Precise and Efficient Analysis of Java Programs

A THESIS

submitted by

MANAS THAKUR

for the award of the degree

of

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

August 2019

THESIS CERTIFICATE

This is to certify that the thesis titled **Precise and Efficient Analysis of Java Programs**, submitted by **Manas Thakur**, to the Indian Institute of Technology Madras, for the award of the degree of **Doctor of Philosophy**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Krishna Nandivada
Research Guide
Associate Professor
Dept. of CSE
IIT Madras, 600 036

Place:

Date:

ACKNOWLEDGEMENTS

I joined IIT Madras as a fresh undergraduate interested in academics and compilers, with no idea about what is research, how to work with large codebases, what is paper writing, how to manage emotional traumas after rejections, and so on. The foremost acknowledgement, hence, is fully owed to my advisor Dr. V Krishna Nandivada, one of the most passionate persons I have ever met, who at times micro-managed my whims with utmost patience, and gradually transformed my interests into a Ph.D. People say that prospective research students should not look for a friend and philosopher and let the advisor be a guide, but I got all of them and more, without asking.

Working towards a Ph.D is a hell lot of long-endured pressure, and we students often claim that only we understand our situation. The presence of a friendly group of colleagues in the workplace is an important requirement to survive. I was fortunate to be associated with the (undisputedly) most active lab of the department. PACE lab was like a home from the first day, with no borders induced by age or experience. Upon joining, I got a warm welcome from always-available seniors like Abhilash Bhandari, T V Kalyan, Tripti Warrier, John Jose, Raghavendra K, Sudharsan J, Suyash Gupta, Pritam Majumder, Gnaneswara Rao, Praveen Alapati, Priyabhashini K, Prasanna Venkatesh, Aditya Kajwe, and Biswabandan Panda (RISE lab). Aman Nougrahiya and Jyothi Vedurada have been constant companions throughout the duration – no words would be enough to acknowledge their roles. We three have a group called *BUGS and Chocolates*, formed when we had started reading the excellent book *BUGS in Writing* by Lyn Dupre; and not unlike the expansion of BUGS, we three have indeed seen and supported each other through the bad, ugly, good, and splendid times alike. Raghesh and Anchu, again close friends, came forward at a tough time and helped in formal writing.

Dr. Rupesh Nasre gave insights on various aspects of the work throughout; I express gratitude for the suggestions for improvement given in my seminars and doctoral committee meetings by him, by Dr. Shankar Balachandran, Prof. N S Narayanaswamy,

Prof. Madhu Mutyam, Prof. Usha Mohan, and Prof. Deepak Khemani. The successive HoDs Prof. P Sreenivasa Kumar, Prof. Krishna Sivalingam and Prof. C Chandra Sekhar, and senior faculty members Prof. C Siva Ram Murthy and Prof. C Pandu Rangan, were not only a source of academic support, but also of thoughtful discussions whenever the need arose. Many people in the administrative staff, specially Sridevi madam from CS office and Ravichandran sir from the Systems lab, were exemplary in the way they let us offload our burdens.

I would have dropped out if I did not get extra-curricular support. My Yoga teacher, Mrs. Katyayini Reddy, completely changed my perspective towards life: I realized that a healthy body, mind, and soul (all three) are not mere catalysts, but a necessity towards a happy life. I am fortunate to have Seetharam sir from Hyderabad in my life, whom I regard as a guiding light; his concerns at tough times give me the courage to face them, and his suggestions provide me the ability to think clearly. I have several teachers from my undergraduate days who never left me (in faith as well as in practice): Dr. Bhaskar Patel, Deshpande sir, Mahatme sir, and Vinay Wardhan sir; they form the continuity of my life. Sudarsun Santhiappan, a batchmate and a very understanding friend, has been a support in all the endeavors, small or big. Above all is the role of my mother Mala Thakur who knows me like being the same soul, and whom I am proud to be a son of; may the God(dess) bestow everyone with such a beautiful parent-child relationship.

The list of friends and the happy times spent with them is never-ending. Aman Sharma got me into the habit of Friday movies; Saurabh Kalikar, Arun T and Somesh Singh were my daily companions for tea/coffee; life would be much boring without them. Though she is a relatively new entrant to PACE lab, my bond with Anju M A will long outlive my stay at IIT Madras. I have troubled and have been helped by Siva (Jyothi's husband) and Suresh (Anju's husband); sincere thanks to both of them. I have a special place in my heart for my young friends Abhimanyu (Anju's son), Vignesh (Jyothi's son), and Duggu (a kid in our neighbourhood at the IIT Madras campus, who made the life ever more joyous by uttering *Bhaai* in a most loving manner on seeing me everyday, for two years). I have good memories with many more current- and ex-PACErs, including Joe Augustine, Jyothi Krishna, Rajendra Dangwal, Shouvick Mondal, Surya Raj, Ganesh K, Rahul Shrivastava, Shashidhar G, Rakesh

Patil, Dennis Varkey and Indu K, and department-colleagues Priyatosh Mishra, Nauman Dawalatabad, Siba Narayan, Jilt Sebastian, Jom Kuriakose, Mari Ganesh and Shrinivas Devshatwar. I am thankful to my childhood friends who have been in constant touch: Abhay Jaiswal, Aditya Kumar, Awanish Shukla, Manohar Mrinal, Mohit Choudhary, Navlok Kumar, Pradeep Yadav, and Rahul Chimkar.

I plead forgiveness from anyone whose contributions I forgot to acknowledge; your wishes would remain a part of me forever. I also apologize to anyone whom I have knowingly or unknowingly hurt in any manner. Finally, I urge the readers to make sure they take out time to appreciate the beauty that lies outside of our regular spheres. I had a very good time with the monkeys at the beautiful IIT Madras campus, and I am going to really miss my favorites: *Tikoni*, *Pithoni*, *Chanchala*, *Katoni* (with her new-born baby), to name a few. The nature reminds us that we have a bigger purpose to serve, and that our contributions in our respective fields should ensure the triumph of truth as well as the preservation of harmony. Best wishes to everyone.

Manas Thakur

ABSTRACT

KEYWORDS: Program analysis, Context-sensitivity, Just-in-time compilers

While translating programs from one form to another, compilers and related tools perform a series of analyses on the input program. The results of these program analyses, apart from many other applications such as program understanding and debugging, are also used to drive several dependent optimizations that result in the generation of an optimized binary. In general, the higher the precision of a program analysis, the higher is the expected number of generated optimization opportunities. However, performing precise analyses usually affects the compilation time adversely, leading to tradeoffs between precision and efficiency. These tradeoffs also play a prominent role in virtual machines with just-in-time (JIT) compilers, where an increased analysis time directly affects the execution time of the program. This thesis proposes several methodologies that help compilers perform JIT as well as static analyses that are precise-yet-efficient, especially for programs written in Java-like object-oriented languages.

To balance the tradeoff between the efficiency and precision of performing program analyses during JIT compilation, this thesis proposes a two-step (static+JIT) analysis framework called PYE that helps generate precise analysis-results at runtime, at a very low cost. PYE achieves the twin objectives of precision and performance during JIT compilation, by using a two-pronged approach: (i) It performs expensive analyses during static compilation, while accounting for the unavailability of the runtime libraries by generating partial results, in terms of *conditional values*, for the input application. (ii) During JIT compilation, PYE resolves the conditions associated with these values, using the pre-computed conditional values for the libraries, to generate the final results. The extensive evaluation results for two instantiations of PYE show that the proposed strategy works quite well and fulfills both the promises it makes: enhanced precision while maintaining efficiency during JIT compilation.

To improve the scalability and precision of static analyses, wherein, in order to scale value-contexts based whole-program heap analyses, this thesis proposes a three-staged approach. The approach is based on a novel idea of level-summarized relevant value-contexts (*LSRV-contexts*), which take into account an important observation that we do not need to compare the complete value-contexts at each call-site. The three stages of the overall approach are: (i) a fast pre-analysis stage that finds the portion of the caller-context which is actually needed in the callee; (ii) a main-analysis stage that uses LSRV-contexts to defer the analysis of methods that do not impact the callers' heap and analyze the rest efficiently; and (iii) a post-analysis stage that analyzes the deferred methods separately. The evaluation of two LSRV-contexts based non-trivial analyses against their traditional value-contexts based versions shows that the proposed approach not only reduces the analysis time and memory consumption significantly, but also succeeds in analyzing otherwise unanalyzable programs in less than 40 minutes.

This thesis next improves the precision of LSRV-contexts by cloning the heap and considering the k-level object context as the context abstraction (resulting in the notion of LSRV$k$objH). The thesis instantiates LSRV$k$objH with $k = 1$ for performing control-flow analysis, and compares the same with a one-level object-sensitive analysis with one level of heap cloning (abbreviated as 1obj1h). The results show that the proposed scheme scales up to large benchmarks (terminates within an hour), while significantly enhancing the precision (for example, about 5% over plain LSRV-contexts and about 12% over 1obj1h, for one of the precision-indicating clients for control-flow analysis).

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **v**

LIST OF TABLES **xi**

LIST OF FIGURES **xiv**

ABBREVIATIONS **xv**

NOTATION **xvii**

1 INTRODUCTION **1**

1.1 Precise and Efficient Just-In-Time Analyses 2

1.2 Scalable Context-Sensitive Static Analyses 6

1.3 Contributions of the Thesis 9

1.4 Organization of the Thesis 11

2 BACKGROUND **13**

2.1 Points-to Analysis 13

2.2 Points-to Graphs 14

2.3 Applications of Points-to Analysis 15

2.3.1 Thread-escape Analysis 15

2.3.2 Null-check Elimination 16

2.3.3 Control-flow Analysis 16

2.4 Analysis Dimensions 16

2.4.1 Flow-sensitivity 17

2.4.2 Field-sensitivity 17

2.4.3 Analysis Scope 18

2.4.4 Context-sensitivity 18

2.5 Analysis Frameworks 20
2.5.1 The Soot Framework 21
2.5.2 The HotSpot JVM 21

3 PRECISE AND EFFICIENT JUST-IN-TIME ANALYSES 23
3.1 The PYE Framework 23
3.1.1 Typical Modular Analyses 23
3.1.2 PYE: A Practical Alternative 25
3.1.3 Partial Summaries 26
3.1.4 Simplifying Partial Summaries 29
3.1.5 Efficient Storage of Partial Summaries for PYE 31
3.1.6 Writing an Analysis in PYE 32
3.2 Points-to Analysis for null-Check Elimination 33
3.2.1 Partial-analyzer for PACE 34
3.2.2 Efficient Storage of Partial Summaries for PACE 40
3.2.3 Results-adapter for PACE 41
3.3 Escape Analysis for Synchronization Elimination 41
3.3.1 Partial-analyzer for EASE 42
3.3.2 Efficient Storage of Partial Summaries for EASE 45
3.3.3 Results-adapter for EASE 45
3.4 PYE: A Discussion 47
3.4.1 PYE: Correctness 50
3.5 Implementation and Evaluation of PYE 52
3.5.1 Evaluation Setup for PYE 52
3.5.2 Evaluation of the Partial-analyzer 54
3.5.3 Evaluation of the Fast-precise-analyzer 56

4 SCALABLE CONTEXT-SENSITIVE STATIC ANALYSES 65
4.1 Challenges in Scaling Context-Sensitive Analyses 65
4.2 LSRV-contexts 67
4.2.1 Pre-analysis 68
4.2.2 Main-analysis 71

4.2.3 Post-analysis 73
4.3 Instantiations of LSRV-contexts 74
4.3.1 Thread-escape Analysis using LSRV-contexts 75
4.3.2 Control-flow Analysis using LSRV-contexts 76
4.4 LSRV-contexts: A Discussion 76
4.5 Implementation and Evaluation of LSRV-contexts 78
4.5.1 LSRV-contexts: Analysis Time 81
4.5.2 LSRV-contexts: Number of Contexts 82
4.5.3 LSRV-contexts: Peak Memory Usage 83

5 HEAP CLONING AND OBJECT-SENSITIVITY 85
5.1 LSRV-contexts with Heap Cloning 86
5.2 Comparing LSRV-contexts and Object-sensitivity 87
5.3 Merging LSRV-contexts and Object-sensitivity 90
5.4 Precision Lattice for the LSRV Approaches 90
5.5 Implementation and Evaluation of LSRV$k$objH 92
5.5.1 Scalability of LSRV$k$objH 92
5.5.2 Precision of LSRV$k$objH 93

6 RELATED WORK 97
6.1 Staged Analysis 97
6.2 Modular Analysis 98
6.3 Points-to Analysis for Null-Check Elimination 100
6.4 Escape Analysis for Synchronization Elimination 100
6.5 Context-sensitive Analysis 101

7 CONCLUSION AND FUTURE WORK 105

LIST OF PAPERS BASED ON THE THESIS 115

LIST OF TABLES

3.1 The definitions of $\mathcal{D}$ and `Val` for the analyses PACE and EASE. . . 34

3.2 Details of the benchmarks used for PACE and EASE, storage overhead for .res files, and the percentage of elements with dependent conditional values (`DCVal`) in the generated .res files. 53

4.1 Some names used to describe the pre-analysis. 68

4.2 Details of the benchmarks used to evaluate LSRV-contexts. 79

4.3 Evaluation results for escape analysis using LSRV-contexts. 80

4.4 Evaluation results for control-flow analysis using LSRV-contexts. . . 80

5.1 Evaluation results with heap cloning for control-flow analysis. . . . 93

LIST OF FIGURES

1.1 Example to motivate PYE. 3

1.2 Example to motivate LSRV-contexts. 7

2.1 Example to demonstrate escape analysis and points-to graphs. . . . 14

2.2 Example to motivate context-sensitivity. 19

3.1 Java program analysis: (a) Two traditional approaches. (b) Approach adopted by PYE. 24

3.2 Example to illustrate the generation of conditional values. 27

3.3 Partial-analysis rules for PACE. 36

3.4 Definition of the `updateDeref` macro used in PACE. 37

3.5 Definition of the `extendCVals` macro used in PACE and EASE. . 38

3.6 Example to demonstrate the working of PACE. 40

3.7 Partial analysis rules for EASE. 42

3.8 Example of conditional values. 44

3.9 Example to illustrate some limitations of the heap abstraction used for PACE and EASE. 48

3.10 Time taken (in seconds) by the partial analyzers of (a) PACE and (b) EASE. 55

3.11 Number of explicit null-checks inserted by the existing analyzer of the C2 compiler and PACE. 57

3.12 Number of synchronization operations elided by the existing analyzer of the C2 compiler and that by EASE. 58

3.13 Comparison of the time taken by PACE and EASE with that by the existing analyzers of C2. 59

3.14 Improvement in JIT compilation time for EASE. 61

3.15 Performance improvement of (a) PACE and (b) EASE, over the respective existing analyzers of C2. 62

4.1 Block diagram of the proposed approach for scaling value-contexts. 67

4.2 Obtaining access-depths in the pre-analysis. 69

4.3 Handling call-stmts in the main-analysis. 72

4.4 Number of contexts created per method during the control-flow analysis of the benchmark PMD. 83

5.1 Example to show the effect of heap cloning. 87

5.2 An example where a control-flow analysis using LSRV-contexts is more precise than one using object-sensitivity. 88

5.3 An example where an LSRVH control-flow analysis is less precise than one based on $k$obj1h. 89

5.4 Precision lattice for various control-flow analyses. 91

5.5 Normalized precision of various context abstractions (lower the better). 94

ABBREVIATIONS

JVM	Java Virtual Machine
JIT	Just-In-Time (compilers/compilation)
PYE	The Precise-Yet-Efficient Framework
PACE	Points-to Analysis for null-Check Elimination
EASE	Escape Analysis for Synchronization Elimination
LSRV	Level-Summarized Relevant Value (context)
LSRV$k$objH	LSRV-contexts with k levels of object-sensitivity and Heap cloning

NOTATION

G_m	The points-to graph for method m.
g_m	The partial summary for method m.

CHAPTER 1

INTRODUCTION

The world of twenty-first century runs using computer programs. Whether it is a hand-held gadget or a powerful supercomputer, humans write programs, often in a high-level programming language (such as C, Java, Python, R, OCaml, and so on), to communicate with these digital devices. However, in order to *execute* a program on a given machine, the program first has to be converted to a form that can be *understood* by that machine. Akin to the communication between any two persons who speak and understand different languages, human-computer interaction, thus, also requires a *translator*. Two primary examples of translators that convert the high-level language of computer programs to the low-level language of machines are *interpreters* and *compilers*.

An interpreter executes a program instruction-by-instruction. On the other hand, typical compilers, while translating code from one form to another, perform a series of analyses on the input program. The results of these *program analyses* are used to drive a plethora of applications: understanding the semantics of a program, debugging an erroneous program, finding out whether a program is secure, and optimizing the program in terms of the resources it may consume (time, memory, energy, etc.), to name a few. Depending on the time at which a compiler translates a program, there are two kinds of compilers and program analyses: *static* and *just-in-time* (or JIT).

A static compiler analyzes a program before the program executes, that is, offline. On the other hand, a JIT compiler analyzes a program while the program is already executing, that is, on-the-fly. In general, the higher the precision of a program analysis, the higher is the expected number of generated optimization opportunities. However, performing precise analyses usually affects the compilation time adversely, leading to trade-offs between precision and efficiency. Though the compilation time should not be prohibitively high for static analyses, the trade-offs between precision and efficiency play a more prominent role in just-in-time compilers, where an increased analysis time directly affects the execution time of the program. This thesis proposes

several methodologies that help compilers perform just-in-time as well as static analyses that are precise-yet-efficient. The focus is on the compilers for programs written in Java (Arnold *et al.*, 2005), which is a popular object-oriented language driving a large number of applications, on computers and mobiles alike (Wilson, 2016).

1.1 Precise and Efficient Just-In-Time Analyses

Modern languages like Java and C# follow a two-step process for compilation and execution: the input program is statically compiled to an intermediate language (for example, Bytecode for Java and CIL for C#), which is then executed on a possibly remote virtual machine (for example, JVM and .NET). Many virtual machines (Paleczny *et al.*, 2001; Alpern *et al.*, 2005) use inbuilt just-in-time (JIT) compiler(s) to generate optimized assembly code that can be directly executed on the hardware. While this can lead to significant performance gains compared to the "interpreter only" mode, it also brings in some interesting challenges.

One of the main challenges in JIT compilation arises from the fact that the time spent in compilation, which includes program-analysis time, gets added to the execution time of the program. Hence, it is important that the time spent in JIT compilation is not prohibitively high. Consequently, typical JIT compilers in popular virtual machines (such as the HotSpot JVM (Paleczny *et al.*, 2001) and the Jikes RVM (Alpern *et al.*, 2005)) perform imprecise analyses in place of precise whole-program analyses and end up sacrificing precision for efficiency.

An alternative to performing imprecise analyses during JIT compilation is to perform expensive whole-program analyses during static compilation, and use the results during JIT compilation. However, the runtime libraries (such as the JDK) on the machine where the program is executed may differ from those available statically on the machine where the program is compiled. As a result, though this alternative does not impact the JIT compilation time much, the static analyses have to handle calls to library methods in a conservative manner, which may again lead to imprecision.

Thus, both the practical alternatives – (i) whole-program analysis at compile-time

```
1  class X { Y g; }
2  class Y { Z h; }
3  class B {
4    X f;
5    B() {
6      f = new X();
7      f.g = new Y();
8      f.g.h = new Z();
9    }
10   void bar() {
11     B r1 = new B();
12     List r2 = new AList();
13     r2.add(r1);
14     X x = r1.f;
15     Y y = x.g;
16     Z z = y.h;
17     ...
18   }
19 }
```

(a)

```
1  class AList<E> extends List<E> {
2    // AList is a fixed size list.
3    // 1. arr is a final field
4    // allocated in the constructor.
5    // 2. size is a private field
6    // initialized in the constructor.
7
8    void add(E elem) {
9      arr[size++] = elem;
10   }
11 }
```

(b)

Figure 1.1: (a) A snippet of a synthetic Java program. (b) Simplified code for the library method `AList.add`. While analyzing the method `bar`, the code for `AList.add` is not available, and vice-versa.

and (ii) fast analysis during JIT compilation – may lead to imprecise results. These issues can be illustrated in the context of a points-to analysis that is used to remove unnecessary "null-dereference-checks" in Java programs.

In Java, before executing each statement that dereferences an object, the JVM needs to check whether the object being dereferenced is null; and if so, a *NullPointerException* must be thrown. Consider the Java code snippet shown in Figure 1.1. In the method `bar`, the statements 13, 14, 15 and 16 dereference objects. As the variables `r1` and `r2` point-to concrete objects allocated at lines 11 and 12, respectively, the null-dereference checks at lines 13 and 14 can be safely skipped. Further, the null-dereference checks at lines 15 and 16 can also be skipped, if (i) `r2` is known to point to an object of type `AList`, and (ii) the method `AList.add` does not modify the object pointed-to via the fields of its parameter. Thus, the number of null-dereference checks that can be skipped (or eliminated) depends directly on the precision of the underlying points-to analysis used. The next two paragraphs discuss the impact of the two above-discussed analysis alternatives on null-dereference-check elimination, in the context of the Java code snippet shown in Figure 1.1.

Alternative A1: Analysis during static compilation. A statically performed whole-program flow- and field-sensitive points-to analysis must assume the code of the method `AList.add` as unavailable (else risk the results being unsound). Thus, the alternative A1 can elide the null-checks at lines 13 and 14, but not the ones at lines 15 and 16.

Alternative A2: Analysis during JIT compilation. Typical JIT compilers restrict themselves to very imprecise analyses. For example, the points-to analysis used by the HotSpot Server Compiler (C2) is only intraprocedural. Thus C2 can again elide the null-checks only at lines 13 and 14.

As a step towards realizing precise and efficient JIT analysis, this thesis proposes a two-step analysis framework called PYE ("Precise-Yet-Efficient" framework) that addresses all the issues discussed above. PYE helps generate highly precise analysis-results for application programs during JIT compilation, at a very low cost. PYE achieves this objective using a two-pronged approach: (i) It offloads expensive analyses to the static Java compiler, where, in contrast to traditional summaries for each method, it generates "partial summaries". To avoid the imprecision arising out of the unavailable runtime libraries, PYE is based on the novel notion of "conditional values", as a way to store the dependencies between the application and the libraries. For example, in the context of null-dereference-check elimination, using traditional simple values, we say that a variable x may point to either a concrete-object or a null-object (concrete- and null-objects are the simple values). In contrast, the proposed conditional values allow us to reach conclusions of the form: variable x may point to a concrete object, if another variable y points to a concrete object, and null otherwise. The partial summaries consist of a set of conditional values for each program element in the method being analyzed. (ii) PYE passes the output of the static compiler (class files + partial summaries) to the JVM, where a newly added component of the JIT compiler evaluates the conditional values in the partial summaries, after merging the partial summaries of the libraries (pre-computed, once for each library installation), and generates final analysis-results.

PYE addresses the three challenges that can be envisaged in such a multi-step analysis framework: (i) It handles the possible imprecision arising out of the unavailable parts of a program while performing precise whole-program analyses. (ii) It makes

sure that the generated partial summaries are succinct and do not lead to any significant storage overhead. (iii) It loads and resolves the partial summaries efficiently without increasing the time spent during JIT compilation.

This thesis uses PYE to design two context-, flow-, and field-sensitive heap-based analyses. The first one is a Points-to Analysis to perform null-Check Elimination (PACE, in short), which elides unnecessary null-dereference checks in Java programs. For the code shown in Figure 1.1, PACE generates partial summaries which indicate that while the null-dereference checks at lines 13 and 14 can be unconditionally elided, the same at lines 15 and 16 can be elided only if the method `AList.add` does not assign null to the fields of its first parameter. During JIT compilation, after loading the library-partial-summary (which indicates that the method `AList.add` does not modify any field of the first parameter), PACE resolves the partial summary for `bar` and elides all the null-dereference checks in `bar`. Importantly, PACE achieves high precision without incurring any significant overhead during JIT compilation. This thesis also uses PYE to design an Escape Analysis (Blanchet, 2003) and demonstrate its effects on Synchronization Elimination (EASE, in short). Escape analysis finds objects that are local to a thread, and is widely used for eliminating useless synchronization (Blanchet, 2003; Ruf, 2000; Choi *et al.*, 1999); see Section 2.3.1 for a brief background on escape analysis. These two analyses were chosen because though both are based on pointer analysis, they have different types of lattices, and are quite pedagogical and illustrative of the intricacies involved in their design.

The core of the PYE framework as well as the two analyses PACE and EASE have been implemented in two parts: (i) the components associated with the static compiler – implemented in the Soot optimization framework (Vallée-Rai *et al.*, 1999); and (ii) the components associated with the JIT compiler – implemented in the HotSpot Server Compiler (C2) of the OpenJDK HotSpot JVM (Paleczny *et al.*, 2001).

The thesis evaluates PYE using PACE and EASE on a series of benchmarks from the SPECjvm (2008), DaCapo (Blackburn *et al.*, 2006) and JGF (Daly *et al.*, 2001) suites, and SPECjbb (2005). The evaluation shows that the strategy adopted by PYE works quite well: (i) PACE inserts 17.36% fewer null-checks during JIT compilation, on average, than the existing technique employed by C2. (ii) Compared to the existing

escape-analyzer of C2 (which elides only 0.03 synchronization operations, on average), EASE elides more synchronization operations (1.13, on average) during JIT compilation. Importantly, compared to the existing analyzers of C2, the improved precision of PACE does not significantly affect the JIT compilation time; and in case of EASE, it actually improves the JIT compilation time by 1.9%, on average. Further, the storage overheads for partial summaries are quite low: 6.41% and 3.96% over the class files for PACE and EASE, respectively.

PYE can, in general, be used to perform any whole-program modular dataflow analysis having: (i) a finite-height lattice of dataflow values; (ii) inter-dependent application and library analysis-results; and (iii) dynamically-refinable static-analysis results. Similarly, the discussed points-to and escape analyses can be extended to other respective related JIT optimizations, such as method inlining (Muchnick, 1997), garbage collection (Domani *et al.*, 2002), and so on. Though this thesis presents PYE in the context of Java, the techniques proposed are general enough to be extended to other languages such as C# that deploy a two-step compilation process.

1.2 Scalable Context-Sensitive Static Analyses

Heap analysis refers to a broad category of program analyses that statically approximate the information about the runtime heap of a program. For example, thread-escape analysis (Choi *et al.*, 1999; Blanchet, 2003) identifies objects that do not escape the thread of their allocation, interprocedural control-flow analysis (Palsberg and Schwartzbach, 1991; Shivers, 1991) identifies the potential targets of method calls, and so on. The precision of heap analyses determines the precision of several analyses and optimizations, and has been a prominent area in compiler research (Lhoták and Hendren, 2006; Smaragdakis *et al.*, 2011; Milanova *et al.*, 2005; Whaley and Lam, 2004; Xu and Rountev, 2008; Thiessen and Lhoták, 2017).

The precision and scalability of interprocedural heap analyses may vary based on whether the analysis is context-sensitive or not (Shapiro and Horwitz, 1997). A context-insensitive analysis does not differentiate among the various calls to a method, and generates a single summary that can be used at all of its call-sites. A context-sensitive

analysis, on the other hand, distinguishes between the "contexts" in which a method is called, and generates a summary for the method in each distinct context. Though the results generated by context-sensitive analyses have been shown to be more precise than context-insensitive analyses (Lhoták and Hendren, 2006), the scalability of the former in analyzing large programs continues to be a cause of concern.

The classical call-strings approach (Sharir and Pnueli, 1978; Shivers, 1991), which identifies contexts based on the call-string formed by a method's callers, is one of the oldest and widely-used approaches of defining the context abstraction. For example, consider the snippet of Java code shown in Figure 1.2a. A call-string based context-sensitive analysis would analyze the method `bar` in two contexts (created at lines 5 and 6), and the method `fb` in four contexts (two contexts for each context of `bar`). A major drawback of the call-string based approach is that in the presence of recursion and deep nesting of multiple calls, the length of the call-strings, and hence the number of contexts, may grow combinatorially. This makes the analysis unscalable to large real-world programs. Consequently, the call-string based analyses usually impose a limit on the call-string length, and treat the contexts of greater lengths conservatively. While such an approach improves the scalability of the analysis, it compromises on the resulting precision.

The clever value-contexts approach (Khedker and Karkare, 2008; Padhye and Khedker, 2013) addresses the scalability challenges in the call-strings approach by using dataflow values to restrict the potentially unbounded growth of call-strings, without sacrificing precision. For the code shown in Figure 1.2a, if Figure 1.2b and Figure 1.2c depict the points-to graphs at lines 5 and 6 respectively, then the value-contexts approach would analyze (i) `bar` in two contexts, as the value-contexts (points-to graphs reachable from formal parameters) at lines 5 and 6, shown in Figures 1.2d and 1.2e, respectively, are different; (ii) `fb` in only one context for each context of `bar`, as the points-to graphs at lines 11 and 12 match. Though the value-contexts approach is a breakthrough in performing precise context-sensitive analyses, it still does not scale for various popular heap analyses. Based on a study of many such heap analyses, this thesis identifies two main reasons for the lack of scalability: (i) time overheads involved in comparing the value-contexts and in redundantly analyzing many methods;

```
1 class A {
2   A f1,f2;
3   void foo(){
4     ...
5     c.bar(a);
6     d.bar(b);
7   } /*foo*/
8   void bar(A p){
9     A x = new A();
10    p.f1.f2 = x;
11    p.fb();
12    p.fb();
13  } /*bar*/
14  void fb(){
15    /*Doesn't read
16    or affect the
17    caller's heap*/
18  } /*fb*/
19 }
```

(a)

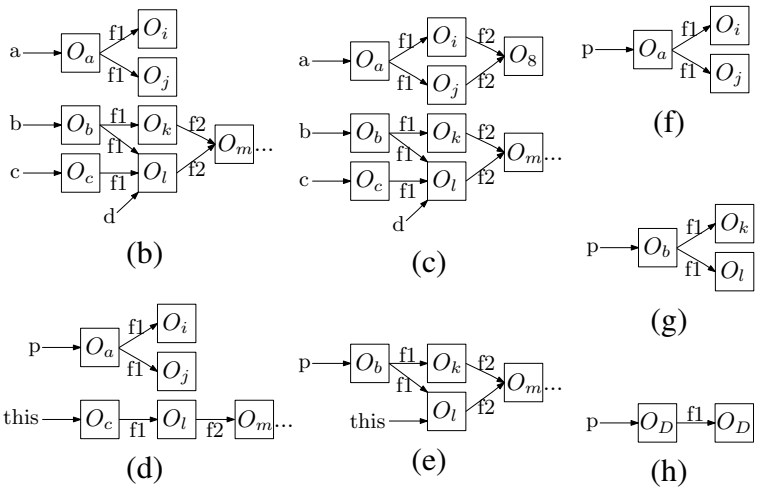


Figure 1.2: (a) A Java code snippet. (b) The assumed points-to graph at line 5. (c) The points-to graph at line 6. (d) The value-context for `bar` at line 5. (e) The value-context for `bar` at line 6. (f) The relevant value-context for `bar` at line 5. (g) The relevant value-context for `bar` at line 6. (h) The LSRV-context for `bar` at lines 5 and 6 (for escape analysis), assuming O_a, O_b, O_i, O_j, O_k and O_l do not escape; O_D represents a universal non-escaping object.

and (ii) memory overheads due to a large number of contexts. For example, for the call to `bar` at line 6, the whole points-to graphs shown in Figures 1.2d and 1.2e are compared; in practice, depending on the size of the graphs and the number of existing contexts, this could be expensive.

This thesis proposes several novel techniques that together help perform complex top-down whole-program value-contexts based heap analyses for large programs in less than 40 minutes. The proposed scheme comprises of three analysis stages: *pre*, *main*, and *post*. The pre-analysis is a lightweight stage that gains insights about the context-dependency of all the methods of a program. It computes the portion of the caller contexts that is actually needed in the callee and stores this information as parameter-wise *access-depth*. The main-analysis uses the access-depths to not only reduce the comparison performed (compares "relevant" parts of contexts) in identifying whether two value-contexts are equal, but also in deferring the analysis of *caller-ignorable* methods that do not impact the callers' heap. Deferring reduces the overheads of analyzing

those methods multiple times and in merging the results with the callers' heap, during the costly main-analysis. The main-analysis also uses a novel analysis-specific abstraction called *level-summarization* to improve the precision of identifying two contexts as equivalent. Finally, the post-analysis analyzes the deferred methods without losing precision. For the code in Figure 1.2a, an escape analysis based on the proposed approach identifies that both `bar` and `fb` are invoked only in a single level-summarized relevant value-context each. Consequently, `bar` is analyzed only once in the main-analysis, and `fb` (deferred in the main-analysis) is analyzed only once in the post-analysis.

The thesis demonstrates the effects of the proposed techniques by using them to perform fully context- and flow-sensitive thread-escape analysis and interprocedural control-flow analysis of Java programs (along with the JDK). The thesis evaluates the analyses against their corresponding traditional value-context versions on a multitude of benchmarks. The results show that the proposed techniques not only reduce the analysis time and memory consumption of the presented analyses significantly, but also help analyze previously unanalyzable large programs in a reasonable time. To the best of the author's knowledge, this is the first work that scales these heap analyses while realizing the precision of unbounded call-strings, especially using the practical value-contexts approach. Further, the proposed techniques are general enough to scale other context-sensitive heap analyses, even for programs written in other OO languages.

Motivated by the scalability of LSRV-contexts, the thesis next improves their precision by adding heap cloning (Nystrom *et al.*, 2004), to lead to the idea of LSRVH, which has an enhanced precision and comparable scalability with respect to plain LSRV contexts. Further, the thesis identifies cases where LSRVH may miss out on some precision-enhancement opportunities compared to k-object-sensitive analyses with heap cloning. In order to additionally capture the opportunities enabled by object-sensitivity, the thesis next adds k-object-sensitivity (Milanova *et al.*, 2005; Smaragdakis *et al.*, 2011) to LSRVH and proposes a new context abstraction termed LSRV$k$objH. This proposal also gives rise to a novel way of connecting the lattices of object-sensitive analyses and LSRV-contexts (and hence call-string based analyses). The thesis compares the precision and scalability of the proposed techniques with k-object-sensitive analyses by implementing them for performing control-flow analysis, and instantiating them for

$k = 1$. The results show that LSRV1objH not only generates a higher number of optimization opportunities, but also scales well to all the benchmarks under consideration.

1.3 Contributions of the Thesis

The main contributions of this thesis are listed below.

(i) Towards improving just-in-time analyses:

• The thesis proposes a new and efficient strategy to obtain precise analysis-results during just-in-time (JIT) compilation, and formalizes it as the PYE framework.

• The thesis introduces the novel notion of *conditional values* as a way to store the dependencies between an application and the libraries. These conditional values help in maintaining *partial summaries* for the application being analyzed statically, and generating final results during JIT compilation, without losing precision.

• The thesis instantiates PYE for performing two context-, flow-, and field-sensitive heap-based analyses (PACE and EASE), coupled with optimizations to store the generated partial summaries in a succinct manner, and to efficiently process the partial summaries at runtime.

• The thesis demonstrates the efficacy of PYE by performing an extensive evaluation of PACE and EASE in a production Java Virtual Machine (OpenJDK HotSpot JVM), and comparing the results with those generated by the existing implementations in the JVM. The evaluation shows that PYE fulfills both the promises it makes – enhanced *precision* of analysis results while maintaining the *efficiency* of the JIT compiler.

(ii) Towards improving static analyses:

• The thesis presents the novel idea of level-summarized relevant value-contexts (*LSRV-contexts*), which take into account an important observation that we do not need to compare the complete value-contexts while performing top-down context-sensitive heap analyses. This also helps classify more value-contexts as equivalent.

• The thesis devises a lightweight pre-analysis stage that gathers insights about the impact of a method on the callers' heap. The results of the pre-analysis are used to (i) further reduce the comparison of contexts in the "main-analysis" stage, and (ii) defer the analysis of "caller-ignorable" methods.

• The caller-ignorable methods are analyzed in a "post-analysis" stage, again in a context-sensitive manner (that is, without loss of precision). The proposed three-staged approach thus achieves the precision of a fully context-sensitive analysis for the whole program (including the JDK).

• The thesis also extends LSRV-contexts with heap cloning (to form LSRVH), and with object-sensitivity (to form LSRV$k$objH). The resultant ideas lead to a new connection between the LSRV and object-sensitivity lattices.

• The thesis presents an elaborate evaluation of LSRV-contexts on two nontrivial heap analyses (escape analysis and control-flow analysis), for DaCapo and JGF benchmarks. The results show that LSRV-contexts (i) succeed in analyzing previously unanalyzable benchmarks in less than 40 minutes; and (ii) significantly reduce the memory requirements. Further, the thesis evaluates the LSRVH and LSRV$k$objH approaches and compares them with k-object-sensitive control-flow analysis (for $k = 1$). The results show that LSRV1objH is a new sweet-spot that improves the precision of both LSRV-contexts based and object-sensitive analyses, while also scaling to large benchmarks.

1.4 Organization of the Thesis

This thesis is organized as follows. Chapter 2 gives an overview (along with references for further reading) of various preliminary concepts for understanding the concepts presented in the rest of the thesis. It begins with a discussion of points-to analysis, which is the basis of all the applications of the approaches proposed in this thesis. The chapter later highlights various precision and scalability related issues and opportunities in various points-to analyses. The chapter concludes with an overview of the analysis frameworks used in this thesis – for static as well as JIT analyses.

Chapter 3 describes the PYE framework, which is the main contribution of this the-

sis towards improving the precision and efficiency of JIT analyses. The chapter first describes PYE as a conceptual model compared to existing practices, then formalizes the various static and JIT components of PYE, followed by the description of two analysis instantiations: points-to analysis for null-check elimination, and escape analysis for synchronization elimination. This is followed by an extensive evaluation of PYE and its instantiations over a series of application programs from different benchmark suites, which establishes the proposed approach as a practical alternative to the currently adopted approach for Java program analysis.

Chapter 4 describes one of this thesis's contributions towards improving the scalability of precise context-sensitive static heap analyses. The chapter first highlights the challenges in call-string based and value-contexts based context-sensitive analyses, followed by insights based on a study to alleviate those challenges. This is followed by the description of the proposed three-staged approach that computes and uses a novel context abstraction called LSRV-contexts, which rely on the newly introduced notions of relevance and level-summarization. The chapter finally evaluates LSRV-contexts by comparing them with the standard value-contexts based implementations of escape and control-flow-analyses, asserting that LSRV-contexts significantly scale the analyses under consideration, while being able to analyze previously unanalyzable large benchmarks in a reasonable time.

Chapter 5 describes the contribution of this thesis towards improving the precision of LSRV-contexts, using heap cloning. The chapter also studies some differences between the optimization opportunities generated when heap cloning is added to LSRV-contexts versus to object-sensitivity. The chapter then describes a novel connection between the lattices of LSRV-contexts and k-object-sensitive analyses by combining the benefits of both into LSRV$k$objH contexts. Based on a comparison among the heap-cloning based approaches for $k = 1$, the chapter finally asserts LSRV1objH as a scalable-yet-precise context abstraction for static heap analyses.

Chapter 6 discusses some of the prior works related to the approaches presented in this thesis, for both just-in-time and static analyses. Finally, Chapter 7 concludes the work done as part of this thesis and highlights some possible future directions.

CHAPTER 2

BACKGROUND

This chapter describes various program-analysis concepts that are used in the rest of this thesis. The chapter first introduces *points-to analysis*, which is a fundamental program analysis that enables several other analyses and optimizations, for object-oriented languages like Java. It then describes *points-to graphs*, which are arguably the most common data structure used to compute points-to information for Java programs. This is followed with a discussion of three important applications of points-to analysis in Java: thread-escape analysis, null-check elimination, and control-flow analysis. The chapter next discusses the various dimensions along which the precision of a program analysis may vary, along with the challenges associated with scaling the same; a special focus is given to context-sensitivity, scaling which is an important goal of the techniques proposed in this thesis. The chapter finishes with an overview of the analysis frameworks used in this thesis for analyzing Java programs: Soot (Vallée-Rai *et al.*, 1999) for static analyses, and the HotSpot JVM (Paleczny *et al.*, 2001) for just-in-time analyses.

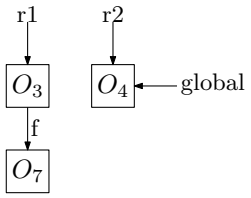
2.1 Points-to Analysis

Points-to analysis is a static program-analysis technique that establishes which pointers, or reference variables, may point to which objects or storage locations, at runtime. The results obtained by points-to analysis are key to several other heap analyses and related optimizations; for example, alias analysis, shape analysis, escape analysis, null-check elimination, call-graph construction, method inlining, and so on.

We represent objects with the line number at which they are allocated. We say a variable var may point-to a set S, if the elements of the set S represent the objects that may be pointed-to by the variable during program execution. For example, in the code shown in Figure 2.1a, the may-points-to sets of the reference variables `r1` and `r2` are $\{O_5\}$ and $\{O_6\}$, respectively.

```
1 class B {B f;}
2 class C {
3  static B global;
4  void foo() {
5     B r1 = new B();
6     B r2 = new B();
7     synchronized(r2) {...}
8     global = r2;
9     r1.f = new B();
10    List lst = new AList();
11    lst.add(r1);
12    B x = r1.f;
13    synchronized(x) {...}
14  }
15 }
```

(a)



(b)

Figure 2.1: (a) The method `foo` of class `B` shown in Figure 1.1. (b) The corresponding points-to graph after line 9.

There is often a cyclic dependence between the precision of points-to analysis and other optimizations. For example, points-to analysis helps resolve the set of methods that could be called at a method-call statement (resulting in a *call-graph*, which is imperative for *interprocedural* analyses). On the other hand, call-graph information is necessary to compute points-to facts across method calls. Thus, a precise points-to analysis allows the construction of a precise call graph, and a precise call graph is needed for computing precise interprocedural points-to information. Such cyclic dependencies are usually resolved by iterating the analysis being performed up to a fixed point.

2.2 Points-to Graphs

Points-to graphs and their variations are widely used (Whaley and Rinard, 1999; Dietrich *et al.*, 2015; Sălcianu and Rinard, 2005; Tan *et al.*, 2017) for representing the points-to relations in Java programs. A points-to graph $G(N, E)$ comprises of (i) a set N of nodes that represent variables and abstract objects in the program; and (ii) a set E of edges that represent points-to relationships among the nodes in the program. An edge can optionally have a label representing the field in the corresponding points-to relationship. For example, an edge (a, O_x) from a reference variable a to a node O_x in a points-to graph implies that the variable a may point to the object O_x. Similarly, an

edge $(O_x, \texttt{f}, O_y)$ from node O_x to O_y with a label `f` implies that O_x.`f` may point to O_y.

In this thesis, while analyzing a method m, the current points-to graph G_m for m returns the points-to information as follows: (i) $G_m(a)$ returns the points-to set of the variable a; and (ii) $G_m(O_x,\ \texttt{f})$ returns the points-to set of O_x.`f`. Figure 2.1b shows the points-to graph after line 9 for the code shown in Figure 2.1a. The points-to sets represented by the graph are: $G_{\texttt{foo}}(\texttt{r1})$ = $\{O_5\}$, $G_{\texttt{foo}}(\texttt{r2})$ = $\{O_6\}$, $G_{\texttt{foo}}(\texttt{global})$ = $\{O_6\}$, and $G_{\texttt{foo}}(O_5, \texttt{f})$ = $\{O_9\}$.

2.3 Applications of Points-to Analysis

We now discuss three important applications that use points-to information, namely, thread-escape analysis, null-check elimination, and control-flow analysis.

2.3.1 Thread-escape Analysis

Thread-escape analysis (Blanchet, 2003), hereafter called escape analysis, partitions the objects allocated in a thread *t* into two categories: (i) those that are local to *t* (that is, do-not-escape); and (ii) those that can be accessed by threads other than *t* (that is, escape). An object may escape to other threads if it is reachable (possibly via a sequence of field dereferences) from a static (global) variable, or from a thread object. Escape analysis has many applications: synchronization elimination (Blanchet, 2003; Ruf, 2000; Choi *et al.*, 1999), data-race detection (Choi *et al.*, 2002), efficient garbage-collection (Domani *et al.*, 2002), and so on. For example, the synchronization operation in the Java synchronization statement '`L: synchronized(`v`) S`' can be elided if the escape analysis finds that the object(s) pointed to by v do not escape before `L`.

Consider the Java code snippet shown in Figure 2.1a. Assume that the code shown by '...' does not affect the heap. In Figure 2.1a, the objects O_5, O_9 and O_{10} do not escape (assuming that the method `AList.add` does not make the objects reachable from its parameters escape). Further, O_6 does not escape until line 8. Thus, the synchronization operations at lines 7 and 13 can be safely elided.

Escape analysis can be performed using points-to graphs by checking whether a node in the points-to graph is reachable from static variables or nodes representing thread objects. Given a points-to graph G_m, we use a function $G_m.reachables(a)$ to get the nodes reachable from a in G_m. In Figure 2.1b, $O_6 \in G_{\texttt{foo}}.reachables(\texttt{global})$, and hence, the object O_6 in method `foo` escapes its allocating thread.

2.3.2 Null-check Elimination

In Java, before executing each statement that dereferences an object, the JVM needs to check whether the object being dereferenced is null; and if so, a *NullPointerException* must be thrown. Consider the Java code snippet shown in Figure 2.1a. In the method `foo`, the statements 7, 9, 11, 12 and 13 dereference objects. As variables `r1`, `r2` and `lst` point-to concrete objects allocated at lines 5, 6 and 9, respectively, the null-dereference checks at lines 7, 9 and 12 can be safely skipped. Further, the null-dereference checks at lines 12 and 13 can be skipped if the method `AList.add` does not set the object pointed-to via the field `f` of its first parameter to null. Thus, the number of null-dereference checks that can be skipped (or eliminated) depends directly on the precision of the underlying points-to analysis used.

2.3.3 Control-flow Analysis

Interprocedural control-flow analysis (Palsberg and Schwartzbach, 1991; Shivers, 1991), is used to determine the targets of a method call in dynamically-dispatched languages. This analysis is also a prerequisite to, and controls the precision of, several interprocedural analyses (for example, call-graph construction (Grove and Chambers, 2001), points-to analysis, escape analysis, and so on). A common way to perform control-flow analysis is by maintaining a points-to graph and using a dataflow lattice whose elements (indicating the possible type of each object) are the set of all the classes in the input program; the meet operation for the lattice is simply the set-union operation.

2.4 Analysis Dimensions

There are several dimensions along which one can vary the precision of a program analysis. In this section, we give an overview of some of the important analysis dimensions, with small examples wherever required, and discuss their effects on the precision of the points-to analyses discussed in the previous section.

2.4.1 Flow-sensitivity

An analysis is considered flow-sensitive, if it respects the control flow in the program, that is, it maintains information specific to each program point. On the other hand, a flow-insensitive analysis computes a single information independent of the order of the program statements. For the code shown in Figure 2.1a, a flow-insensitive escape analysis would simply conclude that the object O_6 escapes, and hence the synchronization operation at line 7 would not be elided. A flow-sensitive analysis, on the other hand, will be able to identify that O_6 does not escape at line 7, and hence will be able to elide the synchronization operation at line 7 successfully.

The standard way to perform a flow-sensitive analysis is to maintain IN and OUT dataflow sets with each statement, and then use a worklist-based algorithm until a fixed-point is reached: if the OUT set (IN set in a backward analysis) of any statement st changes, then the successors (predecessors in a backward analysis) of st are added to the worklist.

2.4.2 Field-sensitivity

A program analysis is field-sensitive, if it maintains the information for each field of an object separately. On the other hand, a field-insensitive analysis aggregates the information of all the fields of an object into the information for that object. For example, after line 9, a field-sensitive analysis would maintain separate points-to sets $\{O_5\}$ and $\{O_6\}$ for `r1` and `r1.f`, respectively. A field-insensitive analysis, on the other hand, will maintain a single (conservative) points-to set $\{O_5, O_6\}$ for `r1` as well as `r1.f`.

In general, it is difficult to perform field-sensitive analyses on data structures whose size is not known at compile time (for example, arrays). Even if the upper bound on the number of elements is known, it may not be feasible to store information for all the elements of a list or an array, because of huge memory requirements. It is common practice to maintain a summary of such data structures (say with a single field "[]"), whose (conservative) information represents all the constituent elements.

2.4.3 Analysis Scope

The scope of an analysis is said to be intraprocedural, if it computes the information for a method independent of the effects of the call sites within that method. Such an analysis assumes some unknown conservative value for the arguments passed to, and the return value of, a callee method at a call statement. An interprocedural analysis tries to accommodate the effects of the callees at the call sites within the method being analyzed. Such an analysis needs the call graph of the program, and the results are heavily dependent on the precision of the call graph. Interprocedural analyses can be performed either: (i) *top-down*, where at each call site, the potential callees are analyzed to get their potential effects; or (ii) *bottom-up*, where the effects of a method are computed in prior, and used at the call sites.

In case of an interprocedural analysis for the code snippet shown in Figure 2.1a, if the analysis for the method `AList.add` tells that the objects pointed-to by its parameters do not escape `AList.add`, then the synchronization operation at line 13 can be removed. An intraprocedural analysis, on the other hand, must (conservatively) assume that all the objects passed to `AList.add` might escape, and hence will not be able to elide the synchronization operation at line 13.

2.4.4 Context-sensitivity

An interprocedural analysis is context-insensitive, if it stores a single information for a method, and the same information is used at each call-site where that method may be called. On the other hand, a context-sensitive analysis may store different information for the different contexts in which a method may be called. A context-sensitive

```
1 class A {A f;}
2 class C {
3    static A tee;
4    public void foo() {
5       A a1 = new A();
6       tee = a1;
7       A r1 = a1.bar();
8       a1 = new A();
9       A r2 = a1.bar();
10      synchronized(r2) {...}
11   }
12   public A bar() {
13      A obj = new A();
14      obj.f = this;
15      return obj.f;
16   }
17 }
```

Figure 2.2: Example to motivate context-sensitivity.

analysis lets the caller use the behavior of a method specific to the calling context, and is likely to provide more precise results for various pointer analyses. One of the oldest and widely-used approaches of defining what constitutes a context is the classical call-strings approach (Sharir and Pnueli, 1978; Shivers, 1991), which identifies contexts based on the call-string formed by a method's callers.

Consider the Java code snippet shown in Figure 2.2. A context-insensitive analysis maintains a single (conservative) summary for the method `bar`. Consequently, it would conclude that the object O_{13}, and hence the object pointed-to by `r2`, escapes, and thus would not be able to elide the synchronization operation at line 10. A call-string based context-sensitive analysis, on the other hand, analyzes `bar` separately for the calls at lines 7 and 9. Consequently, it would be able to identify that in the second call to `bar`, the object O_{13}, and hence the object pointed-to by `r2`, does not escape, and thus would be able to elide the synchronization operation at line 10.

A major drawback of the call-strings approach is that in the presence of recursion and deep nesting of multiple calls, the length of the call-strings, and hence the number of contexts, may grow combinatorially. This makes the analysis unscalable to large real-world programs. Consequently, call-string based analyses usually impose a limit on the call-string length, and treat the contexts of greater lengths conservatively. While such an approach improves the scalability, it compromises on the resulting precision.

Value-contexts. The idea of value-contexts, proposed by Khedker and Karkare (2008), was used by Padhye and Khedker (2013) to perform top-down context-, flow-, and field-sensitive points-to analysis for constructing a call-graph. The analysis starts from the main method, and maintains a points-to graph at each statement. On reaching a call-statement for a method m, the method is (re-)analyzed, if the current value-context is different from the prior value-contexts (if any) in which m was analyzed. Here, the value-context at a call to m is the points-to (sub) graph passed to m – referred to as the *parameter-reachable graph* of m.

Object-sensitivity. An object-sensitive analysis (Milanova *et al.*, 2005) distinguishes the contexts of a method based on the allocation site of the receiver object (the object pointed-to by the `this` pointer). Similar to call-string based analyses, for scalability, object-sensitive analyses also use a limit k on the length of the chain formed by the receivers. For example, a one-level object-sensitive analysis would analyze the method `bar` (see Figure 2.2) in two contexts – at lines 8 and 10 – as the receiver objects at both the sites are different (O_6 and O_9).

Observe that the contexts created in the call-string based and object-sensitive approaches are quite different: in the former, the contexts created for a method can be directly mapped to the runtime call-stack; whereas in the latter, the contexts created depend on the possible receiver objects. Consequently, the per-context precision of object-sensitive analyses cannot be compared with those of call-string (and hence value-contexts) based analyses.

2.5 Analysis Frameworks

Java follows a two-stage program translation mechanism. In order to guarantee platform independence, the static Java compiler first translates Java programs to platform-independent Bytecodes in the form of class files. These class files are then transferred to the target machine, where based on certain heuristics such as the number of times a method is executed, the Bytecodes are either interpreted, or compiled just-in-time (JIT), or interpreted as well as compiled, by a Java Virtual Machine (JVM). As a consequence of this two-stage translation, a Java program can be analyzed at two levels: during

static compilation, and during JIT compilation. We now give an overview of the Soot optimization framework (which we use for writing static analyses), and the OpenJDK HotSpot JVM (which we use for writing JIT analyses).

2.5.1 The Soot Framework

Soot (Vallée-Rai *et al.*, 1999) is a Java optimization framework. Soot can either be used to analyze as well as to develop optimizations and transformations on Java Bytecode. It is freely available and is licensed under the GNU Lesser General Public License, and has been extensively used in various works on program analysis for Java programs.

Soot provides four intermediate representations for analyzing and transforming Java Bytecode: Baf, Jimple, Shimple, and Grimp. Jimple, our choice for implementing the static analyses proposed in this thesis, has a small set of typed three-address instructions, and Soot provides various inbuilt methods to extract and manipulate Jimple statements. Each Jimple instruction has at most three operands, wherein temporary variables are used to break down complex instructions. For simplicity, all declarations are done at the beginning of a method, and the control flow is modeled using `goto` instructions and labels.

2.5.2 The HotSpot JVM

The HotSpot JVM (Paleczny *et al.*, 2001; Kotzmann *et al.*, 2008), shipped by Oracle, is a popular production JVM for executing Java Bytecode. In this thesis, we use the OpenJDK HotSpot JVM, which is the open-source version of Oracle's production version. The HotSpot JVM ships with an interpreter and two JIT compilers: (i) client or C1; and (ii) server or C2. The C1 compiler aims at fast compilation and performs simpler optimizations. The C2 compiler, on the other hand, performs sophisticated optimizations, with a corresponding effect on the compilation time.

In order to execute a program, the HotSpot JVM starts with interpreting the Bytecode, while performing extensive profiling in parallel. Based on the profiled information, when the number of times a method is called or a loop is executed exceeds a

particular threshold (that is, becomes "hot"), it is deemed to be fit for compilation. Depending on several different heuristics, it is either compiled by C1 or by C2, in one of total four modes of compilation. During JIT compilation, the compiler makes several assumptions about the code (for example, the target of a call-statement), and if the assumptions fail (say due to dynamic classloading or branch misprediction), the compiled unit is discarded and the control for that unit is given back to the interpreter (in a process called deoptimization).

Apart from various other analyses and optimizations, the C2 compiler of the HotSpot JVM performs aggressive analyses and optimizations based on points-to information, including null-check elimination, escape analysis for synchronization elimination, control-flow analysis for call-graph construction, and so on. Improving the precision and efficiency of many of these analyses is the aim of our techniques in this thesis, and we would keep discussing them in the later chapters.

CHAPTER 3

PRECISE AND EFFICIENT JUST-IN-TIME ANALYSES

This chapter describes our contributions towards performing precise and efficient JIT analyses for Java programs. First, Section 3.1 describes our proposed framework PYE along with a novel notion of conditional values. Sections 3.2 and 3.3 discuss the design of the two instantiations of PYE, that is, PACE and EASE, respectively. Section 3.4 highlights some subtle aspects in the design of PYE and proves its correctness. Finally, Section 3.5 presents a detailed evaluation of PACE and EASE.

3.1 The PYE Framework

In this section, we first discuss some of the challenges in typical modular dataflow analysis techniques, and then describe PYE in the context of analyzing Java applications.

3.1.1 Typical Modular Analyses

To maintain scalability, typical modular analyses (Whaley and Rinard, 1999; Choi *et al.*, 1999) process one method at a time and maintain its *summary*. For a given dataflow analysis Ψ, the summary of a method m can be seen as a map f_m from the domain $\mathcal{D}$ of Ψ to the set of dataflow values `Val` of Ψ. That is:

$$f_m : \mathcal{D} \rightarrow \texttt{Val} \tag{3.1}$$

We assume that `Val` forms a lattice with a meet operation $\sqcap$, a supremum $\top$ (the most precise element), and an infimum $\bot$ (the most conservative element). For example, in typical escape analysis algorithms (Bogda and Hölzle, 1999; Ruf, 2000; Blanchet,

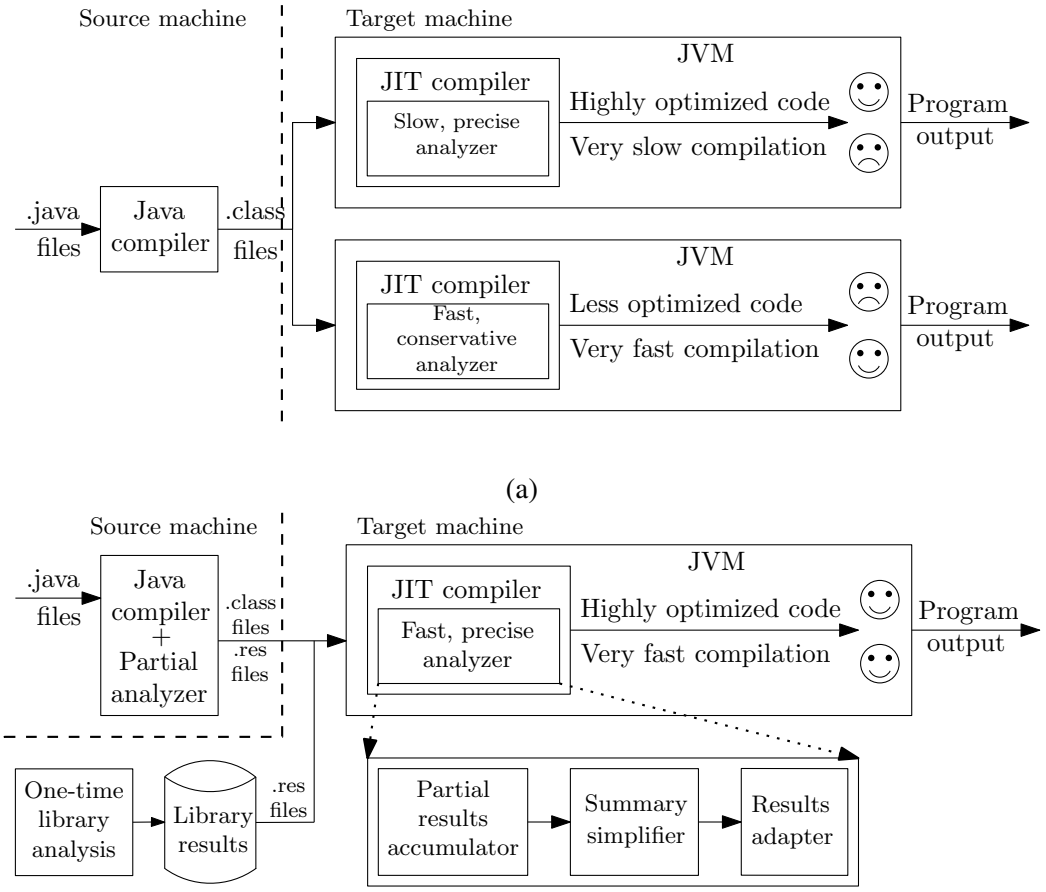


Figure 3.1: Java program analysis: (a) Two traditional approaches. (b) Approach adopted by PYE.

2003), (i) $\mathcal{D}$ includes object-allocation sites, function parameters and return values; and (ii) the lattice `Val` has elements from the set $\{\top, \bot\}$, organized as a chain, indicating the escape-status ($\top$ = *DoesNotEscape*, and $\bot$ = *Escapes*).

For a method m, its summary f_m may depend on the summaries of a set of other methods. Thus to compute f_m precisely, all the dependent summaries must be available. In the context of JIT compilation (for example, in the HotSpot JVM (Paleczny *et al.*, 2001)), the summaries dependent on the runtime-libraries can only be computed at runtime. This can usually be achieved using one of the two approaches shown in Figure 3.1a. A JIT compiler can perform either very precise analyses and incur the large overheads caused by the compilation time, or it can target fast compilation time, and perform imprecise analyses. Note that there could be several other configurations that explore the trade-offs between these two approaches, such as k-limited context-

sensitivity (Sharir and Pnueli, 1978), flow-insensitivity (Hardekopf and Lin, 2007), cutoff-based approaches (Vivien and Rinard, 2001), and so on. However, for simplicity and efficiency, typical JIT compilers (such as the ones in the HotSpot (Paleczny *et al.*, 2001) and Jikes (Alpern *et al.*, 2005) virtual machines) limit themselves to mostly intraprocedural analyses. Even though many JIT compilers may perform early-inlining, its impact on the precision of the analysis is limited, due to the standard restrictions (Paleczny *et al.*, 2001) on inlining (such as the iCache size, deep nesting of methods, recursion, profiling, and so on).

An alternative to costly analyses during JIT compilation is to perform the analysis statically at compile time and export the results to the JVM. However, as the JDK installation on the source machine (where the analysis is performed) may be different from the target machine (on which the analysis results will be used), using such results may lead to unsound optimizations. Examples of such changes include the removal or addition of methods (for example, compared to Java 8, *LogManager.addPropertyChangeListener* was removed in Java 9), changes in method signatures, newer implementations overriding parent-class methods, and so on. As a result, all the referred library methods are considered unavailable at compile time, and the summary of each library method is conservatively approximated to the special "bottom" function $\lambda x.\bot$. Such a scheme can lead to overly imprecise results.

3.1.2 PYE: A Practical Alternative

To overcome the issues of both (i) fully static analysis (too imprecise), and (ii) whole-program analysis at runtime (prohibitively expensive), we propose PYE: a framework for **p**recise-**y**et-**e**fficient just-in-time analyses for Java programs. Figure 3.1b shows the block diagram of PYE. Compared to the traditional scheme of analysis in the JIT compiler, PYE analyzes an application using two inter-related components: the *partial-analyzer* (added to the static Java compiler), and the *fast-precise-analyzer* (added to the JIT compiler of the JVM).

For each method in the application being analyzed, the partial-analyzer skips the calls to the unavailable library methods, and generates *partial summaries*. Compared

to a traditional method-summary f_m (a map from the domain $\mathcal{D}$ to one of the dataflow values in `Val`), a partial summary maps each element in the domain of the analysis to a set of *conditional values*. We propose the novel notion of conditional values as a way to encode the dependence of the analysis results for the associated program element on other unavailable program element(s). For each application, the partial summaries generated by the partial-analyzer are stored as a ".res file". Similarly, another instance of the partial-analyzer analyzes the library methods of the target machine offline, independent of the application, and generates a partial summary for each library method. As the static-compilation time does not get added to the execution time of the program, an analysis-writer using PYE is free to pick highly precise variants of analyses to be performed by the partial-analyzer.

As shown in Figure 3.1b, when a program is executed, along with the .class files (of the application and the libraries), the corresponding partial summaries in the form of .res files are made available to the JVM on the target machine. In the JIT compiler, the fast-precise-analyzer reads the required partial summaries (*partial-results-accumulator*), resolves the dependencies between the application and the library summaries to generate final results (*summary-simplifier*), and then populates the appropriate JVM data structures to perform the related optimizations (*results-adapter*).

We first describe the notion of conditional values as a way to encode the dependencies between various parts of a program, and define partial summaries (Section 3.1.3). Later, we describe how these partial summaries are simplified to obtain precise results during JIT compilation (Section 3.1.4).

3.1.3 Partial Summaries

For a given dataflow analysis Ψ, to compute the summary f_m for each method m, the partial-analyzer of PYE first computes the *partial summary* g_m, which is a map from the domain $\mathcal{D}$ of Ψ to the power set of *conditional values* (`CVal`):

$$g_m : \mathcal{D} \rightarrow P(\texttt{CVal}) \tag{3.2}$$

```
1 class C {
2   void foo(D p) {
3     A q = p.bar();
4   }
5 }
```

```
1  interface D { A bar(); }
2  class D1 implements D {
3    A bar() {
4      return new A1();
5    }
6  }
7  class D2 implements D {
8    A bar() {
9      return new A2();
10   }
11 }
```

Figure 3.2: Example to illustrate the generation of conditional values.

Definition 1. *A conditional value is a three-tuple of the form*

$$\mathcal{T} = \langle \Theta, val, val' \rangle$$

where:

- Θ *is a pair of the form* $\langle u, x \rangle$*, in which:*
  - u *is a method*
  - x *is a program element in* u
- val *and* val' *are elements from the set* `Val` *of* Ψ

A conditional value $\mathcal{T} = \langle \langle u, x \rangle, val, val' \rangle$ in $g_m(e)$ encodes a condition on the final analysis-result $f_u(x)$ for the element x in method u. That is, if $f_u(x) = val$, then $\mathcal{T}$ *evaluates* to val'. If u is a library method, then $f_u(x)$ is not available statically while analyzing the application, and hence it is not possible to resolve the dependence (and evaluate $\mathcal{T}$) till runtime.

Example. Consider the code snippet in Figure 3.2. Suppose the goal is to perform a flow-analysis that maps each reference variable (such as `p` and `q` in the example) to the set of classes whose objects could be pointed-to by the variable. The parameter `p` could either point to an object of class `D1` or of class `D2`. Considering the flow-analysis starting from the method `foo`, the set of conditional values generated by the partial-analyzer for `q` would be:

$$g_{\texttt{foo}}(\texttt{q}) = \{\langle\langle \texttt{foo}, \texttt{p}\rangle, \{\texttt{D1}\}, \{\texttt{A1}\}\rangle, \langle\langle \texttt{foo}, \texttt{p}\rangle, \{\texttt{D2}\}, \{\texttt{A2}\}\rangle\}$$

Here, the conditional values indicate that the flow-set of `q` would include `A1` if the flow-set of `p` is {`D1`}, and the flow-set of `q` would include `A2` if the flow-set of `p` is {`D2`}.

In Section 3.1.4, we explain how such conditional values are simplified in a systematic manner.

Notation. If the analysis-result of an element e does not depend on any other element, then $g_m(e)$ is a singleton with a lone conditional value, whose condition is a tautology. We denote the set of such *simple* ("vacuously true") conditional values as `SCVal`, which can be seen as the conditional representation of the set `Val`. We represent each $v \in$ `Val` as a conditional value $\langle \Theta_v, v, v \rangle \in$ `SCVal`, where Θ_v can be seen as a special global element for which $f_m(\Theta_v)$ is set to v for all methods m. We refer to the rest of the conditional values, that is the ones in the set `CVal` $-$ `SCVal`, as *dependent* conditional values, `DCVal`.

Say x and y are program elements in methods u and m, respectively. In analyses like PACE (Section 3.2) and EASE (Section 3.3), in each conditional value, the respective third and the fourth components match, and if $\exists\ v \in$ `Val`, such that $\langle \langle u, x \rangle, v, v \rangle \in g_m(y) \Rightarrow \forall v' \in$ `Val`, $\langle \langle u, x \rangle, v', v' \rangle \in g_m(y)$. For such analyses, for brevity, we abbreviate the set of all the conditional values dependent on $\langle u, x \rangle$, by only $\langle u, x \rangle$. For uniformity, a simple conditional value $\langle \Theta_v, v, v \rangle \in$ `SCVal` in those analyses is abbreviated to Θ_v.

Example. We now illustrate the above-discussed concepts using another analysis that performs a points-to analysis to elide the null-dereference-checks for which the dereference is guaranteed to be performed on a concrete object. Say the set `Val` for this analysis is $\{C$ (for *Concrete*), N (for *Null*)$\}$. The simple conditional values corresponding to C and N are Θ_C and Θ_N, respectively. We use the code shown in Figure 1.1 as the input for this analysis. The conditional values generated by the partial-analyzer for each dereference O_l (denoting the dereference at line l) in the method `bar` are:

$$g_{\texttt{bar}}(O_{13}) = g_{\texttt{bar}}(O_{14}) = \{\Theta_C\}$$
$$g_{\texttt{bar}}(O_{15}) = \{\langle \texttt{AList.add}, O_{p_1}\texttt{.f} \rangle\}$$
$$g_{\texttt{bar}}(O_{16}) = \{\langle \texttt{AList.add}, O_{p_1}\texttt{.f.g} \rangle\}$$

Here, as the variables `r1` and `r2` point to concrete objects allocated at lines 11 and 12 respectively, the set of conditional values for the dereferences at lines 13 and 14 is the singleton Θ_C. The conditional value for the dereference at line 15 indicates

that the dereferenced object would be null if the object pointed-to by O_{p_1}.f, where O_{p_1} is the object pointed-to by the first parameter of AList.add, is null, and concrete otherwise. Similarly, the conditional value for the dereference at line 16 indicates that the dereferenced object would be null (or concrete) if the object pointed-to by O_{p_1}.f.g is null (or concrete).

3.1.4 Simplifying Partial Summaries

The partial-analyzer generates a partial summary g_m for each method m statically available for analysis. Say, the set of all such summaries for an application $\mathcal{A}$ is $\mathcal{F}_{\mathcal{A}}$. On the target machine, another instance of the partial-analyzer computes a similar summary for each library method offline; say the combined set for a library installation $\mathcal{L}$ is $\mathcal{F}_{\mathcal{L}}$. At runtime, the summaries generated by the partial-analyzer for all the methods (the application as well as the library) are available. When the application is executed by the JVM on the target machine, the fast-precise-analyzer of PYE takes $\mathcal{F}_{\mathcal{A}}$ and $\mathcal{F}_{\mathcal{L}}$, and computes the final analysis-results (in the summary-simplifier; see Figure 3.1b) for all the elements of a method m that is compiled just-in-time by the JVM.

For each element e in the analysis-domain for the method m being compiled by the JIT compiler, the summary-simplifier of the fast-precise-analyzer evaluates each conditional value $\mathcal{T} = \langle\langle u, x\rangle, val, val'\rangle \in g_m(e)$, by looking up the value of $f_u(x)$, and returns the evaluated value $[\![\mathcal{T}]\!] \in$ Val. If the condition specified in $\mathcal{T}$ evaluates to true (that is, $f_u(x)$ = val), then $[\![\mathcal{T}]\!]$ is val'; else $[\![\mathcal{T}]\!]$ is $\top$ (the top value of the lattice Val):

$$[\![\langle\langle u, x\rangle, val, val'\rangle]\!] = (f_u(x) == val)\ ?\ val'\ :\ \top \tag{3.3}$$

For analyses where val is always equal to val', we use the shorthand notation introduced in Section 3.1.3, and simplify equation 3.3 as:

$$[\![\langle u, x\rangle]\!] = f_u(x) \tag{3.4}$$

Finally, given the set of evaluated conditional values $S = \{[\![\mathcal{T}]\!] \mid \mathcal{T} \in g_m(e)\}$, the

analysis-result $f_m(e)$ is computed as the meet over all the elements in S. The semantics of the meet operation is specific to the individual analysis under consideration. Thus, if the meet operation of an analysis is denoted by $\sqcap$, then:

$$f_m(e) = \sqcap_{\forall \mathcal{T} \in g_{m(e)}} [\![\mathcal{T}]\!] \tag{3.5}$$

Given a set of program elements and their conditional values, evaluating all the conditional values may require repeated solving of equations 3.3 (or 3.4) and 3.5 till a fixed-point. Such a fixed-point computation is necessary to take into consideration the dependence of the conditional values of one program element on those of other program elements. We do so by using a standard worklist-based algorithm in the summary-simplifier of the fast-precise-analyzer. As `Val` is a finite-height lattice, the evaluation is guaranteed to terminate and give us the most precise solution for the conditional values generated by the partial-analyzer. The presence of un-evaluated conditional values even after attaining a fixed-point indicates mutually cyclic dependencies. In such cases, we use $f_m(e) = \top$ for each element involved in the cycle (as we have already achieved a fixed-point, it is safe to do so). The cost of performing this fixed-point computation mostly depends on the number of dependent conditional values (theoretically bound by the product of the number of abstract objects in a program and the size of the lattice `Val`, but usually a small percentage compared to the total number of program elements) generated for a particular analysis. Further, the amortized cost required to resolve a dependence (one lookup per dependence) is $O(1)$. As we show in Section 3.5, for the analyses under consideration, the time spent in summary-simplification is very small (order of milliseconds).

Example. For the code shown in Figure 1.1, the partial summary generated by the partial-analyzer for the method `AList.add`, after analyzing the library offline, would be: $g_{\texttt{AList.add}}(O_9) = \{\Theta_C\}$, which indicates that the dereference performed at line 9 in `AList.add` is done on a concrete object, and the absence of any information about its parameters implies that none of them are modified in the method. Note: The meet ($\sqcap$) operation for this analysis is defined as: $\sqcap(C, C) = C$ and $\sqcap(C, N) = \sqcap(N, C) = \sqcap(N, N) = N$. While compiling the method `bar`, the fast-precise-analyzer looks up the partial summary of the method `AList.add` to resolve

the conditions for the elements of `bar`, and generates the following final analysis-results for the dereferences in `bar` (after solving the equations 3.4 and 3.5):

$f_{\texttt{bar}}(O_{13}) = f_{\texttt{bar}}(O_{14}) = \sqcap\{[\![\Theta_C]\!]\} = \sqcap\{C\} = C$

$f_{\texttt{bar}}(O_{15}) = \sqcap\{[\![\langle \texttt{AList.add}, O_{p_1}\texttt{.f}\rangle]\!]\} = \sqcap\{C\} = C$

$f_{\texttt{bar}}(O_{16}) = \sqcap\{[\![\langle \texttt{AList.add}, O_{p_1}\texttt{.f.g}\rangle]\!]\} = \sqcap\{C\} = C$

// $\langle \texttt{AList.add}, O_{p_1}\texttt{.f}\rangle$ and $\langle \texttt{AList.add}, O_{p_1}\texttt{.f.g}\rangle$ cannot be simplified further,

// and hence evaluate to $\top = C$.

The results indicate that all the dereferences in the method `bar` are done on concrete objects, and hence, the corresponding null-pointer-dereference checks can be safely elided. Compared to the results generated by the analysis alternatives A1 and A2 in Section 1.1, it is evident that PYE is able to achieve a higher precision by combining the partial summaries for the application and the library at runtime. We show in Section 3.5 that this precision comes at a very low cost; that is, with negligible associated overheads.

3.1.5 Efficient Storage of Partial Summaries for PYE

The partial summaries generated by the partial-analyzer of PYE for each application are stored in a .res file on the machine where the analysis is performed. This .res file needs to be transferred to the target machine, along with the .class files for the application. On the target machine, the .res files for the application and the libraries are read by the JVM during execution. The speed of performing all the above operations depends a lot on the size of these .res files. Thus, smaller the .res files, fewer will be the storage, transfer, and file-reading overheads. For each analysis implemented in PYE, in order to efficiently maintain and store the partial summaries, we perform an optimization to pre-apply the meet operation in the partial-analyzer: For each method m, for each $g_m(x)$, we pre-apply the meet operation on the conditional values and store either only a single simple conditional value, or one or more dependent conditional values along with at most one simple conditional value. For example, consider the null-check removal analysis discussed in Section 3.1.3. Say for an object O in the method m, a dependent conditional value $\langle u, x\rangle$ gets added to $g_m(O)$. If $g_m(O)$ previously consisted

of a simple conditional value Θ_C, we remove Θ_C from $g_m(O)$ and only keep $\langle u, x \rangle$. This optimization reduces the number of conditional values we carry while performing the static analysis.

3.1.6 Writing an Analysis in PYE

PYE can, in general, be used to perform any whole-program modular dataflow analysis (for languages like Java/C#) having: (i) a finite-height lattice of dataflow values, (ii) inter-dependent application and library analysis-results, and (iii) dynamically-refinable static-analysis results. Examples include the inclusion-based points-to analysis (Andersen, 1994), unification-based analysis (Steensgaard, 1996), partial escape analysis (Stadler *et al.*, 2014), MHP analysis (Naumovich *et al.*, 1999), and so on. In this section, we give an overview of how to implement an existing analysis Ψ in PYE. First, the analysis writer needs to specify the domain $\mathcal{D}$ of Ψ, and the lattice formed by the dataflow values `Val` of Ψ. In the modified analysis, each value $v \in$ `Val` is converted to a special conditional value of the form Θ_v. As part of the partial-analyzer, the modified analysis then processes each statement similar to Ψ, except for the following three scenarios, which need to take into consideration the generation of conditional values:

(i) Unavailable callee. Say while analyzing a method, we encounter a call to an unavailable method (say from a library). Here, the analysis writer needs to encode the dependence of the actual arguments on the method u, using conditional values.

(ii) Unavailable caller. Here, the analysis writer needs to encode the dependence of the formal parameters on the actual arguments that may be passed to the method u, using conditional values.

(iii) Unavailable object-dereference. Say we encounter a load statement of the form $a = b.f$, and b depends on another element from an unavailable method; for example, b holds the return value of a library method u. Here, the analysis writer needs to encode the dependence of $b.f$ on the method u, using conditional values.

As an example of how the dependencies need to be encoded, consider the call to the unavailable method `AList.add` at line 11 in Figure 2.1a. For the object O_5 pointed-to by `r1`, at this call, a traditional static escape analysis would record the escape-status of

O_5 to be the value E (or *Escapes*). Whereas the same analysis implemented in PYE would record the fact that the escape-status of O_5 depends on the first parameter of `AList.add`, using the conditional value $\langle$`AList.add`$, O_{p_1}\rangle$.

In addition to the above changes in the partial-analyzer, the analysis writer needs to provide the implementation of the results-adapter in the fast-precise-analyzer. This simply involves populating the appropriate data structures in the JVM from the results generated by the summary-simplifier, such that they can be accessed directly by the optimizers of the JIT compiler. Note that an otherwise fully just-in-time analysis, in addition to writing the analysis, also needs to appropriately populate the data structures for any dependent optimization passes, and hence the effort required to do the same in PYE is arguably never more.

Overall, PYE achieves the precision of a whole-program analysis with very low analysis overheads at runtime. As can be seen, PYE replaces complex program-analysis phases of the JIT compiler with basic operations like reading the pre-computed summaries and simplifying the summaries based on the summaries of other methods. This strategy pays off quite well by improving the precision of analysis results without significantly affecting the time required for JIT compilation (see Section 3.5).

3.2 Points-to Analysis for null-Check Elimination

In this section, we illustrate the usage and effectiveness of PYE, by using it to efficiently perform a top-down context-, flow-, and field-sensitive points-to analysis for null-pointer-dereference check elimination (or PACE, in short) in Java programs. The analysis is based on points-to graphs (see Section 2.2 for an overview), and is used to remove the implicit null-dereference checks in translated Java programs for the dereferences that are guaranteed to be made on concrete objects (as discussed in Section 1.1).

As mentioned in Section 3.1.6, in order to perform an analysis using PYE, we need to specify the set $\mathcal{D}$, the lattice formed by `Val`, the processing of each relevant statement by the partial-analyzer, and the results-adapter. We now describe the same for PACE.

Set		Description
$\texttt{ALC}_m$	$=$	$\{O_l \mid l$ is an allocation statement in method $m\}$
$\texttt{PAR}_m$	$=$	$\{O_{p_i} \mid p_i$ is the i^{th} parameter of method $m\}$
$\texttt{RET}_m$	$=$	$\{O_x \mid O_x$ is returned by $m\}$
$\texttt{OUT}_m$	$=$	$\{O_{d@l} \mid O_{d@l}$ represents an unavailable object dereference at line l in $m\}$
$\texttt{UCS}_m$	$=$	$\{O_{u@l} \mid l$ is a call-statement in method m and the callee u is unavailable$\}$
$\texttt{DRF}_m$	$=$	$\{O_l \mid l$ is an object-dereferencing statement in method $m\}$ *// specific to PACE*
$\texttt{SYN}_m$	$=$	$\{O_l \mid l$ is a synchronization statement in method $m\}$ *// specific to EASE*
		Domain of program elements
$\mathcal{D}_{pace}$	$=$	$\forall m\ \texttt{ALC}_m \cup \texttt{PAR}_m \cup \texttt{RET}_m \cup \texttt{OUT}_m \cup \texttt{UCS}_m \cup \texttt{DRF}_m$
$\mathcal{D}_{ease}$	$=$	$\forall m\ \texttt{ALC}_m \cup \texttt{PAR}_m \cup \texttt{RET}_m \cup \texttt{OUT}_m \cup \texttt{UCS}_m \cup \texttt{SYN}_m$
		Lattice of dataflow values
$\texttt{Val}_{pace}$	$=$	$\{$*Concrete* (C *or* $\top$), *Null* (N *or* $\bot$)$\}$
$\texttt{Val}_{ease}$	$=$	$\{$*DoesNotEscape* (D *or* $\top$), *Escapes* (E *or* $\bot$)$\}$
Meet_{pace}	:	$C \sqcap C = C;\ C \sqcap N = N \sqcap C = N \sqcap N = N$
Meet_{ease}	:	$D \sqcap D = D;\ D \sqcap E = E \sqcap D = E \sqcap E = E$

Table 3.1: The definitions of $\mathcal{D}$ and $\texttt{Val}$ for the analyses PACE and EASE.

3.2.1 Partial-analyzer for PACE

Table 3.1 shows the domain $\mathcal{D}_{pace}$ of relevant program elements and the lattice of dataflow values $\texttt{Val}_{pace}$ for PACE. For each method m, $\mathcal{D}_{pace}$ consists of six sets of abstract objects: (i) $\texttt{ALC}_m$: one object O_l per allocation statement labeled l; (ii) $\texttt{PAR}_m$: one object O_{p_i} representing the objects pointed-to by the parameter p_i; (iii) $\texttt{RET}_m$: all the objects returned by m; (iv) $\texttt{OUT}_m$: one object $O_{d@l}$ for an unavailable dereference at a statement labeled l; (v) $\texttt{UCS}_m$: one object $O_{u@l}$ per unavailable method u at a call-statement labeled l, indicating the return-value of u at l; and (vi) $\texttt{DRF}_m$: one object O_l per object-dereferencing statement labeled l.

The set $\texttt{Val}_{pace}$ forms a lattice with two elements: *Concrete* (C or $\top$) and *Null* (N or $\bot$). The corresponding conditional values are Θ_C and Θ_N, respectively. The definition of the meet ($\sqcap$) operation is standard (see Table 3.1). Note that the lattice for the underlying points-to analysis for PACE (and for EASE) is standard (Whaley and Rinard, 1999), with the corresponding meet operation defined as the union of the points-to sets; the dataflow values for PACE (and EASE) rely on the information computed by this points-to analysis. On the other hand, while performing PACE and EASE, it

is possible to query the underlying points-to analysis to obtain the points-to sets of different variables and field references.

Our static analysis is a standard top-down, forward, context-, flow-, and field-sensitive iterative dataflow analysis. The analysis of an application begins at the entry of the main method of the application. For analyzing the libraries (on the target machine), we start the analysis afresh at the entry of each public method of the library. For simplicity, we assume that each intraprocedural Java statement is in a "three address" representation, and that a field-dereference occurs to the right-hand side of an assignment only in a load statement of the form $a = b.f$. Further, we skip a detailed discussion of statements involving array references, and briefly highlight the changes required to process them, if any, while discussing the statements of a similar form. For the ease of analysis, we assume that each method has two special statements `entry` and `exit`, denoting the single point of entry and exit, respectively. We also assume that each statement has a unique label associated with it.

For each method m, we maintain two data structures before and after each statement: (i) a points-to graph G_m (see Section 2.2 for an overview); and (ii) the partial summary g_m, which is a map from abstract objects to a set of conditional values. We use a worklist-based algorithm and analyze the statements of a method repeatedly till a fixed-point. After analyzing a method m, instead of storing the analysis information at each program point, we store the points-to graph (standard rules) and the partial summary as observed at the `exit` of m – which reduces the size of the partial summary. Even then, we still realize flow-sensitivity for performing null-dereference check elimination as we separately track each of the object dereferences (the set $\texttt{DRF}_m$); see Section 3.4 for a detailed discussion. While this may theoretically increase the number of objects in the points-to graph by $O(N)$, where N is the program size, overall this scheme reduces the amount of stored information, as it avoids storing the points-to graph at each instruction.

We now describe the processing of each statement that could affect either the points-to graph or the set of conditional values for any element. Figure 3.3 shows the inference rules for updating the points-to graph G_m and the partial summary g_m while analyzing the statements of a method m.

[allocation]	$l : a = new\ B();$	$\begin{cases} g_m[O_l \leftarrow \{\Theta_C\}] \\ G_m[a \leftarrow \{O_l\}] \end{cases}$
[null-assignment]	$a = null;$	$\{\ G_m[a \leftarrow \{O_{null}\}]$
[copy]	$a = b;$	$\{\ G_m[a \leftarrow G_m(b)]$
[store]	$l : a.f = b;$	$\begin{cases} \forall O_a \in G_m(a) \\ \quad G_m[(O_a, f) \cup\!\leftarrow G_m(b)] \\ \texttt{updateDeref}(O_l, a) \end{cases}$
[load]	$l : a = b.f;$	$\begin{cases} G_m[a \leftarrow \cup_{\forall O_b \in G_m(b)} G_m(O_b, f)] \\ \texttt{updateDeref}(O_l, b) \\ \text{if } (\exists O_b \in G_m(b),\ s.t.\ G_m(O_b, f) = \emptyset) \text{ then} \\ \quad \forall \langle u, x\rangle \in g_m(O_b) \cap \texttt{DCVal} \\ \quad\quad g_m[O_{d@l} \cup\!\leftarrow \langle u, x.f\rangle] \\ \quad\quad G_m[(O_b, f) \cup\!\leftarrow \{O_{d@l}\}] \\ \quad\quad G_m[a \cup\!\leftarrow O_{d@l}] \end{cases}$
[throw]	$l : throw\ a;$	$\{\ \texttt{updateDeref}(O_l, a)$
[synch]	$l : synchronized(a)$	$\{\ \texttt{updateDeref}(O_l, a)$
[array-length]	$l : k = a.length;$	$\{\ \texttt{updateDeref}(O_l, a)$
[return]	$return\ a;$	$\{\ \texttt{RET}_m \leftarrow \texttt{RET}_m \cup G_m(a)$
[unavailable-callee]	$l : b = a_0.u(a_1, ..., a_n);$	$\begin{cases} \forall O_{a_i} \in G_m(a_i), \forall O_x \in G_m(O_{a_i}, f) \\ \quad \texttt{extendCVals}(O_x, \langle u, O_{p_i}.f\rangle) \\ g_m[O_{u@l} \leftarrow \{\langle u, O_u\rangle\}] \\ G_m[b \leftarrow \{O_{u@l}\}] \\ \texttt{updateDeref}(O_l, a_0) \end{cases}$
[unavailable-caller]	$m(B\ p_1, ..., B\ p_n)$	$\begin{cases} g_m[O_{p_i} \leftarrow \{\langle \overline{m}, O_{a_i}\rangle]\} \\ G_m[p_i \leftarrow \{O_{p_i}\}] \end{cases}$

Figure 3.3: Partial-analysis rules for PACE. Notation: (i) $\beta[O_x \leftarrow Y]$ means β is extended with $\beta(O_x)$ set to Y. (ii) $X \cup\!\leftarrow Y$ is an abbreviation for $X \leftarrow X \cup Y$.

• *Allocation,* $l : a = new\ B()$. We use the abstract object $O_l \in \texttt{ALC}_m$ to represent the object(s) allocated at line l; the conditional value associated with O_l is set to Θ_C, denoting that O_l is a concrete object. We then set the points-to set of a to $\{O_l\}$.

• *Null-assignment,* $a = null$. In case of an explicit assignment of *null* to a variable a, we set the points-to set of a to a set containing the special object O_{null} (for which $g_m(O_{null})$ is set to $\{\Theta_N\}$), implying that a points to null.

• *Copy,* $a = b$. Here we set the points-to set of a to that of b.

• *Store,* $l : a.f = b$. First, for each object O_a in $G_m(a)$, we add the objects in the points-to set of b to the points-to set of $O_a.f$. Next, to denote the dereference done at l, we call a macro $\texttt{updateDeref}(O_l, a)$ (see Figure 3.4), where $O_l \in \texttt{DRF}_m$ represents

```
1 Macro updateDeref(O_l, a)
2     g_m(O_l) ← ∅;
3     foreach O_a ∈ G_m(a) do
4         g_m(O_l) ← g_m(O_l) ∪ g_m(O_a);
```

Figure 3.4: The `updateDeref` macro. G_m is the current points-to graph and g_m is the current partial summary.

the object(s) being dereferenced at l. For each object O_a in the points-to set of a, the macro $\texttt{updateDeref}(O_l, a)$ adds all the conditional values in $g_m(O_a)$ to $g_m(O_l)$.

• *Load,* $l : a = b.f$. Here, for each object O_b pointed-to by b, we first add the objects in the points-to set of $O_b.f$ to the points-to set of a, and then handle the dereference done at l by calling the macro $\texttt{updateDeref}(O_l, b)$. In case for a certain $O_b \in G_m(b)$ the set $G_m(O_b.f)$ is empty (for example, when O_b is an object returned by an unavailable method), then each dependent conditional value $\langle u, x\rangle$ in $g_m(O_b)$ indicates that O_b depends on the element x of an unavailable method u. In such a case, $O_b.f$ might be modified in u. To denote this dependence, we add the abstract object $O_{d@l} \in \texttt{OUT}_m$ to $G_m(O_b, f)$, and add $\langle u, x.f\rangle$ to $g_m(O_{d@l})$. Finally, we add $O_{d@l}$ to the points-to set of a.

If a store or load is made to/from an array (that is, $a[i] = b$ or $a = b[i]$, respectively), we repeat the same steps as done for a normal store or load statement, except that instead of f, we use the special array field "[]". See Section 3.4 for a discussion on our choice of heap abstraction.

• *Other dereference statements*. For Java statements that involve an implicit null-dereference, such as $l : throw\ a$, $l : k = a.length$ and $l : synchronized(a)\ \{...\}$, we use the object $O_l \in \texttt{DRF}_m$ to denote the dereference, and we update $g_m(O_l)$ by calling the macro $\texttt{updateDeref}(O_l, a)$.

• *Return,* $return\ a$. For each method m, we maintain a set $\texttt{RET}_m$ containing the objects that could be returned by m. At a return statement *return a* in method m, we add all the objects in $G_m(a)$ to the set $\texttt{RET}_m$.

• *Method call,* $l : b = a_0.u(a_1, ..., a_n)$. The processing of a method-call statement depends on whether the callee (method u) is available for analysis or not. For example, when analyzing a Java application, the methods from the JDK library are considered

```
1 Macro extendCVals(O_x, ⟨u, e⟩)
2     g_m(O_x) ← g_m(O_x) ∪ {⟨u, e⟩};
3     foreach edge (O_x, f, O_y) in G_m do
4         if ¬O_y.visited then
5             O_y.visited ← true;
6             extendCVals(O_y, ⟨u, e.f⟩);
```

Figure 3.5: The macro `extendCVals`. G_m is the current points-to graph and g_m is the current partial summary.

unavailable for analysis. In case of multiple possible callees (due to virtual dispatch), we take a union of the conditional values generated due to each callee.

(i) Available-callee (standard, rule not shown). In this case, we first merge the points-to graph G_u at the exit of the called method u into the points-to graph G_m for the caller, using the standard mapping algorithm presented by Whaley and Rinard (1999). For each object O_k added from G_u to G_m while merging, we add the conditional values in $g_u(O_k)$ to $g_m(O_k)$.

(ii) Unavailable-callee. In this case, for each object O_{a_i} pointed-to by the argument a_i, the object(s) reachable from any field of O_{a_i} might change in the method u. Say the object pointed-to by the formal parameter p_i at the entry of u is represented by O_{p_i}. For each field f of O_{a_i}, we denote the dependence of each object $O_x \in G_m(O_{a_i}, f)$ on u by adding a conditional value $\langle u, O_{p_i}.f\rangle$ to $g_m(O_x)$. Note that such conditional values need to be added to all the nodes in the subgraph reachable from O_x as well. We use a macro `extendCVals`$(O_x, \langle u, O_{p_i}.f\rangle)$ that extends the set of conditional values transitively for all the objects reachable from O_x; see Figure 3.5. Next, say the object O_u represents the objects returned by the method u. After the assignment, to store the dependence of the object pointed-to by b on O_u, we add the conditional value $\langle u, O_u\rangle$ to $g_m(O_{u@l})$, where $O_{u@l} \in \texttt{UCS}_m$. We also set the points-to set of b to a singleton containing $O_{u@l}$.

Irrespective of whether the callee is available or not, we handle the dereference performed by the receiver object a_0 by calling the macro `updateDeref`(O_l, a_0), where $O_l \in \texttt{DRF}_m$.

Note that at a call statement, we need not analyze the callee m if the context c at

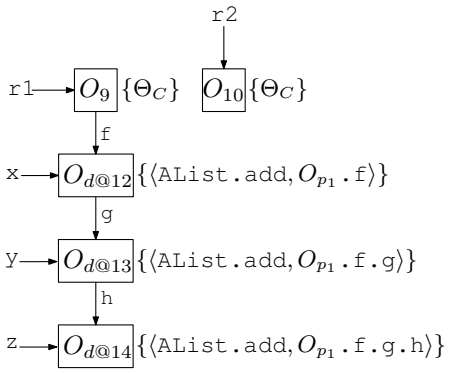
the call-site is the same as another context c' in which m has already been analyzed. We define a context by applying the idea of *level-summarized value contexts* (see Chapter 4), which is an extension of the idea of value-contexts (Khedker and Karkare, 2008); we consider the set $\{G_m, g_m\}$ as the context at the call-site. We terminate recursion in the standard way, that is, by iterating over the strongly-connected components of the call-graph till a fixed-point.

• *Method entry,* $m(B\ p_1, ..., B\ p_n)$. The initialization of G_m and g_m at the entry of a method m depends on whether the points-to graph and the partial summary at the call-site c are available or not. We discuss both the cases below.

(i) Available-caller (standard, rule not shown). To construct the points-to graph G_m at the entry of m, for each formal parameter p_i, we add each object O_{a_i} pointed-to by the corresponding actual argument a_i in the points-to graph G_c of the caller c, to $G_m(p_i)$. Next, we copy the subgraph reachable from O_{a_i} in G_c, to G_m. For each object O_x added to G_m in the previous steps, we set $g_m(O_x)$ to $g_c(O_x)$.

(ii) Unavailable-caller. In PACE, the callers of the entry method(s) are not available. For each such method m, we use $\overline{m}$ to denote all its callers at runtime. For each formal parameter p_i of m, we associate an abstract object $O_{p_i} \in \texttt{PAR}_m$, with $g_m(O_{p_i})$ containing the conditional value $\langle \overline{m}, O_{a_i} \rangle$, and add O_{p_i} to the points-to set of p_i. This conditional value indicates the dependence of O_{p_i} on the actual argument O_{a_i} passed to m by its callers.

Example. Figure 3.6 shows the points-to graph $G_{\texttt{bar}}$ and the partial summary $g_{\texttt{bar}}$ at the exit of the method `bar` shown in Figure 1.1. O_{11} and O_{12} are objects allocated at lines 11 and 12, respectively. The statement at line 14 is a load where the object dereferenced via the field `f` is unavailable. We use the abstract object $O_{d@14} \in \texttt{OUT}_{\texttt{bar}}$ to denote the dereferenced object and add an edge from `x` to $O_{d@14}$ to denote that `x` points to $O_{d@14}$. Similarly, the abstract objects $O_{d@15}$ and $O_{d@16}$ are added to $G_{\texttt{bar}}$ at lines 15 and 16, respectively. For each abstract object, Figure 3.6 also shows the corresponding conditional values in the map $g_{\texttt{bar}}$. The conditional values for O_{13} and O_{14} imply that the corresponding dereferences are guaranteed to be made on concrete objects. The conditional values for O_{15} and O_{16} imply that the objects dereferenced at lines 15 and 16 would be null (or concrete) if the objects pointed-to via $O_{p_1}.\texttt{f}$



$\texttt{ALC}_{\texttt{bar}} = \{O_{11}, O_{12}\}$
$\texttt{OUT}_{\texttt{bar}} = \{O_{d@14}, O_{d@15}, O_{d@16}\}$
$\texttt{DRF}_{\texttt{bar}} = \{O_{13}, O_{14}, O_{15}, O_{16}\}$

$g_{\texttt{bar}}(O_{13}) = \{\Theta_C\}$
$g_{\texttt{bar}}(O_{14}) = \{\Theta_C\}$
$g_{\texttt{bar}}(O_{15}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$
$g_{\texttt{bar}}(O_{16}) = \{\langle \texttt{AList.add}, O_{p_1}.\texttt{f.g}\rangle\}$

Figure 3.6: $G_{\texttt{bar}}$ and $g_{\texttt{bar}}$ at the exit of the method `bar` shown in Figure 1.1. For the nodes in $\texttt{ALC}_{\texttt{bar}}$, $\texttt{UCS}_{\texttt{bar}}$ and $\texttt{OUT}_{\texttt{bar}}$, the conditional values are shown next to the node. Redundant conditional values are not shown.

and O_{p_1}.f.g, where O_{p_1} is the object pointed-to by the first parameter of `AList.add`, are null (or concrete), respectively.

3.2.2 Efficient Storage of Partial Summaries for PACE

In addition to the pre-apply-meet optimization discussed in Section 3.1.5, we apply the following three additional optimizations in PACE.

(i) *Store only what is needed.* As discussed in Section 3.2.1, the domain of program elements for PACE consists of six sets of abstract objects. However, we do not need the analysis information about all the abstract objects for performing null-dereference-check elimination, and hence we store only a subset of these sets. For example, for an application method m, it is sufficient to store the conditional values only for the objects in the set $\texttt{DRF}_m$. Similarly, for each library method u, apart from the objects in the set $\texttt{DRF}_u$, we store the conditional values for the elements reachable from the objects in $\texttt{PAR}_u$ and $\texttt{RET}_u$, which may be needed for simplifying the partial summaries of the application methods.

(ii) *Do not store what can be interpreted.* For an element e of a method m, if the set of conditional values $g_m(e)$ is a singleton of the form $\{\Theta_C\}$, we avoid storing the information about e in the .res file. During JIT compilation, if the information about a dereference done at a statement labeled l is missing, we interpret that the dereference is guaranteed to be done on a concrete object. This greatly reduces the number of program

elements whose information needs to be printed in the .res files. We could have done the same for elements whose set of conditional values consisted only of Θ_N (in place of Θ_C); however, the space-saving we get would be much lower in practice, considering the small number of dereferenced objects that are guaranteed to be null.

(iii) *Implication.* Say we have a sequence of statements `x = p.f1; y = p.f2`. At runtime, `p` either points to *null* and an exception will be thrown at the first dereference (the second statement will not be executed), or `p` points to a concrete object. In either case, we can elide the null-dereference check at the second statement. We use this observation to further reduce the number of abstract objects created to denote the dereferences in PACE. For example, in the sequence of statements shown above, we do not generate any conditional values for the dereference in the statement `y = p.f2`. This further reduces the size of the stored .res files.

3.2.3 Results-adapter for PACE

During the JIT compilation of a method m, the summary-simplifier of the fast-precise-analyzer first simplifies the partial summary g_m and generates f_m. Then for each Bytecode instruction at offset l in m, if the instruction makes a dereference, the results-adapter for PACE checks the value of $f_m(O_l)$. If $f_m(O_l)$ is found to be C (for *Concrete*), the corresponding null-dereference check is elided; else the existing JVM mechanism is used to proceed with the insertion of the null-dereference check. In Section 3.5, we show that for a multitude of benchmarks, PACE is able to elide a substantial number of null-dereference checks during JIT compilation, without impacting the JIT compilation time negatively.

3.3 Escape Analysis for Synchronization Elimination

We now give an overview of the second analysis that we have implemented in PYE: a top-down context-, flow-, and field-sensitive thread-escape analysis for Java programs (a variation of the approach of Whaley and Rinard (1999); its brief summary can be found in Section 2.3.1). We use the results of this analysis to elide the acquire/release

[allocation]	$l : a = new\ B();$	$\begin{cases} \text{if } (isMultiThreaded(B)) \text{ then} \\ \quad g_m[O_l \leftarrow \{\Theta_E\}] \\ \text{else} \\ \quad g_m[O_l \leftarrow \{\Theta_D\}] \end{cases}$
[store]	$a.f = b;$	$\begin{cases} \text{if } (isStatic(f)) \text{ then} \\ \quad \forall O_x \in G_m.reachables(b) \\ \quad\quad g_m[O_x \leftarrow \{\Theta_E\}] \\ \text{else} \\ \quad \forall O_x \in G_m.reachables(b) \\ \quad\quad g_m[O_x \cup\!\leftarrow g_m(O_a)] \end{cases}$
[load]	$l : a = b.f;$	$\begin{cases} \text{if } (\exists O_b \in G_m(b),\ s.t.\ G_m(O_b, f) = \emptyset) \text{ then} \\ \quad \forall \langle u, x\rangle \in g_m(O_b) \cap \texttt{DCVal} \\ \quad\quad g_m[O_{d@l} \cup\!\leftarrow \{\langle u, x\rangle, \langle u, x.f\rangle\}] \end{cases}$
[synchronization]	$l : synchronized(a)$	$\{\ g_m[O_l \leftarrow \cup_{\forall O_a \in G_m(a)} g_m(O_a)]$
[unavailable-callee]	$l : b = a_0.u(a_1, ..., a_n);$	$\begin{cases} \forall O_{a_i} \in G_m(a_i) \\ \quad \texttt{extendCVals}(O_{a_i}, \langle u, O_{p_i}\rangle) \\ g_m[O_{u@l} \leftarrow \{\langle u, O_u\rangle\}] \end{cases}$

Figure 3.7: Partial-analysis rules for EASE. The updates to the points-to graph G_m, and the rules [unavailable-caller] and [return], are similar to those for PACE (see Figure 3.3), and hence skipped. Notation: $X \cup\!\leftarrow Y$ is an abbreviation for $X \leftarrow X \cup Y$.

synchronization operations associated with objects that do not escape their allocating thread. We call this instantiation the Escape Analysis for Synchronization Elimination (or EASE, in short).

As discussed in Section 3.1.6, in order to perform an analysis using PYE, we need to specify the domain $\mathcal{D}$, the lattice formed by `Val`, the processing of each relevant statement by the partial-analyzer, and the results-adapter. We now describe the same for EASE.

3.3.1 Partial-analyzer for EASE

Figure 3.1 shows the domain $\mathcal{D}_{ease}$ and the lattice $\texttt{Val}_{ease}$ for EASE. For each method m, $\mathcal{D}_{ease}$ consists of six sets of abstract objects: (i) $\texttt{ALC}_m$: one object O_l per allocation statement labeled l; (ii) $\texttt{PAR}_m$: one object O_{p_i} representing the objects pointed-to by the parameter p_i; (iii) $\texttt{RET}_m$: all the objects returned by m; (iv) $\texttt{OUT}_m$: one object $O_{d@l}$ for an unavailable dereference at a statement labeled l; (v) $\texttt{UCS}_m$: one object $O_{u@l}$ per unavailable method u at a call-statement labeled l; and (vi) $\texttt{SYN}_m$: one object O_l per

synchronization statement labeled l. The set $\texttt{Val}_{ease}$ forms a lattice with two elements: *DoesNotEscape* (D or $\top$) and *Escapes* (E or $\bot$). The corresponding conditional values are Θ_D and Θ_E, respectively. The definition of the meet ($\sqcap$) operation is standard. At runtime, for each method m, the goal of EASE is to compute the *escape-status* (D or E) of each object in $\texttt{SYN}_m$.

Figure 3.7 shows how the proposed partial-analyzer of EASE processes each statement that may affect the partial summary g_m for the method m being analyzed. As the rules for maintaining the points-to graph G_m and the processing of method entry and return statements are similar to those for PACE (see Section 3.2.1), we skip showing/discussing the same. Similar to PACE, we assume that each statement has a unique label associated with it.

• *Allocation,* $l : a = new\ B()$. At an allocation statement $a = new\ B()$ at line l, we use the abstract object $O_l \in \texttt{ALC}_m$ to represent the object(s) allocated at l. The conditional value associated with O_l is either Θ_E or Θ_D, depending on whether B is a *multi-threaded* class or not. We term a class as multi-threaded, if it is a subclass (immediate or otherwise) of `java.lang.Thread` or implements `java.lang.Runnable`.

• *Store,* $a.f = b$. At a statement of the form $a.f$ = b, if f is a static field, for each object O_x reachable from b, we set $g_m(O_x)$ to the singleton Θ_E, indicating that O_x now escapes; else we indicate the dependence of the escape-status of O_x on that of O_a by adding the conditional values in $g_m(O_a)$ to $g_m(O_x)$.

• *Load,* $l : a = b.f$. Similar to PACE, for an object $O_b \in G_m(b)$, if $g_m(O_b)$ consists of a dependent conditional value $\langle u, x\rangle$, it implies that the escape-status of O_b depends on the program element x in method u. In such a case, the object(s) pointed-to by $O_b.f$ might change/escape in u. To denote this dependence, we add the abstract object $O_{d@l} \in \texttt{OUT}_m$ to $G_m(O_b, f)$, and add the conditional values $\langle u, x\rangle$ and $\langle u, x.f\rangle$ to $g_m(O_{d@l})$. These conditional values indicate that $O_{d@l}$ escapes if either of the program elements x or $x.f$, in the method u, escape.

• *Synchronization,* $l : synchronized(a)$. Here we use the abstract object $O_l \in \texttt{SYN}_m$ to represent the synchronization statement labeled l. For each object O_a in the points-to set of a, we add the conditional values in $g_m(O_a)$ to $g_m(O_l)$. This indicates

$g_{\texttt{foo}}(O_5)$ = {$\langle$`AList.add`, $O_{p_1}\rangle$}
$g_{\texttt{foo}}(O_6)$ = {Θ_E}
$g_{\texttt{foo}}(O_9)$ = {$\langle$`AList.add`, $O_{p_1}\rangle$, $\langle$`AList.add`, O_{p_1}.`f`$\rangle$}
$g_{\texttt{foo}}(O_{10})$ = {$\langle$`AList.add`, $O_{p_0}\rangle$}
$g_{\texttt{foo}}(O_7)$ = {Θ_D}
$g_{\texttt{foo}}(O_{d@12}) = g_{\texttt{foo}}(O_{13})$ = {$\langle$`AList.add`, $O_{p_1}\rangle$, $\langle$`AList.add`, O_{p_1}.`f`$\rangle$}

$\texttt{ALC}_{\texttt{foo}} = \{O_5, O_6, O_9, O_{10}\}$
$\texttt{SYN}_{\texttt{foo}} = \{O_7, O_{13}\}$
$\texttt{OUT}_{\texttt{foo}} = \{O_{d@12}\}$

Figure 3.8: The conditional values in $g_{\texttt{foo}}$ for the method `foo` in Figure 2.1a. Redundant conditional values are not shown.

that the synchronization operation at l can be elided if none of the objects in the points-to set of a at l escape.

• *Unavailable-callee,* $l : b = a_0.u(a_1, ..., a_n)$. The handling of a method call where the callee u is unavailable is similar to that for PACE except for a minor difference. Here, for each argument a_i, even the top-level object $O_{a_i} \in G_m(a_i)$ may escape in u (if it is assigned to a static field in u, for example). We indicate the dependence by adding the conditional value $\langle u, O_{p_i}\rangle$ to $g_m(O_{a_i})$, indicating that O_{a_i} might escape if the object O_{p_i}, representing the object pointed-to by the formal parameter p_i at the entry of u, escapes. Similarly, all the objects reachable from O_{a_i} in G_m also depend on u. We call the macro `extendCVals`$(O_{a_i}, \langle u, O_{p_i}\rangle)$ to add the corresponding conditional values (see Figure 3.5).

Example. Figure 3.8 shows the conditional values generated by the partial-analyzer of EASE as seen at the exit of the method `foo` shown in Figure 2.1a. As the object O_5 pointed-to by the variable `r1` is passed to the unavailable method `AList.add`, the conditional value in $g_{\texttt{foo}}(O_5)$ indicates that the escape-status of O_5 depends on the escape-status of the object O_{p_1} pointed-to by the first formal parameter at the entry of `AList.add`. Similar is the case for the object O_{10}. As the object O_6 becomes reachable from the static variable `global` at line 8, $g_{\texttt{foo}}(O_6)$ consists of the conditional value Θ_E, implying that O_6 escapes. However, as we separately keep track of abstract objects in the set $\texttt{SYN}_{\texttt{foo}}$, we are able to capture the fact that O_6 does not escape at line 7, and hence $g_{\texttt{foo}}(O_7)$ consists of the conditional value Θ_D. This enables the elision of the synchronization at line 7 (at runtime). The conditional values for the synchronization statement at line 13 indicate that the corresponding synchronization

operation can be elided if the object O_{p_1} pointed-to by the formal parameter p_1 and the object pointed-to by $O_{p_1}.\texttt{f}$ do not escape in `AList.add`.

Synchronized methods. In Java, apart from synchronized statements, methods can also be declared as *synchronized* to indicate that any code in those methods cannot be executed by concurrent threads. If an instance method is declared as synchronized, it is equivalent to a synchronized statement on the receiver object, enclosing the body of the method. If the synchronized method is static, the synchronization operation is performed on the global object associated with its declaring class. Similar to the approach used by Lee and Midkiff (2006), we elide the synchronization operation associated with such a method, if the method is never called in a sequence of calls originating from a `run` method of a multithreaded class. For both these cases, we store the conditional values in a special abstract object associated with the corresponding method.

3.3.2 Efficient Storage of Partial Summaries for EASE

In addition to the pre-apply-meet optimization discussed in Section 3.1.5, we apply the following two additional optimizations in EASE.

(i) *Store only what is needed.* For an application method m, we store the conditional values only for the objects in the set $\texttt{SYN}_m$. For each library method u, apart from the objects in the set $\texttt{SYN}_u$, we also store the conditional values for the elements reachable from the objects in $\texttt{PAR}_u$ and $\texttt{RET}_u$.

(ii) *Do not store what can be interpreted.* For an element e of a method m, if the set $g_m(e)$ of conditional values is a singleton of the form $\{\Theta_D\}$, we avoid storing the information about e in the .res file. During JIT compilation, if the information about an abstract object ($\in \texttt{SYN}_m$) is missing, we interpret that the associated synchronization operation can be safely elided.

3.3.3 Results-adapter for EASE

To perform synchronization elimination in the HotSpot JVM, the optimizer needs to know whether the object associated with a synchronization statement does-not-escape

and consequently if the synchronization operation can be eliminated. Our results-adapter reads the escape-status of the abstract object corresponding to each of the synchronization statement and sets a field `isEliminatable` in the synchronization statement accordingly.

Example. The conditional values generated by the partial-analyzer for the method `AList.add` (see Figure 1.1) include the dependencies listed below; recall that for each method m with unavailable-caller, we use $\overline{m}$ to denote its callers at runtime.

$$g_{\texttt{AList.add}}(O_{p_0}) = \{\langle \overline{\texttt{AList.add}}, O_{a_0}\rangle\}$$
$$g_{\texttt{AList.add}}(O_{p_1}) = \{\langle \overline{\texttt{AList.add}}, O_{a_0}\rangle, \langle \overline{\texttt{AList.add}}, O_{a_1}\rangle\}$$

The conditional values $\langle \texttt{AList.add}, O_{a_0}\rangle$ and $\langle \texttt{AList.add}, O_{a_1}\rangle$ are added to the map $g_{\texttt{AList.add}}$ for O_{p_0} and O_{p_1} respectively, at the entry of the method `AList.add`. When the object O_{p_1} is stored into the array pointed-to by $O_{p_0}.\texttt{f}$ (at line 9), the conditional values in $g_{\texttt{AList.add}}(O_{p_0})$ are added to $g_{\texttt{AList.add}}(O_{p_1})$.

During JIT compilation of the application method `foo`, the summary-simplifier evaluates the conditional values (see Figure 3.8) for the synchronization objects of lines 7 and 13 (O_7 and O_{13}, respectively). For O_7, the conditional value Θ_D simply evaluates to the value simple conditional value $D \in \texttt{Val}_{ease}$. To simplify the conditional values for O_{13}, our algorithm starts with a list L of conditional values $\{\langle \texttt{AList.add}, O_{p_1}\rangle, \langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle\}$ (given by $g_{\texttt{foo}}(O_{13})$ shown in Figure 3.8). Simplifying these conditional values adds the elements of the set $g_{\texttt{AList.add}}(O_{p_1})$ to L. As O_{10} and O_5 are the actual arguments O_{a_0} and O_{a_1} respectively, our algorithm adds the elements of $g_{\texttt{foo}}(O_{10})$ and $g_{\texttt{foo}}(O_5)$ to L to obtain: $L = \{\langle \texttt{AList.add}, O_{p_1}\rangle$, $\langle \texttt{AList.add}, O_{p_1}.\texttt{f}\rangle$, $\langle \overline{\texttt{AList.add}}, O_{a_0}\rangle$, $\langle \overline{\texttt{AList.add}}, O_{a_1}\rangle$, $\langle \texttt{AList.add}, O_{p_0}\rangle\}$.

At this stage, no further simplification is possible and we reach a fixed-point. As mentioned in Section 3.1.4, after attaining the fixed-point, for each conditional value $\mathcal{T}_i$ in the worklist, we set $[\![\mathcal{T}_i]\!]$ to D (the top of the lattice). Thus, the summary-simplifier generates the escape-status: $f_{\texttt{foo}}(O_7) = f_{\texttt{foo}}(O_{13}) = D$. Thereby the results-adapter sets the `isEliminatable` field for the synchronization statements at lines 7 and 13 to `true`. This field is used by the following pass of synchronization elimination to perform the necessary optimization.

Note that a fully static (flow-sensitive) analysis would be able to elide the synchronization operation only at line 7. On the other hand, the C2 compiler of the HotSpot JVM performs a connection-graph (Choi *et al.*, 1999) based partially-interprocedural escape analysis during JIT compilation, which can elide the synchronization operation at line 13, but not at line 7. Using PYE, and by maintaining abstract objects separately in $\texttt{SYN}_{\texttt{foo}}$, EASE is able to elide the synchronization operations both at lines 7 and 13. We show in Section 3.5 that the overheads involved for achieving this precision during JIT compilation are quite small; in fact, less than the existing analysis performed by C2.

3.4 PYE: A Discussion

In this section, we first discuss some subtle aspects in the current design of PYE, followed by its correctness argument.

1. Deoptimization. PYE handles deoptimization scenarios (for example, because of dynamic classloading, failed speculative type-checks, and so on) in a natural manner: the set of new and old methods to be re-compiled are obtained by analyzing the call graph and optimized using their partial summaries (new or existing). This set can be made further precise using a scheme similar to that of Cooper *et al.* (1986).

2. Callbacks. PYE analyzes each library installation independent of the application. Consequently, if a library method m has a callback to a method in the application program, the called method cannot be analyzed in this context. In such a scenario, we compile m (and the dependent methods) with the existing analyses built in the JVM, and not with the analysis results generated by PYE. Since callbacks are used infrequently in practice (during our evaluation over real-world benchmarks, we have not found any case where we lose precision because of this design choice), we believe it to be an acceptable limitation of PYE. There could be multiple ways to handle callbacks more precisely. For example, we could perform a fast (perhaps imprecise) analysis to find out whether the called-back method may indeed affect the results for the given analysis and fall-back only if it does. Another way to handle callbacks is to statically create *gaps* (Arzt and Bodden, 2016) at call-sites that may be involved in a callback, and fill these gaps with more precise information during JIT compilation. We leave the

```
1  class B { ... // from Figure 1.1
2    void baz() {
3      B r1 = new B();
4      List l = new AList();
5      l.add(r1);
6      l.add(null);
7      X x = l.first();
8      Y y = x.g;
9    }
10 }
```

```
1  class AList<E> { ... // from Figure 1.1
2    E first() {
3      return arr[0];
4    }
5  }
```

(a)

```
1  class C {
2    void foo() {
3      X r1 = new X();
4      bar(null);
5      X r2 = bar(r1);
6      ... = r2.g;
7    }
8    void bar(X rx) {
9      Y y1 = new Y();
10     y1.f = rx;
11     return y1.f;
12   }
13 }
```

(b)

Figure 3.9: Example to illustrate some limitations of the heap abstraction used for PACE and EASE.

integration of the techniques of Arzt and Bodden into PYE as future work.

3. Verification. The summaries (generated by the partial-analyzer) transferred along with the class files to the target machine may get corrupted (intentionally, or otherwise). Consequently, the fast-precise-analyzer may derive wrong analysis-results. Currently, we resolve this issue by using public-key cryptography (Stinson, 1995). However, keeping in mind its limitations (for example, trusting the partial-analyzer), we are working on a fast verifier (similar to Java Bytecode type-checking) to validate the results in the JVM itself.

4. Heap abstraction. We abstract all the elements of an array in a field-insensitive manner, which leads to well-understood imprecision. Consider the methods `baz` of class `B` and `first` of class `AList`, as shown in Figure 3.9, in the context of PACE. For the dereference at line 8, the set of conditional values generated by the partial-analyzer would be: $\{\langle \texttt{AList.add}, O_{\texttt{AList.add}} \rangle\}$. However, as the array `arr` does not distinguish among its various elements (it is standard to treat arrays field-insensitively for scalability), the method `first` would conservatively generate the value $\{\Theta_N\}$ for its return-value, and hence the dereference during JIT compilation would not be elided. Similarly, the abstraction of objects by their allocation site (that is, no heap-cloning (Nystrom *et al.*, 2004)) leads to understandable imprecision. For example, in

Figure 3.9b, the partial-analyzer finds that at line 6, `r2` may point to `null` and hence cannot elide the null-check. We believe it would be an interesting future work to extend PYE to support more precise forms of heap abstractions.

5. Conditional values. The conditions used in the conditional values depend on the specific analysis being performed. For example, for EASE the conditions are based on the escape-information, leading to conditional values such as x escapes if y escapes. In contrast, for implementing the taint analysis by Arzt and Bodden (2016), each 'source' of the taint may be treated as a tainted object, and the conditions may be based on points-to/alias relationships, leading to conditional values such as x points to a tainted object z', if y points to z. These points-to conditions can be on the argument-objects passed, and the return values of the unavailable methods. Importantly, note that such variations (naturally, analysis-dependent) still fit into the general two-pronged approach presented as part of PYE.

6. Levels of granularity. Though we discuss PYE at method-level granularity, these ideas can also be extended to other levels of granularity (for example, storing/simplifying summaries per class, package, and so on) without impacting the precision. The choice of the appropriate granularity levels can lead to interesting trade-offs: while storing/reading/simplifying summaries at higher levels of granularity can amortize (and speedup) the overall disk reads, it could also lead to increased overheads from reading/simplifying summaries of even those methods that may not be compiled.

7. Flow-sensitivity. It is worthy to highlight the novel way in which we maintain flow-sensitive results in an efficient manner. As an instance, traditional connection-graph based escape analyses (Choi *et al.*, 1999) first construct a connection graph, then perform a reachability test to determine the objects that escape, and then, based on the escape-status of the corresponding objects, a second pass then elides unnecessary synchronization operations. As discussed in Section 3.3, such an approach might not be able to elide the synchronization operations for which the associated object(s) escape after the synchronization statement. On the other hand, in this thesis, we use a special abstract object ($\in \texttt{ALC}_m$) with each synchronization statement, which allows us to elide more synchronization operations (by taking advantage of flow-sensitivity) without the need to later store the results of all the objects in the .res files.

3.4.1 PYE: Correctness

It may be noted that the precision of an analysis implemented in PYE depends upon the algorithm used in the partial-analyzer to generate the partial summaries. Thus, PYE can be thought to be parametric on the analysis algorithm. If we represent the analysis being performed as Ψ, then the PYE variation of Ψ can be denoted as PYE(Ψ). We now state the correctness theorem for our proposed approach for programs that may call library methods which in turn invoke no callbacks. Later, we extend the argument to library methods that may invoke callbacks.

Definition 2. *The set $\{\langle y_1, v_1^1, v_2^1\rangle, \langle y_2, v_1^2, v_2^2\rangle, \cdots\}$ of conditional values generated by PYE(Ψ) for a variable x, for any program P at a program point L, is considered to be "correct", if during the whole-program analysis, the following conditions hold: (i) $\exists$ a set S of integers, such that $\forall i \in S$, Ψ computes the value of y_i to be v_1^i at L; and (ii) Ψ computes the dataflow value of x to be $\sqcap_{\forall i \in S} v_2^i$. That is, a correct set of conditional values includes all the dependencies and nothing more.*

Theorem 3.4.1. *Given a whole-program analysis algorithm Ψ, for any program P, the analysis results obtained using PYE(Ψ) for the program elements of P will match those obtained using Ψ. That is, if $\mathcal{D}$ is the set of program elements of P, then after simplification of partial summaries, $\forall x \in \mathcal{D}$, PYE$(\Psi)(x) = \Psi(x)$.*

Proof. (Proof Sketch) We prove the theorem by contradiction. Say there exist a whole-program analysis Ψ and a program element x in method m, such that at some program point L, PYE$(\Psi)(x) = s_1$ and $\Psi(x) = s_2$, and $s_1 \neq s_2$.

This implies that while Ψ has found the dataflow value of x at L (the most precise solution) to be s_2, the summary-simplifier has found the dataflow value of x at L to be s_1. There can be two cases under which the summary-simplifier can find the dataflow value of x at L to be s_1 ($\neq s_2$):

(i) In the set of summaries provided by the partial-results-accumulator, at program point L, $g_m(x)$ was a singleton containing s_1 (a simple conditional value). This implies that the partial-results-accumulator has incorrectly obtained the dataflow value as s_1, from the partial-analyzer. But as the partial-analyzer implements Ψ for performing the

static analysis and gives a singleton with a simple conditional value only for objects that do not depend on any library calls, Ψ would also find the dataflow value of x as s_1 (and hence $s_1 = s_2$). A contradiction.

(ii) In the set of summaries provided by the partial-results-accumulator, at program point L, $g_m(x)$ consists of a set of dependent conditional values, and $g_m(x)$ was simplified as s_1 by the summary-simplifier. There are two sub-cases:

(a) the partial-analyzer generated a correct set of conditional values for x, but the summary-simplifier generated an imprecise solution. As discussed in Section 3.1.4, the summary-simplifier iteratively solves the dependent conditional values until no more of them can be simplified further, which generates the most precise solution for the *given* set of conditional values. A contradiction.

(b) the partial-analyzer generated an incorrect set of conditional values for x. Note that the partial-analyzer is an implementation of Ψ with the only difference being in the output for the ones related to the conditional values, such that $\forall x \in \mathcal{D}$, at each program point L, if PYE(Ψ)(x) is a not a simple conditional value, then the partial-analyzer adds all the required dependent conditional values that denote a dependence of $\Psi(x)$ on the unavailable parts of P. Further, we can see that the partial-analyzer adds only the required dependent conditional values – as it adds a dependent value only as a result of some unavailable code (at method calls, method entries, and at load statements). That is, PYE marks all the required dependencies and nothing more. That is, as per Definition 2, the summary generated by the partial-analyzer (PYE($\Psi(x)$) is correct. A contradiction. □

Theorem 3.4.1 guarantees that during JIT compilation, a whole program analysis Ψ can be equivalently replaced by PYE(Ψ), when the called library methods do not in turn invoke callbacks. Even in cases where the library methods may invoke callbacks, extending PYE with techniques proposed by Arzt and Bodden (2016) can lead to similar precision. However, when using our above discussed conservative scheme to handle callbacks, the Theorem 3.4.1 statement can be modified to state that after the simplification of partial summaries, $\forall x \in \mathcal{D}$, PYE($\Psi$)($x$) $\sqsubseteq \Psi(x)$, where the relation $x \sqsubseteq y \Rightarrow x \sqcap y = x$.

Corollary 3.4.2. In a JIT compiler, a semantics-preserving optimization based on a whole program analysis Ψ can use PYE(Ψ) instead and still remain semantics-preserving.

Proof. Proof follows directly from Theorem 3.4.1 and the above discussion thereof. □

3.5 Implementation and Evaluation of PYE

We implemented the PYE framework (see Figure 3.1b) in two parts: (i) interfacing of the partial-analyzer in the Soot optimization framework (Vallée-Rai *et al.*, 1999) version 2.5.0 – approximately 2000 lines of code; (ii) different components of the fast-precise-analyzer in the HotSpot Server Compiler (C2) of the OpenJDK HotSpot JVM (Paleczny *et al.*, 2001) version 7 – approximately 1000 lines of code. To understand the usability of PYE, we used PYE to instantiate PACE (Section 3.2) and EASE (Section 3.3). The associated respective partial-analyzers consist of about 4100 and 4000 lines of Java code (in Soot), and the respective results-adapters consist of about 550 and 250 lines of C++ code (in C2). In addition, we use the extremely helpful tool TamiFlex (Bodden *et al.*, 2011) version 2.0.3 to eliminate the reflection-based code from the original benchmarks, so that they can be analyzed by Soot.

3.5.1 Evaluation Setup for PYE

We evaluated PYE and its two instantiations on twenty-two benchmarks from four benchmark suites: (i) AES, COMPILER, COMPRESS, FFT, LU, MONTECARLO, RSA, SIGNVERIFY, SOR, SPARSE and SUNFLOW from SPECjvm (2008) – using the 'default' input; (ii) AVRORA, ECLIPSE, FOP, H2, LUINDEX, LUSEARCH, PMD and XALAN from the DaCapo suite (Blackburn *et al.*, 2006) version 9.12 – using the 'default' input; (iii) MOLDYN and RAYTRACER from Section C of the JGF suite (Daly *et al.*, 2001) – using 'SizeB' inputs; and (iv) SPECjbb (2005) – using the default 30 seconds ramp-up time and 240 seconds measurement window. In the case of BATIK (from DaCapo), we could not run the original program on our system for the 'default' input (threw *TruncatedFileException*). The rest of the benchmarks excluded from the original DaCapo and SPECjvm suites could not be analyzed – either by Soot or by TamiFlex. Our exper-

S.No.	Bench -mark	#class files	.class size (MB)	.res size (MB)		Overhead (%)		DCVal (%)	
				PACE	EASE	PACE	EASE	PACE	EASE
1.	aes	297	2.1	0.24	0.16	11.33	7.38	42.97	10.0
2.	compiler	306	2.1	0.24	0.15	11.29	7.33	42.96	10.0
3.	compress	308	2.1	0.24	0.16	11.19	7.48	42.59	10.0
4.	fft	301	2.1	0.24	0.16	11.29	7.38	43.72	9.09
5.	lu	300	2.1	0.24	0.16	11.33	7.38	43.38	9.09
6.	montecarlo	300	2.1	0.23	0.16	11.14	7.38	43.50	9.09
7.	rsa	297	2.1	0.30	0.16	11.05	7.38	43.30	10.0
8.	signverify	297	2.1	0.24	0.16	11.29	7.38	43.39	10.0
9.	sor	301	2.1	0.24	0.16	11.19	7.38	42.51	9.09
10.	sparse	300	2.1	0.24	0.16	11.19	7.38	42.68	9.09
11.	sunflow	406	2.7	0.32	0.22	11.85	8.26	35.55	4.76
12.	avrora	527	2.6	0.04	0.02	1.54	0.85	62.55	0.0
13.	eclipse	1344	10	1.50	0.84	15.0	8.42	18.17	5.05
14.	fop	1027	5.8	0.36	0.20	6.21	3.38	35.51	11.11
15.	h2	324	2.2	0.04	0.02	1.91	1.05	62.78	0.0
16.	luindex	200	1.3	0.04	0.03	2.92	1.85	61.00	100.0
17.	lusearch	198	1.2	0.04	0.03	3.33	2.08	59.54	100.0
18.	pmd	688	3.7	0.04	0.03	1.19	0.68	56.16	0.0
19.	xalan	638	3.7	0.04	0.03	1.19	0.78	49.74	100.0
20.	moldyn	14	0.06	0.003	0.002	5.33	2.67	52.2	0.0
21.	raytracer	22	0.09	0.005	0.004	5.76	3.91	88.8	0.0
22.	specjbb	82	0.51	0.07	0.03	12.99	6.69	73.96	12.27
GeoMean		268	1.7	0.11	0.07	6.41	3.96	47.27	7.1

Table 3.2: Details of the benchmarks used for PACE and EASE, storage overhead for .res files, and the percentage of elements with dependent conditional values (DCVal) in the generated .res files.

iments have been performed on a 2.3 GHz AMD Abu Dhabi system (DVFS disabled) with 64 cores and 512 GB of memory, running CentOS 6.4.

Table 3.2 lists some static characteristics of the benchmarks included in the study. The sizes of the benchmarks listed in Table 3.2 varied from 60 KB (small programs) to 10 MB (large applications), and the number of class files varied from 14 to 1344. In the SPECjvm suite, it can be seen that the benchmarks from the same group have similar sizes; for example, AES, RSA and SIGNVERIFY from the CRYPTO group, and FFT, LU, MONTECARLO, SOR and SPARSE from the SCIMARK group. This similarity occurs because most of the code among the benchmarks in the same group is shared (for example, the harness and the utilities).

We now present an evaluation to study the impact of our proposed framework PYE. We divide the evaluation into two parts: (i) evaluation related to the static-compilation

time analysis (involves the partial-analyzer of PYE); and (ii) evaluation related to the JIT-compilation time analysis (involves the fast-precise-analyzer of PYE).

3.5.2 Evaluation of the Partial-analyzer

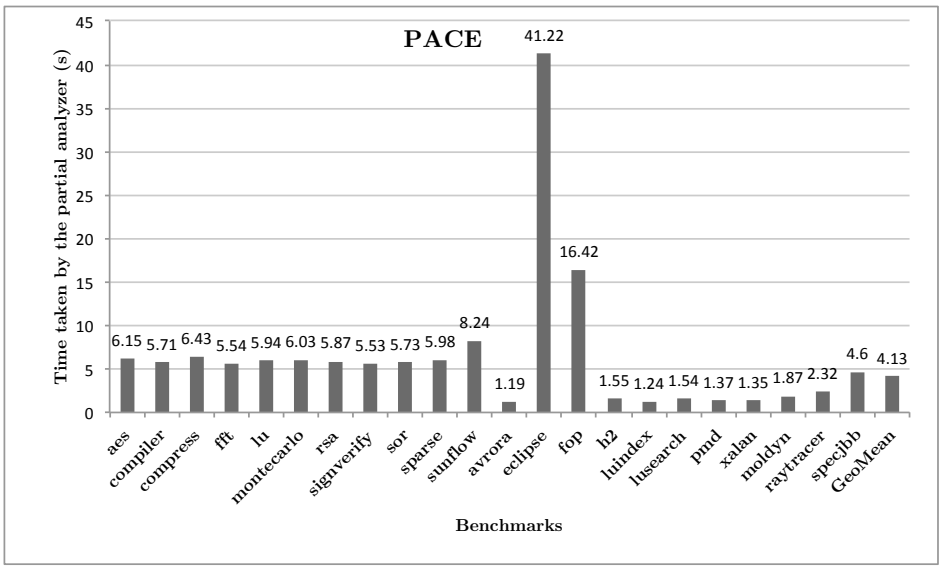
In this section, we evaluate the partial-analyzers for PACE and EASE by focusing on the time taken to compute the partial summaries, the storage overhead of the generated .res files, and the precision of the conditional values in the .res files.

Time taken by the partial-analyzer. Figure 3.10 shows the time taken by the partial-analyzer for all the benchmarks. On average, the partial-analyzer for PACE took 4.13 seconds across all the benchmarks, and that for EASE took 3.92 seconds across all the benchmarks.

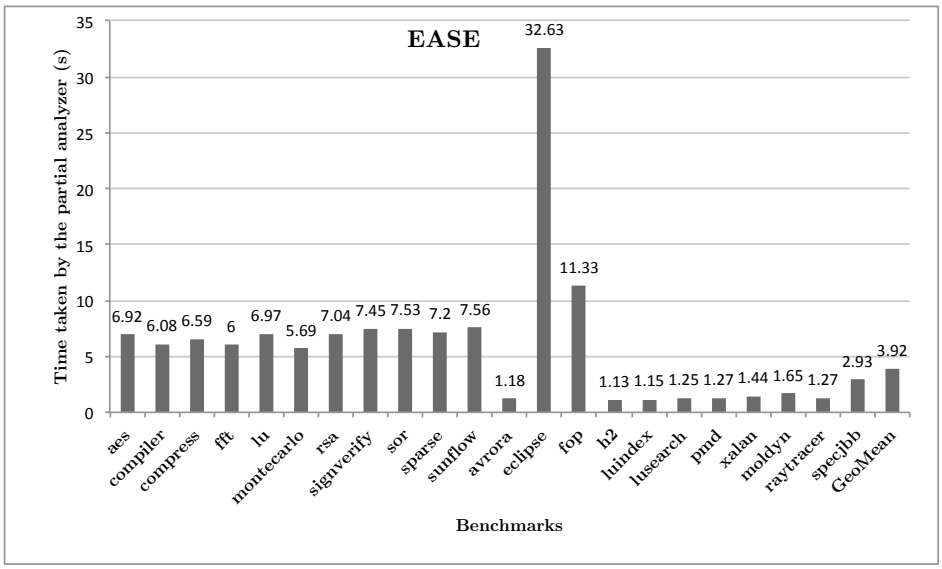
We observe that the time required is mainly dependent on the size of the individual benchmark: less time for smaller benchmarks (for example, JGF), and more time for larger DaCapo benchmarks (for example, ECLIPSE and FOP). Considering that the analyses performed by PACE and EASE are very precise (context, flow, and field-sensitive), we argue that the analysis time is quite reasonable. Further, this analysis time (once per static-compilation) does not get added to the time for final execution (may happen many times).

Storage overhead. The summaries generated by the partial-analyzer for each benchmark are stored in plain text as a file `<benchName>.<analysis>.res`. These summaries are piggybacked with the class-files of the benchmark and transferred to the JVM in which the benchmark is executed. Table 3.2 lists the sizes of the .res files (in MB) for each benchmark, for both PACE and EASE. It is evident that these files are very small (110 KB and 70 KB on average, respectively, for PACE and EASE). As compared to the sizes of the corresponding benchmarks, the average storage overheads for PACE and EASE are 6.41% and 3.96% respectively – arguably quite low.

Lee and Midkiff (2006) had proposed a two-phase escape analysis for the Jikes RVM. They compute connection graphs (a representation similar to points-to graphs) for different methods offline, and merge the connection graphs to complete an interprocedural analysis during JIT compilation. Compared to the overhead reported by Lee



(a)



(b)

Figure 3.10: Time taken (in seconds) by the partial analyzers of (a) PACE and (b) EASE.

and Midkiff for storing the connection graphs – 68% over the class files, the storage overhead for partial summaries in EASE is quite low – only 3.96% over the class files.

We also evaluated the effects of our proposed techniques for efficient storage of partial summaries and found that the improvements were significant. For example, for PACE, without the optimizations proposed in Section 3.2.2, we found an additional overhead of 35.6% for the .res files. We expect a similar trend for escape analysis, where, in general, the number of objects that do not escape is much higher than the ones that escape.

Precision of conditional values. For a program element e in method m, the conditional values in $g_m(e)$ may be either simple or dependent (see Section 3.1.3). If all the conditional values in $g_m(e)$ are simple, it implies that the analysis-result $f_m(e)$ for e does not depend on any other element. For the rest of the elements, the dependent conditional values need to be evaluated in the fast-precise-analyzer (at runtime). The last column of Table 3.2 shows the percentage of stored elements for which the partial summary consists of at least one dependent value. On average, the analysis-results for 47.27% and 7.1% of the elements for PACE and EASE, respectively, consist of dependent conditional values, and hence cannot be computed precisely using any static whole-program analysis that handles library calls conservatively (alternative A1 in Section 1.1). These numbers show that the potentially achievable precision for PACE and EASE over a static whole-program analysis is significant.

3.5.3 Evaluation of the Fast-precise-analyzer

In this section, we evaluate the fast-precise-analyzers for PACE and EASE by focusing on the precision of the generated analysis-results and the time taken during JIT compilation, compared to the existing state-of-the-art analyses in the C2 compiler of the HotSpot JVM version 7. The goal of this comparison is to demonstrate that PYE leads to the generation of more precise analysis-results, while spending time similar to (and in some cases lower than) the existing imprecise analyzers of C2. For each of the analyses, we evaluate the fast-precise-analyzer in the default setting of the HotSpot JVM (called the *mixed* mode) that uses an interpreter, the client C1 compiler, and the server C2 compiler. In this mode, the C2 compiler is invoked only for those methods/loops that are invoked/iterated more than a threshold number of times (usually 10000-15000).

Precision of generated results. For statements that dereference an object, the JVM needs to perform a null-dereference check – if the dereferenced object is null, then the check fails and a *NullPointerException* is thrown. The C2 compiler applies the implication optimization discussed in Section 3.2.2 to avoid inserting several checks; however, the rest of the checks remain explicit and need to be performed during program execution. Figure 3.11 compares the number of explicit null-dereference checks inserted by PACE, with that by the existing analyzer of C2. As evident, the number of checks in-

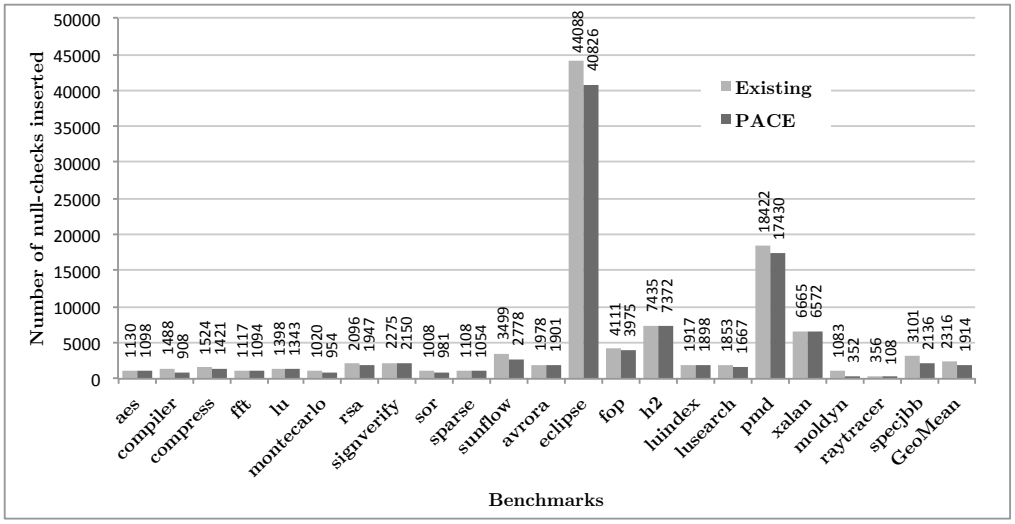
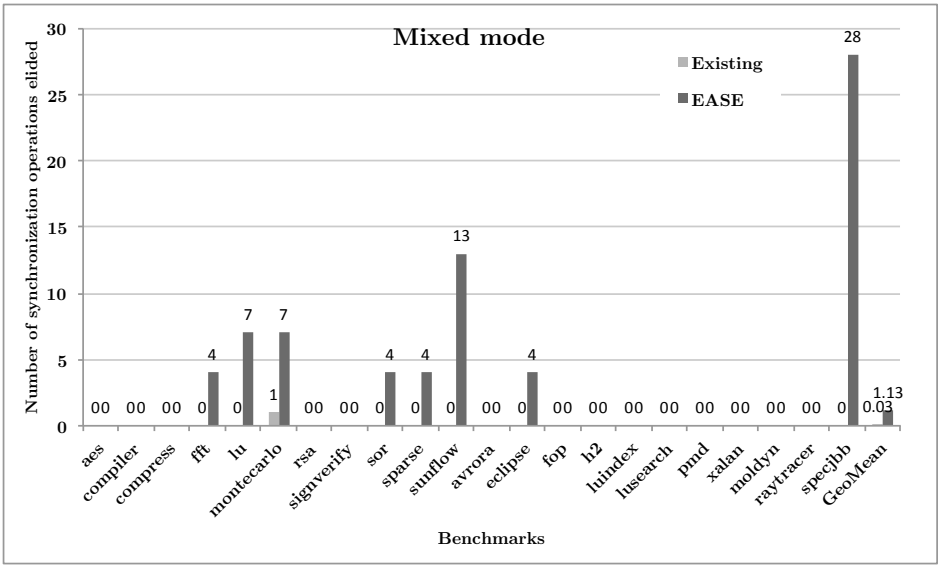


Figure 3.11: Number of explicit null-checks inserted by the existing analyzer of the C2 compiler and PACE; smaller the better.

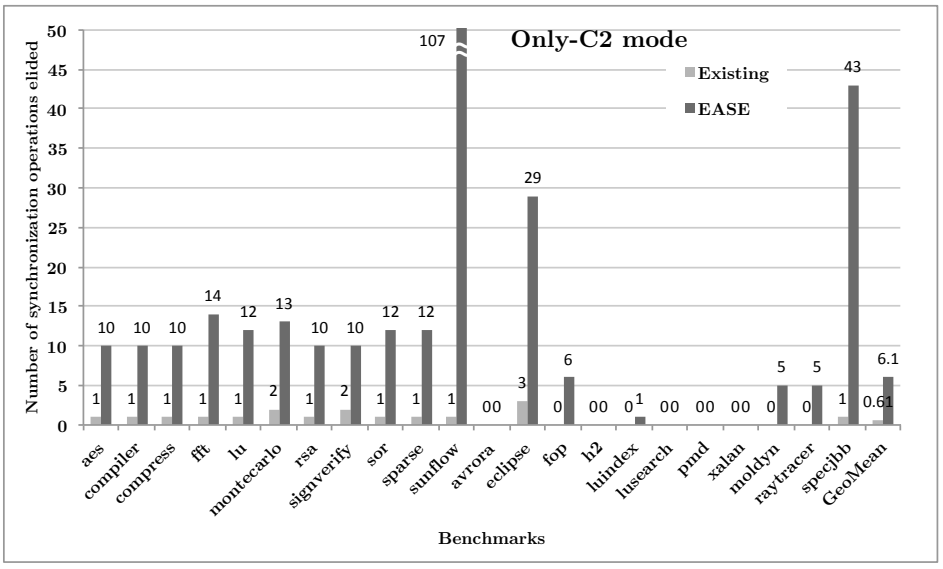
serted by PACE is significantly lower than that by the existing analyzer of C2: 17.36% on average. This clearly demonstrates the enhanced precision achieved by PACE. Later in this section, we show that the cost to achieve this precision is negligibly low.

In Java programs, the synchronization statement `synchronized(a){S}` is used to execute the statement S in a mutually exclusive manner. The JVM implements mutual exclusion by acquiring the lock associated with the object pointed-to by the variable `a`. Based on the work of Choi *et al.* (1999), the C2 compiler performs a partially interprocedural escape analysis: the connection-graphs are intraprocedural, but a Bytecode-level flow-insensitive analysis is performed at method call-sites. This analysis is used to identify and elide the synchronization operations for which the associated object is guaranteed to be accessed by a single thread. Figure 3.12a compares the number of synchronization operations elided by EASE with that by the existing escape analyzer of C2. We can see that while the existing analyzer of C2 does not elide any synchronization operation across most of the benchmarks (except MONTECARLO), the precise nature of EASE leads to the elision of a significant number of synchronization operations in many benchmarks (up to 28 elisions, in case of SPECJBB). Later in this section, we show that the time to obtain this precision is much less compared to the time spent in performing escape analysis in the existing C2 compiler.

Note that the low number of elisions is due to two reasons: (i) In most of the programs, the synchronization constructs are indeed necessary. (ii) The methods containing



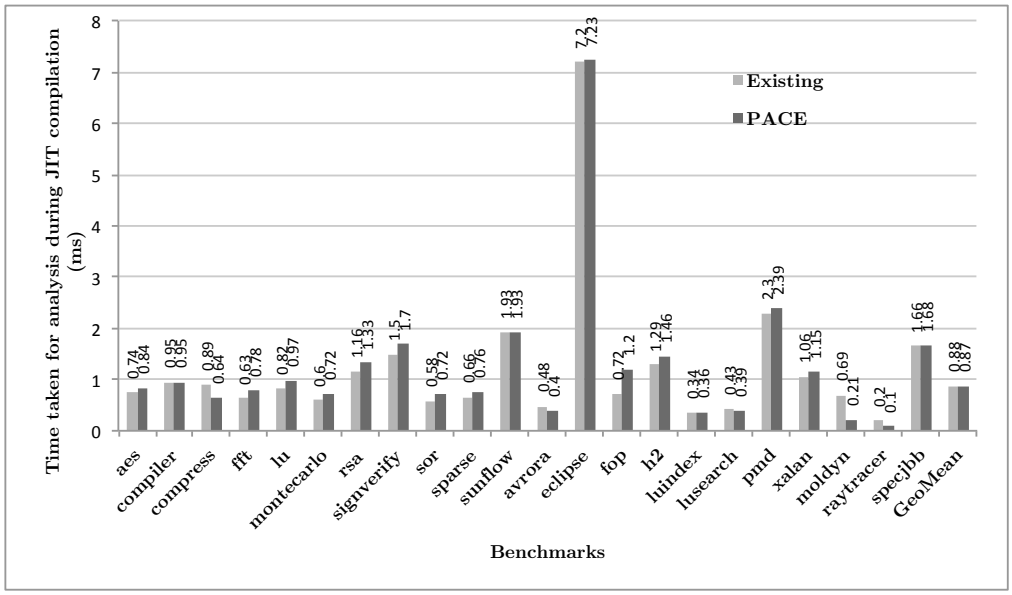
(a)



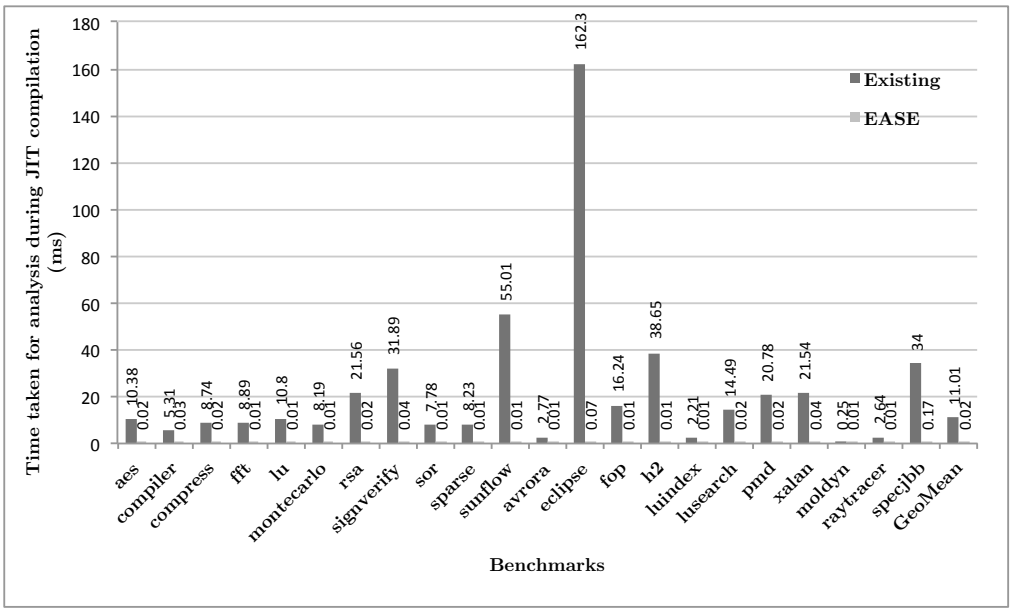
(b)

Figure 3.12: Number of synchronization operations elided by the existing analyzer of the C2 compiler and that by EASE: (a) in mixed mode, and (b) in only-C2 mode; larger the better.

the synchronization statements are not compiled by C2, because of the high threshold limit. As a side study, to get an estimate on the upper limit on the number of synchronization operations that could be elided for long-running programs, where more methods may get compiled by C2, we evaluated EASE and the existing analyzer of C2 in an *only-C2* mode. Here, we disabled the interpreter and the C1 compiler, and compiled every method using C2. Figure 3.12b compares the number of synchronization operations elided by EASE with that by the existing analyzer in the only-C2 mode. It



(a)



(b)

Figure 3.13: (a) Time taken (in milliseconds) to insert explicit null-dereference checks by the existing analyzer of C2 and by PACE during JIT compilation. (b) Time taken (in milliseconds) to perform synchronization elimination by the existing analyzer of C2 and by EASE during JIT compilation.

can be seen that compared to Figure 3.12a, the existing analyzer of C2 finds opportunities (albeit small in number) for synchronization elimination in more benchmarks. In contrast, EASE is able to find many more opportunities (up to 107). This shows that for long-running programs, EASE may lead to the elision of more synchronization operations than the existing analyzer of C2.

Time taken during JIT compilation. We now evaluate the time taken by the fast-precise-analyzers of PACE and EASE during JIT compilation. For reading the .res files, we spawn a separate thread for the partial-results-accumulator during the initialization of the JVM, and wait for the thread to terminate before using the results in the C2 compiler. The time to read the .res files varies with the size of the .res files, and on average, it is 13 and 8 milliseconds for PACE and EASE, respectively. However, we found that the spawned thread finished long before the results were needed for all the benchmarks in both the modes, for both the analyses. Thus, the effective time required by the partial-results-accumulator for both PACE and EASE is zero.

Figure 3.13a compares the time taken by PACE with that by the existing analyzer of C2 to insert explicit null-dereference checks during JIT compilation, in milliseconds. As evident, the execution times of both the analyzers are very small (less than a millisecond, on average) and comparable. The time spent by the existing analyzer is quite small because the corresponding analysis is only intraprocedural. The important point to observe is that the time spent by PACE to obtain significantly more precise results is also very small and does not cause any noticeable overhead.

Figure 3.13b compares the time taken by EASE with that by the existing analyzer of C2 to perform synchronization elimination during JIT compilation, in milliseconds. As evident, the escape-analysis time in EASE is clearly lower (on average, 99.91% less) compared to the existing analyzer. This is because EASE completely alleviates the need to construct connection graphs and propagate escape-analysis information therein, as is done by the existing analyzer of C2 during JIT compilation. Note that these savings also imply a drop in the overall JIT compilation time. For the benchmarks under consideration, we found the mean saving in the JIT compilation time due to EASE was about 1.9% (see Figure 3.14). Note that these improvements may also include the effects on the JIT compilation time, by any additional optimizations enabled by EASE (for example, in the IR simplification passes).

Impact of null-check elimination and synchronization elimination. Figure 3.15 shows the performance improvements obtained using PACE and EASE over the respective existing analyzers of C2, in the presence of all other optimizations of the HotSpot JVM. It is well known (Georges *et al.*, 2007) that the performance of Java programs

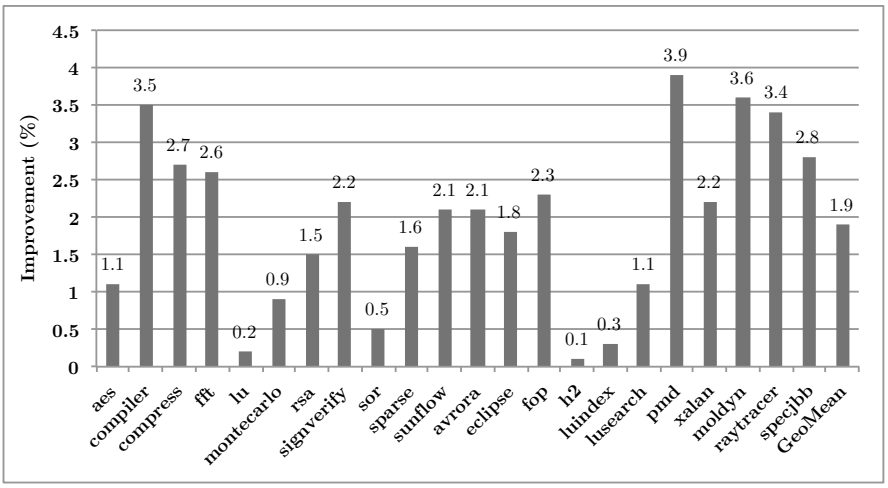
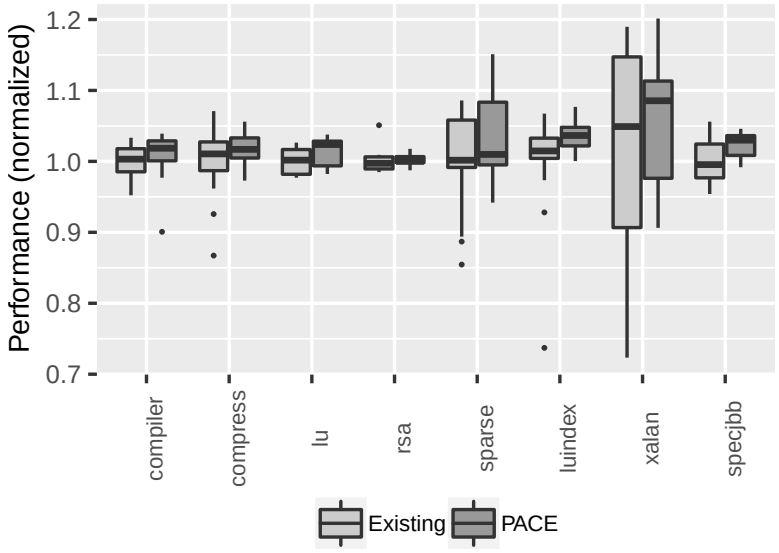


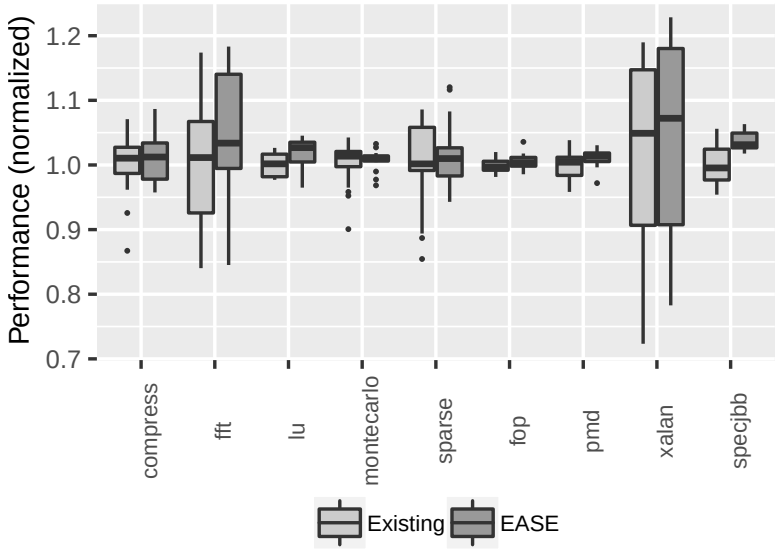
Figure 3.14: Improvement in JIT compilation time for EASE.

varies significantly across multiple iterations and even across JVM invocations, due to several non-deterministic factors. To measure the steady-state performance, we ran each benchmark as follows. For the SPECjvm and the DaCapo benchmarks, we executed K (=11) warmup iterations and used the following iteration to measure the performance metric: operations-per-second for SPECjvm and time taken for DaCapo. For SPECjbb, we used the score (in operations-per-second) reported by the benchmark harness (after warming up) in each run. For the JGF benchmarks, we use the time for each run as calculated using the `time` command. Finally, in order to account for systemic bias and the variations across JVM invocations, we ran the three analyses in alternating order twenty times, and show the variations as box plots (in Figure 3.15), normalized over the mean of the twenty runs of the corresponding base version (labeled Existing in the plots). We report only those benchmarks where the mean performance difference was more than 0.5% (thereby accounting for possible noise).

For PACE, we find that the geomean performance improvements for most benchmarks are modest: about 1%, except for SPARSE (3.5%), LUINDEX (3.4%), and SPECJBB (3.6%). Note that though the improvements for XALAN (geomean 5.7%) look comparatively high, we also observe a high variance in its execution times across both the versions (Existing and PACE in Figure 3.15a); this makes it difficult to conclude much about this benchmark. As most null-checks in the HotSpot JVM are actually handled using hardware traps (whose cost is not very high), the improvements obtained by PACE are along the expected lines.



(a)



(b)

Figure 3.15: Performance improvement of (a) PACE and (b) EASE, over the respective existing analyzers of C2.

For EASE (see Figure 3.15b), the mean performance improvements go up to 4.4% (for FFT), the geomean being 1.79% (for the shown benchmarks). For the benchmarks where EASE elided some synchronization operations but the improvements were negligible (e.g., SUNFLOW), we found that most elided synchronization operations were associated with static synchronized methods that were called infrequently.

Considering that the above performance improvements are observed in the presence of a host of other optimizations performed by the HotSpot JVM, we believe that these gains are important. Note that PACE and EASE may improve the performance

of an application in two ways: (i) by saving time during JIT compilation; and (ii) by enabling additional optimizations due to the enhanced precision. Note that the performance gains depend on multiple runtime factors, such as the number of times the statements containing the elided null-checks and synchronization operations are actually executed (post-compilation) at runtime, the overall execution time, and so on.

Comparison with whole-program analysis during JIT compilation. An alternative approach to performing precise analyses in JIT compilers could be to create separate “analysis-threads" that analyze the methods being compiled in the background. However, such an approach is impractical as: (i) the time taken to perform precise analyses can be prohibitively high, and (ii) the analysis-threads may reduce the amount of parallelism available to the application. To establish this argument, we tried to perform a whole-program context-, flow- and field-sensitive analysis including the libraries statically in Soot. We set the cutoff to perform such an analysis as twice the actual execution time of the benchmarks under consideration. Let alone the larger SPECjvm and DaCapo benchmarks, we found that the analysis for even our smallest benchmark MOLDYN did not terminate within the set cutoff. Thus, performing such expensive analyses during JIT compilation would take more time than the actual program-execution time itself, and is fundamentally impractical.

Overall, we see that PYE can be used to perform highly precise program analyses without incurring any significant overheads during JIT compilation. The evaluation of PACE and EASE establishes them as practical alternatives for the existing analyzers of C2. We also note that as the overheads involved are quite small, the partial summaries generated using PYE can be used to enable sophisticated optimizations, which are currently performed only by complex JIT compilers such as C2, in faster compilers such as C1 and possibly in the interpreter as well. Even though we have implemented PYE and its instantiations in Soot and the HotSpot JVM, the proposed techniques can as well be implemented in other static analyzers such as WALA (2019), and other Java runtime environments such as the Jikes RVM (Alpern *et al.*, 2005).

CHAPTER 4

SCALABLE CONTEXT-SENSITIVE STATIC ANALYSES

This chapter describes our contributions to scale value-contexts based whole-program heap analyses, leading to the novel notion of LSRV-contexts. First, Section 4.1 describes the challenges in scaling traditional value-contexts based heap analyses, along with insights to resolve the same. Section 4.2 describes LSRV-contexts and our three-staged approach to compute and use the same. Section 4.3 discusses how we use LSRV-contexts to perform two non-trivial heap analyses: thread-escape analysis and control-flow analysis. Section 4.4 highlights some subtle aspects in the proposed approach and presents its correctness arguments. Finally, Section 3.5 presents a detailed evaluation of the proposed approaches by comparing LSRV-contexts based analyses against their corresponding versions based on traditional value-contexts.

4.1 Challenges in Scaling Context-Sensitive Analyses

Traditional value-contexts use dataflow values to restrict the combinatorial explosion of contexts in classical call-string based context-sensitive analysis. However, they do not scale well, both in terms of analysis time and memory usage, for performing common top-down context-sensitive heap-based analyses (such as escape analysis, control-flow analysis, and so on). In this section, we first illustrate the underlying problems and then describe our scalable precise solutions.

Problem 1: *Too much comparison.* In value-contexts based analyses, when a previously analyzed method is called from a new site, the value-context at the new call-site is compared with the ones at the previous call-site(s). For heap analyses, this involves comparing the whole parameter-reachable points-to graphs. Comparing such potentially large graphs for exact equality may lead to significant overheads.

Insight 1. The whole of the points-to graph reachable from the parameters is usually not *relevant* for the callee. For example, for the method `bar` in Figure 1.2a, the relevant part of the points-to graph (value-context) at its entry, for escape analysis, consists only of the objects pointed-to by `p` and `p.f1`. Hence, the relevant value-contexts for `bar` for the calls at lines 5 and 6 (shown in Figures 1.2f and 1.2g, respectively) are much smaller than the complete value-contexts (shown in Figures 1.2d and 1.2e, respectively).
Proposal: Identify/use *relevant value-contexts* for comparison.

Problem 2: *Too many contexts.* An important challenge that any context-sensitive analysis throws up for scalability is the number of contexts created during the analysis. As the number of contexts keeps increasing during the analysis, the associated method needs to be analyzed again and again in each new context. Each re-analysis consumes time (more so, if the analysis is flow-sensitive), and the generated summary increases the memory usage of the analysis, thus leading to scalability problems while performing precise analyses for large programs. Analyzing a method in a large number of contexts also implies more context-comparisons to be performed at subsequent call-sites for the method, thus aggravating the scalability issues further. This problem is pertinent even in case of the value-contexts based approaches (Khedker and Karkare, 2008; Padhye and Khedker, 2013), where the number of contexts is bound by the size of the lattice of the points-to graph, though they improve the scalability compared to the traditional call-string based approaches (Sharir and Pnueli, 1978; Shivers, 1991) (where the number of contexts can be unbounded).

Insight 2a. A major reason leading to a blow-up in the number of contexts is the failure to detect the equality of two contexts. For example, for performing escape analysis, consider the two relevant value-contexts (at lines 5 and 6) for the method `bar` in Figures 1.2f and 1.2g. Even though the relevant value-contexts are different, if the objects O_i, O_j, O_k and O_l do not escape, the escape-status of the object O_9 remains the same (*DoesNotEscape*) in both the contexts. Hence, once `bar` has been analyzed for the value-context at line 5, it need not be analyzed again at line 6, as the "level-wise" summary for the objects pointed to by `p` and `p.f1` match (see Figure 1.2h).
Proposal: Use *level-summarized relevant value-contexts*.

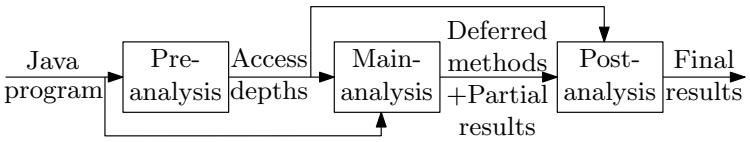


Figure 4.1: Block diagram of the proposed approach for scaling value-contexts.

Insight 2b. Not all the methods of a program modify the heap of its callers (for example, the method `fb` in Figure 1.2a). The analysis of such methods does not affect the heap of their callers. When we encounter a call to such methods (*caller-ignorable*), we can defer the analysis of that method and simply proceed to the next statement. These deferred methods can be analyzed in a *post-analysis* pass after the main-analysis is over, without losing any precision. The advantage of such deferring is that we can save the time and memory spent in analyzing them (in multiple contexts), and merging their points-to graphs with those of the callers, while performing the costly main-analysis.
Proposal: Identify/defer *caller-ignorable methods*.

4.2 LSRV-contexts

We now use the insights discussed above and describe our proposed three-stage approach (Figure 4.1 shows the block diagram) to scale value-contexts based top-down context-sensitive heap analyses. The three proposed stages are:

1. A fast flow-insensitive interprocedural *pre-analysis* that estimates, for each method m, the maximum depth of the parameter-reachable points-to graph till which the effects of m may be visible in its caller.

2. A flow- and context-sensitive *main-analysis* that takes advantage of the information gathered by the pre-analysis to (i) reduce the amount of comparison performed while checking value-contexts for equality (using the notion of relevant value-contexts), and (ii) identify and defer the analysis of caller-ignorable methods (methods with zero parameter-depth for all the parameters). Further, for comparing the relevant value-contexts for equality, we propose a novel abstraction called level-summarization that leads to fewer and more compact relevant value-contexts.

3. A flow- and context-sensitive *post-analysis* that analyzes the deferred methods, without compromising on precision.

Name	Meaning
G_m	Points-to map for method m.
$\texttt{ret}_m$	Set of objects returned by method m.
$\texttt{mInfo}_m$	Pre-analysis summary of method m.
`deferredMethods`	Map from deferred methods to context.
O_x.`accDpth`	Access-depth of object O_x.
O_x.`callerNode`	Indicates if O_x is passed from the caller.

Table 4.1: Some names used to describe the pre-analysis.

4.2.1 Pre-analysis

For each parameter p_i of a method m, the goal of the pre-analysis is to conservatively approximate its *access-depth*, which is the maximum depth of the caller-allocated portion of the points-to graph reachable from p_i that is accessed in m. We use $\texttt{mInfo}_m(i)$ to give the access-depth of p_i in m. Intuitively, $\texttt{mInfo}_m(i)$ is k, if there exists a maximal list of k fields such that either a caller-allocated object O_x pointed-to by $p_i.f_1.f_2...f_k$ is read in m, or $p_i.f_1.f_2...f_{k-1}$ points-to a caller-allocated object and m stores an object to $p_i.f_1.f_2...f_k$. We obtain this information by performing a lightweight bottom-up analysis on the call-graph. We assume the program to be in three-address code (Muchnick, 1997). For ease of presentation, we list the names of data structures and fields used in this section in Table 4.1.

The pre-analysis maintains a flow-insensitive points-to graph for each method m. The analysis begins by making each of the non-primitive parameters of m point to a dummy object. The fact that these dummy objects are created in one of the callers of m is noted by setting a special boolean field `callerNode` that is associated with the corresponding object node. We also maintain a field `accDpth` with each object, whose initial value is set to zero. For brevity, we skip the standard rules to update the points-to graphs, and show the extra processing required to obtain the depth information on processing load, return, method-exit and method-call statements, using the *proc*Pre* methods (Figure 4.2) described below. Note that copy (of the form a=b), store (of the form $a.f$=b) and other statements do not impact the access-depth.

procLoadPre: At a *load* statement L: a=$b.f$, if b is a parameter, and say the object pointed-to by b is O_b, then the points-to set of $O_b.f$ would be empty (as we started with dummy nodes for the parameters). To handle such loads, we add to the points-to graph

```
1  Function procLoadPre(st:load-stmt) // st is "L: a=b.f"
2      foreach O_b ∈ G_m(b) do
3          if G_m(O_b.f) ≠ ∅ then continue;
4          G_m(O_b.f) = O_L;
5          O_L.callerNode = true;
6          O_b.accDpth = max(O_b.accDpth, 1);
7          if L is inside a loop then O_L.accDpth = ∞;
8          else O_L.accDpth = max(O_L.accDpth, O_b.accDpth);

9  Function procRetPre(st:return-stmt) // st is "return r"
10     ret_m.add(G_m(r));

11 Function procCallPre(st:call-stmt) //st is "L: r=m'(b_1, ..., b_k)"
12     foreach b_i ∈ b_1, ..., b_k do
13         foreach O_i ∈ G_m(b_i) do
14             if mInfo_m' has been populated then
15                 O_i.accDpth = max(O_i.accDpth, mInfo_m'(i));
16             else O_i.accDpth = ∞; // possible due to recursion.
17     G_m(r) = O_L;
18     if mInfo_m' has been populated then
19         O_L.accDpth = max(∀O_r ∈ ret_m' O_r.accDpth);
20     else O_L.accDpth = ∞; // possible due to recursion.

21 Function procExitPre()
22     foreach parameter p_i (at index i) of m do
23         mInfo_m(i) = max(∀O_x ∈ G_m(p_i) findLvl(O_x));

24 Function findLvl(O_x)
25     if O_x.visited then return O_x.accDpth;
26     O_x.visited = true;
27     if ¬O_x.callerNode then return 0;
28     if O_x is a leaf in G_m then return O_x.accDpth;
29     let lvlDn = max(∀O_y ∈ G_m(O_x) findLvl(O_y));
30     O_x.accDpth= max(O_x.accDpth, 1+lvlDn);
31     return O_x.accDpth;
```

Figure 4.2: Obtaining access-depths in the pre-analysis.

G_m a node O_L representing the object(s) obtained from the dereference at statement L, and add O_L to the points-to set of $O_b.f$. As the object O_L actually flows from the caller of the current method, we set the field O_L.`callerNode` to *true*. If the `accDpth` of O_b is zero, we set it to one to indicate a dereference. Next, we update the value of O_L.`accDpth` based on whether or not L is inside a loop: if not, we set O_L.`accDpth` using O_b.`accDpth`; else, we update O_L.`accDpth` to ∞ (to conservatively imply that we do not know how much access-depth does O_L.`accDpth` represent).

procRetPre: At a *return* statement $return\ r$, if r is a non-primitive variable, we add the pointees of r to a set $\texttt{ret}_m$ that contains the objects returned by the method m.

procCallPre: At a *call* statement L: r=$m'(b_1, ..., b_k)$, for each object O_i pointed-to by each b_i, we update its `accDpth` using the access-depth information of the corresponding parameter of the method m' (using the summary $\texttt{mInfo}_{m'}(i)$; line 15). Next, we add a new object O_L representing the objects returned by m', and set $O_L.\texttt{accDpth}$ (line 19) to the maximum `accDpth` of the objects returned by m'. If the information about m' is not available (possible in the context of recursion), we conservatively set the `accDpth` fields of O_i and O_L to ∞ (to keep the analysis light; lines 16 and 20).

procExitPre: Once all the statements of a method m have been processed, we invoke the function *procExitPre*, which stores in $\texttt{mInfo}_m(i)$ the access-depth of the parameter p_i. For each object pointed-to by each parameter p_i, *procExitPre* invokes the recursive function *findLvl* (line 23), which returns the maximum access-depth of the argument object, by performing a depth-first search. For any object O_x that may be obtained from the caller (identified using $O_x.\texttt{callerNode}$), its access-depth is computed as the maximum of $O_x.\texttt{accDpth}$ and 1 + the maximum access-depth of its children nodes.

If an object in the points-to graph is reachable from a static (global) field, then that object might be accessed by a parallel thread. We handle this case (not shown in Figure 4.2) by setting the value of the `accDpth` field of all such objects to ∞, before performing the depth-first search.

Example. For Figure 1.2a, as the method `bar` does not access parameter# 0 (`this`) and stores to `p.f1.f2` (`p` is parameter# 1), $\texttt{mInfo}_{\texttt{bar}}$ would be $\{\langle 0, 0\rangle, \langle 1, 2\rangle\}$. Similarly, as the method `fb` takes no explicit arguments and does not affect the heap of its callers, $\texttt{mInfo}_{\texttt{fb}}$ would be $\{\langle 0, 0\rangle\}$.

After pre-analyzing all the methods of the input program, the computed `mInfo` is made available to the main-analysis. We next describe how we scale the main value-contexts based analysis using the information in `mInfo`.

4.2.2 Main-analysis

We now describe our changes to existing points-to graph based context-, flow-, and field-sensitive analyses that use value-contexts (in terms of the points-to graphs reachable from actual arguments at call-sites). We assume that the underlying analyses maintain flow-sensitivity in the standard way, that is, as "in" and "out" points-to graphs, before and after each statement, respectively. We now highlight how we scale such context-sensitive analyses (using the results of the pre-analysis), by focusing on the method-call statements. The handling of all other statements is standard (Whaley and Rinard, 1999; Choi *et al.*, 1999).

The function *procCallMain* in Figure 4.3 shows the handling of a call-statement st, with H_{in} as the points-to graph, in the main-analysis stage. For a method m, the entry points-to graph H_e is a copy of the points-to graph reachable from the formal parameters of m, obtained by invoking *getEntryPTG* (code not shown). If the access-depth for every parameter of m is zero (based on $\texttt{mInfo}_m$ computed in the pre-analysis), we say that m is *caller-ignorable*, that is, m does not affect the heap of its caller(s). If so, we defer the analysis of m at st, and use H_{in} as the points-to graph after st (stored in $H_{out'}$; line 4). We use the map `deferredMethods` to store the fact that the analysis of m was deferred in the value-context H_e.

If m is not caller-ignorable, we invoke the function *getSummary*. For each context c in which m was previously analyzed, we compare H_e with the entry points-to graph H_c at c. To avoid the overheads of comparing the whole of H_e and H_c, we use two important optimizations: (i) For each considered parameter, we compare the value-contexts only till the access-depth for that parameter (obtained from $\texttt{mInfo}_m$, populated in the pre-analysis) – *relevant value-contexts*. (ii) We use an efficient analysis-specific technique, which we call *level-summarization*, to summarize the contexts level-wise, and compare the level-summarized relevant value-contexts (*LSRV-contexts*) level by level. We describe the details of level-summarization for two analyses, thread-escape analysis and control-flow analysis, in Section 4.3. These two optimizations compact the points-to graphs representing the heaps being compared into smaller subgraphs, and importantly the comparison of just those subgraphs is sufficient to conclude about the equality of the heaps under consideration.

```
1  Function procCallMain(st: call-stmt, m: method, H_in: ptg)
2      ptg H_e = getEntryPTG(H_in, st); ptg H_out';
3      if m is caller-ignorable then
4          H_out' = H_in;
5          deferredMethods.put(m, H_e);
6      else H_out' = getSummary(m, H_e);
7      putSummary(m, H_e, H_out');
8      // H_out = H_out' combined with relevant parts of H_in.

9  Function getSummary(m: method, H_e: ptg)
10     let prevContext = null;
11     foreach context c in which m was analyzed do
12         prevContext = c;
13         let H_c be the entry points-to graph at c;
14         foreach non-primitive parameter p_i of m do
15             maxLvl = mInfo_m(i);
16             if maxLvl == 0 then continue; // p_i has no impact
17             curSumm = level-summary(H_e(p_i), maxLvl);
18             oldSumm = level-summary(H_c(p_i), maxLvl);
19             if curSumm ≠ oldSumm OR ¬eqDown(H_e,H_c,H_e(p_i),H_c(p_i),1,maxLvl)
                 then return analyze(m, H_e); // no matching context found
20     return the exit points-to graph at prevContext;

21 Function eqDown(H_e: ptg, H_c: ptg, curPts: set, oldPts: set, curLvl: int,
       maxLvl: int)
22     if maxLvl < curLvl then return true;
23     if curPts and oldPts have been visited then return true;
24     Mark curPts and oldPts as visited;
25     foreach field f do
26         curPts = ∪_{∀O∈curPts} H_e(O.f);
27         oldPts = ∪_{∀O∈oldPts} H_c(O.f);
28         curSumm = level-summary(curPts, maxLvl);
29         oldSumm = level-summary(oldPts, maxLvl);
30         if curSumm ≠ oldSumm then return false;
31     return eqDown(H_e, H_c, curPts, oldPts, 1+curLvl, maxLvl);
```

Figure 4.3: Handling call-stmts in the main-analysis. Abbreviation: *ptg*: points-to graph. The function *putSummary* stores the summary; details skipped.

Say the LSRV-contexts for H_e and H_c are `curSumm` and `oldSumm`, respectively. We conclude that the current context does not match H_c (line 19) and analyze m afresh in the context H_e, if either `curSumm` and `oldSumm` (which are the parameter-wise level-summaries) do not match, or if the recursive comparison of the next levels of the LSRV-contexts (done by calling the function *eqDown*) fails; see line 19.

The function *eqDown* performs a level-wise comparison for each field of the objects

being compared. The function is recursive; it returns *true* (implying that the level-wise comparison for both the graphs is equal) under one of the following conditions: (i) the access-depth for the current parameter (as computed by the pre-analysis) is less than the current level (line 22); or (ii) the objects at the current level for both the graphs have already been compared (line 23) – a situation possible in points-to graphs with cycles. The function *eqDown* returns *false* if at any level it finds that the level-summary for the two points-to graphs does not match (line 30).

If the entry points-to graph H_e for the method m at statement st matches a previous value-context (say `prevContext`), then we need not analyze m at st. In such a case, we can simply use the points-to graph at the exit of the analysis performed in `prevContext` as the summary of m at st (line 20). Note that using the summary computed in `prevContext` is sound as `prevContext` is equivalent to the current context for the analysis under consideration. In all the cases, after obtaining the summary $H_{out'}$, we merge $H_{out'}$ with the non-parameter reachable portions of H_{in} (line 8) to obtain the points-to graph H_{out} after st (using standard merging rules (Whaley and Rinard, 1999)).

Example. The method `bar` in Figure 1.2a does not affect any object(s) reachable from the receiver (`this`). Further, the access-depth of the parameter `p` of `bar` is 2. Thus, the relevant value-contexts for `bar`, for the calls made at lines 5 and 6, are shown in Figures 1.2f and 1.2g, respectively. Similarly, the method `fb` does not affect any object(s) reachable from the receiver, and hence, the relevant value-context for `fb` is empty. Thus, `fb` is a caller-ignorable method and need not be analyzed at any of its call-sites during the main-analysis.

4.2.3 Post-analysis

The main-analysis generates a summary for each method in the program in each LSRV-context in which it was called, except for the deferred methods. The deferred methods do not affect the heap reachable from the parameters of their callers. However, their own summary may depend on the heap of the caller. For example, say we are performing an analysis to determine which of the dereferences in a program are guaranteed

to be made on concrete (non-null) objects. Say the method m being analyzed has a statement *L: p.foobar()*, where p is a parameter of m and *foobar()* is a caller-ignorable method. The pre-analysis (Section 4.2.1) would infer that m is caller-ignorable as it does not affect the heap reachable from the parameter p, and hence for each value-context, the main-analysis would defer the analysis of m. However, whether the dereference performed at statement *L* is done on a concrete object or not, depends on the points-to information of the actual argument in the points-to graph of m's caller(s). Thus, in each skipped value-context, we still need to analyze m. We perform this task in the post-analysis stage by iterating over the entries in the `deferredMethods` map and invoking *getSummary*(m, H_e) for each (m, H_e) in the map, to generate the corresponding context-sensitive summary. Thus, by the end of the post-analysis, we obtain context-sensitive summaries for all the methods in the program.

There are two clear advantages of deferring the analysis of caller-ignorable methods and performing the post-analysis stage as a separate pass. First, as the deferred methods are guaranteed not to affect the caller's heap, we need not spend time in merging the heap of the callee with that of the caller, after each call statement (a potentially costly operation). Second, top-down context-sensitive heap analyses may, in general, consume a large amount of memory, as the points-to graphs reachable from the callers keep flowing towards the leaves of the call-graph till the points-to graphs of the callees are merged back. Deferring the analysis of certain methods may avoid the traversal of certain long branches of the call-graphs and thus save the memory required to propagate the points-to graphs therein, during the costly main-analysis.

4.3 Instantiations of LSRV-contexts

In this section, using the techniques proposed in Section 4.2, we present two popular context, flow-, and field-sensitive points-to graph based analyses: (i) thread-escape analysis; and (ii) control-flow analysis. Though both these analyses are based on points-to information, the corresponding lattices of dataflow values are quite different (see Section 2 for an overview), and hence they offer a wide illustration of our proposed techniques. For both the analyses, we mainly highlight the computation of the level-

summary functions (see Figure 4.3). Note that apart from the definition of the function *level-summary*, the rest of the handling of a method-call statement remains the same as in Figure 4.3.

4.3.1 Thread-escape Analysis using LSRV-contexts

Consider a method m that is being analyzed. Say p is one of the parameters of m, and H_e is the value-context (points-to graph at the entry of m). Say the graph-front induced by the edge sequence $f_1, f_2, ..., f_k$ in H_e is given by the set $S = \{O_1, O_2, ..., O_n\}$. For escape analysis, we claim that the relevant information represented by the objects in S is the set of escape-statuses of the objects $O_1, O_2, ..., O_n$. For example, if a variable q is set to the object obtained by dereferencing p using an expression of the form $p.f_1.f_2...f_{k-1}$, then for a sound escape-analysis technique, the following two observations hold: (i) If none of the objects in S escapes, then any object stored into any field f_i of q will not escape due to the assignment involving this dereference. (ii) If any of the objects in S escapes, then any object stored into $q.f_i$ will also be marked as escaping. Hence for escape analysis, we define the level-summary of a set S of objects as the meet of the escape-statuses of the objects in S.

The *level-summary* function for escape analysis takes two arguments: H and k, where H is a points-to graph (value-context) and k is the access-depth (see Section 4.2.2). It computes the level-summaries for each possible edge-sequence of size at most k and then returns a graph, which includes a unique node for each unique edge sequence and if s_1 and s_2 are the nodes (level-summaries) corresponding to the edge sequences $f_1, f_2, \cdots f_{k_1}$ and $f_1, f_2, \cdots f_{k_1}, f_k$, respectively, then there is an edge between s_1 and s_2, labeled k.

Example. Consider the calls to the method `bar` in Figure 1.2a, with the respective relevant value-contexts as shown in Figures 1.2f and 1.2g. As none of the objects O_i, O_j, O_k and O_l escape (implying that O_a and O_b also must not escape), Figure 1.2h represents the LSRV-context for both Figures 1.2f and 1.2g. As a result, with LSRV-contexts, we need not analyze `bar` for the call made at line 6, and can use the analysis results obtained for the call at line 5 itself.

4.3.2 Control-flow Analysis using LSRV-contexts

Consider a method m that is being analyzed. Say p is one of the parameters of m, and H_e is the value-context (points-to graph at the entry of m). Say the graph-front induced by the edge sequence $f_1, f_2, ..., f_k$ in H_e is given by the set $S = \{O_1, O_2, ..., O_n\}$. For control-flow analysis, we claim that the relevant information represented by the objects in S is the set of types represented by the objects $O_1, O_2, ..., O_n$. For example, if a variable q is set to the object obtained using an expression of the form $p.f_1.f_2...f_{k-1}$, and if m consists of a statement *L: q.foo()*, then the set of possible callees at *L* depends only on the types of the objects in S. Hence for control-flow analysis, we define the level-summary of a set S of objects as the union of the types of the objects in S.

Example. Consider the calls to the method `bar` in Figure 1.2a, with the respective relevant value-contexts as shown in Figures 1.2f and 1.2g. Here, if the types of O_a and O_b are same, then the two LSRV-contexts at level 1 from the parameter p would be the same. Similarly, if the set of types for $\{O_i, O_j\}$ and $\{O_k, O_l\}$ are same, then the LSRV-contexts at level 2 would also be the same. In such a case, we need not analyze `bar` for the call made at line 6.

In Section 4.5, we show that LSRV-contexts not only lead to a significant reduction in the escape and control-flow analysis time and memory of several benchmarks, but also facilitate the analysis of previously unanalyzable benchmarks.

4.4 LSRV-contexts: A Discussion

1. Scope. In this chapter, in order to scale context-sensitive heap analyses that use value-contexts, we have presented three main ideas: relevant value-contexts, level-summarization, and deferred methods. While the second idea is analysis-specific and needs to be customized for each analysis, the other two are general in nature and can be used directly for any heap analysis.

2. Cost. It can be argued that an approach using LSRV-contexts is likely to never increase the cost compared to that using traditional value-contexts. The relevant value-contexts, as discussed in Section 4.2, are always a subset of the points-to graphs, and

level-summarization is analysis-specific but usually leads to a smaller lattice of dataflow values (as shown in Section 4.3). For the analyses under consideration, we show in Section 4.5 that LSRV-contexts are much cheaper (in terms of time as well as memory) than traditional value-contexts.

3. Special treatment of methods. In the JDK library, the methods `equals` and `toString` are overridden heavily, and often call other `equals` and `toString` methods. For these methods, we found that the Spark tool (Lhoták and Hendren, 2003), which we used to build our base call-graph, led to huge strongly-connected components (clique with up to 356 JDK methods), which blew up the analysis time and memory. Hence, just for these `equals` and `toString` methods, we suggest using the approach of Smaragdakis *et al.* (2014) and analyzing them conservatively (intra-procedural). We believe it to be a reasonable design decision, as based on our analysis of the complete JDK library (version 8), these methods do not explicitly modify their callers' heap. To be consistent, we handle these methods uniformly, in all the implementations evaluated in Section 4.5. Note: we do analyze their implementations in the applications like any other method.

4. Correctness. As described in Section 4.2, in order to determine whether two value-contexts for a method are equivalent, we compare only the LSRV-contexts. Further, we defer the analysis of caller-ignorable methods, and analyze them in a separate pass. We now sketch the correctness argument of our design.

Conjecture 4.4.1. The pre-analysis (see Figure 4.2) computes a conservative approximation of the relevance information for each parameter.

Conjecture 4.4.2. For a given analysis Ψ, level-summarizing a points-to graph based on the lattice of Ψ does not lead to any loss of precision for Ψ.

Theorem 4.4.3. *For a particular analysis under consideration, compared to value-contexts, LSRV-contexts do not lose any relevant information.*

Proof. (Proof Sketch) For a given value-contexts based analysis $\mathcal{A}$, say the LSRV-contexts variant be represented by $\mathcal{A}'$. Assuming Conjectures 4.4.1 and 4.4.2 to be true, during the main-analysis, at a certain call-site, if $\mathcal{A}$ re-analyzes a target method m, then either $\mathcal{A}'$ also re-analyzes m, or there exists a prior instance of $\mathcal{A}'$ analyzing

m such that: (i) the abstract heap obtained by $\mathcal{A}$ on re-analyzing m matches the one obtained by $\mathcal{A}'$ in the prior analysis-instance; and (ii) in $\mathcal{A}$, during the re-analysis of m, for each accessed abstract heap-location, the abstract value read/written matches that of $\mathcal{A}'$ in the prior analysis-instance. □

Corollary 4.4.4. Instead of comparing complete value-contexts, it is sufficient to check the equality of the level-summaries of the corresponding relevant value-contexts.

Proof. Follows from Theorem 4.4.3. □

Theorem 4.4.5. *Analyzing a deferred method m in the post-analysis does not affect the precision of its callers, and of m.*

Proof. (Proof Sketch) Assuming Conjecture 4.4.1 to be true, for each caller-ignorable (and hence deferred) method m, if $\mathcal{H}$ is the reachable abstract heap present before analyzing the call to m, and $\mathcal{H}'$ is the reachable abstract heap after analyzing the call-statement, then $\mathcal{H} = \mathcal{H}'$. That is, the precision of the callers of m does not get affected. Further, as the access-depths for all the parameters of m is zero, the results computed for m do not depend on the callers' heap $\mathcal{H}$. □

Theorems 4.4.3 and 4.4.5 ensure that the precision of an analysis using our proposed approach is the same as that using traditional value-contexts.

4.5 Implementation and Evaluation of LSRV-contexts

We have implemented both of our proposed instantiations of thread-escape analysis and control-flow analysis, in the Soot framework (Vallée-Rai *et al.*, 1999). The implementation spans 3967 lines of Java code for the escape analysis, and 3899 lines of Java code for the control-flow analysis. We have performed our experiments using the OpenJDK HotSpot JVM (version 8), on a 2.3 GHz AMD Abu Dhabi system with 64 cores and 512 GB of memory.

We have evaluated our techniques on seven benchmarks from the DaCapo-9.12 suite (Blackburn *et al.*, 2006), and the three multithreaded benchmarks from Section C

1	2	3	4
Benchmark	**Application**		**#Referred**
	#classes	**size (MB)**	**JDK classes**
avrora	527	2.7	1588
batik	1038	6.0	3700
eclipse	1608	14.0	2589
luindex	199	1.3	1485
lusearch	198	1.3	1481
pmd	697	4.1	1607
sunflow	225	1.7	3509
moldyn	13	0.15	1555
montecarlo	19	0.67	1555
raytracer	19	0.21	1555

Table 4.2: Details of the benchmarks used. The number of classes has been computed using Spark's (Lhoták and Hendren, 2003) call graph.

of the JGF suite (Daly *et al.*, 2001), listed in Table 4.2. We used the extremely helpful tool TamiFlex (Bodden *et al.*, 2011) to resolve reflective calls in the original DaCapo benchmarks, so that they could be analyzed by Soot. The benchmarks excluded from the DaCapo suite are the ones which either could not be translated by TamiFlex, or could not be analyzed by Soot (using OpenJDK8) after the TamiFlex pass.

Table 4.2 shows some static characteristics of the used benchmarks. The sizes of the benchmarks (excluding the JDK library) varied from 150 KB (small programs) to 14 MB (large applications), and the number of application classes varied from 13 to 1.6K. Table 4.2 also shows the number of JDK classes referred by each benchmark (gives the total number of analyzed classes), computed using the call-graph generated by the Spark (Lhoták and Hendren, 2003) tool (our default call-graph).

We now present an evaluation to study the impact of our proposed techniques on the scalability of context-, flow- and field-sensitive escape and control-flow analyses. For both the analyses, we compare four different versions: (i) Base: a standard value-contexts based implementation, where the points-to graphs at method entries are considered as the contexts. (ii) OM: a version where only the main analysis (as proposed in Sections 4.2 and 4.3) is performed. Thus, the level-wise summaries are used as contexts, but no pre-analysis (that is, trimming) and post-analysis (that is, deferring) are performed. (iii) PM: both the pre and the main analyses are performed. Thus, the level-wise summaries are trimmed based on the access-depths computed by the pre-

1	2	3	4	5	6	7	8
Benchmark	**Analysis time (seconds)**					**Average #contexts**	**Memory (GB)**
	B_e	**Pre**	**Post**	**PM**	**PMP**	**PMP**	**PMP**
avrora	-	1.0	0.4	603	225	1.4	21
batik	-	2.2	1.8	3483	1722	1.4	45
eclipse	-	2.7	6.0	-	2275	1.9	57
luindex	-	1.1	0.4	204	70	1.3	6
lusearch	-	1.0	0.5	343	87	1.3	10
pmd	-	1.3	0.4	531	157	1.3	11
sunflow	-	2.1	1.6	1477	486	1.3	21
moldyn	-	0.9	0.4	178	55	1.3	6
montecarlo	-	0.9	0.4	183	57	1.3	6
raytracer	-	0.9	0.4	183	54	1.3	6
geomean	-	1.3	0.7	444	192	1.4	13

Table 4.3: Evaluation results for escape analysis. Abbreviations: B_e: $Base_e$; PM: Pre and Main; PMP: Pre, Main, and Post. A '-' implies that the analysis did not terminate in 3 hours.

1	2	3	4	5	6	7	8	9	10	11	12
Benchmark	**Analysis time (seconds)**						**Average #contexts**			**Memory (GB)**	
	B_c	**Pre**	**Post**	**OM**	**PM**	**PMP**	**B_c**	**OM**	**PMP**	**B_c**	**PMP**
avrora	1322	1.0	0.5	231	71	55	9.5	2.6	1.2	54	11
batik	-	2.2	2.4	-	1033	946	-	-	1.3	-	64
eclipse	-	2.7	6.1	-	1312	988	-	-	1.4	-	49
luindex	1175	1.1	0.7	507	60	46	10.6	2.9	1.2	58	11
lusearch	1215	1.0	0.9	561	58	57	10.5	3.6	1.2	54	11
pmd	5769	1.3	0.7	1243	130	108	11.9	4.5	1.2	127	13
sunflow	-	2.1	2.2	-	692	684	-	-	1.2	-	53
moldyn	929	0.9	0.6	222	54	53	9.5	2.5	1.3	29	11
montecarlo	925	0.9	0.3	238	60	53	9.4	2.6	1.2	29	9
raytracer	940	0.9	0.3	211	61	53	9.4	2.6	1.2	29	10
geomean	1364	1.3	0.9	351	151	130	10.1	3.0	1.2	47	18

Table 4.4: Evaluation results for control-flow analysis. Abbreviations: B_c: $Base_c$; OM: Only Main; PM: Pre and Main; PMP: Pre, Main and Post. A '-' implies that the analysis did not terminate in 3 hours; the geomean was computed by excluding non-terminating benchmarks.

analysis. (iv) PMP: the full proposed version where all the three analyses (pre, main, and post) are performed. We compare these versions on three parameters: (i) analysis time, (ii) average number of created contexts, and (iii) peak memory consumption. As prior works (Khedker and Karkare, 2008; Padhye and Khedker, 2013) have already shown that the classical call-string based approach does not scale well compared to the value-contexts based approach, we omit a comparison against the same.

4.5.1 LSRV-contexts: Analysis Time

Tables 4.3 and 4.4 show the time taken for performing the escape analysis and the control-flow analysis, respectively, by the different versions. Base$_e$ implements the (Base) thread-escape analysis of Whaley and Rinard (1999), and Base$_c$ implements the (Base) control-flow analysis of Padhye and Khedker (2013); both Base$_e$ and Base$_c$ use value-contexts based context-sensitivity. We found that Base$_c$ did not terminate within 3 hours (our set cutoff) for three large DaCapo benchmarks, and Base$_e$ did not terminate in 3 hours for any of the benchmarks. It clearly shows the scalability issues with the Base variations, which simply use the parameter-reachable points-to graph at the entry of a method as the value-context.

Columns 3-4 in Tables 4.3 and 4.4 show the times taken by the pre and the post analyses while performing the escape and the control-flow analyses, respectively. As the proposed pre-analysis is agnostic to the heap analysis being performed, it takes the same time (on average, 1.3 seconds) for both the analyses. We observe that together the pre and the post analyses take very less time – just 2.0 seconds for escape analysis, and 2.2 seconds for control-flow analysis, on average.

Column 6 in Table 4.3 and column 7 in Table 4.4 (labeled PMP) show the full analysis time of our proposed technique, which includes the pre, the main, and the post-analysis times. Not only is our proposed technique able to analyze all the benchmarks within the set cutoff, the average required time is just 192 seconds for escape analysis, and 130 seconds for control-flow analysis (the largest value being less than 40 minutes, for ECLIPSE). It can be seen that the time required depends mostly on the size of the benchmark. Note that though SUNFLOW appears to be a relatively small benchmark (see column 3, Table 4.2), the corresponding analysis time is high, as SUNFLOW references a large number of the JDK library classes (see column 4, Table 4.2).

In order to individually estimate the effects of the pre and the post analyses (that is, the insights presented in Section 4.1) on the scalability, we studied the analysis times of the OM and the PM versions (see column 5 in Table 4.3 and columns 5-6 in Table 4.4). For escape analysis, the OM version did not terminate in 3 hours for any benchmark (hence not shown), while for control-flow analysis, it did not terminate

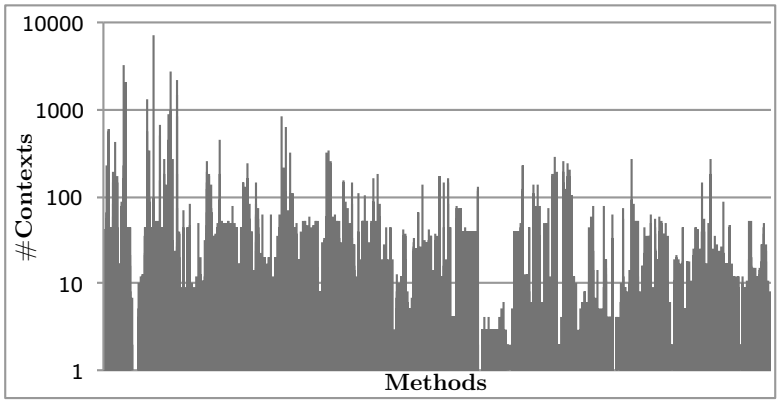
for three large DaCapo benchmarks (BATIK, ECLIPSE, and SUNFLOW). This indicates that only level-wise summarization (done in OM version, *Insight 2a*) is not sufficient to scale all the analyses under consideration. We can see that though the time savings due to the pre-analysis alone (PM version, *Insights 1+2a*) are significant (except for the escape analysis of ECLIPSE, where it did not terminate), the post-analysis improves it further. The PMP version (*Insights 1+2a+2b*) runs faster than the PM version by about 56% and 14% for the escape and the control-flow analyses, respectively. Thus, by spending just $\sim$2 seconds for the pre and the post analyses over the OM version, the PMP version successfully scales both the analyses.

Overall, we see that the combination of the pre, the main, and the post analyses helps us perform previously non-terminating analyses in less than 40 minutes, for all the benchmarks under consideration.

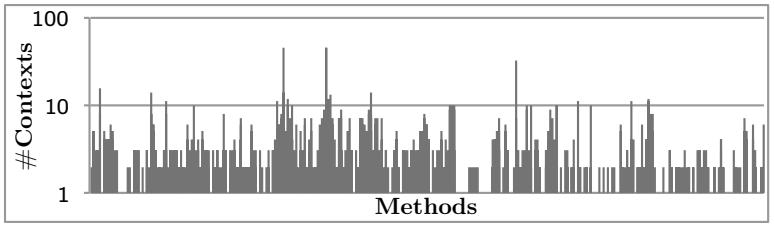
4.5.2 LSRV-contexts: Number of Contexts

Columns 7 in Table 4.3 and 9-11 in Table 4.4 show the average number of contexts created during escape analysis and control-flow analysis, respectively, over all the methods. The numbers for the PM version are not shown separately; they match the ones for the PMP version, as the deferred methods in PMP are later analyzed in all the deferred contexts. For escape analysis, on average, PMP creates 1.4 contexts per method (Base_e and OM did not terminate, and hence average contexts not reported). For control-flow analysis, for the cases where Base_c terminated, we can see that the average number of contexts created per method is about 10.1. On the other hand, the number is around 3.0 for the OM version (for control-flow analysis), and just 1.2 for the PMP version. This implies that our proposed techniques significantly reduce the number of times a method is analyzed ($\sim 8\times$, on average).

To further visualize the difference in the number of contexts created, we have plotted two histograms for the benchmark PMD for control-flow analysis; see Figure 4.4. The charts show the number of contexts created per method in Base_c (Figure 4.4a) and PMP (Figure 4.4b) versions, respectively. The methods (in the x-axis) are arranged in alphabetical order, and the number of contexts (y-axis) is shown growing logarithmi-



(a) Base$_c$ version.



(b) PMP version of LSRV-contexts.

Figure 4.4: Number of contexts created per method during the control-flow analysis of the benchmark PMD.

cally. The high density of the bars for Base$_c$ clearly shows the large number of contexts created for a large number of methods. For instance, the maximum number of contexts created in the Base$_c$ version was 7324 for the method *java.lang.Object: void <init>*, for which there was only one context created in the PMP version. We observed a similar trend for all the benchmarks.

Overall, we see that our proposed techniques were able to significantly reduce the number of contexts created per method, which in turn led to a significant reduction in the resources spent in analyzing those methods, thus making the analyses scalable.

4.5.3 LSRV-contexts: Peak Memory Usage

Tables 4.3 and 4.4 also show the peak memory consumption (computed using the `time -v` command) of the Base and the PMP versions for the escape and the control-flow analyses, respectively. As Base$_e$ did not terminate within the cutoff of 3 hours, the corresponding statistic is not shown. However, as a point of indication, the usage at

the end of 3 hours for the smallest benchmark MOLDYN was about 373 GB, which indicates that for larger benchmarks the analysis may run out of memory, if run for a longer time. The high memory usage is due to the large number of contexts created during the analysis and the resultant flow-sensitive points-to graphs maintained therein.

In comparison, the memory usage in the PMP version was several magnitudes lesser, and was just 13 GB and 18 GB respectively, on average, for the two analyses. We argue that given the range and size of the benchmarks under consideration, this is quite reasonable. We also observe that the memory usage varied mostly with the size of the benchmark: lower for the JGF benchmarks (smaller) and higher for BATIK, ECLIPSE, and SUNFLOW (larger), for both escape and control-flow analysis.

Overall, we note that our proposed techniques allow performing precise whole-program heap analyses, which earlier did not scale even with a large amount of memory ($\sim$512 GB), now on systems with much less memory ($\sim$32-64 GB).

CHAPTER 5

HEAP CLONING AND OBJECT-SENSITIVITY

The idea behind performing context-sensitive analyses is to be able to partition the dataflow facts (points-to information for heap analyses) across different contexts in which a method is analyzed. The partitioning happens by specializing the elements in a method to the current context. In the scheme of analysis presented so far (that is, in Chapter 4), while analyzing a method m in a context c (irrespective of the context abstraction), we specialized the stack variables in m with c. Another technique to enhance the partitioning efficacy is to also specialize the heap objects with the context in which they are created, a process termed as *heap cloning* (Nystrom *et al.*, 2004).

In this chapter, we first enhance the precision of LSRV-contexts by adding heap cloning to the same. We then study some common patterns where heap cloning improves the precision of existing context-sensitive analyses. Here we perform a deeper comparison with recent object-sensitive analyses that include heap cloning as a part of the definition of object-sensitivity (Smaragdakis *et al.*, 2011). In particular, we note the cases where LSRV-contexts may lead to more precise results than object-sensitive analyses, and cases where compared to object-sensitive analyses, LSRV-contexts may miss out on potential precision-enhancement opportunity offered by heap cloning. Motivated by the scalability of LSRV-contexts, we next extend the definition of LSRV-contexts to also consider the k-level object context as the context abstraction. We find that this leads to a very novel experiment of performing a very precise context-sensitive analysis, and uniquely connects the lattices of call-site-sensitive and object-sensitive analyses.

We implement our two ideas, the first of LSRV-contexts extended with heap cloning (say LSRVH), and the second one of LSRVH extended with the precision of k-object-sensitivity (say LSRV$k$objH), for performing control-flow analysis of Java programs, in the Soot framework. In order to deal with the potential overheads, we apply the final extended definition selectively for a subset of methods, which are identified using little additional computation over the pre-analysis of the LSRV approach. We evaluate

the implementations by comparing them with standard k-object-sensitive analyses with heap cloning. The results show that not only do the newer proposals scale well to large benchmarks, they also enhance the precision of LSRV-contexts and object-sensitive analyses significantly.

5.1 LSRV-contexts with Heap Cloning

Heap cloning (Nystrom *et al.*, 2004) is a technique that distinguishes different instances of objects allocated on heap, based on the context in which they are created. This may lead to the removal of some spurious points-to facts, and hence may improve the precision of the analysis being performed. For example, consider the Java code fragment shown in Figure 5.1. Say we are performing a call-string based context-sensitive points-to analysis. Without heap cloning, though there are two contexts (`foo:4` and `foo:5`, created at lines 4 and 5, respectively) in which the method `getB` is analyzed, it returns the same object O_{10} in both the contexts. As a result, in the points-to graph for the method `foo`, the variables `b1` and `b2` (conservatively) point to the same object, and hence are potential aliases (though they will point to different `B` objects at runtime). With heap cloning, the object created at line 10 is qualified by the context in which created. Say, the object returned by the first call to `getB` is denoted by $O_{foo:4\_10}$, and the one returned by the second call to `getB` is denoted by $O_{foo:5\_10}$. As a result, in the points-to graph for `foo`, the variables `b1` and `b2` would point to different `B` objects ($O_{foo:4\_10}$ and $O_{foo:5\_10}$, respectively), and would (correctly) not be identified as aliases, hence possibly triggering other optimizations.

In Chapter 4, we represented abstract objects with the (unique) label of their allocation site; thus, all the abstract objects allocated at a statement with label l were represented as O_l. We now extend this definition to also include the context in which a corresponding abstract object was created. First, we assume that each new context is associated with a unique integer label. Next, we denote an abstract object allocated in a context c and statement l as O_{c_l}. In addition to the extended object abstraction, we also use a map `contextMap`, which associates each LSRV-context with a unique integer.

Note that without heap cloning, the number of objects while performing a heap

```
1  class B {}
2  class D {
3    void foo() {
4      B b1 = getB();
5      B b2 = getB();
6      // b1 and b2 are aliases without heap cloning,
7      // but not with heap cloning
8    }
9    static B getB() {
10     return new B();
11   }
12 }
```

Figure 5.1: Example to show the effect of heap cloning.

analysis is bound by the number of allocation statements in the program, which is finite. However, with heap cloning, the finiteness (and hence the termination of a given heap analysis) also depends on the number of contexts that could be created. Thus, for LSRV-contexts, if the lattice of the analysis under consideration is finite (which bounds the number of possible contexts), the number of objects that could get created with heap cloning is also finite, and the analysis would still be guaranteed to terminate.

Apart from the above-extended definition, in order to add heap cloning, the rest of the implementation of LSRV-contexts remains unchanged from what was proposed in Chapter 4. We call the resultant version as LSRVH (for LSRV-contexts with Heap cloning). Note that the precision of LSRVH can be more but never less than LSRV-contexts, with a possible tradeoff in the analysis time. In Section 5.5, we evaluate LSRVH by comparing its precision with plain LSRV-contexts.

5.2 Comparing LSRV-contexts and Object-sensitivity

Milanova *et al.* (2005) proposed another context abstraction called *object-sensitivity*, which distinguishes contexts based on the allocation site of the receiver object (the object pointed-to by the `this` pointer). Similar to call-string based analyses, for scalability, object-sensitive analyses also use a limit k on the length of the chain formed by the receiver objects.

It is well understood (Milanova *et al.*, 2005; Lhoták and Hendren, 2008) that as the contexts created in the call-site-sensitive and object-sensitive approaches are dif-

```
1  class B {
2    void foo{...}
3  }
4  class C extends B {
5    void foo{...}
6  }
7  class D {
8    void bar() {
9      B b1 = new B();
10     B b2 = b1.parid(new B());
11     B b3 = b1.parid(new C());
12     b2.foo();
13   }
14   B parid(B parm) {
15     return parm;
16   }
17 }
```

Figure 5.2: An example where a control-flow analysis using LSRV-contexts is more precise than one using object-sensitivity.

ferent, these approaches are in-principle incomparable. As value-contexts, and hence LSRV-contexts, are also based on call-site-sensitivity, they are also theoretically incomparable with object-sensitive analysis. Nevertheless, there may be cases where one of the approaches, with or without heap cloning, may be able to enable an optimization opportunity, and not the other.

Figure 5.2 shows an example of a case where a control-flow analysis based on LSRV-contexts would fare better in terms of precision than another that is based on object-sensitivity. Say both the classes `B` and `C` define a method `foo`. Here, the method `parid` would be analyzed in two different LSRV-contexts, one each at lines 10 and 11. As a result, `b2` would point only to a `B` object, and hence would infer that only `B`'s `foo` could be called at line 12. On the other hand, as the receiver object in both the calls to `parid` is the same (that is, O_9), an object-sensitive control-flow analysis would analyze `parid` in only one context. As a result, `b2` would point to two objects: one of class `B` and another of class `C` (the two parameters to `parid`), and hence would (conservatively) infer that both `B`'s and `C`'s `foo` could be called at line 12; that is, both the calls would be polymorphic (leading to imprecision).

Another interesting behaviour can be observed when heap cloning is added to both LSRV-contexts (to obtain LSRVH) and object-sensitivity. Figure 5.3 shows an example of a case where an LSRVH control-flow analysis may be less precise than a k-level

```
1  class W {
2    X f;
3    W() { f = new X(); }
4    void setG(Y y) {
5      f.g = y;
6    }
7    Y getG() {
8      return f.g;
9    }
10 }
11 class X { Y g; }
12 class Y {
13   void m() {...}}
14 class Z extends Y {
15   void m() {...}}
```

```
16 class D {
17   void bar() {
18     W w1 = new W();
19     Y y1 = new Y();
20     w1.setG(y1);
21
22     W w2 = new W();
23     Z z1 = new Z();
24     w2.setG(z1);
25
26     Y p = w1.getG();
27     p.m();
28     Y q = w2.getG();
29     q.m();
30   }}
```

Figure 5.3: An example where an LSRVH control-flow analysis is less precise than one based on $k$obj1h.

object-sensitivity based control-flow analysis with one-level of heap cloning (that is, $k$obj1h). Here, the class `W` is like a container with a field `f` of class `X` (initialized in the constructor of `W`). The field `g` of `X` objects may in turn store an object either of class `Y` or class `Z` (as `Z` extends `Y`).

The method `bar` of class `D` creates two `W` instances at lines 18 and 22, pointed-to by `w1` and `w2`, respectively. However, as the LSRVH contexts for the `W` constructor at both the allocation sites are the same (the level-summarized receiver), an LSRVH-based control-flow analysis would not re-analyze the constructor at line 22. As a result, the object pointed-to by `w1.f` and `w2.f` would be the same, that is, O_3. Consequently, though the method `setG` is analyzed in two LSRVH contexts (due to the types of the parameter being different) at lines 20 and 24, both `w1.f.g` and `w2.f.g` would point to both O_{19} and O_{23}. Thus, the variables `p` and `q` would point to both `Y` and `Z` objects (O_{19} and O_{23}, respectively), leading to both the calls to the method `m` (that is, at lines 27 and 29) being deemed as polymorphic.

Contrary to LSRVH, a $k$obj1h control-flow analysis would forcefully re-analyze the constructor of class `W` at line 22 (as the receiver object is different and a constructor call is like a method call with the object being allocated as the receiver). As a result, `w1.f` and `w2.f` would point to two different objects (O_{18_3} and O_{22_3}, respectively), `w1.f.g` and `w2.f.g` would respectively point to O_{19} and O_{23}, and hence both the calls to `m` would be monomorphic.

5.3 Merging LSRV-contexts and Object-sensitivity

Motivated by the scalability of LSRV-contexts and the additional precision of object-sensitivity (with heap cloning), we now present a novel mix of both the context abstractions, called LSRV$k$objH. Here, we extend the definition of our context abstraction by defining it as a pair of the LSRV-context and the k-level object-sensitive context. Thus, an approach using LSRV1objH as its context abstraction would in effect be using the LSRV-context as well as the one-level object-sensitive context with one level of heap cloning (that is, 1obj1h) to distinguish between the contexts of a method. Hence a method m would be re-analyzed in a new LSRV$k$objH context if any of the LSRV-context or the k-level object context has changed.

Recall (from Chapter 4) that in order to compute the *relevant value-context* for a method m, we performed a pre-analysis pass that computed a conservative approximation of the depth till which m accessed its callers' heap from each parameter p; this was called the *access-depth* of p. In order to keep the LSRV$k$objH approach efficient, we conditionally check the $k$obj context for a method m, only if the access-depth of the receiver (that is, parameter# 0) is non-zero. Thus, for methods for which the access-depth of the receiver is zero, only the LSRV-context is considered, and for methods with non-zero access-depth for the receiver, the complete LSRV$k$objH context is considered.

Recall that if k is a finite positive integer, k-object-sensitive analyses are guaranteed to terminate. Further, as discussed in Section 5.1, analyses using LSRVH contexts are also guaranteed to terminate. As a result, analyses that use LSRV$k$objH contexts, where the maximum number of contexts that could be created is a product of the possible number of LSRVH contexts and that of k-object-sensitive contexts, are also guaranteed to terminate.

5.4 Precision Lattice for the LSRV Approaches

We now discuss the theoretical precision of the various context abstractions discussed so far; see Figure 5.4. Without heap cloning, for a particular analysis, analysis-specific LSRV-contexts are as precise as traditional value-contexts (Khedker and Karkare, 2008),

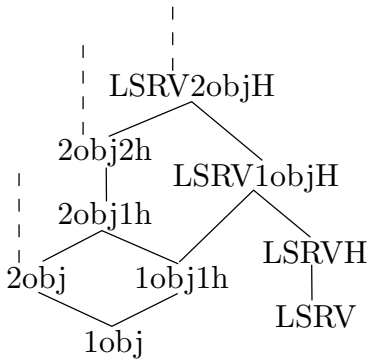


Figure 5.4: Precision lattice for various control-flow analyses.

which in turn are similar to the unbounded call-string based analyses (not shown in the figure). Among k-object-sensitive analyses, the precision increases with the value of k, that is, 2obj is more precise than 1obj, and so on.

With heap cloning, 1obj1h is more precise than 1obj, but it does not compare with 2obj. The 2obj1h analysis is more precise than both 2obj and 1obj1h, and is currently the most popular object-sensitive analysis (Tan *et al.*, 2017; Li *et al.*, 2018*b*,*a*). On the other hand, with heap cloning, LSRVH is more precise than LSRV, but still does not compare with object-sensitive analyses. However, once we merge LSRV-contexts and object-sensitive analyses to obtain LSRV$k$objH analyses, we get a connection between the two lattices. As an instance, LSRV1objH analyses are definitely more precise than both LSRVH and 1obj1h analyses, LSRV2objH analyses are more precise than LSRV1objH as well as 2obj2h (hence 2obj1h) analyses, and so on.

Note that Figure 5.4 is not complete. We can construct a few more relationships as an extension of Figure 5.4. For example, we may have a variant LSRV1obj (without heap cloning), which will be more precise than both LSRV and 1obj. In Section 5.5, we show that in practice, a control-flow analysis based on LSRV1objH contexts, which is more precise than both LSRVH and 1obj1h analyses, scales well for all the benchmarks under consideration. It should be possible to further scale LSRV$k$objH analyses for higher values of k, for different heap analyses, perhaps based on analysis-specific heuristics if required; we leave the same as future work.

5.5 Implementation and Evaluation of LSRV$k$objH

We have implemented our techniques for control-flow analysis of Java programs, in the Soot framework (Vallée-Rai *et al.*, 1999) version 2.5.0. This is a parameterized implementation that takes a command-line argument to choose between LSRVH contexts and LSRV$k$objH versions. We have also implemented k-object-sensitive control-flow analysis (based on *full object-sensitivity* as defined by Smaragdakis *et al.* (2011)) in the Soot framework. For comparison, we have instantiated both the LSRV$k$objH and k-object-sensitive versions with $k = 1$. Thus, in this section, we compare three versions of control-flow analysis: LSRVH, 1obj1h, and LSRV1objH. We evaluate all the versions on the same ten benchmarks from the DaCapo and the JGF suites that were used in Chapter 4; see Figure 4.2 for some of their static characteristics.

5.5.1 Scalability of LSRV$k$objH

We now evaluate the scalability of our proposed approaches in terms of the analysis time, the average number of contexts, and the peak memory requirement; Table 5.1 shows the same for LSRVH (written as lsrvh), 1obj1h (written as 1o1h), and LSRV1objH (written as lsrv1o1h).

Columns 2-4 of Table 5.1 show the analysis time (in seconds) of the three approaches. Compared to the time taken by plain LSRV-contexts (on average, 130 seconds; see Section 4.5), we can see that the time taken by LSRVH (that is, LSRV-contexts with heap cloning) is higher: 224 seconds on average. However, 1o1h takes a significantly higher time than LSRVH and does not terminate in 3 hours for the large benchmark ECLIPSE. On the contrary, LSRV1objH, though more precise than both LSRVH and 1obj, takes only 24.3% higher time than LSRVH, and terminates in less than an hour for all the benchmarks under consideration.

The reason for the better scalability of the LSRV approaches can be attributed to the difference in the average number of contexts created per method; see Table 5.1, columns 5-7. We can observe that 1obj1h creates a large number of contexts compared to LSRVH and LSRV1objH. The savings are a result of our pre-analysis, which leads to

1	2	3	4	5	6	7	8	9	10
Benchmark	**Analysis time (seconds)**			**Average contexts**			**Memory (GB)**		
	lsrvh	**1o1h**	**lsrv1oh**	**lsrvh**	**1o1h**	**lsrv1oh**	**lsrvh**	**1o1h**	**lsrv1oh**
avrora	69	410	70	1.2	8.3	1.3	13.5	15.2	13.7
batik	2529	7661	2168	1.4	10.2	1.6	163.6	98.3	117.2
eclipse	3201	-	3220	1.5	-	1.6	134.4	-	-
luindex	64	347	57	1.2	9.2	1.3	13.0	15.2	12.1
lusearch	82	440	92	1.2	9.3	1.3	17.7	22.6	18.3
moldyn	98	291	97	1.3	8.2	1.4	15.9	15.2	16.1
montecarlo	84	304	89	1.2	8.3	1.4	13.5	22.4	13.5
pmd	156	742	2552	1.3	8.1	2.3	23.5	22.9	77.0
raytracer	86	293	89	1.2	8.2	1.4	13.4	15.0	13.4
sunflow	963	8188	1048	1.4	9.6	1.5	77.1	95.1	101.3
geomean	224	-	296	1.3	-	1.5	28.5	-	-

Table 5.1: Evaluation results with heap cloning for control-flow analysis.

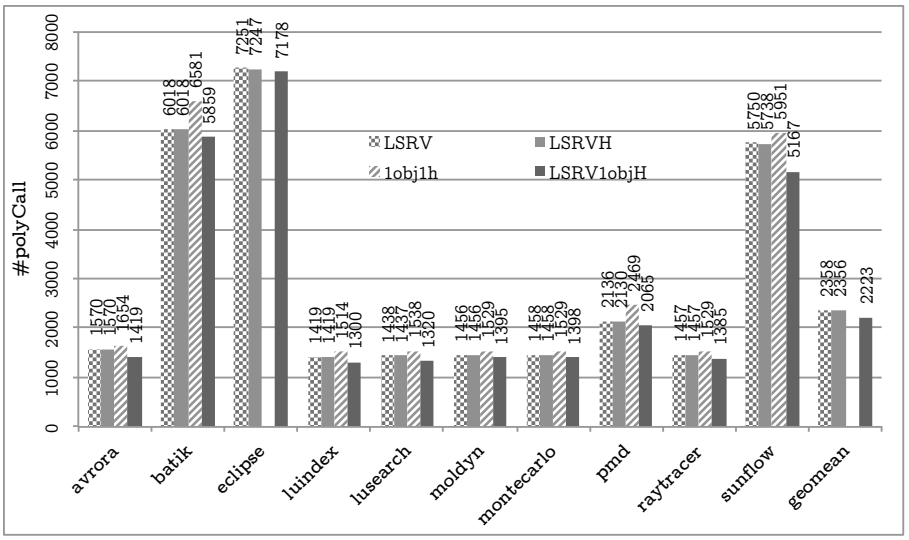
the identification of cases where methods need not be analyzed redundantly in multiple contexts. Note that some of our techniques, such as not (re-)analyzing a method redundantly if the access-depth for its receiver is zero, can be applied to scale object-sensitive analyses as well; we leave it as a future exercise.

Columns 8-10 of Table 5.1 show the peak memory consumption (computed using the `time -v` command) for the three analyses. For small benchmarks, we observe that the memory requirements of all the approaches remain similar (about 16-32 GB). For larger benchmarks (BATIK, ECLIPSE and SUNFLOW), the memory requirements reach 100-200 GB. For ECLIPSE, though the LSRV1objH approach terminated (see column 4), it timed out when we enabled the memory measurement.

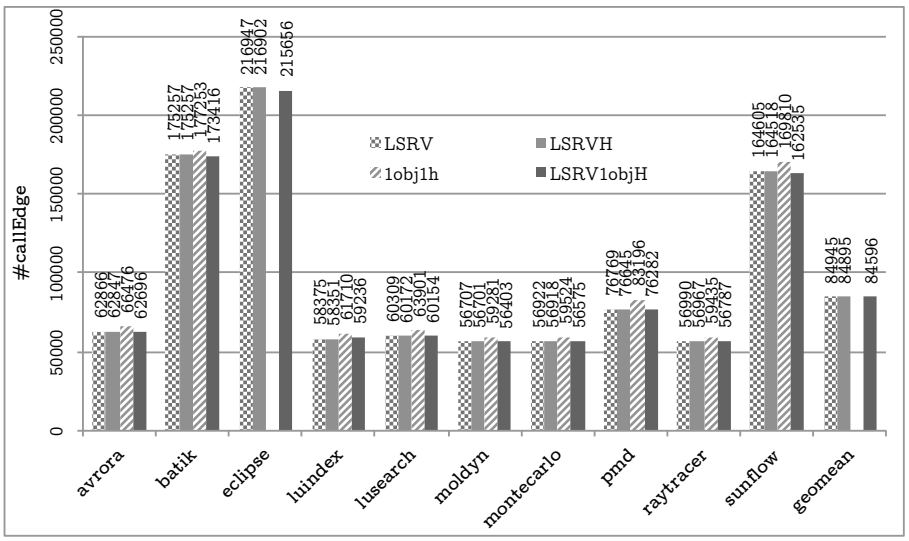
Overall, we can see that LSRVH1objH remains a practical choice and scales well within our cutoff. In the next section, we compare the precision of our approaches to find out how does the scalable LSRV1objH approach fare compared to the rest.

5.5.2 Precision of LSRV$k$objH

For measuring precision, similar to existing works (Lhoták and Hendren, 2008; Smaragdakis *et al.*, 2011; Li *et al.*, 2018*b*,*a*), we normalize over all contexts two popular precision-indicating clients: (i) *#polyCall*: the number of call-sites that could not be resolved to a single method (Figure 5.5a); and (ii) *#callEdge*: the number of edges in



(a) Number of polymorphic calls (lower the better).



(b) Number of edges in the call-graph.

Figure 5.5: Normalized precision of various context abstractions (lower the better).

the on-the-fly call-graph (Figure 5.5b). For both the metrics, a smaller value is better. To study the impact of heap cloning on LSRV-contexts, apart from the three heap-cloning enabled versions (that is, LSRVH, 1obj1h, and LSRV1objH), we also show the corresponding numbers for plain LSRV-contexts.

In Figure 5.5a, we can see that the number of polymorphic calls reduces, albeit marginally, from LSRV-contexts to LSRVH. The number of polymorphic calls in case of 1obj1h is the highest among the four reported versions, which indicates that overall, the number of optimization opportunities enabled (over all contexts) by the LSRV approaches is better than that by 1obj1h. For all the benchmarks, we can see that

LSRV1objH leads to the smallest number of polymorphic calls (5.73% less compared to LSRV-contexts, 5.65% less compared to LSRVH, and 12.23% less compared to 1obj1h for cases where 1obj1h terminates), which establishes it as a superior alternative over existing context-sensitive analyses.

In Figure 5.5b, we see that similar to the number of polymorphic calls, the normalized number of call-graph edges reduces, though by a small amount, from LSRV-contexts to LSRVH. The number of call-graph edges is the highest for 1obj1h, and the reduction is the highest for LSRV1objH.

It is well-known (Li *et al.*, 2018*b*,*a*) that the next more precise object-sensitive analysis used in practice, that is 2obj1h, does not directly scale for many large benchmarks. Further, as indicative by the relatively high time taken by LSRV1objH for the large benchmarks, we believe that with the current choice of efficiency heuristics, the LSRV1objH approach presents the best tradeoff between scalability and precision, among the approaches under consideration. It would be an interesting future direction to explore ways to scale more intricate combinations of k-object-sensitivity and LSRV-contexts, for higher values of k.

Overall, we observe that the precision of the LSRV1objH approach, as expected from its theoretical model (see Section 5.4), generates more optimization opportunities than both LSRV and object-sensitive approaches. Combining this fact with the scalability noted in Section 5.5.1, the proposed LSRV$k$objH context abstraction can be viewed as a new sweet-spot that combines the benefits of LSRV-contexts (and hence classical call-strings) and object-sensitivity.

CHAPTER 6

RELATED WORK

Points-to analysis for Java is a very widely studied discipline and there have been a plethora of works on improving its precision and/or efficiency. We divide the discussion on the related work into five parts: (i) staged analysis; (ii) modular analysis; (iii) points-to analysis for null-check elimination; (iv) escape analysis for synchronization elimination; and (v) context-sensitive analyses.

6.1 Staged Analysis

There have been prior works (Serrano *et al.*, 2000; Ali, 2014) that help perform costly whole-program analyses/optimizations statically for Java. Serrano *et al.* (2000) propose an interesting compilation scheme, in which the application (considered along with the statically available libraries) is statically compiled to a platform-specific optimized binary. This may involve dynamic compilation to support dynamic Java features (such as different runtime libraries). Averroes (Ali, 2014) helps perform whole-program analyses statically, by generating “placeholder" libraries that conservatively approximate the behavior of the actual runtime libraries.

Many prior works have tried to reduce the overheads at runtime (but not during JIT compilation) by taking advantage of the multi-stage nature of Java compilation/execution model. For example, Sreedhar *et al.* (2000) use code specialization to generate multiple versions of code statically, where each version may be optimized differently (not all may be “semantics preserving", or valid). Based on the runtime conditions, one of the valid versions of the code is invoked during execution. Similarly, Guyer *et al.* (2006) annotate the input code with explicit “free" instructions (executed at runtime) based on the liveness information of the heap objects. In contrast, PYE performs expensive analyses on applications statically to obtain results that are conditional on

the specific libraries on the target machine; these partial results are combined with the partial results of the libraries, at runtime, to achieve precision and performance.

Chambers (2002) and Philipose *et al.* (2002) propose a staged compilation scheme in which the generation of native code for different platforms is spread across the different stages of compilation and linking, for C programs. In contrast, PYE is designed for analyzing programs written in languages like Java/C# that follow a two-step process of compilation (static + just-in-time), where the libraries can only be obtained at runtime.

Sharma *et al.* (1998) propose *deferred* data-flow analysis (DDFA) to address the problem of the conservative nature of static analyses. DDFA performs most of the analysis at compile-time and uses control-flow information at runtime to improve the precision of the analysis. In PYE, our focus is on handling the dependence between the application and the library methods. It would be interesting to perform DDFA analyses in the PYE framework.

Our idea of conditional values is partially inspired by the idea of three-valued logic analysis (Sagiv *et al.*, 2002). In PYE, we associate conditions with the indeterminate third values, which are resolved during JIT compilation. This helps us achieve precision without much overhead at runtime.

6.2 Modular Analysis

Modular analysis, as proposed by Cousot and Cousot (2002), is a well-explored technique to scale interprocedural analyses by analyzing different modules (or methods, in Java context) separately, and composing the modular results to obtain whole-program analysis results (Whaley and Rinard, 1999; Vivien and Rinard, 2001; Choi *et al.*, 1999; Calcagno *et al.*, 2011; Sălcianu and Rinard, 2005). In a recent survey, Madhavan *et al.* (2015) evaluate several of these existing techniques in a well-formalized framework.

In the context of Java programs, Whaley and Rinard (1999) compute summaries for methods in the form of points-to escape graphs, and compose them at interprocedural boundaries. Later, Vivien and Rinard (2001) incrementalize the analysis to analyze only those parts of a program that may deliver useful results. The analysis performed by

our proposed framework PYE is also modular, but we do not have the library methods while analyzing the application, and vice-versa. Consequently, our generated summaries contain conditional values. In PACE and EASE, we have borrowed the idea of creating outside nodes from Whaley and Rinard (1999), and added conditional values to the outside nodes to represent the dependence of unavailable object-dereferences on the unanalyzed parts of a program. Further, we use the mapping algorithm presented by Whaley and Rinard (1999) to merge the points-to graphs of analyzed methods at interprocedural boundaries.

It is worth noting that the way PYE performs whole-program analyses is a bit different from both standard call-string based analyses as well as functional analyses (Sharir and Pnueli, 1978). Top-down analyses analyze the whole program without skipping any method calls, whereas bottom-up analyses analyze each method only once and then merge the obtained summary with its callers at each call site. In PYE, even though we construct summaries for application and library methods in a top-down manner (Padhye and Khedker, 2013), the effects (in the form of the final analysis-results obtained during JIT compilation) of merging these summaries at the application and library boundaries is similar to what is done in bottom-up analyses (Bodden, 2018; Gharat, 2018).

There have been works that analyze methods independent of the caller information and generate multiple summaries along with the conditions under which a particular summary may be used. For example, Wilson and Lam (1995) generate *partial transfer functions* for each method m, and a given transfer function for m is selected based on the alias relationship that exists among the actual arguments at a call-site of m. Similarly, Yu *et al.* (2010) perform a whole-program analysis in two phases: a bottom-up pass where conditions are generated for the points-to relations among the (unavailable) actual arguments, and a subsequent top-down pass that completes the points-to results by resolving the conditions and completing the points-to relations with actual arguments. On the other hand, the partial-analyzer of PYE generates conditions at application and library boundaries, propagates the conditions and extends them at subsequent statements, and forms partial results for all the elements of a program (interprocedurally). These conditions are resolved in the fast-precise-analyzer by looking up the results of the statically unavailable methods.

There have been prior works that model the dependencies between the available and unavailable methods while performing static analyses. WALA (2019) models native methods flow-insensitively and merges their summaries with those of their callers. StubDroid (Arzt and Bodden, 2016) summarizes Android libraries for taint analysis by storing conditions on the "taint value" of actual arguments. In contrast, the conditional values proposed in PYE are bidirectional (that is, from application to library methods and vice-versa), and the statically generated partial summaries are resolved during JIT compilation to obtain precise analysis-results in the JVM.

6.3 Points-to Analysis for Null-Check Elimination

There have been works that perform points-to analysis to statically identify unnecessary null-dereference checks (Loginov *et al.*, 2008; Nanda and Sinha, 2009). Loginov *et al.* (2008) perform a static points-to analysis and annotate the statements that are guaranteed to dereference a concrete object. They handle those library calls precisely whose specifications guarantee that the corresponding methods return a non-null object, and treat others conservatively. Nanda and Sinha (2009) perform a path-sensitive points-to analysis statically to mark the dereferences guaranteed to be made on a concrete object. They treat the library methods as unavailable for analysis, and handle the library calls conservatively. Contrary to both these works, PACE is a two-step analysis that neither assumes the specification of library methods, nor handles library calls conservatively. Instead, PACE analyzes the application and the library methods separately, while encoding the dependence between them in the generated partial summaries as conditional values. These dependencies are resolved during JIT compilation to obtain precise analysis-results.

In order to balance the time spent in JIT compilation, the HotSpot Server Compiler (C2) of the HotSpot JVM (Paleczny *et al.*, 2001) performs an intraprocedural points-to analysis to avoid inserting null-checks that are not required. As shown in Section 3.5, PACE offers a much-enhanced precision (on average 23.67% fewer checks than the existing analyzer) in almost the same amount of time (as C2) during JIT compilation.

6.4 Escape Analysis for Synchronization Elimination

There have been many prior research works (Salcianu and Rinard, 2001; Choi *et al.*, 1999; Whaley and Rinard, 1999) that perform precise escape analysis for Java programs. However, they are completely performed either during static compilation (Choi *et al.*, 1999) (make conservative assumptions about libraries), or during JIT compilation (Whaley and Rinard, 1999; Salcianu and Rinard, 2001) (not scalable). Kotzmann and Mössenböck (2005) present an imprecise but fast escape analysis for the HotSpot Client Compiler (C1). In contrast, our proposed analysis EASE generates precise escape-analysis results during JIT compilation in the HotSpot Server Compiler (C2), at speeds comparable to that of the baseline partially interprocedural and partially flow-sensitive escape analysis.

Lee and Midkiff (2006) propose an insightful two-phase escape analysis for the Jikes RVM (Alpern *et al.*, 2005). They compute connection graphs (a representation similar to points-to graphs) for different methods offline, and merge them to complete an interprocedural analysis during JIT compilation. On the contrary, EASE generates precise escape-analysis results at runtime, by resolving the statically generated partial summaries, and has the following advantages: (i) By maintaining the set `SYN` separately, EASE can preserve flow-sensitivity for synchronization elimination (Lee and Midkiff store only per-method connection-graphs and lose flow-sensitivity). (ii) The overhead of the fast-precise-analyzer of EASE at runtime is very less, as it does not perform any actual escape analysis (Lee and Midkiff may have to revisit/modify the connection-graph multiple times). (iii) The storage overheads of the result files for EASE are quite less: average 3.96% over the size of the class files (Lee and Midkiff report 68% overhead).

Stadler *et al.* (2014) proposed *partial* escape analysis, which elides locks only on those branches in which the associated object does not escape, in the Graal (2019) compiler. For instance, before entering a synchronization statement, a data-structure is looked up to check if the associated object has not escaped; and if so, the lock operation is not performed. This is a promising approach for doing escape analysis, and we believe that its efficiency can be further improved by implementing it in PYE.

6.5 Context-sensitive Analysis

There have been several works that propose ways to perform context-sensitive analyses in a scalable manner; we can broadly classify them into two categories: (i) efficiently maintaining the identified contexts (Whaley and Lam, 2004; Xu and Rountev, 2008; Thiessen and Lhoták, 2017); and (ii) defining new context abstractions (Khedker and Karkare, 2008; Smaragdakis *et al.*, 2011; Milanova *et al.*, 2005; Tan *et al.*, 2017).

Whaley and Lam (2004) proposed the use of binary decision diagrams (BDDs) to represent equivalent contexts with lesser memory. Xu and Rountev (2008) explore the non-scalability of BDDs for context-sensitive analyses involving heap-cloning, and merge the calling contexts with similar points-to relationships. However, their presented analysis is imprecise as it lacks flow-sensitivity. Thiessen and Lhoták (2017) present a novel way to scale k-length call-string based analyses by representing points-to information in terms of the input-output values of different contexts, which allows merging of equivalent call-strings. In contrast, in this thesis, we have proposed ways to scale the value-contexts based approach by compacting (level-summarizing) and improving the precision of the process of finding equivalent contexts.

Milanova *et al.* (2005) propose a new context abstraction called k-object-sensitivity, which distinguishes the contexts based on the allocation site of the receiver. Smaragdakis *et al.* (2011) clarify the definition of object-sensitivity for $k > 1$, and propose type-sensitivity as another close sibling. In recent attempts to scale these abstractions, Tan *et al.* (2017) merge type-consistent objects for type-dependent analyses, and Li *et al.* (2018*b*) select among the different variants of object- and type-sensitivity for each method (with a small dip in the precision). On the contrary, we proposed LSRV-contexts: a way to scale call-site-sensitivity based analyses (while maintaining their precision), which, though incomparable, generate similar (higher, with heap cloning) number of optimization opportunities as object-sensitive analyses.

There have been works that scale the main-analysis using a pre-analysis. Oh *et al.* (2014) perform a pre-analysis that estimates the impact of context-sensitivity on different methods for a given set of queries (for C programs), and then reduce the precision of the main-analysis on methods that might not benefit from the enhanced precision. Tan et

al. (Tan *et al.*, 2016) use a pre-analysis to identify and eliminate redundant objects from object-sensitive analyses (for Java programs) to reduce the number of effective contexts. Recently, Karkare (2018) first uses a fast analysis to mark variables whose shape cannot be refined, and skips them in a following precise (slow) pass. Prior works (Smaragdakis *et al.*, 2013, 2014) use a pre-analysis to identify code portions that do not affect the analysis results or may degrade scalability, and analyze them conservatively. In contrast, our proposed pre-analysis identifies the relevant portions of the caller's heap, whose results are then used to scale whole-program context-sensitive analyses for Java programs. We also use the pre-analysis to identify caller-ignorable methods that are deferred by the main-analysis and analyzed as a post-pass without any loss of precision.

The scalability and precision of points-to analyses, and context-sensitive analyses in particular, depend significantly on the abstraction used to denote objects allocated on the heap. In a recent survey, Kanvar and Khedker (2016) classify and formalize several heap abstractions and discuss their relative advantages. In this thesis, we have used the allocation-site abstraction (qualified with its context in Chapter 5) for all the analyses. The techniques proposed could be applied even in presence of other heap abstractions, with corresponding changes in the underlying data structures.

Whole-program escape analysis and control-flow analysis are two very important heap analyses with wide applicability, and there have been works to improve their scalability on large programs. Kotzmann and Mössenböck (2005) propose a fast but imprecise unification-based escape analysis for the HotSpot client compiler (Kotzmann *et al.*, 2008). Padhye and Khedker (2013) present a value-contexts based precise control-flow analysis; however it does not scale well on large programs. Our proposed LSRV-contexts help perform value-contexts based escape and control-flow analyses that are not only precise, but also scale very well to large programs. To the best of our knowledge, ours is the first work that scales these heap analyses while realizing the precision of unbounded call-strings, especially using the practical value-contexts approach.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This thesis proposed several ways to advance the state-of-the-art in Java program analysis in terms of precision as well as efficiency, for both the just-in-time (JIT) compiler of the JVM, and the static Java compiler.

Towards improving the precision and efficiency aspects of the JIT compiler, the thesis proposed a two-step analysis framework called PYE that helps generate highly precise analysis-results during JIT compilation, at a low cost. PYE is based on the idea of generating partial summaries at compile-time, which encode the dependence on the missing libraries in a concise manner, in the form of conditional values. The effectiveness of PYE was shown by using it to design two precise analyses – points-to analysis for null-check elimination (PACE) and escape analysis for synchronization elimination (EASE). Over a wide range of benchmarks, PACE and EASE generate more precise results compared to the existing analyzers of the HotSpot Server Compiler (C2), with negligible overhead (in fact, saving time in case of EASE) during JIT compilation. The evaluation attests PYE to be an effective and practical framework for implementing complicated whole-program analyses and their related optimizations.

Towards improving the scalability of precise static analyses, the thesis proposed a three-stage analysis approach to scale complex whole-program value-contexts based heap analyses for large programs. The approach is based on the novel idea of LSRV-contexts, which take into account an important observation that one need not compare the complete value-contexts at each call-site. LSRV-contexts helped reduce the comparison performed for determining context-equality and classify more value-contexts as equivalent. The precision of LSRV-contexts was further improved by first cloning the heap, and then imparting to them the benefits of object-sensitivity. These approaches were evaluated on two nontrivial heap analyses. The results showed that the proposed approaches not only reduced the analysis time and memory consumption significantly, but also helped analyze previously unanalyzable large programs in a reasonable time. In

addition, the exercise of combining the benefits of LSRV-contexts and object-sensitivity with heap cloning led to a novel connection between the corresponding analysis lattices.

We observe that the techniques proposed in this thesis to improve the scalability of precise context-sensitive analyses can in turn be used as part of the PYE framework to improve the precision of the performed analyses further (for example, we have already used level-summarization in PACE and EASE). Also, even though this thesis presents new techniques for analyzing Java programs, the techniques can be extended to other similar object-oriented languages, such as C#, that support multi-staged analysis.

Future work

It would be quite interesting to implement precise versions of other popular analyses (such as points-to analysis for call-graph construction, may-happen-in-parallel analysis, and so on) in PYE, and study the impact on the precision of the results obtained, possibly for other static+JIT-compiled languages (such as C#). Likewise, it would be an interesting exercise to take the PYE model – of splitting the analysis across the different compilers – as a built-in setting for some of the Java Virtual Machines.

Another interesting idea would be to use the advantages of PYE to improve the precision of less-complicated compilers such as C1 of the HotSpot JVM (and perhaps even the interpreter), without compromising their goal of fast translation. It is also noteworthy that owing to the efficiency imparted by PYE to JIT compilation, more contemporary precise (but time-consuming) static analysis techniques can now be brought to production JIT compilers.

On the static analysis front, it would be interesting to implement more analyses using LSRV-contexts, and improve the efficiency of other context abstractions using the ideas of relevance and summarization. Another promising direction would be to discover more (analysis-specific or otherwise) ways to scale static analyses by doing exactly that amount of work which is required. Motivated by the observations in this thesis, staging the analyses looks like a promising way to go ahead.

REFERENCES

1. **Ali, K.** (2014). *The Separate Compilation Assumption*. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada.

2. **Alpern, B.**, **S. Augart**, **S. M. Blackburn**, **M. Butrico**, **A. Cocchi**, **P. Cheng**, **J. Dolby**, **S. Fink**, **D. Grove**, **M. Hind**, **K. S. McKinley**, **M. Mergen**, **J. E. B. Moss**, **T. Ngo**, and **V. Sarkar** (2005). The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.*, **44**(2), 399–417. ISSN 0018-8670.

3. **Andersen, L. O.** (1994). *Program Analysis and Specialization for the C Programming Language*. Ph.D. thesis, Cornell University.

4. **Arnold, K.**, **D. Holmes**, and **J. Gosling**, *The Java Programming Language, Fourth Edition*. Addison-Wesley Professional, USA, 2005.

5. **Arzt, S.** and **E. Bodden**, StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. *In Proceedings of the International Conference on Software Engineering (ICSE)*. 2016. ISSN 1558-1225.

6. **Blackburn, S. M.**, **R. Garner**, **C. Hoffmann**, **A. M. Khang**, **K. S. McKinley**, **R. Bentzur**, **A. Diwan**, **D. Feinberg**, **D. Frampton**, **S. Z. Guyer**, **M. Hirzel**, **A. Hosking**, **M. Jump**, **H. Lee**, **J. E. B. Moss**, **A. Phansalkar**, **D. Stefanović**, **T. VanDrunen**, **D. von Dincklage**, and **B. Wiedermann**, The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *In Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06. ACM, New York, NY, USA, 2006.

7. **Blanchet, B.** (2003). Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.*, **25**(6), 713–775. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/945885.945886`.

8. **Bodden, E.**, The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distributive, Summary-based Static Analyses (and How to Master Them). *In Companion Proceedings for the ISSTA/ECOOP Workshops*, ISSTA '18. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5939-9. URL `http://doi.acm.org/10.1145/3236454.3236500`.

9. **Bodden, E.**, **A. Sewe**, **J. Sinschek**, **H. Oueslati**, and **M. Mezini**, Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. *In Proceedings of the International Conference on Software Engineering*, ICSE '11. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0445-0. URL `http://doi.acm.org/10.1145/1985793.1985827`.

10. **Bogda, J.** and **U. Hölzle**, Removing Unnecessary Synchronization in Java. *In Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99. ACM, New York, NY, USA, 1999. ISBN 1-58113-238-7. URL `http://doi.acm.org/10.1145/320384.320388`.

11. **Calcagno, C.**, **D. Distefano**, **P. W. O'Hearn**, and **H. Yang** (2011). Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, **58**(6), 26:1–26:66. ISSN 0004-5411. URL http://doi.acm.org/10.1145/2049697.2049700.

12. **Chambers, C.**, Staged Compilation. *In Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02. ACM, New York, NY, USA, 2002. ISBN 1-58113-455-X. URL http://doi.acm.org/10.1145/503032.503045.

13. **Choi, J.-D.**, **M. Gupta**, **M. Serrano**, **V. C. Sreedhar**, and **S. Midkiff**, Escape Analysis for Java. *In Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99. ACM, New York, NY, USA, 1999. ISBN 1-58113-238-7. URL http://doi.acm.org/10.1145/320384.320386.

14. **Choi, J.-D.**, **K. Lee**, **A. Loginov**, **R. O'Callahan**, **V. Sarkar**, and **M. Sridharan**, Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '02. ACM, New York, NY, USA, 2002. ISBN 1-58113-463-0. URL http://doi.acm.org/10.1145/512529.512560.

15. **Cooper, K. D.**, **K. Kennedy**, and **L. Torczon**, Interprocedural Optimization: Eliminating Unnecessary Recompilation. *In Proceedings of the SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86. ACM, New York, NY, USA, 1986. ISBN 0-89791-197-0. URL http://doi.acm.org/10.1145/12276.13317.

16. **Cousot, P.** and **R. Cousot**, Modular Static Program Analysis. *In Proceedings of the International Conference on Compiler Construction*, CC '02. Springer-Verlag, London, UK, UK, 2002. ISBN 3-540-43369-4. URL http://dl.acm.org/citation.cfm?id=647478.727794.

17. **Daly, C.**, **J. Horgan**, **J. Power**, and **J. Waldron**, Platform Independent Dynamic Java Virtual Machine Analysis: The Java Grande Forum Benchmark Suite. *In Proceedings of the Joint ACM-ISCOPE Conference on Java Grande*, JGI '01. ACM, New York, NY, USA, 2001. ISBN 1-58113-359-6. URL http://doi.acm.org/10.1145/376656.376826.

18. **Dietrich, J.**, **N. Hollingum**, and **B. Scholz**, Giga-scale Exhaustive Points-to Analysis for Java in Under a Minute. *In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3689-5. URL http://doi.acm.org/10.1145/2814270.2814307.

19. **Domani, T.**, **G. Goldshtein**, **E. K. Kolodner**, **E. Lewis**, **E. Petrank**, and **D. Sheinwald**, Thread-local Heaps for Java. *In Proceedings of the International Symposium on Memory Management*, ISMM '02. ACM, New York, NY, USA, 2002. ISBN 1-58113-539-4. URL http://doi.acm.org/10.1145/512429.512439.

20. **Georges, A.**, **D. Buytaert**, and **L. Eeckhout**, Statistically rigorous Java performance evaluation. *In Proceedings of the ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07. ACM, New York, NY, USA, 2007.

ISBN 978-1-59593-786-5. URL http://doi.acm.org/10.1145/1297027.1297033.

21. **Gharat, P.** (2018). *Generalized Points-to Graph: A New Abstraction of Memory in Presence of Pointers*. Ph.D. thesis, Indian Institute of Technology Bombay, Mumbai, India.

22. **Graal** (2019). OpenJDK Graal. http://openjdk.java.net/projects/graal/.

23. **Grove, D.** and **C. Chambers** (2001). A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, **23**(6), 685–746. ISSN 0164-0925. URL http://doi.acm.org/10.1145/506315.506316.

24. **Guyer, S. Z.**, **K. S. McKinley**, and **D. Frampton**, Free-Me: A Static Analysis for Automatic Individual Object Reclamation. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06. ACM, New York, NY, USA, 2006. ISBN 1-59593-320-4. URL http://doi.acm.org/10.1145/1133981.1134024.

25. **Hardekopf, B.** and **C. Lin**, The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-633-2. URL http://doi.acm.org/10.1145/1250734.1250767.

26. **Kanvar, V.** and **U. P. Khedker** (2016). Heap Abstractions for Static Analysis. *ACM Comput. Surv.*, **49**(2), 29:1–29:47. ISSN 0360-0300. URL http://doi.acm.org/10.1145/2931098.

27. **Karkare, A.**, TwAS: Two-stage Shape Analysis for Speed and Precision. *In Proceedings of the Annual ACM Symposium on Applied Computing*, SAC '18. ACM, New York, NY, USA, 2018. ISBN 978-1-4503-5191-1. URL http://doi.acm.org/10.1145/3167132.3167330.

28. **Khedker, U. P.** and **B. Karkare**, Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. *In Proceedings of the International Conference on Compiler Construction*, CC'08. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 3-540-78790-9, 978-3-540-78790-7. URL http://dl.acm.org/citation.cfm?id=1788374.1788394.

29. **Kotzmann, T.** and **H. Mössenböck**, Escape Analysis in the Context of Dynamic Compilation and Deoptimization. *In Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05. ACM, New York, NY, USA, 2005. ISBN 1-59593-047-7. URL http://doi.acm.org/10.1145/1064979.1064996.

30. **Kotzmann, T.**, **C. Wimmer**, **H. Mössenböck**, **T. Rodriguez**, **K. Russell**, and **D. Cox** (2008). Design of the Java HotSpot Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, **5**(1), 7:1–7:32. ISSN 1544-3566. URL http://doi.acm.org/10.1145/1369396.1370017.

31. **Lee, K.** and **S. P. Midkiff**, A Two-phase Escape Analysis for Parallel Java Programs. *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '06. ACM, New York, NY, USA, 2006. ISBN 1-59593-264-X. URL http://doi.acm.org/10.1145/1152154.1152166.

32. **Lhoták, O.** and **L. Hendren**, Scaling Java Points-to Analysis Using SPARK. *In Proceedings of the International Conference on Compiler Construction*, CC'03. Springer-Verlag, Berlin, Heidelberg, 2003. ISBN 3-540-00904-3. URL http://dl.acm.org/citation.cfm?id=1765931.1765948.

33. **Lhoták, O.** and **L. Hendren**, Context-Sensitive Points-to Analysis: Is It Worth It? *In Proceedings of the International Conference on Compiler Construction*, CC'06. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 3-540-33050-X, 978-3-540-33050-9. URL http://dx.doi.org/10.1007/11688839_5.

34. **Lhoták, O.** and **L. Hendren** (2008). Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.*, **18**(1), 3:1–3:53. ISSN 1049-331X. URL http://doi.acm.org/10.1145/1391984.1391987.

35. **Li, Y.**, **T. Tan**, **A. Møller**, and **Y. Smaragdakis** (2018*a*). Precision-guided Context Sensitivity for Pointer Analysis. *Proc. ACM Program. Lang.*, **2**(OOPSLA), 141:1–141:29. ISSN 2475-1421. URL http://doi.acm.org/10.1145/3276511.

36. **Li, Y.**, **T. Tan**, **A. Møller**, and **Y. Smaragdakis**, Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. *In Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018. ACM, New York, NY, USA, 2018*b*. ISBN 978-1-4503-5573-5. URL http://doi.acm.org/10.1145/3236024.3236041.

37. **Loginov, A.**, **E. Yahav**, **S. Chandra**, **S. Fink**, **N. Rinetzky**, and **M. Nanda**, Verifying Dereference Safety via Expanding-scope Analysis. *In Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-050-0. URL http://doi.acm.org/10.1145/1390630.1390657.

38. **Madhavan, R.**, **G. Ramalingam**, and **K. Vaswani** (2015). A Framework For Efficient Modular Heap Analysis. *Found. Trends Program. Lang.*, **1**(4), 269–381. ISSN 2325-1107. URL http://dx.doi.org/10.1561/2500000020.

39. **Milanova, A.**, **A. Rountev**, and **B. G. Ryder** (2005). Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, **14**(1), 1–41. ISSN 1049-331X. URL http://doi.acm.org/10.1145/1044834.1044835.

40. **Muchnick, S. S.**, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

41. **Nanda, M. G.** and **S. Sinha**, Accurate Interprocedural Null-Dereference Analysis for Java. *In Proceedings of the International Conference on Software Engineering*, ICSE '09. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-1-4244-3453-4. URL http://dx.doi.org/10.1109/ICSE.2009.5070515.

42. **Naumovich, G.**, **G. S. Avrunin**, and **L. A. Clarke**, An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. *In Proceedings of the European Software Engineering Conference*, ESEC. Springer-Verlag, London, UK, UK, 1999. ISBN 3-540-66538-2. URL http://dl.acm.org/citation.cfm?id=318773.319252.

43. **Nystrom, E. M.**, **H.-S. Kim**, and **W.-m. W. Hwu**, Importance of Heap Specialization in Pointer Analysis. *In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04. ACM, New York, NY, USA, 2004. ISBN 1-58113-910-1. URL http://doi.acm.org/10.1145/996821.996836.

44. **Oh, H.**, **W. Lee**, **K. Heo**, **H. Yang**, and **K. Yi**, Selective Context-sensitivity Guided by Impact Pre-analysis. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2784-8. URL http://doi.acm.org/10.1145/2594291.2594318.

45. **Padhye, R.** and **U. P. Khedker**, Interprocedural Data Flow Analysis in Soot Using Value Contexts. *In Proceedings of the ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2201-0. URL http://doi.acm.org/10.1145/2487568.2487569.

46. **Paleczny, M.**, **C. Vick**, and **C. Click**, The Java HotSpot^{TM} Server Compiler. *In Proceedings of the Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01. USENIX Association, Berkeley, CA, USA, 2001. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

47. **Palsberg, J.** and **M. I. Schwartzbach**, Object-Oriented Type Inference. *In Proceedings of Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91. ACM, New York, NY, USA, 1991. ISBN 0-201-55417-8. URL http://doi.acm.org/10.1145/117954.117965.

48. **Philipose, M.**, **C. Chambers**, and **S. J. Eggers**, Towards Automatic Construction of Staged Compilers. *In Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02. ACM, New York, NY, USA, 2002. ISBN 1-58113-450-9. URL http://doi.acm.org/10.1145/503272.503284.

49. **Ruf, E.**, Effective Synchronization Removal for Java. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '00. ACM, New York, NY, USA, 2000. ISBN 1-58113-199-2. URL http://doi.acm.org/10.1145/349299.349327.

50. **Sagiv, M.**, **T. Reps**, and **R. Wilhelm** (2002). Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, **24**(3), 217–298.

51. **Salcianu, A.** and **M. Rinard**, Pointer and Escape Analysis for Multithreaded Programs. *In Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01. ACM, New York, NY, USA, 2001. ISBN 1-58113-346-4. URL http://doi.acm.org/10.1145/379539.379553.

52. **Serrano, M.**, **R. Bordawekar**, **S. Midkiff**, and **M. Gupta**, Quicksilver: A Quasi-static Compiler for Java. *In Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00. ACM, New York, NY, USA, 2000. ISBN 1-58113-200-X. URL http://doi.acm.org/10.1145/353171.353176.

53. **Shapiro, M.** and **S. Horwitz**, The Effects of the Precision of Pointer Analysis. *In* **P. Van Hentenryck** (ed.), *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-540-69576-9.

54. **Sharir, M.** and **A. Pnueli**, *Two Approaches to Interprocedural Data Flow Analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978. URL https://cds.cern.ch/record/120118.

55. **Sharma, S. D.**, **A. Acharya**, and **J. Saltz** (1998). Deferred Data-Flow Analysis. Technical report, University of Maryland.

56. **Shivers, O. G.** (1991). *Control-flow Analysis of Higher-order Languages or Taming Lambda*. Ph.D. thesis, Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

57. **Smaragdakis, Y.**, **G. Balatsouras**, and **G. Kastrinis**, Set-based Pre-processing for Points-to Analysis. *In Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2374-1. URL http://doi.acm.org/10.1145/2509136.2509524.

58. **Smaragdakis, Y.**, **M. Bravenboer**, and **O. Lhoták**, Pick Your Contexts Well: Understanding Object-sensitivity. *In Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0490-0. URL http://doi.acm.org/10.1145/1926385.1926390.

59. **Smaragdakis, Y.**, **G. Kastrinis**, and **G. Balatsouras**, Introspective Analysis: Context-sensitivity, Across the Board. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2784-8. URL http://doi.acm.org/10.1145/2594291.2594320.

60. **SPECjbb** (2005). SPEC JBB2005. https://www.spec.org/jbb2005/.

61. **SPECjvm** (2008). SPEC JVM2008. https://www.spec.org/jvm2008/.

62. **Sreedhar, V. C.**, **M. Burke**, and **J.-D. Choi**, A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '00. ACM, New York, NY, USA, 2000. ISBN 1-58113-199-2. URL http://doi.acm.org/10.1145/349299.349326.

63. **Stadler, L.**, **T. Würthinger**, and **H. Mössenböck**, Partial Escape Analysis and Scalar Replacement for Java. *In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2670-4. URL http://doi.acm.org/10.1145/2544137.2544157.

64. **Steensgaard, B.**, Points-to Analysis in Almost Linear Time. *In Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96. ACM, New York, NY, USA, 1996. ISBN 0-89791-769-3. URL http://doi.acm.org/10.1145/237721.237727.

65. **Stinson, D. R.**, *Cryptography: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1995, 1st edition. ISBN 0849385210.

66. **Sălcianu, A.** and **M. Rinard**, Purity and Side Effect Analysis for Java Programs. *In Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 3-540-24297-X, 978-3-540-24297-0. URL http://dx.doi.org/10.1007/978-3-540-30579-8_14.

67. **Tan, T.**, **Y. Li**, and **J. Xue**, Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. *In* **X. Rival** (ed.), *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-53413-7.

68. **Tan, T.**, **Y. Li**, and **J. Xue**, Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-4988-8. URL http://doi.acm.org/10.1145/3062341.3062360.

69. **Thiessen, R.** and **O. Lhoták**, Context Transformations for Pointer Analysis. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-4988-8. URL http://doi.acm.org/10.1145/3062341.3062359.

70. **Vallée-Rai, R.**, **P. Co**, **E. Gagnon**, **L. Hendren**, **P. Lam**, and **V. Sundaresan**, Soot - a Java Bytecode Optimization Framework. *In Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99. IBM Press, 1999. URL http://dl.acm.org/citation.cfm?id=781995.782008.

71. **Vivien, F.** and **M. Rinard**, Incrementalized Pointer and Escape Analysis. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '01. ACM, New York, NY, USA, 2001. ISBN 1-58113-414-2. URL http://doi.acm.org/10.1145/378795.378804.

72. **WALA** (2019). The T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

73. **Whaley, J.** and **M. S. Lam**, Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '04. ACM, New York, NY, USA, 2004. ISBN 1-58113-807-5. URL http://doi.acm.org/10.1145/996841.996859.

74. **Whaley, J.** and **M. Rinard**, Compositional Pointer and Escape Analysis for Java Programs. *In Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99. ACM, New York, NY, USA, 1999.

75. **Wilson, B.** (2016). Why is Java the Most Popular Programming Language? https://blogs.oracle.com/oracleuniversity/why-is-java-the-most-popular-programming-language.

76. **Wilson, R. P.** and **M. S. Lam**, Efficient Context-sensitive Pointer Analysis for C Programs. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '95. ACM, New York, NY, USA, 1995. ISBN 0-89791-697-2. URL `http://doi.acm.org/10.1145/207110.207111`.

77. **Xu, G.** and **A. Rountev**, Merging Equivalent Contexts for Scalable Heap-cloning-based Context-sensitive Points-to Analysis. *In Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-050-0. URL `http://doi.acm.org/10.1145/1390630.1390658`.

78. **Yu, H.**, **J. Xue**, **W. Huo**, **X. Feng**, and **Z. Zhang**, Level by Level: Making Flow- and Context-sensitive Pointer Analysis Scalable for Millions of Lines of Code. *In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-635-9. URL `http://doi.acm.org/10.1145/1772954.1772985`.

LIST OF PAPERS BASED ON THE THESIS

1. **Manas Thakur and V. Krishna Nandivada**. Compare Less, Defer More: Scaling Value-contexts Based Whole-program Heap Analyses. *In Proceedings of the 28^{th} International Conference on Compiler Construction (CC 2019)*, ACM, New York, NY, USA, 2019, pp. 135–146.
   URL `http://doi.acm.org/10.1145/3302516.3307359`.

2. **Manas Thakur and V. Krishna Nandivada**. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, New York, NY, USA, 2019. URL `https://doi.org/10.1145/3337794`.

Posters:

1. **Manas Thakur and V. Krishna Nandivada.** Precise, Efficient and Secure Just-In-Time Analysis of Java Programs. *European Conference on Programming Languages (ECOOP 2019)*, London, UK, July 2019.