computes the possible run time values of pointer variables at compile time (i.e., statically), it is a data-flow analysis [34]. The general theory of data-flow analysis applies to pointer analysis as well. Just as in any data-flow analysis, a pointer analysis can be designed to consider (or ignore) certain characteristics of the program.

1.2.1 Context-Sensitivity

An analysis is said to be context-insensitive if it does not differentiate between the different calling contexts of the function being analyzed [34]. A context-insensitive analysis allows values to flow from one call through the function and return to another caller. A context-sensitive analysis on the other hand distinguishes between all [58] or some [49] of the different contexts in which a function can be called. Different approaches to a context-sensitive pointer analysis can be seen in the literature [29]. A study on the importance of context-sensitivity in pointer analysis can be found in [25].

1.2.2 Flow-Sensitivity

An analysis is said to be flow-insensitive if it ignores the control flow in the program. In the context of pointer analysis, this means that the analysis considers all pointer related statements in the program as an unordered set for the purposes of the analysis. A flow-sensitive analysis on the other hand respects program control flow. This means that it differentiates between different program points. Flow-sensitive pointer analyses compute points-to sets for pointers at each program point, whereas a flow-insensitive pointer analysis computes a single points-to set (for each pointer) for the entire program. We illustrate the differences between a flow-sensitive and a flow-insensitive pointer analysis with an example in Section 2.2.

Flow-sensitivity in pointer analysis has been of interest lately and a number of researches have focused on scaling flow-sensitive pointer analysis to large programs [23,38].
Chapter 1. Introduction

Pointer analyses, in general, have to deal with hundreds of thousands of pointers, each having thousands of pointees in their points-to sets. For a flow-sensitive analysis, this complexity multiplies since it requires this huge amount of information to be stored, processed and propagated at each program point during the analysis [23].

Since a flow-sensitive analysis computes points-to sets for pointers at each program point, the points-to sets computed are more precise than what would be computed by a flow-insensitive analysis. This additional precision allows clients analyses to make more accurate decisions. Flow-sensitive pointer analysis has been shown to be of importance to a variety of client program analyses such as analysis of multi-threaded code [57] and detection of security vulnerabilities [12].

1.2.3 Field-Sensitivity

Field sensitivity in an analysis determines whether the analysis keeps track of individual members of an aggregate data type (such as struct in C) [50]. A field-insensitive analysis flattens all aggregates (i.e., considers all members of the aggregate to be one). A field-sensitive pointer analysis differentiates one or more of the aggregate and can thus achieve more accurate points-to results.

In addition to the variations of pointer analysis just mentioned, pointer analyses can also be classified as inclusion based [3] vs unification based [59] or path-sensitive vs path-insensitive [61] etc.

1.3 Why Pointer Analysis?

Having seen what a pointer analysis is, and its various dimensions, we now illustrate how such an analysis can be useful. Consider the program shown in Figure 1.2(a). Assuming that \( x \) is not modified after its definition until its use in the condition, a compiler can deduce that the condition \( (x > 100) \) will always evaluate to false, and hence can eliminate the true part of the branch (dead code elimination).

Suppose the program in Figure 1.2(a) is modified as shown in Figure 1.2(b). Since
1.3. Why Pointer Analysis?

int x, *p;
void foo () {
   x = 20;
   ...
   *p = 302;
}

int x, *p;
void main () {
   x = 20;
   ...
   *p = 302;
   if (x > 100) {
      // Do A
   } else {
      // Do B
   }
}

int x, *p;
void main () {
   x = 20;
   ...
   *p = 302;
   if (x > 100) {
      // Do A
   } else {
      // Do B
   }
}

int x, *p;
void main () {
   x = 20;
   ...
   *p = 302;
   if (x > 100) {
      // Do A
   } else {
      // Do B
   }
}

void main () {
   p = &y;
   foo();
   ...
}

Figure 1.2: Need for pointer analysis

there is a store (*p = 302) in between the definition and use of variable x, the compiler, in the absence of pointer analysis has to assume that this store could potentially redefine (indirectly) x. Hence it cannot deduce whether the value of x is greater than 100, preventing the dead code elimination mentioned earlier. Now consider the program in Figure 1.2(c). If a pointer analysis can determine that p can only point to y, then the compiler can be sure that x is not redefined by the store statement “*p = 302”. Thus, it can eliminate the true part of the condition, performing successful dead code elimination.

A common use of pointer analysis is to determine if two pointers alias. Two pointers are said to alias if the intersection of their points-to sets is not empty [27]. Pointer analysis is also useful in many other compiler code optimization algorithms such as loop invariant code motions, auto parallelization etc [39, 63]. Without precise points-to information, these optimizations will need to make conservative assumptions, resulting in some optimizations not being performed.

In addition to the use of pointer analysis for code optimization by compilers, it is also an important component of error detection and other program analysis tools. Pointer analysis has been used for bug detection [20], race detection [19] and detecting vulnerabilities in web programs [32]. In the absence of precise points-to information, these tools make conservative assumptions, thus limiting their impact.
Chapter 1. Introduction

1.4 Our Contributions

The additional precision offered by a flow-sensitive pointer analysis, along with the challenge of scaling this computationally intensive analysis to large programs has lead to the problem being an important research topic in the area of pointer analysis [22, 23, 35, 38, 39, 66]. In this work, we introduce two new methods to further improve the efficiency of flow-sensitive pointer analysis:

1. Parallelization of flow-sensitive pointer analysis:
   - We formulate flow-sensitive pointer analysis as a graph-rewriting problem.
   - We propose an efficient parallel algorithm to solve the graph-rewriting problem. To the best of our knowledge, this is the first successful attempt at parallelizing fully flow-sensitive pointer analysis.
   - We show how our algorithm can be efficiently implemented using a parallel programming framework such as Intel Threading Building Blocks [51].
   - We demonstrate considerable scaling (upto 4.05x) for 8 threads on a set of 10 benchmarks. Compared to the sequential implementation of staged flow-sensitive analysis [23], a single threaded execution of our implementation performs better in 9 out of the 10 benchmarks.

2. An approximation algorithm for flow-sensitive pointer analysis:
   - We analyze the flow-sensitive points-to sets of a few commonly used benchmarks and observe the frequent occurrence of a few sets of objects in the points-to sets of pointers.
   - We propose a technique to identify and merge such frequently occurring object sets. We show that such a summarization of memory locations is safe. We also illustrate why it may lead to a precision loss.
   - We experimentally evaluate our approximation technique on a set of ten previously used benchmarks. We observe a speedup of upto 12.9x, with an average precision loss of 0.25% and 1.40% on two precision measures, respectively.
1.5 Organization of this Thesis

This thesis is organized as follows. Chapter 2 describes a commonly used intermediate program representation used in compilers and on which our flow-sensitive pointer analysis is performed. The chapter also discusses the correctness and precision aspects of pointer analysis and provides a brief summary of the staged flow-sensitive pointer analysis [23].

In Chapter 3 we describe our method for parallelizing flow-sensitive pointer analysis. We begin the chapter with the challenges involved in parallelizing flow-sensitive pointer analysis and then proceed to our graph-rewriting formulation of the analysis, which helps us extract parallelism. We show that our analysis terminates, and always computes the same result. We also describe and evaluate our implementation, providing experimental results on its performance, scaling and comparison with the staged flow-sensitive pointer analysis.

Our approximation method for flow-sensitive pointer analysis, which uses frequent itemset mining [54] to identify frequent object sets that can be approximated, is explained in Chapter 4. We motivate the work by showing the potential speedup that can be achieved by approximating frequent object sets in the points-to information computed by a pointer analysis. We describe in detail how such frequent object sets can be identified and merged during a flow-sensitive pointer analysis. We also illustrate why such an approximation can lead to a precision loss, but is still safe. We evaluate our approximation method by comparing the execution time with the staged flow-sensitive pointer analysis, and by measuring the precision loss with respect to two metrics.

In Chapter 5, we compare our work with some of the earlier works in pointer analysis. We discuss related research on both flow-insensitive and flow-sensitive pointer analysis. The chapter concludes with a discussion on the theoretical complexities of different variations of pointer analysis. We summarize our work and discuss potential future work in Chapter 6.
Chapter 2

Background

We now present the necessary background information on the static single assignment form and the pointer analysis problem. We also discuss the staged flow-sensitive pointer analysis [23] here, which is the starting point for both of our works in Section 3 and Section 4.

2.1 Static Single Assignment Form

Static single assignment (SSA) [15] is a program representation that restricts each variable to have only a single definition. If a variable has multiple definitions, it is split into different variables (sometimes called versions). Whenever more than one definition reaches a point, a \( \phi \) node, which is a special function that indicates multiple reaching definitions, is inserted.

Figure 2.1(a) shows an example program. When this program is rewritten in SSA form, each definition of a variable is assigned a unique number (to differentiate it from other definitions of the same variable). In this example, variable \( x \) has two definitions, each of which will get a different SSA number. Since a use of \( x \) may have two definitions of \( x \) reaching it, a \( \phi \) node is used to capture these multiple reaching definitions, and create a new (virtual) definition of \( x \). The SSA representation of this program is shown in Figure 2.1(b).
2.1. Static Single Assignment Form

The SSA representation of a program (sometimes referred to as sparse representation) is used as the intermediate representation in most modern compilers [36, 48]. The SSA form provides several advantages [5] such as (1) simpler optimization algorithms due to the single definition property, (2) sparser def-use chains/graph and (3) separating unrelated references to the same variable. Our work described in this thesis uses an SSA based intermediate representation of the program as input to the analysis.

Indirect defs and uses through pointers complicate the process of rewriting a program into the SSA form [14]. In an indirect def or use, the actual variable being defined or used is not known (in the absence of points-to information). To avoid this problem, modern compilers such as GCC [48] or LLVM [36] use a partial SSA representation [23]. In partial SSA form, top-level variables (variables whose address is never taken) are placed in the SSA form while address-taken variables are not placed in the SSA form. Top-level variables are referenced directly in the IR. Address-taken variables are referenced only through indirect loads or stores.

Unless otherwise stated, we will use letters at the end of the alphabet to denote top-level variables (e.g., x, y, w in Figure 2.5) and letters at the beginning to denote address-taken variables (e.g., a, b, c in Figure 2.5). If a variable is sub-scripted, it denotes that the variable is in the SSA form. We will also use the terms objects, locations to mean address-taken variables. Top-level variables may also be called pointers.
2.2 Pointer Analysis

Pointer (or points-to) analysis is the problem of determining at compile time, the possible values that a pointer variable may have at run time. Although there are many variations of pointer analysis (as we saw in Section 1.2), most of these analyses share a few common ideas in their design. We now briefly explain these aspects of a pointer analysis that are helpful in understanding how any pointer analysis works.

Pointer analysis usually involves analyzing the following types of statements in a program.

- \( x = &a \) // address-of statement.
- \( x = y \) // copy statement.
- \( x = *y \) // load statement.
- \( *x = y \) // store statement.

In addition to these statements, memory allocation statements (such as \texttt{malloc} or \texttt{new}) also need to be handled. Static memory allocation statements can be processed in different ways. The strategy used to handle allocation of heap memory (i.e., dynamically allocated memory) is sometimes referred to as a \textit{heap model} [23]. Some analyses treat all memory locations (dynamically) allocated by an allocation statement to be a single abstract memory location. Such a strategy has often been used in different pointer analysis algorithms [27,29] and is the one that we use in our analysis.

Correctness

A pointer analysis is said to be correct (or sound) if, whenever there exists an execution sequence of the program in which a pointer \( p \) points to a memory location \( a \), then the pointer analysis includes memory location \( a \) in the points-to set of \( p \) [35].
2.2. Pointer Analysis

Precision

The definition of correctness above only specifies what memory locations must be present in the points-to set of a pointer $p$. A precise pointer analysis is safe and results in a minimal points-to set for each pointer [27, 29]. However, determining the minimal points-to set for each pointer such that the correctness condition is satisfied, is in general, undecidable [55].

The size of points-to sets of pointers computed by a pointer analysis algorithm is a measure of the precision of the algorithm. The points-to set computed by a more-precise algorithm, for each pointer, will be a subset of the points-to set of that pointer as computed by a less precise algorithm. In the worst case (least precise), an analysis might report that every pointer may point to every memory location in the program. A concrete example to illustrate differences in precision between analyses is provided in Section 2.3.

2.2.1 An Example: Andersen’s Analysis

We now explain Andersen’s pointer analysis [3], a flow-insensitive analysis as an example analysis to elucidate the concepts that were presented in the previous section. Since the analysis is flow-insensitive, it computes a single points-to set for each pointer, for the entire program. Let $\text{pts}(p)$ denote the points-to set of a pointer $p$. Then, the goal of the analysis is to compute $\forall p: \text{pts}(p)$.

Andersen’s analysis processes each statement [21] in the program to update the points-to set(s) of one or more pointers or objects as shown in Figure 2.2. Since the analysis is flow-insensitive, there is no specific order in which the statements need to be processed. All statements are processed until a fixed point, i.e., the analysis stops when none of the points-to sets change. The order in which statements are processed does not change the points-to sets computed. It may however affect the number of iterations needed for the analysis to converge [46].

Consider the statements shown in the first column of Table 2.1. Assuming that the statements are processed in that order, the points-to sets of each variable in the program,
Statement | Update | Explanation
--- | --- | ---
$x = \&a$ | $\{a\} \subseteq pts(x)$ | $pts(x)$ is updated to include $a$
$x = y$ | $pts(y) \subseteq pts(x)$ | $pts(x)$ is updated to include $pts(y)$
$x = *y$ | $\forall a \in pts(y), pts(a) \subseteq pts(x)$ | For each $a$ that $y$ may point to, $pts(x)$ is updated to include $pts(a)$
$*x = y$ | $\forall b \in pts(x), pts(y) \subseteq pts(b)$ | For each $b$ that $x$ may point to, $pts(b)$ is updated to include $pts(y)$

Figure 2.2: Updating points-to sets in Andersen’s analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 pts(x)</td>
</tr>
<tr>
<td></td>
<td>p q r a b</td>
</tr>
<tr>
<td>$p = &amp;a$</td>
<td>a</td>
</tr>
<tr>
<td>$q = p$</td>
<td>a a</td>
</tr>
<tr>
<td>$*q = r$</td>
<td>a a</td>
</tr>
<tr>
<td>$r = &amp;b$</td>
<td>a a b</td>
</tr>
</tbody>
</table>

Table 2.1: Andersen’s analysis: an example

after each iteration as computed by Andersen’s analysis is shown. The computation of points-to sets of $p$, $q$ and $r$ are straightforward. In the first iteration, since $r \xrightarrow{p} b$ is not known when processing the store statement ($*q = r$), the points-to set of $a$ (which $q$ points to) does not get updated. In the second iteration, $a$ gets the updated points-to set ($\{b\}$). There is no change in any of the points-to sets during the third iteration (i.e., the analysis has converged), and hence the analysis terminates. Note that there exists no execution order (including multiple executions) of these four statements that can produce any other points-to relation between the variables, other than what is computed by the analysis. Therefore, the points-to sets computed by the analysis for this example program are correct (sometimes stated as safe). An example comparing the precision of this analysis with that of a flow-sensitive analysis (more precise) is given in the next section.
2.3 Flow-Sensitive Pointer Analysis

Flow-sensitivity is a dimension of pointer analysis that determines the points-to sets for pointers at each program point [34]. A flow-sensitive algorithm respects program control flow, and hence computes points-to sets at each program point. On the other hand, a flow-insensitive algorithm, as we saw previously, computes a single global points-to set for each pointer.

Figure 2.3 illustrates the difference in precision between a flow-sensitive (more precise) and a flow-insensitive (less precise) algorithm. The reader may note that the points-to sets for any pointer are the same at all program points in a flow-insensitive analysis. While the flow-sensitive analysis computes the points-to sets for \( z \) at each of the program points 1, 2 and 3 to be \( \{\} \), \( \{a\} \) and \( \{b\} \) respectively, a flow-insensitive algorithm would compute the less precise points-to set \( \{a,b\} \) at all of these program points.

**Strong and Weak Updates**

In a flow-sensitive pointer analysis, there are two possibilities that occur when processing a store statement \( \ast x = y \) [34]: (1) The pointer \( x \) points only to a single object \( a \), in which case the points-to set of \( a \) is replaced with the points-to set of \( y \). This is called a **strong** update. (2) The pointer \( x \) points to more than one object, in which the points-to sets of all of these objects will be updated to include the points-to set of \( y \) along with their existing value. This is called a **weak** update.
In the example program shown in Figure 2.4, when the store statement \( s5 \) is to be processed, the analysis can only determine that \( p \) may point to either of \( a \) or \( b \). This means that either of \( a \) or \( b \) may be assigned to during a run of this program. Hence, the analysis conservatively weak updates the points-to sets of both \( a \) and \( b \) to include \( c \), but does not discard (kill) the points-to information of \( a \) or \( b \) already computed so far. However when processing statement \( s6 \), the analysis can be sure that \( x \) points to only \( a \), and can thus perform a strong update. It can safely discard the points-to set of \( a \) computed so far and conclude that at the end of this statement, \( a \) may point only to \( d \).

2.3.1 A Formal Description

Flow-Sensitive pointer analysis is a forward, non-distributive [34] data-flow analysis. Let \( G = (N, E) \) be a control flow graph (CFG) where \( N \) is the set of statements and \( E \in N \times N \) corresponds to control flow between statements. Let \( \text{pred}(n) \) denote the CFG predecessor(s) of \( n \). Let \( IN_i \) and \( OUT_i \) denote the points-to sets of variables before and after node \( i \). \( IN_i \) and \( OUT_i \) for each node \( i \) are computed as follows:
\[ \text{IN}_i = \bigcup_{j \in \text{pred}(i)} \text{OUT}_j \]

\[ \text{OUT}_i = \text{GEN}_i \cup (\text{IN}_i - \text{KILL}_i) \]

The solution to these equations is the solution to the flow-sensitive pointer analysis problem on a given program. The combined effect of applying \( \text{GEN}_i \) and \( \text{KILL}_i \) when processing a node \( i \) constitutes the transfer function for \( i \). Let the set of variables in the program be \( V \). Then, \( \text{GEN}_i \) and \( \text{KILL}_i \) for a node (statement) \( i \) are computed as shown next. Note that in the equations for \( \text{KILL}_i \) below, for a store statement \((\ast x = y)\), the first case corresponds to a \textit{strong} update, while the second case is a \textit{weak} update.

\[ \text{GEN}_i = \begin{cases} 
\{ x \rightarrow a \} ; x = \& a \\
\bigcup \{ x \rightarrow a \} ; x = y \\
\bigcup \{ x \rightarrow a \} ; x = *y \\
\bigcup \{ b \rightarrow a \} ; *x = y \\ 
\end{cases} \]

\[ \text{KILL}_i = \begin{cases} 
\{ x \rightarrow c \}, \forall c \in V \\
\{ x \rightarrow c \}, \forall c \in V \\
\{ x \rightarrow c \}, \forall c \in V \\
\{ b \rightarrow c \}, \forall c \in V ; \{ x \rightarrow b \} \in \text{IN}_i \land \forall d (\{ x \rightarrow d \} \notin \text{IN}_i \lor b = d) \\
\{ \} ; \text{otherwise} \\
\end{cases} \]
2.4 Staged Flow-Sensitive Pointer analysis

Traditional flow-sensitive pointer analysis as described in the previous section requires storing the points-to information for every pointer variable at each node in the control flow graph (CFG). After application of the transfer function at each CFG node, the information needs to be propagated. Since all uses of the points-to information at a node are not known initially (knowledge of such uses would in turn require complete points-to information), the traditional approach has been to propagate information to all successors.

Consider the example shown in Figure 2.5. Suppose during an analysis, after processing CFG node 1, it is known that $x_1$ may point to variables $a$ or $b$. Since it may not be known what $z_1$ or $y_1$ points to, the points-to sets of $a$ and $b$ at node 1 will need to be propagated to both node 2 and node 3. If it turns out (as the analysis progresses) that $y_1$ may point only to $b$ and $x_1$ only to $a$, then the propagated points-to sets of $a$ at node 2 and $b$ at node 3 will not be useful. Thus, the lack of complete points-to information may lead to unnecessary propagation of data-flow values.

Since the lack of points-to information prevents selective propagation of data-flow values (in this case, the points-to information), Hardekopf et al. [23] proposed the use of an auxiliary analysis (henceforth referred to as AUX) that is less precise (and hence faster) than the main flow-sensitive analysis (henceforth referred to as the primary analysis). Typically, AUX will be a flow-insensitive context-insensitive analysis. Similar to [23], we use Andersen-style [3] (inclusion based) analysis for AUX.

AUX helps in constructing conservative def-use information for address-taken variables. After performing AUX, flow-insensitive points-to information is available. This means that for each load or store statement, the points-to set (as computed by AUX) of the variable dereferenced at the statement forms the set of address-taken variables that may be indirectly referenced there. These possible indirect uses and defs are denoted by the $\mu$ and $\chi$ functions respectively [14]. Treating each $\chi$ as both a def and use of the variable, and $\mu$ as a use of the variable, the address-taken variables can be easily converted to SSA form using any standard SSA conversion algorithm [15]. Once the address-taken
variables are in the SSA form, inferring def-use information is straightforward since SSA names inherently describe def-use information. With the def-use edges in place, points-to information needs to be propagated only along these def-use edges, thus saving both time and memory. The actual flow-sensitive analysis proceeds much like a traditional flow-sensitive analysis, except that information propagation happens only along def-use edges (corresponding to the variable whose data-flow information is being propagated).

Figure 2.5 illustrates the usage of $\mu$ and $\chi$, assuming that the points-to sets for $x_1$, $y_1$, $z_1$ and $w_1$ as computed by AUX are $\{a, b\}$, $\{b\}$, $\{a\}$ and $\{c\}$. The address-taken variables are shown after conversion to SSA form. Def-use edges (built using results from AUX) are shown using dashed lines. In this example, the adoption of def-use edges lead to the points-to set of $a$ being propagated only to node 3 and the points-to set of $b$ being propagated only to node 2.

A client (user) of this analysis would typically have a query as “what is the points-to set of pointer $x$ at program point $m$” [47]. Since the program is assumed to be in partial SSA form, only top-level variables (which are in the SSA form) are referenced directly in the program. Hence, the points-to query from the client can be translated to (1) Find the SSA version (definition) of $x$ that reaches program point $m$. Let this SSA version be $x_1$. (2) Return the points-to set of $x_1$ as computed by the staged flow-sensitive points-to analysis.

We use the the algorithm and implementation of Hardekopf et al. [23] as our reference and will henceforth refer to it as the reference algorithm/implementation. Hardekopf’s analysis is flow and field sensitive but context-insensitive.
Chapter 3

Parallelization

Flow-sensitive pointer analysis is a computationally expensive problem [29]. Therefore, making better use of existing hardware, in particular, multiple processor cores can help reduce the analysis time. Existing methods of flow-sensitive pointer analysis are restricted to be executed on a single processor core. Since a pointer analysis is best performed on the whole program [2], module level parallelism (i.e., analyzing different modules or functions of a program in parallel) would not be useful. Therefore, to exploit multiple processor cores for a flow-sensitive pointer analysis, we propose a parallelization technique to parallelize the state-of-the-art flow-sensitive pointer analysis method, called the staged flow-sensitive pointer analysis [23]. The reader may refer to Section 2.4 for a brief background on the staged flow-sensitive pointer analysis.

Previously, flow-insensitive pointer analysis (Andersen’s analysis) has been parallelized, by Méndez-Lojo et al. [41]. This method involved formulating the analysis as a graph-rewriting problem, enabling easy extraction of parallelism. We extend this approach for flow-sensitive pointer analysis. Our formulation of flow-sensitive pointer analysis as a graph-rewriting problem retains all the key properties of the graph-rewriting technique for flow-insensitive pointer analysis that allow easy extraction of parallelism.

We begin by briefly explaining Méndez-Lojo et al.’s graph-rewriting formulation of Andersen’s analysis and then demonstrate the challenges in using this for a flow-sensitive pointer analysis, accompanied by a brief introduction of our solutions to these challenges.
3.1 Graph-Rewriting

We later describe our technique concretely with a detailed description of the graph-rewriting formulation and an overall algorithm. We also identify and elucidate the sources of parallelism present in the analysis and how our formulation helps in extracting this parallelism.

3.1 Graph-Rewriting

We now give a brief summary of the graph-rewriting based parallelization of flow-insensitive pointer analysis, whose ideas we adopt in our parallelization of the staged flow-sensitive pointer analysis.

Méndez-Lojo et al. [41] introduced the idea of solving flow-insensitive pointer analysis via graph-rewriting. An initial constraint graph is built using the points-to constraints of the program, followed by repeated application of a set of rewrite rules on the graph, till the graph stops changing.

The constraint graph consists of a node for each pointer variable in the program. Corresponding to the four basic types of points-to constraints, there are four types of edges in the graph. There is a one-one correspondence between each edge in the initial constraint graph and each points-to constraint in the original program.

The rewrite rules for Andersen’s analysis are shown in Figure 3.1. Applying a rewrite rule amounts to processing a points-to constraint. After the rewriting terminates (i.e., no more rewrite rules can be applied), the outgoing “p” edges from a node form the points-to set of the variable corresponding to that node.

As an example, consider the constraints in Figure 3.2(a). The initial constraint graph for this is shown in Figure 3.2(b). After applying the copy rewrite rule (Figure 3.1(a)), the graph would transform to what is shown in Figure 3.2(c). The points-to sets for $x$ and $y$ initially are $\{a\}$ and $\{}$. After applying the rewrite rule, the new points-to sets become $\{a\}$ and $\{a\}$ respectively.

Such a formulation of the analysis as a graph-rewriting problem exposes amorphous data parallelism [52] inherent in the analysis, allowing for easy parallelization. In this
Chapter 3. Parallelization

Figure 3.1: Rewrite rules for Andersen’s analysis. Dashed arrows indicate newly added edges. The edge types are: \( p(\text{points-to}) \), \( c(\text{copy}) \), \( l(\text{load-to}) \), \( s(\text{store-from}) \)

\[
\begin{align*}
(a) & \quad & (b) & \quad & (c) \\
\begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\draw (x) edge[->, bend left] node {$p$} (y);
\draw (x) edge[->, bend right] node {$c$} (a);
\end{tikzpicture} & \quad & \begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\draw (x) edge[->, bend left] node {$p$} (y);
\draw (y) edge[->, bend right] node {$l$} (a);
\end{tikzpicture} & \quad & \begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\draw (y) edge[->, bend right] node {$c$} (a);
\end{tikzpicture}
\end{align*}
\]

Figure 3.2: (a) Example points-to constraints, (b) Initial constraint graph, (c) The constraint graph of (b) after applying the \textit{copy} rewrite rule.

\[
\begin{align*}
(a) & \quad & (b) & \quad & (c) \\
\begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\node (a) at (2.5,-2) {$y$};
\draw (x) edge[->, bend left] node {$p$} (y);
\draw (y) edge[->, bend left] node {$p$} (a);
\draw (y) edge[->, bend right] node {$c$} (a);
\end{tikzpicture} & \quad & \begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\draw (x) edge[->, bend left] node {$p$} (y);
\draw (y) edge[->, bend left] node {$p$} (a);
\draw (y) edge[->, bend right] node {$c$} (a);
\end{tikzpicture} & \quad & \begin{tikzpicture}[node distance=2cm,auto]
\node (x) at (0,0) {$x$};
\node (a) at (0,-2) {$a$};
\node (y) at (1.5,0) {$y$};
\draw (x) edge[->, bend left] node {$p$} (y);
\draw (y) edge[->, bend left] node {$p$} (a);
\draw (y) edge[->, bend right] node {$c$} (a);
\end{tikzpicture}
\end{align*}
\]
3.2 Challenges and Solutions

In adapting flow-insensitive graph-rewriting rules (Section 3.1) for flow-sensitive pointer analysis, two major challenges arise: (1) presence of spurious edges and (2) handling strong and weak updates. Hence we begin this section by explaining these two challenges, along with a brief intuitive explanation of our solution.

3.2.1 Presence of Spurious Edges

Consider the CFG in Figure 3.3(a). The CFG has been annotated with $\chi$ and $\mu$ functions using the results from AUX. The flow-insensitive constraint graph corresponding to load constraints in the CFG is shown in Figure 3.3(b). Here, solid edges are part of the initial constraint graph and dashed edges are the ones added by flow-insensitive rewrite rules. The only definition of variable $a$ reaching node 3 of the CFG is $a_1$ (and similarly only $a_2$ reaches node 5). Hence, for a flow-sensitive pointer analysis, the points-to set of $a_2$ should not be included in the points-to set of $y_1$ (and similarly, the points-to set of $a_1$ should not be included in the points-to set of $y_2$). Hence, for a flow-sensitive analysis, the edges $a_1 \rightarrow y_2$ and $a_2 \rightarrow y_1$ should not be added. These edges reduce the precision of the analysis. However, application of the flow-insensitive rewrite rules lead to the addition of these spurious edges (as shown in the figure).

Potential edges: For any edge type, we introduce the concept of a potential edge. A potential edge of type $t$ indicates that there could be an actual edge of type $t$ between the nodes, depending on the information computed by the flow-sensitive analysis as it progresses. Some of the rewrite rules will be modified to look for a potential edge before inserting an actual edge of that type. This will become clear as we explain the rewrite
rules. We use a prefix “p_” on the edge type to denote potential edges of that type\(^1\).

To illustrate the utility of potential edges, we again consider the example in Figure 3.3. Our goal is now to modify the rewrite rules such that the two spurious edges \(a_1 \rightarrow y_2\) and \(a_2 \rightarrow y_1\) do not get added. Only \(a_1 \rightarrow y_1\) and \(a_2 \rightarrow y_2\) should be added as a result of applying the rewrite rules. To achieve this, we add potential copy edges \(a_1 \xrightarrow{p} y_1\) and \(a_2 \xrightarrow{p} y_2\) during construction of the initial constraint graph corresponding to the load constraints. The flow-insensitive load rewrite rule (Figure 3.1(b)) is modified to add a new copy edge only if there already exists a potential copy edge between the two nodes. The two copy edges \(a_1 \rightarrow y_1\) and \(a_2 \rightarrow y_2\) will get added since potential copy edges exist between the two pairs. However, since the edges \(a_1 \xrightarrow{p} y_2\) and \(a_2 \xrightarrow{p} y_1\) are not present, the two spurious copy edges mentioned earlier do not get added.

### 3.2.2 Strong and Weak Updates

The concept of strong and weak updates relates to whether the incoming points-to information for a variable being (indirectly) updated at a store constraint is killed or not (see Section 2.3 for a detailed discussion). Since a flow-insensitive analysis does not deal with strong updates, handling these updates requires additional logic to be added to the graph-rewriting rules.

\(^1\)The prefix “p_” used for potential edges (which can be of any type) should not be confused with the label “p” used for points-to edges.
3.2. Challenges and Solutions

Consider the store node shown in Figure 3.4(a). The flow-sensitive points-to set of $a_1$ depends not only on whether $x_1$ points to $a$ but also on whether $x_1$ points to any other variable $b$ (as computed by the flow-sensitive analysis). If $x_1$ points only to $a$, then $a_1$ will have a strong update, thus getting a “c” edge from $y_1$. If $x_1$ points to both $a$ and $b$, then the previous points-to set of $a$ (which is in the SSA variable $a_0$) also needs to be copied to $a_1$, making the update weak. If $x_1$ does not point to $a$, $a_1$ will have the same points-to set as $a_0$. This logic (w.r.t updating $a_1$) is shown in Figure 3.4(b).

To handle strong and weak updates, we need a way to group all the indirect defs in a store constraint. We do this by introducing Klique nodes, a new node type which is used to connect all these indirect defs. Every store constraint in the original program will have an associated Klique node in the constraint graph. For each address-taken variable that may be (indirectly) defined in that store, we add a Klique edge (denoted by the letter “k”)\(^2\) from the corresponding SSA version of that variable to the corresponding Klique node. For example, for the store constraint in Figure 3.4(a), a new Klique node is created and Klique edges are added from $a_1$ and $b_1$ to this Klique node. Thus, we have connected all indirect defs in a store constraint. The rewrite rules in the next sub-section will make it clear as to how this helps us perform a weak or a strong update.

\(^2\)Note that we use the term Klique for both nodes and edges. A Klique edge will always be directed to a Klique node.
Chapter 3. Parallelization

3.3 Algorithm

With the intuition given in the previous section to overcome the two challenges for a graph-rewriting formulation of flow-sensitive pointer analysis, we now present the various components of our algorithm, namely the graph structure (node and edge types) and the rewrite rules. We also give the overall algorithm in Section 3.3.3

3.3.1 Graph Structure

Node types: The constraint graph for flow-sensitive analysis can have the following types of nodes:

- **TopLvl**: These nodes correspond to the top-level variables in the program. Top-level variables are expected to be in the SSA form [15]. Since the SSA form guarantees a single definition, the outgoing “p” edges of a TopLvl node will represent the flow-sensitive points-to set of the corresponding top level variables.

- **AddrTakenSSA**: There is one AddrTakenSSA node for each SSA version of each address-taken variable. Since each possible (indirect) definition of an address-taken variable will have its own SSA version, the outgoing “p” edges of these nodes will represent the flow-sensitive points-to sets of the corresponding address-taken variables.

- **NonSSA**: These nodes correspond to address-taken variables in the original program, before converting them to SSA form. We will use these nodes as a representative of all AddrTakenSSA nodes that are from the same address-taken variable. All “p” edges will be directed towards these nodes.

- **Klique**: These nodes are useful to handle strong and weak updates. They are artificial nodes that are introduced to recognize the set of address-taken variables that may (indirectly) be defined at a given store constraint.

As remarked earlier, SSA variables will be sub-scripted with a number.
3.3. Algorithm

Apart from the four edge types (p, c, l and s) that are used in flow-insensitive analysis, we have the following additional edge types:

- **Uses (u)**: In a load constraint, when it is known (as the flow-sensitive analysis progresses) that the TopLvl node being dereferenced points to an address-taken variable, a Uses edge is added from the TopLvl node to the corresponding AddrTakenSSA node. Note that the AddrTakenSSA node will have a $\mu$ function in that load constraint.

- **Defines (d)**: In a store constraint, when it is known (as the flow-sensitive analysis progresses) that the TopLvl node being dereferenced points to an address-taken variable, a Defines edge is added from the TopLvl node to the corresponding AddrTakenSSA node. Note that the AddrTakenSSA node will be on the LHS of a $\chi$ function in that store constraint.

- **Klique edge (k)**: These edges connect AddrTakenSSA nodes on the LHS of $\chi$ functions in a store constraint to a Klique node that is unique to that store constraint.

- **NonKilledCopy (n)**: These edges connect the AddrTakenSSA node on the RHS of a $\chi$ function to the corresponding AddrTakenSSA node on the LHS. They are useful to perform weak updates.

- **AlreadyDefined (a)**: AddrTakenSSA nodes in a store constraint that are already defined (i.e., they have an incoming Defines edge) are marked by having an AlreadyDefined edge from the Klique node of that store constraint.

- **SSAParent (ssa-parent)**: Each AddrTakenSSA node will have exactly one SSA-Parent edge to identify the NonSSA node that corresponds to it.

Some of the edge types above can also be seen as properties of nodes. For example, the SSAParent edge from a node can be considered as a constant property of that node. We present them as edges for the sake of a pure graph formulation.
**Initial constraint graph:** The initial constraint graph is built from points-to constraints in the input program. The nodes and edges added for each type of constraint is shown in Figure 3.5.

The initial constraint graph for address-of constraints (Figure 3.5(a)) captures a direct points-to relation (“p” edge). Figure 3.5(b) shows the initial constraint graph for a copy constraint. The “c” (copy) edge indicates that the points-to information of the RHS variable needs to be copied over to (included in) the points-to set of the LHS variable.

The initial constraint graph for load constraints is shown in Figure 3.5(c). The graph indicates that \( a_1 \) may potentially be used (“p\_u” edge) in this load (“l” edge) and that there is a potential copy from \( a_1 \) to \( x_1 \) (“p\_c” edge). As mentioned earlier, potential edges are used here to avoid adding spurious edges during the analysis. The initial constraint graph for \( \phi \)-nodes (Figure 3.5(e)) just consists of “c” (copy) edges indicating that the destination of the \( \phi \) will get points-to information from each of the argument.

The initial constraint graph for a store constraint (Figure 3.5(d)) consists of the following components:

1. The top-level variable (\( x_1 \) here) potentially defines (“p\_d”) the address-taken taken variables (\( a_1 \) and \( b_1 \)) on the LHS of a \( \chi \) function.

2. The store edge, labelled “s” (here \( x_1 \) stores from \( y_1 \)).

3. Due to the store (from \( y_1 \)), there is a potential copy (“p\_c”) from the RHS top-level variable (\( y_1 \)) to each variable that may potentially be defined (\( a_1 \) and \( b_1 \)).

4. AddrTakenSSA nodes on the RHS of \( \chi \) functions (\( a_0 \) and \( b_0 \)) have a NonKilledCopy edge (labelled “n”) to the corresponding nodes (\( a_1 \) and \( b_1 \) respectively) on the LHS of the \( \chi \) function. Upon a weak update, this edge leads to a copy edge.

5. Klique nodes and edges (labelled “k”) are added to group together variables that may (indirectly) be defined in this constraint (here, \( a_1 \) and \( b_1 \) are connected to the Klique node \( k_1 \)). As mentioned, this grouping helps in performing weak updates.
Figure 3.5: Initial graph for different types of constraints. In (a), $k_1$ is a Klique node. In (e), $v_1$, $v_2$, $v_3$ may be either top-level or address-taken.
3.3.2 Rewrite Rules

Once the initial constraint graph is constructed from the input program as shown in Figure 3.5, the analysis needs to be solved. The rewrite rules to solve the analysis are shown in Figure 3.6. We apply these rewrite rules till no more edges are added to the graph, indicating a fixed point. At the end of the analysis, the “p” edges emanating from a node form the points-to set for the variable corresponding to that node.

Rules (a), (b) and (c) (Figure 3.6) are adaptations of the three flow-insensitive rewrite rules, taking spurious edges into account. These rules now look for a potential edge before adding an actual edge. Rules (d) and (e) add Defines and Uses edges based on whether the TopLvl node being dereferenced points to the concerned address-taken variable. This requires knowing the SSAParent of AddrTakenSSA nodes that are potentially defined or used (in χ or µ functions). Rules (f) and (g) handle weak updates. Rule (f) marks a node belonging to a Klique as already defined (i.e., it has an incoming Defines edge). This is the first step in performing a weak update. For an AddrTakenSSA node on the LHS of a χ function, if another AddrTakenSSA node in the same Klique is already defined, then the value on the RHS of the χ function needs to be copied. This is carried out by rule (g), completing the weak update process. Thus, rules (c), (f) and (g) together handle the logic for strong and weak updates (Figure 3.4).

Similar to the flow-insensitive rewrite rules in [41], our rewrite rules have the property that they may be applied on different nodes in parallel, as long as the underlying data structure that represents edges can handle concurrent updates.

3.3.3 Putting It All Together

A summary of the steps in our method is presented here.

1. Convert TopLvl variables to SSA form. Most compilers already have an SSA based representation, and hence this step may not be necessary.

2. Perform AUX. The less precise points-to information computed by AUX is used to annotate the program with χ and µ functions.
Figure 3.6: Rewrite rules for flow-sensitive pointer analysis
3. Convert the *address-taken* variables (which are in the $\chi$ and $\mu$ functions) to SSA form using any standard SSA conversion algorithm.

4. Build the initial constraint graph according to Figure 3.5.

5. Apply the flow-sensitive graph rewrite rules (Figure 3.6), in parallel, repeatedly till fixed point.

6. The points-to set for each *top-level* variable can be obtained from the corresponding *TopLvl* graph node. The points-to ("p") edges from this node form the points-to set. Since only *top-level* variables are directly referenced in partial SSA form, the points-to sets of *address-taken* variables are not needed after the algorithm terminates.

### 3.3.4 Properties of Our Graph-rewriting System

In this section, we show that our algorithm described in Section 3.3.3: (1) Terminates. (2) Computes a unique solution whose precision is equivalent to that of the reference algorithm. (3) Easily exploits parallelism.

**Theorem 3.3.1** The algorithm described in the previous section terminates in a finite number of steps.

*Proof.* Since the total number of edges that can be added to the graph is finite and we add at least one edge to the graph in each iteration (we stop if no edges are added in an iteration), the algorithm terminates after a finite number of iterations. □

**Theorem 3.3.2** The fixed point (final graph) obtained when the algorithm terminates is unique.

*Proof.* Similar to the rewrite rules in [41], our rewrite rules are locally confluent. Combining this with the result of Theorem 3.3.1, we can conclude (by Newman’s lemma [7]) that the rewriting system is globally confluent, and hence the final graph (fixed point) is unique. □
Theorem 3.3.3 The results computed by our algorithm is equivalent to that of the reference algorithm.

Proof. Every TopLvl node in our graph corresponds to a top-level variable. The points-to sets for these variables are flow-sensitive since they are in the SSA form. For address-taken variables, the reference algorithm stores the points-to sets at each program point where the variable is in a $\chi$ or $\mu$ function. We achieve the same by having different nodes for each SSA version of the address-taken variable, with these SSA versions corresponding to the use/def of the variable in a $\chi$ or $\mu$ function. Actual data-flow propagation in the reference algorithm happens through def-use edges built using the SSA names of variables. However, our method does not have explicit def-use edges. The node for a variable holds the points-to information. Update of points-to set of a variable happens through incoming copy edges to the node (that corresponds to the variable) in the constraint graph. These copy edges are added wherever points-to set propagation happens in the reference algorithm. $\square$

Observation 3.3.1 Our formulation of the analysis exposes parallelism.

In the analysis performed by the reference algorithm, any points-to constraint that has new incoming (IN) information can be processed independently of other constraints. This means that there are multiple items on the worklist that can be processed in parallel. The second source of parallelism in the analysis is the update of variables in a single constraint. Since a load or store constraint can involve more than one address-taken variable, each of them can be updated independently of the other while processing the constraint. Our graph formulation exposes both these sources of parallelism in a natural way. Any two graph nodes can be processed in parallel. Synchronization is required only when two threads simultaneously try to update edges of the same node. Such a parallelism in which different nodes in the graph can be processed in parallel, but with certain constraints, is called amorphous data-parallelism [52].
3.4 Implementation and Optimizations

We implemented our parallel algorithm in C++, using Intel Threading Building Blocks [51] (referred to as TBB henceforth) for parallel work management.

The initial constraint graph for all benchmarks were generated using Ben Hardekopf’s LLVM implementation of the staged flow-sensitive pointer analysis [23]. Hence, our implementation has the same properties (w.r.t various dimensions of pointer analysis) as Ben Hardekopf’s implementation. Our implementation is context-insensitive, field-sensitive and flow-sensitive.

Edge representation: In our graph-rewriting approach, although different nodes can be processed independently, synchronization is required whenever two threads try to add edges to the same node. To handle this efficiently, we use concurrent data structures that allow parallel updates by different threads. Although Binary Decision Diagrams (BDDs) are known to be well suited for representing points-to sets [6], we were unable to obtain a good concurrent BDD implementation. So we used the concurrent_unordered_set data structure provided by TBB in our implementation. This is a hash table based data structure that allows concurrent addition of set elements.

_Potential_ edges can be represented in two ways. Since they lose importance as soon as an actual edge of the same type is added between the two nodes, one way to represent them would be to have two bits per edge. 00 represents no edge, 01 represents a _potential_ edge and 10 (or 11) can represent an actual edge. The other way to implement these edges is to treat them as a different edge type altogether. We use the latter approach in our implementation. In such an implementation, there is an option to delete _potential_ edges once an actual edge has been added in its place. Deleting may improve performance since deleted edges will not be redundantly checked when applying a rewrite rule.

Worklist approach: Although it is possible to try and apply each rewrite rule to all nodes until no more edges are added to the graph, it is very inefficient. We use a worklist based approach to apply rewrite rules. “p” edges in the initial constraint graph are used to build the initial worklists. Any new edge that is added during the analysis can trigger more nodes to be added to one or more worklists. For example, a new “p”
edge can trigger the rules (a), (d) and (e) in Figure 3.6.

We employ a dual worklist based approach (similar to double buffering [4]) to add and remove items from the worklists. Rewrite rules are applied to nodes on the current worklist, while we add new nodes (which need processing) to the next worklist. Once the current worklist becomes empty, the current and next worklists are swapped and the process repeats. This is done until both the worklists are empty. We implement worklists using vectors provided by the C++ Standard Template Library. Further, the next worklist is maintained per thread (as thread local storage) and at the end of an iteration, before swapping next and current, we combine the per thread worklists. The current worklist is a global worklist. We use a per node flag to determine if a node is in a worklist or not. We rely on the parallel_for construct of TBB to manage work scheduling. TBB implements work-stealing [51] [9] to manage parallel workload.

**Incremental updates:** When new “p” edges are added to a node, the node gets added to the worklist of rewrite rule (a) (in Figure 3.6). When this node is processed again in the next iteration to apply the rule, only the new “p” edges need to be propagated (through the “c” edges). Points-to edges that were added before the previous iteration can be ignored here. This optimization can be applied to other rewrite rules as well. Thus it helps to maintain newly added edges as a delta over existing edges. This idea has already been used by Méndez-Lojo et al. in their GPU implementation of Andersen’s analysis [40]. We adapt this idea to our approach. This optimization ensures that no redundant work is carried out when a rewrite rule is applied.

## 3.5 Evaluation

In this section, we evaluate our algorithm and implementation by comparing it to the current state-of-art algorithm\(^3\) by Hardekopf et al. [23]. We first show some properties of the benchmarks that we used followed by the performance results.

We used a dual socket 8-core machine with 16GB of memory, running Debian Linux

\(^3\)http://www.cs.ucsb.edu/~benh/research/downloads.html
Chapter 3. Parallelization

Table 3.1: Number of nodes of each type

<table>
<thead>
<tr>
<th>benchmark</th>
<th>TopLvl</th>
<th>NonSSA</th>
<th>AddrTakenSSA</th>
<th>Klique</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>9852</td>
<td>1229</td>
<td>7953</td>
<td>148</td>
</tr>
<tr>
<td>254.gap</td>
<td>45946</td>
<td>2393</td>
<td>635639</td>
<td>469</td>
</tr>
<tr>
<td>176.gcc</td>
<td>114994</td>
<td>5908</td>
<td>255774</td>
<td>1770</td>
</tr>
<tr>
<td>nethack</td>
<td>85994</td>
<td>11977</td>
<td>124448</td>
<td>223</td>
</tr>
<tr>
<td>197.parser</td>
<td>8514</td>
<td>1020</td>
<td>3631</td>
<td>129</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>50538</td>
<td>2829</td>
<td>270833</td>
<td>543</td>
</tr>
<tr>
<td>sendmail</td>
<td>45155</td>
<td>4136</td>
<td>22220</td>
<td>347</td>
</tr>
<tr>
<td>svn</td>
<td>99181</td>
<td>8740</td>
<td>6328901</td>
<td>2738</td>
</tr>
<tr>
<td>vim</td>
<td>238031</td>
<td>8935</td>
<td>1017678</td>
<td>724</td>
</tr>
<tr>
<td>255.vortex</td>
<td>17910</td>
<td>3304</td>
<td>12138</td>
<td>107</td>
</tr>
</tbody>
</table>

6.0, to conduct these experiments. The machine had version 4.0 of Intel Threading Building Blocks [51] installed.

We use a subset of benchmarks from SPEC2006 [26] that are relatively larger (in terms of lines of code). Program names with a numbered prefix are from SPEC2006. In addition, we also use a few other programs that have all been used in previous studies [23] [41]. Ex is a text processor, Nethack is a text based game, Sendmail is an email server, Svn (subversion) is a revision control system and Vim is a text editor.

Table 3.1 shows the number of nodes (of each type) present in the initial constraint graph. Table 3.2 shows the average (across iterations) number of nodes in each worklist for each benchmark. The worklists (a) to (g) correspond to rewrites rules from (a) to (g) in Figure 3.6. Worklist (h) corresponds to the rewrite rule for processing pointer arithmetic statements. While these columns give us some idea of the extent to which the analysis for the benchmarks can be parallelized, it is important to note that not all of it can be exploited fully. There may be contention among threads (when applying a rewrite rule) to access the same node, which may slow down the accesses. Note that although we talk about different worklists for different rewrite rules, this is only for conceptual clarity. In our implementation we combine work-items from different worklists.
3.5. Evaluation

Table 3.2: Average number of nodes in each worklist

<table>
<thead>
<tr>
<th>benchmark</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
<th>(f)</th>
<th>(g)</th>
<th>(h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>86</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td>11</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>254.gap</td>
<td>335</td>
<td>23</td>
<td>20</td>
<td>20</td>
<td>23</td>
<td>20</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td>176.gcc</td>
<td>220</td>
<td>76</td>
<td>27</td>
<td>27</td>
<td>76</td>
<td>27</td>
<td>23</td>
<td>72</td>
</tr>
<tr>
<td>nethack</td>
<td>1147</td>
<td>270</td>
<td>23</td>
<td>23</td>
<td>275</td>
<td>23</td>
<td>278</td>
<td>956</td>
</tr>
<tr>
<td>197.parser</td>
<td>57</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>9</td>
<td>16</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>106</td>
<td>15</td>
<td>9</td>
<td>9</td>
<td>16</td>
<td>9</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>sendmail</td>
<td>225</td>
<td>31</td>
<td>17</td>
<td>17</td>
<td>32</td>
<td>17</td>
<td>31</td>
<td>22</td>
</tr>
<tr>
<td>svn</td>
<td>713</td>
<td>180</td>
<td>82</td>
<td>82</td>
<td>180</td>
<td>82</td>
<td>315</td>
<td>26</td>
</tr>
<tr>
<td>vim</td>
<td>4638</td>
<td>81</td>
<td>34</td>
<td>34</td>
<td>85</td>
<td>34</td>
<td>625</td>
<td>66</td>
</tr>
<tr>
<td>255.vortex</td>
<td>260</td>
<td>40</td>
<td>10</td>
<td>10</td>
<td>40</td>
<td>10</td>
<td>151</td>
<td>64</td>
</tr>
</tbody>
</table>

3.5.1 Comparison with SFS

Figure 3.7 shows the performance of our graph-rewriting implementation w.r.t SFS. The speedup shown here is for a single threaded execution of our implementation. It is to be noted that the difference in execution time seen here is an artifact of the data structure that we used to represent points-to sets. While SFS uses BDDs to represent points-to sets, we use a hash table based data structure provided by TBB. For the smaller benchmarks (Table 3.1), our use of hash table based data structures to represent points-to sets results in faster execution. However for the larger benchmarks such as svn, the execution times are similar. For vim, our graph-rewriting implementation is slower. This is due to the higher density of the points-to graph (more edges at each node) of vim. Although BDDs are known to scale well for such programs (by exploiting regularity in the points-to sets) [6], we were unable to use them for our work since we require a concurrent data structure to represent edges in our graphs.

3.5.2 Scaling

Figure 3.8 shows thread scaling of our parallel graph-rewriting implementation of flow-sensitive pointer analysis. We observe significant scaling for upto four threads on most benchmarks. However, some of the benchmarks (such as parser) do not scale well. As expected, the number of nodes on each worklist (Table 3.2) for a benchmark limits the
amount of parallelism that can be extracted. For example, \textit{svn}, \textit{vim} and \textit{nethack} show good scaling as they have a larger number of nodes (on an average) on their worklists (\textit{svn} and \textit{vim} also scale to eight threads easily). On the other hand, \textit{parser} and \textit{perl} have relatively lower number of nodes (on an average) on the worklists, and hence do not show good scaling.

\textbf{Discussion}

Although the thread scaling of our method has a correlation with the average number of workitems on the worklist (as pointed out in the previous section), we also analyzed a few of these benchmarks using Intel VTunes to verify that the poor scaling is due to synchronization overhead between threads. VTunes provides a count of the number of waits (conflicts) that were incurred during an execution of the program. Although the absolute number of conflicts reported by VTunes cannot be used to infer synchronization overhead, the increase in number of conflicts, as the number of threads increase, can be an indicator of the synchronization overheads.

Figure 3.9 shows the increase in number of conflicts (with the number of conflicts for a 2-threaded execution as the base) as the number of threads increase, for \textit{parser}, \textit{ex}, \textit{nethack} and \textit{vim} (the first two scaled poorly while the second two scaled well). While for

Figure 3.7: Comparison of 1-thread execution with SFS
parser and ex, the number of conflicts grew sharply as we moved to 4 and 8 threads, for vim and nethack, the increase in number of conflicts is lesser. This scaling of conflicts with threads correlates to the scaling of running time that we saw in Figure 3.8.

3.6 Chapter Summary

In this work, we identified two sources of parallelism in flow-sensitive pointer analysis and introduced a graph-rewriting formulation of the analysis that can easily extract this parallelism. Our algorithm is efficient and easy to implement. It scales considerably upto 8 threads.
Figure 3.9: Scaling of number of conflicts
Chapter 4

An Approximation Algorithm

In this chapter, we propose an approximation algorithm for flow-sensitive pointer analysis. Our algorithm is based on certain characteristics of the points-to sets computed by a flow-sensitive pointer analysis. We observe that some sets of objects appear frequently in the points-to sets of many pointers. Propagation of these frequent object sets contributes to a significant part of the total analysis time. Therefore, we focus our work on optimizing the time spent by the analysis in propagating these frequent object sets.

We use a known data-mining technique, frequent itemset mining, to identify frequent object sets in the points-to sets of pointers. Since the points-to sets of the flow-sensitive analysis are not known till the completion of the analysis itself, we use the points-to sets from a less precise (but fast) flow-insensitive analysis to determine the frequent object sets. We then approximate each of the frequent object set into a single summary object. Such an approximation leads to only the summary object being propagated in place of the entire set of objects, thus, reducing the analysis time significantly. We call our technique as FOSM (Frequent Object Set Merging). Although FOSM incurs a slight loss in precision, it is shown to be safe.

We motivate this work (Section 4.1) by our observation of the occurrence of frequent object sets in the points-to sets of a flow-sensitive pointer analysis, and the potential improvement in efficiency possible by taking advantage of this phenomenon in computing flow-sensitive points-to sets. Section 4.3 gives details on how we actually take advantage
of these frequent object sets, along with a discussion on the correctness and precision aspects of such an approach. We evaluate our algorithm, both in terms of improvements in efficiency, and in terms of precision loss in Section 4.4.

4.1 Motivation

A key observation made by us in this work is that there are object sets that occur together and frequently in the points-to sets of many pointers [17, 64]. We motivate this with the help of data obtained from the staged flow-sensitive analysis [23], which is flow and field-sensitive, but context-insensitive. We analyze the results of this analysis for a few benchmarks to find frequently occurring object sets (i.e., sets of memory objects that are pointed-to by many pointers).

Table 4.1 details the number of frequent objects in the points-to sets of each benchmark along with their frequency of occurrence. The object sets here are all maximal\(^1\). Some benchmarks had more than one such maximal frequent object set. Except in the case of ex and python, the sizes of other frequently occurring sets were insignificant. Hence we report only the most frequently occurring set here (although we do take advantage of the others in our implementation).

The frequency of occurrence of the object sets reported here is computed as a percentage of the number of occurrences of the object set in the points-to sets of top-level variables. Additionally, these object sets may appear in the points-to sets of many memory objects (address-taken variables) at various program points too. Hence optimizing the time spent by the analysis in propagating points-to information related to frequent object sets is likely to speed up the analysis.

Figure 4.1(a) is a plot of the number of pointers that contain the frequent object sets in their points-to sets, as the analysis progresses. The progress of the analysis is measured (x-axis) in terms of the number of worklist items processed. The graph (plotted for staged flow-sensitive analysis (SFS) [23] on the benchmark “vim”), shows

\(^1\)The term “maximal” is defined in Section 4.2
that after 150,000 worklist items are processed, the frequent object sets start to form and their growth and propagation is steep (happens between 185,000 to 225,000 worklist items).

What happens if the analysis is seeded with the frequent object set information, right at the beginning? We plot the cumulative execution time of the analysis against the same measure of progress (number of worklist items processed) in Figure 4.1(b). Using the oracle information of frequent object sets in SFS, we plot the execution time for SFS without and with frequent object set optimization. Without optimization, as the number of pointers containing the frequent object sets (in their points-to sets) increase, the analysis time also starts to increase. This is mainly due to the large sizes of these frequent object sets, propagation of which is an expensive operation. SFS with frequent object set information abstracts large frequent object sets as single objects, and hence is able to reduce the time taken to propagate them. Another interesting point here is that, after optimizing, the analysis converges faster, i.e., there are fewer worklist items processed.

This motivates that identifying the group of objects that occur frequently and abstracting them as a single object can speedup the analysis. But, how do we identify these object sets? This problem is the well known frequent itemset mining problem [54]. In the following section, we present the necessary background for this.

Table 4.1: Details of frequently occurring objects

<table>
<thead>
<tr>
<th>benchmark</th>
<th>num_freq_objects</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>34 (1.4%)</td>
<td>2.8%</td>
</tr>
<tr>
<td>176.gcc</td>
<td>157 (1.1%)</td>
<td>19.6%</td>
</tr>
<tr>
<td>nethack</td>
<td>412 (2.1%)</td>
<td>14.1%</td>
</tr>
<tr>
<td>254.gap</td>
<td>615 (7.2%)</td>
<td>26.2%</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1353 (29.4%)</td>
<td>30.5%</td>
</tr>
<tr>
<td>vim</td>
<td>3368 (1.9%)</td>
<td>6.6%</td>
</tr>
<tr>
<td>python</td>
<td>5598 (25.5%)</td>
<td>20.3%</td>
</tr>
<tr>
<td>svn</td>
<td>6131 (25.3%)</td>
<td>18.1%</td>
</tr>
<tr>
<td>pine</td>
<td>22053 (5.4%)</td>
<td>16.5%</td>
</tr>
<tr>
<td>gdb</td>
<td>15164 (20.8%)</td>
<td>18.4%</td>
</tr>
</tbody>
</table>
Figure 4.1: Progress of the analysis vs (a) Growth of frequent object sets (b) Cumulative time: with and without optimizing frequent object sets.
4.2 Frequent Itemset Mining

Let $U$ be a set of items. A transaction is a subset of $U$. The term arises from a set of items bought in a transaction (sometimes the term basket is used instead of transaction). A Frequent Itemset $I$ is a set of items that appears in (is a subset of) at least $s$ transactions, where $s$ is called the support threshold. The frequency of occurrence of $I$ is called its support. An itemset is considered maximal if no proper superset of the itemset is frequent, i.e., appears more than the support threshold.

Typically, the number of transactions is high, and the size of a transaction (i.e., number of items in the transaction) is smaller. Given a set of transactions, the problem of finding all itemsets whose support is at least a specified support threshold is called frequent itemset mining [54].

Most algorithms for frequent itemset mining rely on the monotonicity property: If a set $I$ of items is frequent, then so is every subset of $I$. A well known algorithm for frequent itemset mining is the apriori algorithm [1]. The algorithm performs multiple passes on the set of transactions, computing frequent itemsets of increasing sizes in each pass. It utilizes the set of frequent items from the previous pass to obtain a pruned set of candidates for the current pass. Although historically significant, the apriori algorithm is generally considered inefficient and several newer algorithms have been proposed.

In this work, we use the ECLAT [67] algorithm to find frequent itemsets. This algorithm employs better heuristics (such as equivalence class clustering) to quickly determine maximal frequent itemsets. In our work, we are only interested in maximal frequent itemsets. In our problem context, the set of all memory objects (address-taken variables) form the universe $U$ of items. The points-to sets of pointers form the transactions. We aim to find those object sets that occur frequently in the points-to sets of many pointers.

4.3 Frequent Object Set Merging (FOSM)

The frequent occurrence of some memory objects in the points-to sets of pointers indicates that modifying the analysis to handle these object sets better can make the analysis faster.
A simple way to take advantage of these frequent object sets is to consider all objects in a frequently occurring set as a single abstract summary location. The only requirement on such a summary location is that it may not be subjected to strong updates [18] (see Section 2.3 for a discussion on strong and weak updates). Other correctness aspects of merging memory objects into a single summary location are discussed in Section 4.3.1.

To merge sets of frequent memory objects into a single location, we need to know which object sets are frequent. One way to gather this information is to keep track of pointers that have similar points-to sets during the analysis. A similarity measure such as the Jaccard similarity [54] can be used to measure similarity between two sets. Jaccard similarity is measured as the size of intersection of sets divided by the size of their union. i.e., the similarity measure of two sets $A$ and $B$ is defined as $|A \cap B|/|A \cup B|$, where $|A|$ denotes the cardinality of set $A$. The similarity value can range from 0 (dissimilar) to 1 (identical). As the analysis progresses, objects pointed-to by similar pointers, satisfying certain properties (such as a minimum size for its points-to set) can be merged into a single object. Such an approximation has already been experimented with for Andersen’s inclusion based (flow-insensitive) analysis [43]. We discuss some disadvantages of such an approach in Section 5.

Instead of incrementally determining objects that need to be merged, we determine candidate objects for merging and merge them even before the analysis begins. However, the data on frequent object sets (as mentioned in Section 4.1) is not available until the actual analysis itself finishes running. We therefore need an approximation for it in order to do the merging prior to the analysis. Since the flow-sensitive analysis we use is a staged analysis [23], the frequent object sets from the result of a less precise analysis (in this case flow-insensitive Andersen’s analysis) can be used to approximate the frequent object sets of the flow-sensitive analysis.

We found out experimentally that, although the frequency (support) of the frequent object sets vary between the flow-insensitive analysis and the flow-sensitive analysis (this is expected as the two analyses compute results with different precisions), the frequent object sets themselves are almost the same. Table 4.2 shows the similarity between the
4.3. Frequent Object Set Merging (FOSM)

Table 4.2: Similarity between the frequent object sets of the flow-insensitive and the flow-sensitive pointer analyses

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Jaccard Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex</td>
<td>1</td>
</tr>
<tr>
<td>176.gcc</td>
<td>1</td>
</tr>
<tr>
<td>nethack</td>
<td>0.99</td>
</tr>
<tr>
<td>254.gap</td>
<td>1</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1</td>
</tr>
<tr>
<td>vim</td>
<td>0.98</td>
</tr>
<tr>
<td>python</td>
<td>0.93</td>
</tr>
<tr>
<td>svn</td>
<td>0.92</td>
</tr>
<tr>
<td>pine</td>
<td>0.95</td>
</tr>
<tr>
<td>gdb</td>
<td>0.75</td>
</tr>
</tbody>
</table>

The most frequent frequent object set of the flow-insensitive analysis and that of the flow-sensitive analysis for each benchmark. The similarity measure used here is the previously mentioned Jaccard similarity. As can be seen from the table, the similarity between maximal frequent object sets of a flow-sensitive and a flow-insensitive analysis is very high, greater than 90% for most of the benchmarks.

Thus, we analyze results from the flow-insensitive analysis to find frequent object sets and use this as an approximation to merge the objects before the flow-sensitive analysis begins. The flow-sensitive algorithm itself remains as before, but now works on a modified input in which frequent object sets are replaced by their summary objects. We use the ECLAT [67] algorithm to determine frequent objects. An overview of our approach is shown in Algorithm 1.

4.3.1 Correctness

We now establish the correctness of our algorithm (FOSM) where objects in the frequent object sets are merged. We claim that the points-to set computed by a flow-sensitive analysis after merging objects is a superset of the points-to set computed without merging, for each pointer in the program, at every program point.

Let \( o_1 \) and \( o_2 \) be two objects that are to be merged (which is done prior to the main
Algorithm 1 High level description of Frequent Object Set Merging (FOSM)

1: procedure FrequentObjectSetMerging
2:    Perform Andersen’s flow-insensitive points-to analysis (first step in SFS)
3:    Use ECLAT to identify maximal frequent object sets in the results computed
      by Andersen’s analysis
4:    Build data-flow graph, required as input to SFS
5:    For each frequent object set $f$, define a summary object $o_f$
6:    For each frequent object $o \in f$, replace $o$ by $o_f$ in the data-flow graph
7:    Run the main analysis of SFS using the modified data-flow graph as input
8: end procedure

analysis), into a summary object $o'$. This means that every reference to both $o_1$ and $o_2$ in
the program will be replaced\(^2\) with $o'$. During the analysis, any object that would have
been added to the points-to set of $o_1$ (or $o_2$) at a program point $p$, will now be added to
the points-to set of $o'$ at that program point. In other words, the points-to set of $o'$ at a
program point $p$ will include the points-to sets of both $o_1$ and $o_2$ at that point.

A similar argument can also be applied to pointers (top-level variables). Any pointer
that would have pointed to either of $o_1$ or $o_2$ (or both) would now point to the summary
object $o'$. This can be interpreted as the pointer including both $o_1$ and $o_2$ in its points-to
set.

Last, since it may be unsafe to perform strong updates on abstract objects that may
represent more than one concrete memory location (such as arrays and heaps) [18], we
perform only weak updates on the merged summary object, as stated in the previous
section.

4.3.2 Precision

Here, we give examples to illustrate why merging memory objects into a single summary
location can lead to loss in precision of the final points-to results.

1. Consider two pointers $p, q$ for which a pointer analysis computes the points-to sets,
   respectively as $\{o_1\}, \{o_2\}$. Suppose we merge $o_1$ and $o_2$ into a single object, say $o'$
\(^2\)It may be desirable to keep track which object is replaced by which summary object in a real
implementation, but it does not affect our discussion here.
before the analysis. The analysis would now compute both $p$ and $q$ as having the same points-to set $\{o'\}$. Thus the two pointers which didn’t alias earlier now do.

2. Extending the previous example, suppose at a program point $p_1$, the points-to sets of $o_1$ and $o_2$ as computed by a pointer analysis are $\{o_3\}$ and $\{o_4\}$. Merging $o_1$ and $o_2$ would cause the merged object $o'$ to have its points-to set as $\{o_3, o_4\}$. Thus, a use of either of $o_1$ or $o_2$ in a load/store will now propagate the combined points-to set, resulting in a loss of precision.

These examples show that (1) pointers that point to objects being merged will lose precision, and (2) points-to sets at each program point, of the objects being merged may lose precision.

### 4.3.3 Implementation

We build on Hardekopf’s LLVM implementation of the staged flow-sensitive analysis \(^3\). After the first phase of the analysis (i.e., Andersen’s flow-insensitive analysis), we use the points-to sets computed by it as input to the frequent itemset mining algorithm. We use a publicly available implementation\(^4\) of the ECLAT algorithm to find frequent object sets in the points-to results of the flow-insensitive pointer analysis. This implementation [10] has many improvements over the original ECLAT algorithm [67] and can quickly compute maximal frequent itemsets even on some of the larger benchmarks we use.

Once the frequent object sets are determined, we modify the input to the main flow-sensitive analysis by merging frequently occurring object sets into single summary objects. The actual flow-sensitive analysis remains unmodified.

As a post-processing step, we compute the percentage of pointers that alias (see Section 4.4.3). Two pointers are said to alias if the intersection of their points-to sets is non-empty [27]. Checking all pairs of pointers in the program for possible aliasing is a time consuming process, especially for larger programs. We use the Pestrie data structure [64] \(^5\)

---

4. [http://www.borgelt.net/eclat.html](http://www.borgelt.net/eclat.html)  
5. [https://github.com/richardxx/pestrie](https://github.com/richardxx/pestrie)
to quickly compute the number of pointer pairs that alias. This data structure is designed for efficient persistent storage of pointer information. It also efficiently supports a number of queries on the stored pointer information, including aliasing between different pointers.

4.4 Evaluation

In this section, we evaluate our FOSM method by comparing the execution time and precision of the analysis with that of the staged flow-sensitive pointer analysis [23].

As in the previous chapter, we conducted our experiments on a set of 10 benchmarks. We use some of the bigger benchmarks from SPEC2006 [26] and a few other benchmarks all of which have been used previously in pointer analysis research [23, 41].

We used an 8-core machine with 16GB of memory, running Debian GNU/Linux 6.0, to conduct these experiments. However, our flow-sensitive analysis runs as a single thread utilizing only one core in the system.

4.4.1 Performance

We first compare the execution time of flow-sensitive pointer analysis with frequent object set merging (FOSM) against the original staged flow-sensitive (SFS) analysis [23]. Table 4.3 shows the execution times (in seconds) for each benchmark. The table also lists the time taken by the frequent itemset mining algorithm (ECLAT). We add the time taken by the actual analysis after merging frequent object sets and the time taken by ECLAT when computing speedup w.r.t SFS.

Even smaller benchmarks such as ex and 176.gcc, whose frequent object sets are not significantly large (see Table 4.1) show a speedup of at least 2x. Most of the bigger benchmarks (such as pine, gdb) show higher speedup ranging from 4x to 13x. Python and pine are two of the larger benchmarks where the time taken by ECLAT is more than the time for the actual analysis (column 3 in the table). While this limits the speedup to 5.91x in python, merging the large frequent object set (containing 22053 objects) in pine brings down its actual analysis time significantly, enabling a good speedup.
4.4. Evaluation

4.4.2 Convergence

By approximating many memory locations (objects) with a single summary location, we not only reduce propagation of large points-to sets, but also achieve faster convergence of the analysis. Table 4.3 (columns 7,8,9) shows the number of worklist items processed after merging frequent object sets, as a fraction of the number of worklist items processed without the approximation. On an average (geomean), about 33% less worklist items are processed after merging frequent object sets.

Table 4.3: Speedup in execution time of the analysis and the fraction of worklist items processed after frequent object set merging

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution time (s)</th>
<th>Worklist items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SFS</td>
<td>FOSM</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
<td>ECLAT</td>
</tr>
<tr>
<td>ex</td>
<td>0.17</td>
<td>0.08</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3.06</td>
<td>0.60</td>
</tr>
<tr>
<td>nethack</td>
<td>5.73</td>
<td>0.96</td>
</tr>
<tr>
<td>254.gap</td>
<td>19.71</td>
<td>0.76</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>95.57</td>
<td>6.32</td>
</tr>
<tr>
<td>vim</td>
<td>168.73</td>
<td>20.38</td>
</tr>
<tr>
<td>python</td>
<td>411.28</td>
<td>21.36</td>
</tr>
<tr>
<td>svn</td>
<td>4269.46</td>
<td>396.52</td>
</tr>
<tr>
<td>pine</td>
<td>6671.23</td>
<td>110.97</td>
</tr>
<tr>
<td>gdb</td>
<td>10591.46</td>
<td>653.66</td>
</tr>
</tbody>
</table>

4.4.3 Precision

In this section, we study the effects of merging memory objects on the precision of the analysis. We consider two client analysis queries and study the impact of precision loss on them.

Alias Pairs

A typical use of pointer analysis is to compute alias pairs in the program. Two pointers are said to alias if the intersection of their points-to sets is non-empty. In this experiment, we consider all pairs of pointers in the program and test how many of them alias.
Chapter 4. An Approximation Algorithm

We show the results in terms of the percentage of possible pointer pairs that may alias. This is a commonly used precision measure for points-to analysis [8, 17, 47]. Higher the percentage, lower is the precision of the analysis. The percentage of pointer pairs that alias for both our approximate analysis (frequent object set merging - FOSM) as well as Hardekopf’s fully flow-sensitive staged analysis (SFS) [23] are shown in Table 4.4.

While some benchmarks such as 176.gcc and pine show very little precision loss, python and 253.perlbmk incur relatively higher precision loss. We observe that even for these benchmarks which have a higher percentage of their objects (29.4% and 25.5% respectively) occurring with a higher frequency (30.5% and 20.3%), indicating that a larger number of frequent objects were merged, the precision loss is within 5%.

**Dependent Load/Store**

Some program analysis techniques such as Taint Analysis [62] and Loop Invariance Detection [65] require the knowledge of conflicting (dependent) loads and stores in the program. Any load or store that may refer to the same memory object as a given store can be considered to be in conflict with the given store. Similarly, stores that may refer to the same memory object as a given load can be considered to be in conflict with the given load.

To study the effects of precision loss due to our approach, we compute the percentage of conflicting load/store pairs in each function of the program. Higher the percentage of conflicting pairs, lower is the precision of the analysis. We again compare the precision of our analysis (frequent object set merging - FOSM) with Hardekopf’s fully flow-sensitive staged analysis (SFS) [23]. The results are shown in Table 4.4.

The precision loss characteristics here are similar to what we observed when using Alias Pairs as a precision measure, although the actual numbers vary. Ex, 176.gcc and 254.gap have very low precision loss while 253.perlbmk, python and svn have relatively higher precision loss compared to the others (less than 7%). This is again due to a higher percentage of their objects occurring with a higher frequency. Note that the actual numbers used to measure precision here are more than what we saw when using Alias
4.4. Evaluation

*Pairs* as a measure since the candidates for conflicting loads/stores considered here are local to a function, while in measuring *Alias Pairs*, we considered all pointers in the program.

Table 4.4: Percentage of pointer pairs that alias and the percentage of dependent load/stores in each function.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Alias Pairs % alias</th>
<th>% loss</th>
<th>Dependent Ld/St % conflict</th>
<th>% loss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SFS</td>
<td>FOSM</td>
<td>SFS</td>
<td>FOSM</td>
</tr>
<tr>
<td>ex</td>
<td>1.72</td>
<td>1.88</td>
<td>0.15</td>
<td>9.59</td>
</tr>
<tr>
<td></td>
<td>9.77</td>
<td>9.77</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>176.gcc</td>
<td>4.73</td>
<td>4.74</td>
<td>0.01</td>
<td>7.75</td>
</tr>
<tr>
<td>nethack</td>
<td>2.81</td>
<td>2.86</td>
<td>0.05</td>
<td>14.28</td>
</tr>
<tr>
<td></td>
<td>16.68</td>
<td>2.39</td>
<td>7.76</td>
<td>0.01</td>
</tr>
<tr>
<td>254.gap</td>
<td>13.38</td>
<td>13.83</td>
<td>0.45</td>
<td>23.48</td>
</tr>
<tr>
<td></td>
<td>23.84</td>
<td>0.36</td>
<td>14.28</td>
<td>2.39</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>18.58</td>
<td>22.26</td>
<td>3.67</td>
<td>29.81</td>
</tr>
<tr>
<td></td>
<td>36.28</td>
<td>6.46</td>
<td>23.48</td>
<td>2.39</td>
</tr>
<tr>
<td>vim</td>
<td>2.27</td>
<td>3.44</td>
<td>1.16</td>
<td>30.99</td>
</tr>
<tr>
<td>python</td>
<td>9.71</td>
<td>14.22</td>
<td>4.50</td>
<td>20.86</td>
</tr>
<tr>
<td></td>
<td>25.53</td>
<td>4.66</td>
<td>23.48</td>
<td>2.39</td>
</tr>
<tr>
<td>svn</td>
<td>7.47</td>
<td>7.65</td>
<td>0.18</td>
<td>21.22</td>
</tr>
<tr>
<td></td>
<td>28.18</td>
<td>6.96</td>
<td>20.86</td>
<td>4.66</td>
</tr>
<tr>
<td>pine</td>
<td>12.16</td>
<td>12.23</td>
<td>0.07</td>
<td>8.99</td>
</tr>
<tr>
<td></td>
<td>12.32</td>
<td>3.33</td>
<td>9.77</td>
<td>12.32</td>
</tr>
<tr>
<td>gdb</td>
<td>12.74</td>
<td>13.05</td>
<td>0.31</td>
<td>14.69</td>
</tr>
<tr>
<td></td>
<td>19.74</td>
<td>5.05</td>
<td>9.77</td>
<td>12.32</td>
</tr>
</tbody>
</table>

4.4.4 Choice of Objects to be Merged

In addition to the experiments in the previous two subsections, we also tried to merge the frequent object sets as computed by the flow-sensitive analysis, to measure the loss that could happen because of our use of a flow-insensitive analysis to approximate the frequent object sets. In other words, if we had oracle knowledge of frequent objects sets in the flow-sensitive pointer analysis, would the precision loss reduce? We found that using the frequent object sets from the final flow-sensitive analysis (and feeding it back to the frequent itemset mining algorithm in a subsequent run) does not result in any significant improvement in precision (less than 0.05% in any of the benchmarks).

Selecting a subset (which may have a higher frequency of occurrence) of the maximal frequent object sets as a candidate for merging may reduce precision loss further. However, determining the right subset may require iterating through all the subsets. We leave this exploration for future work.
4.5 Chapter Summary

In this work, we studied the structure of points-to sets of pointers in a flow-sensitive pointer analysis and identified the frequent occurrence of certain object sets in these points-to sets. We used the results from a less-precise analysis to determine these frequent object sets as an approximation to the frequent object sets in a flow-sensitive analysis. We showed that these frequent object sets can be merged into a single summary location at the cost of some precision loss. Such a merging of frequent object sets leads to significant improvements in the execution time of the flow-sensitive analysis.
Chapter 5

Related Work

The area of pointer analysis is rich in literature. A good survey can be found in [29, 44]. We highlight some related papers in flow-sensitive analyses in more detail.

5.1 Flow-Insensitive Analysis

5.1.1 Early Works

Probably the two most influential early research on pointer analysis are the works by Andersen [3] and Steensgaard [59]. Steensgaard’s analysis is still used in production compilers such as LLVM [36], while Andersen’s analysis is the basis for many papers in the area of pointer analysis [21, 27, 29, 41]. Andersen’s analysis, also referred to as inclusion-based analysis, processes points-to constraints by updating the points-to set of variables on the LHS of pointer assignments by including the points-to set of the RHS variable in it. This means that, after a pointer assignment statement is processed, the points-to set of the variable on the LHS is a superset of the points-to set of the variable on the RHS. On the other hand, Steensgaard’s analysis is a unification based analysis, i.e., after processing a pointer assignment statement, the points-to set of variables on the LHS and RHS are considered to be equal (unified). The loss of information due to unification makes Steensgaard’s analysis almost linear time, but less precise compared to Andersen’s analysis.
Das [16] achieves precision in between that of Steensgaard’s and Andersen’s analysis. Unification of symbols at the top levels of pointer chains in the points-to graph is avoided by using a “one level flow” algorithm, thus achieving scaling similar to Steensgaard’s algorithm, but being almost as precise as Andersen’s algorithm.

5.1.2 Speeding Up Andersen’s Analysis

A number of papers have looked at reducing the running time of Andersen’s inclusion based flow-insensitive analysis, each optimizing different aspects of the analysis. Rountev et al. [56] propose a method to detect variables that always have equal points-to sets and represent it with a single summary variable. Since the variables being grouped are guaranteed to have the same points-to sets, there is no precision loss involved. Their method looks for equivalent variables beyond strongly connected components. For example, if a top-level variable has a single incoming edge, then the variable has the same points-to set as the variable at the source of the edge. Hardekopf et al. [21] extend the idea of finding equivalent variables by involving both online (as the algorithm runs) and offline (before the actual algorithm begins) searches for variables having equivalent points-to sets. This results in merging more variables into a single summary variable, allowing for a faster analysis.

A few recent papers have explored parallelization of Andersen’s pointer analysis. Méndez-Lojo et al. [41] were the first to parallelize Andersen’s analysis. Their work involved formulating the analysis as a graph-rewriting problem so as to expose amorphous data parallelism [52]. Our work on parallelizing flow-sensitive analysis uses Méndez-Lojo’s work as base and extends it for flow-sensitivity. Méndez-Lojo et al. also extend this work [40] to perform the analysis on GPUs. Putta et al. [53] take advantage of the monotonicity property of the analysis to keep multiple copies of points-to sets, thereby reducing dependence across constraints. This helps them achieve an average speedup of 3.4x on 8 threads. Su et al. [60] scale Andersen’s analysis to heterogeneous systems using a dynamic workload distribution scheme.
5.2. Flow-Sensitive Pointer Analysis

5.1.3 Approximate Flow-Insensitive Analysis

The first attempt at approximating Andersen’s (flow-insensitive) analysis via merging variables, at the cost of precision loss was by proposed by Nasre [43]. This work allowed merging of two variables based on them having similar points-to or pointed-to-by sets. A known similarity measure, called the Jaccard similarity [54] was used. The disadvantage with this method is that the quality of the final result depends on the order of similarity checks and merging of objects. Also, repeated checks for similarity during the analysis is an overhead. In our work in Chapter 4, we do not incrementally merge objects based on similarity. Instead, we determine the objects to be merged beforehand using a frequent itemset mining algorithm [54,67] and merge objects before the actual analysis begins. We take advantage of the staged flow-sensitive analysis by using results from the first stage (flow-insensitive pointer analysis) to determine frequent object sets for the second stage (flow-sensitive pointer analysis). Another advantage of performing object merging prior to the main analysis is that it decouples resource optimization from the main analysis, making it independent of the implementation of the main analysis.

Nasre et al. [47] proposed bloom filters as a container data structure for points-to sets. They show how to utilize these probabilistic data structures for context-sensitive (but flow-insensitive) Andersen’s analysis, at the cost of some precision loss. Another method to approximate pointer analysis by Nasre [44] analyses parts of the program, chosen randomly, in a context, flow or field sensitive way, while analyzing the rest of the program in an insensitive way. Points-to results from a few such random runs are merged together to obtain a significantly precise result.

5.2 Flow-Sensitive Pointer Analysis

Traditional flow-sensitive pointer analysis [34] uses an iterative data-flow analysis, which is extremely inefficient for pointer analysis, mainly due to the conservative propagation of all data-flow information from each node in the control flow graph to every other reachable node. The conservative propagation is due to the fact that the analysis cannot
know which node actually needs what information. Since there can be hundreds of thousands of pointers and each pointer can have thousands of pointees, a huge amount of information is stored, processed and propagated.

A frequently used method to optimize flow-sensitive data-flow analyses is to perform a sparse analysis [13] [28]. These analyses directly connect variable definitions to their uses, so that, data-flow values need to be propagated only to their uses. However, pointer information is needed to compute these def-use chains [2]. This results in a cyclic dependence.

Hardekopf et al. [22] presented a semi-sparse analysis that performs a sparse analysis on some variables and an iterative data-flow analysis on other variables. Hardekopf et al. [23] also proposed a fully sparse analysis to minimize the amount of data propagated during the analysis. They adopted a staged approach to solve the cyclic dependence problem mentioned in the last paragraph. A less precise (and fast) pointer analysis was used as a first step to the flow-sensitive pointer analysis. Points-to sets from the less precise analysis can be used to build conservative def-use chains in the program, allowing for propagation of points-to sets only along these def-use chains, thus minimizing the amount of data propagated. The flow-sensitive pointer analysis methods proposed in this thesis are based on Hardekopf’s staged flow-sensitive (SFS) method. Quantitative comparison of our methods against SFS have been reported in Chapters 3 and 4.

Li et al. [38] proposed a method in which the problem of flow-sensitive pointer analysis is reduced to a general graph reachability problem in a value flow graph. The value flow graph captures dependencies between pointers, both direct and indirect through memory. They also compute pointed-to-by sets for each memory object during their analysis. This information can be useful in some client analyses such as escape analysis. Li et al. [39] extend the use of value flow graphs for a fully context-sensitive analysis, applying context-sensitivity to both local and heap objects.

A recent work by Khedker et al. [35] uses the idea that points-to information of a pointer needs to be propagated only to those program points in which the pointer is live. They combine flow-sensitive pointer analysis with liveness analysis [2]. Yu et al. [66]
5.2. Flow-Sensitive Pointer Analysis

Figure 5.1: Simple example to illustrate imprecision of Hasti and Horwitz’s algorithm

propose a flow and context sensitive analysis in which pointers are analyzed in decreasing order of their *points-to level*. At each level, pointers at that level are re-written into an SSA form and analyzed. Steensgaard’s analysis [59] is used at the beginning to obtain the levels of all pointers.

Lhoták et al. observe that the higher precision seen in a flow-sensitive pointer analysis is due to its ability to perform strong updates. Hence they propose an algorithm in which dereferenced singleton points-to sets, where strong updates can happen are analyzed flow-sensitively, while larger dereferenced sets, where strong updates are not applicable are analyzed flow-insensitively. Another approximation to flow-sensitive pointer analysis by Nasre [45] uses bloom filters to store points-to information. Since bloom filters are probabilistic data structures, there is a loss of precision. This work utilizes GPUs to speedup the analysis.

The paper by Hasti and Horwitz [24] talks about an algorithm to improve precision of flow-insensitive analysis by performing repeated rewrites into the SSA form [15]. They however do not comment on whether the method described will achieve the precision of a flow-sensitive analysis. This question has also been raised in more recent papers [23,37], but without an answer. We answer the question with the following example which shows that their algorithm will not always achieve full flow-sensitive precision. In Figure 5.1, *a* in block 2 can only point to *x*. However, Hasti and Horwitz’s algorithm would compute that *a* may point to either of *x* or *y*. This is because block 2 and block 3 depend on each other to become precise.
5.3 Complexity of Pointer Analysis

The variations and dimensions possible in designing a pointer analysis has lead many researchers to analyze the theoretical complexity of different pointer analyses. While the complexity results for some common forms of the analysis are well established, the complexities of certain others are still unknown.

Kam et al. [33] established that solving non-distributive data-flow analyses in general, to obtain a Meet of Paths solution, is undecidable. Ramalingam [55] showed that flow-sensitive pointer analysis (which is a non-distributive data-flow analysis), in the presence of dynamic memory allocation and aggregate data types is undecidable. This proof involved reducing the Post Correspondence problem to flow-sensitive alias analysis of programs. Muth et al. [42] prove that flow-sensitive pointer analysis on programs with no dynamic memory accesses, and having only scalar variables (no aggregate types) is PSPACE complete, even when only two levels of indirections are allowed.

Horwitz [30] proved that flow-insensitive alias analysis on programs with only scalar variables and no dynamic memory allocation, but with any number of indirections allowed is NP-Hard. Whether the problem is in NP (and hence NP-Complete) remains an open question. It is also not known whether flow-insensitive analysis in the presence of dynamic memory is decidable.

Chakaravarthy [11] illustrates why Andersen’s analysis fails to be as precise as a general flow-insensitive analysis and provides complexity results on a restricted version of flow-insensitive pointer analysis. Specifically, he shows that the flow-insensitive pointer analysis on programs with strong typing (i.e., arbitrary casts not being allowed), but with any number of indirection levels, can be solved in polynomial time. However the complexity of pointer analysis on programs with no strong typing, but with limited indirection levels remains an open question. While Ramalingam [55] had shown that flow-sensitive pointer analysis, on programs with aggregate data types, and with dynamic memory allocation is undecidable, Chakaravarthy showed that even when only scalar variables are allowed (i.e., no aggregate types), the analysis remains undecidable.

Blackshear et al. [8] give a witness search based algorithm for precise flow-insensitive
pointer analysis. This algorithm, although not polynomial in its time complexity, helps them show that there is almost no difference between the precision of a general flow-insensitive pointer analysis and Andersen’s analysis in practice.
Chapter 6

Conclusion and Future Work

In the last few years, flow-sensitivity has been the focus of many research papers in the area of pointer analysis. Since flow-sensitivity requires computing and storing points-to sets (which may contain thousands of objects) at each program point, scaling the analysis to large programs was a challenge. Many recent works have proposed different approaches to achieve this. The staged flow-sensitive analysis was one such method that used a less precise (but fast) analysis prior to the actual analysis to minimize the amount of points-to information propagated and stored at program points.

In this thesis, we proposed two methods to speedup the staged flow-sensitive pointer analysis. In the first method, we introduced a technique to parallelize flow-sensitive pointer analysis. Our parallelization technique involved formulating the analysis as a graph-rewriting problem, making it easy to extract amorphous data parallelism. Our parallel implementation of this graph-rewriting formulation using TBB achieves considerable scaling (upto 4.05x) for 8 threads on a set of 10 benchmarks. To the best of our knowledge, this was the first method for parallelizing fully flow-sensitive pointer analysis.

Our second proposed method to speed up flow-sensitive pointer analysis is based on the observation that certain objects occurred together in the points-to sets of many pointers. We showed that by summarizing sets of frequent objects in terms of single objects, significant speedup in the execution time of flow-sensitive pointer analysis can be achieved. Such an approximation of many objects into a single summary object can
lead to a precision loss. However, we observed only a small average precision loss of 0.25% and 1.40% on two precision metrics. Our approximation method achieves a speedup of upto 12.9x, with an average speedup of 6.2x.

**Future Work**

Both of our methods to speed up flow-sensitive pointer analysis make way for more explorations in further speeding up the analysis. We point out a few such ideas here.

In our graph-rewriting formulation of flow-sensitive pointer analysis, we achieve parallelism based on the presence of multiple nodes in the graph to which rewrite rules may be applied. In our implementation, we let TBB manage the workload, that is, the dynamic partitioning of application of rewrite rules to different threads. It would be an interesting experiment to influence this partitioning with the layout of the graph, so that nodes nearby in the graph get assigned to the same thread. This can potentially reduce contention among different threads trying to update edges of the same graph node. Different threads will now tend to update different parts of the graph. The challenge here is that the structure of the graph changes dynamically due to addition of edges. This leads to a varying behavior in the propagation of points-to sets (addition of more edges) in different parts of the graph, as the analysis progresses.

The original staged flow-sensitive pointer analysis work described a context-insensitive analysis. Our work consequently is context-insensitive. Extending the staged flow-sensitive pointer analysis to be context-sensitive would require defining and framing the data-flow graph for different calling contexts. Formulating such a context-sensitive version of the staged flow-sensitive pointer analysis as a graph-rewriting problem to exploit parallelism is an interesting food for thought.

In our approximation of the staged flow-sensitive pointer analysis, we merged maximal frequent object sets (i.e., object sets, none of whose supersets were frequent) into single summary objects. It is however not necessary that only maximal frequent sets be chosen for merging. Choosing an appropriate subset of a maximal frequent object set for merging may further reduce precision loss, while still giving performance speedup. However,
identifying the right subset to achieve a finer control over the precision loss requires good heuristics to avoid iterating over every possible subset.

Our two contributions for a fast flow-sensitive pointer analysis described in this thesis are independent of each other. Conceptually, combining the two techniques is simple. Since our approximation algorithm does not modify the main analysis of the staged flow-sensitive pointer analysis (it just modifies the input data-flow graph), the graph-rewriting analysis can be directly used in its place, rewriting the graph corresponding to the modified data-flow graph (in which frequent object sets are merged). The key challenge in such a setup would be to scale the analysis on the modified input, which may have different workload characteristics.
References


REFERENCES


[59] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the*
REFERENCES


[63] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool.


