Explorations in Static Pointer Analysis: Adaptive Scalability and Strong Guarantees

George Kastrinis[1]

National and Kapodistrian University of Athens,
Department of Informatics and Telecommunications, Athens, Greece

Abstract. Static program analysis aims to automatically reason about certain properties a given program might exhibit under all possible executions without actually observing such executions. Static *pointer* analysis is a major subcategory that focuses on the objects that program expressions might point to during program executions. The evolution of programming languages has led to the addition of many abstraction layers that, as a result, have made any automatic reasoning about a program a challenging task at best or an infeasible one at worst. Thus, any practical static pointer analysis algorithm has to compromise and aim to approximate results in some way—either computing more or less than what is actually true.

This dissertation shows how we can obtain *precise* yet also *scalable* static pointer analysis algorithms by carefully differentiating policies for different parts of the program. Furthermore, since a static pointer analysis algorithm with global soundness guarantees and meaningful results throughout is not realistic, we show that it is possible to design analyses that offer *strong guarantees* on the soundness of the results for specific parts of the program.

Pointer analyses in the past introduced the concept of *context-sensitivity* in order to tackle the ever growing problem of imprecision versus scalability. Context is used to annotate analysis components so that the analysis can be more precise without at the same time sacrificing scalability. We show beneficial ways to combine different context flavors for different parts of the program without paying the cost that a naive combination would incur.

Another attempt at producing precise yet scalable analyses leads us to an introspective analysis. We employ a common adaptive pattern in which a cheap imprecise analysis is run first so various metrics can be gathered, and then a more precise (and costly) analysis can be used only in parts of the program—under the assumption that more precise handling of the rest would only incur performance penalties.

Subsequently, we shift our attention to an analysis that *under*-approximates results (instead of the norm of *over*-approximating) so that it might report less but can guarantee those properties to always hold. We build upon observations on the properties that such analyses have in order to apply a specialized data structure that speeds up our algorithm by nearly two orders of magnitude.

---

[1] Dissertation Advisor: Yannis Smaragdakis, Professor.

Finally, in our last contribution, we revisit an analysis formulation that over-approximates results to create an analysis algorithm that is truly sound but at the same time highly efficient. Our analysis is conservative, guaranteeing soundness even in the presence of arbitrary unknown code, but avoids wasting any work on computations that will later be invalidated due to soundness concerns.

Keywords: Pointer Analysis ◆ Alias Analysis ◆ Object-Oriented Programming ◆ Precision ◆ Performance ◆ Context-Sensitivity

1. Introduction

Static program analysis is the cornerstone of several modern programming facilities and tools for program development and aided program understanding. Nowadays, it is an umbrella term for many different methodologies (Hoare logic [10, 11, 21, 25], model checking [4, 5, 9,23], symbolic execution [2, 12, 18, 22], abstract interpretation [6–8], data-flow analysis [13,16, 17, 20, 24, 26], and so on) all with the ultimate goal of inferring a program's properties,without the need of an actual execution. It is routinely employed in many different contexts:compilers, bug detectors, verifiers, security analyzers, IDEs, and a myriad other tools.

The main intention of any static program analysis algorithm is to reason about the set of all feasible behaviors (under some abstraction of behaviors) that a given program might exhibit under all possible executions. For example, could this method throw a runtime exception? or is that type cast possible to fail under some program input? etc. As a result, virtually all interesting static program analysis questions are undecidable—indeed the prototypical undecidable problem, the halting problem, is a static program analysis question: will a program terminate under all inputs?

Pointer analysis (also known as points-to analysis) is a fundamental subdomain of static program analysis that consists of computing some abstract memory model for a given program. The essence of such an analysis is to compute a set of possible objects that a program variable or expression may point to during program execution. A straightforward endeavor at first, it quickly gets too complicated in practice due to all of the intricate details one has to take into account and the multitude of different features that mutually depend on each other.[2] Although a challenging task, smart implementations of pointer analysis can bear many benefits to client analyses that will subsequently consume the results to reason about specialized behaviors (e.g., security vulnerabilities or potential optimization opportunities).

A closely related analysis, sometimes confused with pointer analysis, is alias analysis in which one computes sets of program expressions that may alias (i.e., point to common objects) with each other. Pointer analysis could—although it is not the only possible alternative—be used to implement an alias analysis algorithm, and vice versa.

At the same time, programming languages are evolving, becoming ever higher-level and more complex. Many abstraction levels are added throughout the years with the aim of making the very task of programming easier for developers allowing them to express more with less effort (e.g., in terms of lines

[2] The analysis inputs are large and the analysis algorithms are typically quadratic or cubic, but try to maintain near-linear behavior in practice, by exploiting program properties and maintaining precision—more precise (i.e., smaller) inference sets lead to less work.

of code). Frequently, new features come with complicated semantics regarding their possible implementations and usually they interact in intricate ways with pre-existing ones.

Additionally, modern software paradigms have evolved as well. Complex design patterns have become the norm for experienced developers, immense libraries and frameworks are accepted as a prerequisite for any non-trivial software, and over-involved build tools often make even the task of understanding all of the program's dependencies a challenge.

It comes as no surprise that any kind of static analysis has struggled to keep up with this ever-increasing complexity both in programming languages and software. Even the seemingly simple task of computing a program's call-graph (i.e., which methods are called at every invocation site) requires sophisticated analysis for achieving acceptable precision. Thus, the main emphasis of pointer analysis algorithms is on combining fairly precise modeling of pointer behavior and memory abstractions with scalability.

Thesis.

> Precise yet scalable static pointer analysis algorithms can be obtained by careful choice of different policies for different parts of the program. In a complementary fashion, analyses can be designed to offer (uniquely) strong guarantees on the soundness of results, but for a part of the program only.

We provide a number of techniques for implementing scalable static pointer and alias analyses in the setting of Java programs by configuring the analysis strategy differently for different code parts. Additionally, we present a couple of defensive algorithms for reporting highconfidence results even in the presence of hostile or unknown program points.

1.1 Pointer Analysis Crash Course

Before enumerating the scientific contributions of this dissertation, it is mandatory to introduce certain concepts related to pointer analysis, that comprise the scientific and technical base of this work. This is by no means a detailed presentation of said concepts—a more elaborate introduction will follow in Chapter 2.

Implementation Platform & Target Language. Most of the following work and algorithms have been expressed in the Doop framework [3]. Doop is a well established pointer analysis framework offering a wide variety of full-fledged algorithms for static pointer analysis of Java programs. More in Section 2.11.

Context Sensitivity. Implementing any sophisticated pointer analysis algorithm quickly turns out to be a balancing act between precision and performance tradeoffs. Any attempt for a scalable algorithm might inadvertently be accompanied by significant precision losses whereas an endeavour for highly precise results might also enforce huge performance penalties.

Throughout the years, the scientific community has amassed a few tools in its arsenal in order to tackle this conundrum of precision versus performance. Among those tools, a widely employed notion, that aims to improve precision without having to pay an unbearable performance cost, is that of context

resulting in context-sensitive algorithms. An algorithm will use additional information (also known as context) to annotate analysis components with the aim of countering potential precision losses. The key idea is that the analysis will differentiate the handling of program elements under some contexts while it will collapse it under others. For instance, an algorithm might differentiate the analysis of a method when called from method A or method B or anywhere else (thus under three different contexts).

Two main kinds of context have been widely used in the past; in call-site-sensitive analyses call instructions comprise the context elements, whereas in object-sensitive analyses context is based on the identity of the calling object at each method invocation. More in Section 2.2.

May vs. Must Analyses. The goal of any static program analysis algorithm is to reason about a set of behaviors under all potential program executions. This endeavour is an undecidable problem for any set of behaviors other than the most trivial ones. As a consequence, any practical algorithm has to approximate results in one of two directions; either over-approximate and both report all possible behaviors and also some that will never actually arise, or to under-approximate and be conservative by reporting only a subset of potential arising behaviors. Analyses are often categorized as may-analyses when they overapproximate results, and as must-analyses when they under-approximate results. More in Section 2.6.

Soundness. A formal term often used to accompany static pointer analysis algorithms is that of soundness. In layman's terms, an algorithm is said to be sound when it actually does what it claims. For instance, a may-pointer analysis claims that it aims to over-approximate the set of objects that various program expressions may point to in all possible program executions. If the results are not missing any such inference that could arise in a program execution, then the algorithm is sound. Due to various factors, most may-pointer analysis algorithms forgo soundness in order to maintain scalability. A more detailed discussion regarding soundness will follow in Sections 2.7-2.9.

1.2 Scientific Contributions

In this section, we will briefly explain the main scientific contributions of this dissertation. As already mentioned, the exploration happens in the context of analyzing Java—mainly by use of the Doop framework—although it is not far-fetched to generalize results to other languages that offer similar features and follow similar paradigms.

Ever since the introduction of object sensitivity by Milanova et al. [19], there has been increasing evidence that it is the superior context choice for programs expressed in object oriented languages, yielding a high precision to cost ratio. Such has been its success that in practice it has almost superseded the use of more traditional call-site-sensitive analyses in object-oriented languages. Nevertheless, a call-site-sensitive analysis is not always inferior as there are language features and code patterns that may partially favor this kind of context abstraction.

Consequently, one might consider an approach where both context flavors are—naively—combined in every program point with the goal of increasing the precision of the end result. Truly, such a combination would bear some precision benefits but in most cases it would be accompanied by an infeasibly high cost.

First contribution. Our first scientific contribution is a step towards a more sophisticated handling, aiming to achieve a beneficial combination of both context flavors. We propose a hybrid context flavor for defining a family of analyses where classical contexts are mixed and combined only in those program points where it is profitable for the analysis. The resulting selective combination of both context kinds vastly outperforms not only analyses following the naive non-selective combination approach, but also their "normal" object-sensitive counterparts. This result holds for a large array of context-sensitive analyses establishing a new set of performance/precision sweet spots. Figure 1 depicts performance vs. precision metrics for eight of our benchmarks over all analyses.

Second contribution. The second scientific contribution tries to tackle an oft-reported issue with context-sensitive analyses, in that they mostly operate in two extremes: either the analysis is precise enough that it manipulates only manageable sets of data, and thus scales impressively well, or the analysis gets quickly derailed at the first sign of—massive— imprecision and becomes orders-of-magnitude more expensive than would be expected given the program's size. Currently, there is no approach for a precise, context-sensitive (of any context flavor) analysis that would scale across the board at a level comparable to that of a context-insensitive one. Instead, we propose a two step process by means of introspective analysis: the approach uniformly scales context-sensitive analyses by eliminating the performance-detrimental behavior, only at a small precision expense.

Introspective analysis employs a common adaptive pattern: it first performs a context insensitive analysis and then it uses the results to selectively refine (i.e., analyze contexts ensitively) only those program elements that are expected not to cause an explosion in running time or memory space. The technical challenge is to appropriately identify such program elements. We show that a simple but principled approach can be remarkably effective, achieving scalability (often with dramatic speedup) for benchmarks previously completely out-of-reach for deep context-sensitive analyses. Figure 2 depicts experimentals results for a subset of our analyses.

For the last two contributions, we shift our attention towards analyses that aim for the highest confidence in their claims. Although quite reluctant and conservative in making a claim, when they actually do they make certain that it is the correct decision.

Third contribution. The next, third, contribution features a different flavor of static program analysis. Instead of the more commonly researched paradigm of may-analyses, we chose to explore the alternative approach of a must-analysis. More specifically, we focus on an instance of a must-alias (also known as definite-alias) analysis that aims to infer aliasing relationships among program expressions that are guaranteed to always hold.[3] The applications of a must-alias analysis are manifold: (1) it is useful for enabling optimizations such as constant folding and register allocation, (2) it can increase the precision of bug detectors, e.g., greatly benefiting a null-reference detector and a non-termination detector, and (3) it can be used internally as part of more complex analyses, e.g., one that can reason correctly about "strong updates" at instructions that modify the heap. In order to compute high-confidence, non-trivial results, the analysis needs to be flow-sensitive, i.e., compute information at each program point and propagate it forward while respecting the control flow of the program.

---

[3] As previously mentioned, a must-analysis will aim to compute an under-approximation of behaviors that will happen in every possible program execution.

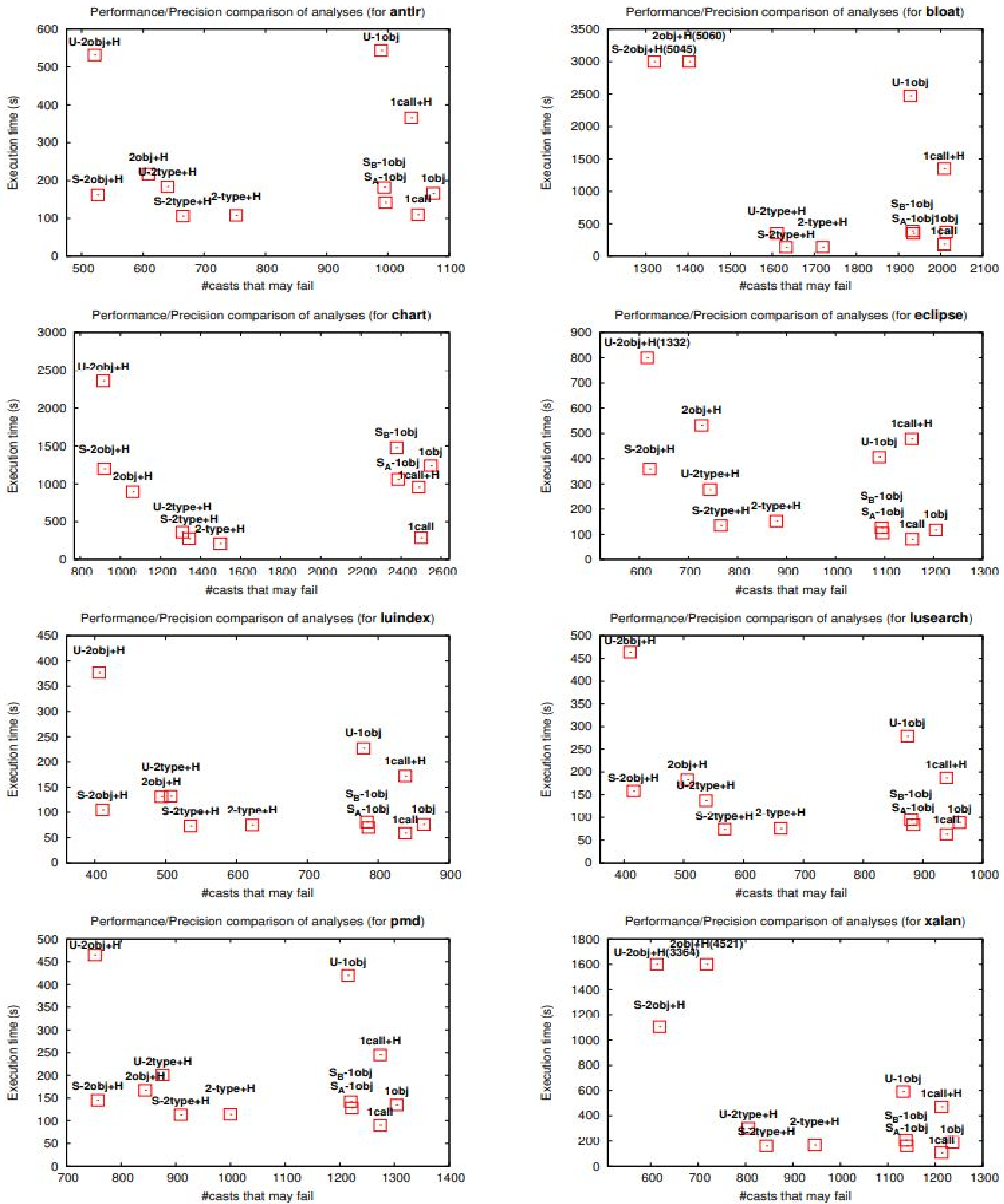

Figure 1: (*re 1st Contribution*) Graphical depiction of performance vs. precision metrics for eight of our benchmarks over all analyses. Lower is better on both axes. The Y axis is truncated for readability. Out-of-bounds points are included at lower Y values, with their real running time in parentheses.

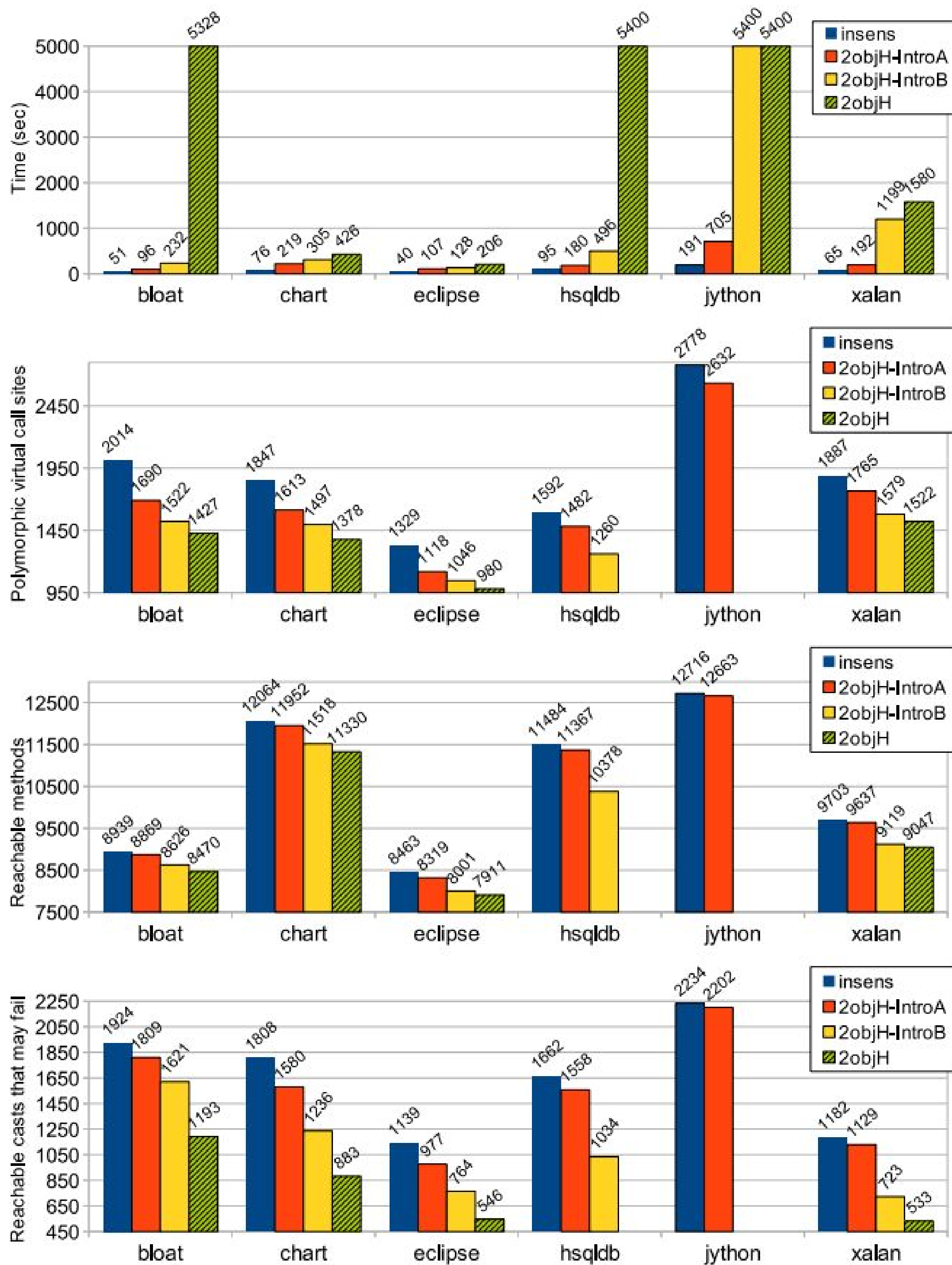

Figure 2: (*re 2nd Contribution*) Performance and precision (3 separate metrics: calls that cannot be devirtualized, reachable methods, casts that cannot be eliminated) for introspective context-sensitive variants of a 2objH analysis, compared with baselines (2objH and insensitive).

Furthermore, we observe that a must-alias analysis exhibits certain properties that can be exploited in order to achieve a more efficient algorithm without any compromise in the precision or the validity of its results. We present a custom specialized data structure (implemented both in Java and in Datalog) that speeds up a must-alias analysis by nearly two orders of magnitude. The data structure achieves its efficiency by encoding multiple alias sets in a single linked structure, and compactly representing the aliasing relations of arbitrarily long program expressions. Under this approach, must-alias analysis can be performed efficiently, over large Java benchmarks, in under half a minute, making the analysis cost acceptable for most practical uses. The performance benefits of the specialized data structure are shown in Figure 3.

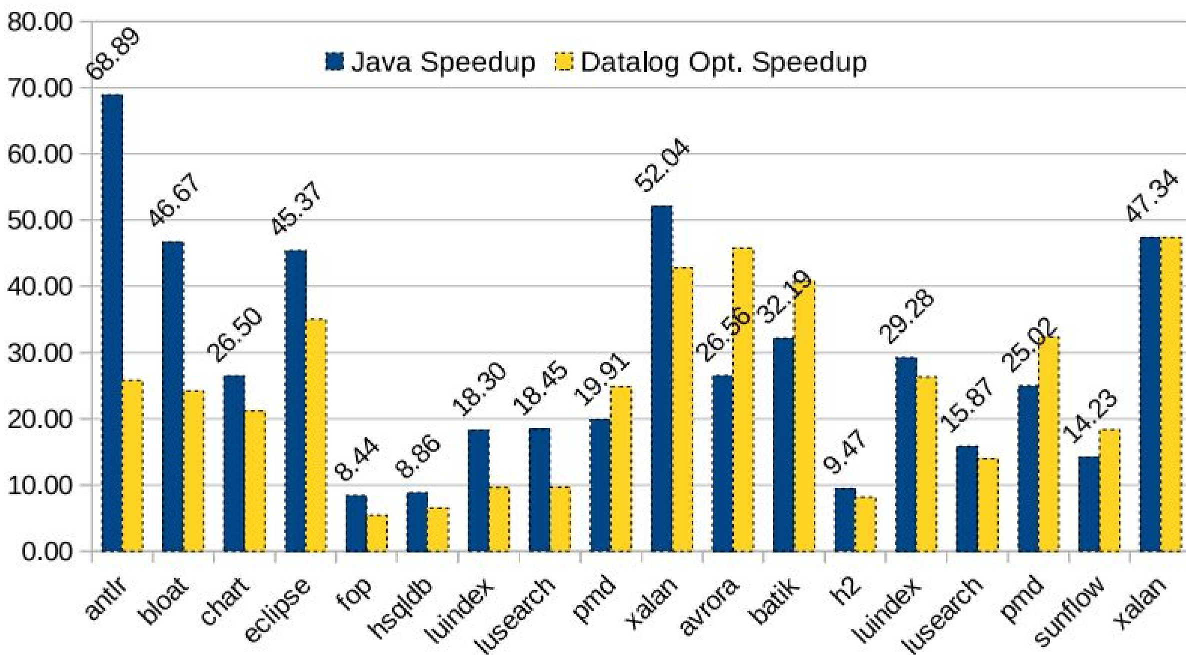

Figure 3: (*re 3rd Contribution*) Speedups of employing the optimized data structure.

Fourth contribution. For our last contribution, we revisit the setting of a may-analysis but this time while aiming to explore the potential of a truly sound—instead of just soundy—yet practical analysis. We present such an approach in a defensive may-point-to analysis, which can guarantee soundness even in the presence of arbitrary opaque code.[4] A key design tenet of our approach is laziness: the analysis computes points-to relationships only for program expressions that are guaranteed to never escape into opaque code.

The defensive nature of our analysis means that it might miss some valid inferences, but because of its laziness it will never waste work to compute sets that are not "complete", i.e. that may be missing elements due to opaque code. This frugal approach is what enables the great efficiency of the algorithm, allowing for a highly precise points-to analysis (such as a 5-call-site-sensitive, flow-sensitive analysis). Despite its conservative nature, the analysis yields sound, actionable results for a large subset of the

[4] Code that cannot be analyzed such as dynamically generated or native code, or dynamic language features such as reflection, invokedynamic, etc

program code, achieving (under worst-case assumptions) 34-74% of the program coverage of an unsound state-of-the-art analysis for real-world programs.

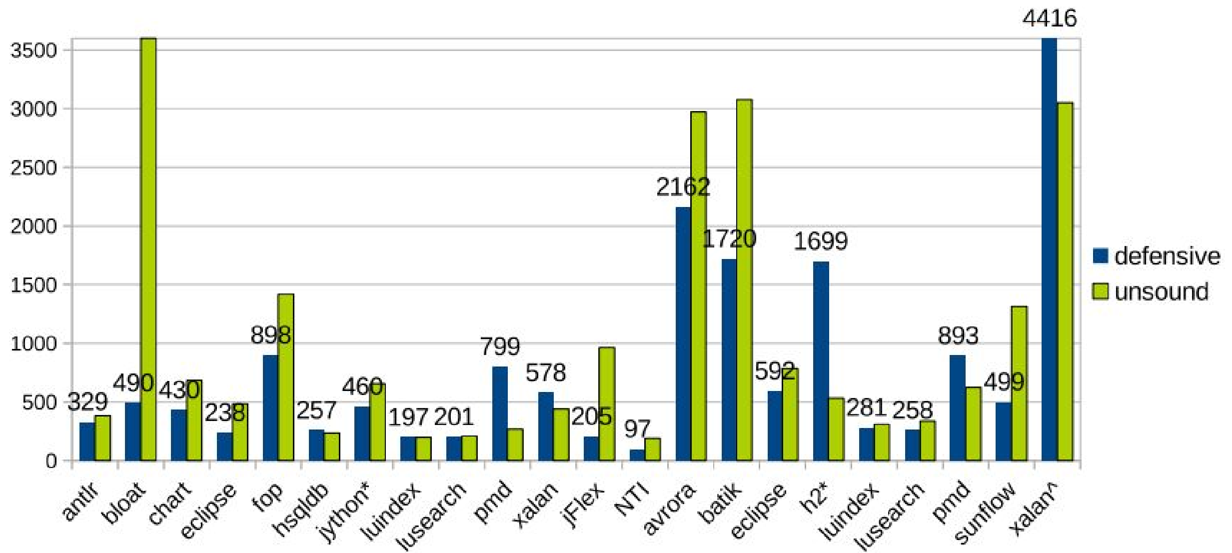

Figure 4: (*re 4th Contribution*) Execution time (in seconds) of defensive analysis, with running time of 2objH (with unsound reflection handling) shown as a baseline. Labels are shown for defensive analysis only to avoid crowding the plot.

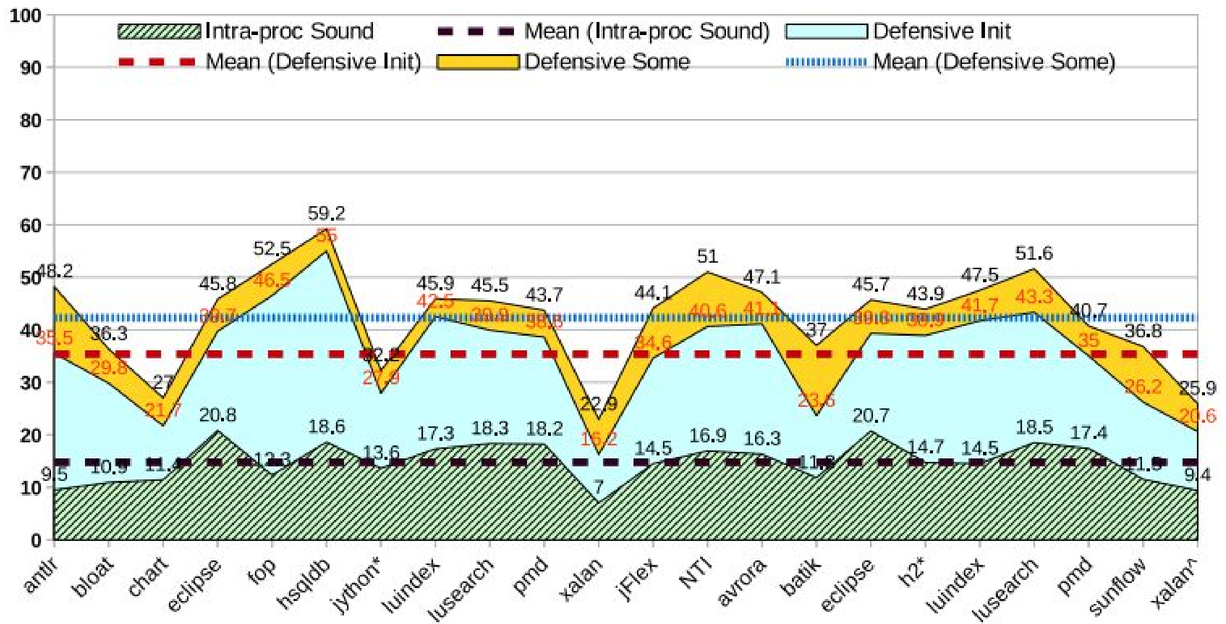

Figure 5: (*re 4th Contribution*) Virtual call sites that are found to have receiver objects of a single type. These call sites can be soundly devirtualized. Numbers are shown as percentages of devirtualization achieved by unsound 2objH analysis.

1.3 Outline

The dissertation is organized as follows:

- Chapter 2 offers a quick yet non-trivial introduction to certain notions or properties that are important to take under consideration when designing a sophisticated static pointer analysis algorithm.

- Chapter 3 examines how a naive combination of object sensitivity and call-site sensitivity into a single analysis can be massively penalizing in terms of performance. Following that, we presents a hybrid context-sensitive approach for implementing points-to analyses that leverage the benefits of combining both object and a call-site sensitivity while avoiding to pay most of the cost of a naive combination. This chapter presents research previously published in "***Hybrid Context-Sensitivity for Points-To Analysis***" [14].

- Chapter 4 examines the well-known, bi-modal nature of classical static program pointsto analyses in regards to scalability; they are either quite scalable or not scalable at all. In order to counter that discrepancy, we propose an adaptive approach in introspective analysis, where an imprecise analysis is used as a stepping stone in order to fine-tune program points in which a more precise handling is both beneficial and not detrimental to the overall analysis's performance. This chapter presents research previously published in "***Introspective Analysis: Context sensitivity, Across the Board***" [28].

Both aforementioned contributions aim for more scalable analyses that achieve superior performance without foregoing precision. The next three contributions aim for analyses that although more restrained on what they report, they do so with much more confidence in the accuracy of their claims.

- Chapter 5 examines how to compose a declarative model of a rich family of must-alias analyses, with emphasis on careful and compact modeling, while at the same time exposing the key points where the algorithm's inference power can be adjusted. This chapter presents research previously published in "***A Datalog Model of Must-Alias Analysis***" [1].

- Chapter 6 builds upon the previous chapter and goes forth to provide a specialized data structure that by exploiting the nature of a must-alias analysis it achieves high performance without any sacrifice on the accuracy of its results. We explore the data structure's performance in both an imperative (implemented in Java) and a declarative (implemented in Datalog) setting and contrast it extensively with prior techniques. This chapter presents research previously published in "***An Efficient Data Structure for Must-Alias Analysis***" [15].

- Chapter 7 examines how a defensive reasoning in the presence of opaque code can be combined along with computational laziness in order to produce a highly efficient, highly precise and truly sound may-points-to analysis. This chapter presents research previously published in "***Defensive Points-To Analysis: Effective Soundness via Laziness***" [27], that also received a Distinguished Paper award.

- Chapter 8 first discusses related work that is specific to previous chapters, and then expands to various other interesting subjects in the broader realm of static analysis.
- Chapter 9 concludes this dissertation by assessing our initial thesis and discussing future work.

2. Conclusions

To summarize, we advocate that modern, sophisticated, static pointer analyses need not make a sacrifice over precision or scalability, to achieve the other. Both properties are achievable with appropriate tuning and design choices, for different parts of the program. Complementary, it is possible for analyses to compute results alongside with strong soundness guarantees, again focusing at specific parts of the program. To conclude, a static pointer analysis algorithm doesn't have to use a one-size-fits-all handling of every language feature and program point, but instead it is favorable to methodically differentiate its policies for different parts of the code, towards different desired outcomes.

References

1. George Balatsouras et al. "A Datalog Model of Must-Alias Analysis". In: International Workshop on State Of the Art in Program Analysis (SOAP). SOAP '17
2. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT—a Formal Systemfor Testing and Debugging Programs by Symbolic Execution". In: SIGPLAN Notices 10.6 (1975)
3. Martin Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses". In: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). OOPSLA '09
4. Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: Logics of Programs, Workshop. Vol. 131. LOP '81
5. Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: ACMTrans. on Programming Languages and Systems 8.2 (1986)
6. Patrick Cousot and Radhia Cousot. "Abstract Interpretation and Application to LogicPrograms". In: Logic Programming 13.2 & 3 (1992)
7. Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: Logic and Computation 2.4 (1992)
8. Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: Principles of Programming Languages (POPL). POPL '77
9. E. Allen Emerson and Edmund M. Clarke. "Characterizing Correctness Properties of Parallel Programs Using Fixpoints". In: Proc. of the 7th International Colloquiumon Automata, Languages and Programming. Vol. 85. ICALP '80
10. Robert W Floyd. "Assigning Meanings to Programs". In: Proc. of Symp. in Applied Mathematics. Mathematical Aspects of Computer Science. Vol. 19

11. C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: Commun.ACM 12.10 (1969)
12. William E. Howden. “Symbolic Testing and the DISSECT Symbolic Evaluation System”. In: IEEE Trans. Software Engineering 3.4 (1977)
13. John B. Kam and Jeffrey D. Ullman. “Monotone Data Flow Analysis Frameworks”. In: Acta Informatica 7 (1977)
14. George Kastrinis and Yannis Smaragdakis. “Hybrid Context-Sensitivity for Points-To Analysis”. In: Programming Language Design and Implementation (PLDI). PLDI ’13
15. George Kastrinis et al. “An efficient data structure for must-alias analysis”. In: International Conference on Compiler Construction (CC). CC ’18
16. Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. Data Flow Analysis - Theory and Practice. CRC Press, 2009
17. Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: Proc.of the 1st ACM Symp. on Principles of Programming Languages. POPL ’73
18. James C. King. “Symbolic Execution and Program Testing”. In: Commun. ACM 19.7 (1976)
19. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to and side-effect analyses for Java”. In: International Symposium on Software Testing and Analysis (ISSTA). ISSTA ’02
20. Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997
21. Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures”. In: Proc. of the 15th International Workshop on Computer Science Logic. Vol. 2142. CSL ’01
22. Corina S. Pasareanu and Neha Rungta. “Symbolic PathFinder: symbolic execution of Java bytecode”. In: Proc. of the 25th IEEE/ACM International Conf. on Automated Software Engineering. ASE ’10
23. Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: Proc. of the 5th International Symp. on Programming. Vol. 137. Springer, 1982
24. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. POPL ’95
25. John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: Proc. of the 17th IEEE Symp. on Logic in Computer Science. LICS ’02
26. Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data flow analysis”. In: Program Flow Analysis: Theory and Applications. Prentice-Hall, 1981
27. Yannis Smaragdakis and George Kastrinis. “Defensive Points-To Analysis: Effective Soundness via Laziness”. In: European Conference on Object-Oriented Programming (ECOOP). ECOOP ’18
28. Yannis Smaragdakis, George Kastrinis, and George Balatsouras. “Introspective Analysis: Context-sensitivity, Across the Board”. In: Programming Language Design andImplementation (PLDI). PLDI ’14