SCALING CONTEXT-SENSITIVE POINTS-TO ANALYSIS

A Thesis

Submitted For the Degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Rupesh Nasre.

under the supervision of

Prof. R. Govindarajan

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

AUGUST 2011

Contents

Chapter 3

A Survey of Pointer Analysis Methods

3.1 Introduction

We present a survey of pointer analysis methods in this chapter. Summarizing three decades of research work on pointer analysis in one chapter is a difficult task. Therefore, instead of trying to cover the work exhaustively, we attempt to discuss the most interesting and important works on pointer analysis.

We divide the survey in 12 categories. The categories are not completely mutually exclusive. However, we try to classify the related work into these categories based on the focus of the work.

1. **Surveys** (Section 3.2): In this category, we discuss the surveys on pointer analysis. Typically, a survey contains a broader perspective than individual algorithms and although it may not add a new innovation to the area, it helps in understanding the higher level picture of the area in the form of trends, pitfalls and lessons.

2. **Complexity Results** (Section 3.3): In this category, we discuss the work which focuses on computational complexity of pointer analysis algorithms. Such a work identifies the complexity classes for variants of the algorithm and proposes polynomial solutions to the possible variants.

3. **Use of Novel Data Structures** (Section 3.4): In this category, we discuss the work

which proposes the use of novel data structures to store points-to information which helps in scaling pointer analysis, e.g., binary decision diagrams. We also discuss the work that compares the same pointer analysis algorithm using different data structures.

4. **Optimizations and Techniques** (Section 3.5): In this category, we discuss the work which proposes techniques to speed-up points-to analyses. These techniques are generic enough to be applicable to multiple points-to analysis algorithms.

5. **Exact Methods** (Section 3.6): In this category, we discuss various novel algorithms for pointer analysis which use an inclusion-based approach.

6. **Methods Achieving Explicit Trade-offs** (Section 3.7): In this category, we discuss the work which offers an explicit control over the trade-off between precision and scalability.

7. **Client-driven and Demand-driven Methods** (Section 3.8): Exhaustive algorithms compute all possible points-to facts from a given set of constraints. In this category, we discuss their variants, namely, client-driven analysis which prioritizes various processing elements (functions, constraints, etc.) to suit the needs of a client, and demand-driven analysis which cumulatively processes only the required program fragments to answer a specific query.

8. **Incremental and Probabilistic Methods** (Section 3.9): In this category, we discuss incremental analysis and probabilistic analysis. An incremental analysis allows for dynamic addition or removal of constraints from the original set of constraints and computes points-to information only for the affected set of constraints. A probabilistic analysis assigns probabilities to the computed points-to facts based on the likelihood of the fact being realized at run-time.

9. **Analysis of Parallel Programs and Parallel Analyses** (Section 3.10): In this category, we discuss the work that deals with (sequential) pointer analysis of multi-threaded programs and also the parallel versions of pointer analyses.

10. **Applications of Points-to Analysis** (Section 3.11): Pointer analysis has been extensively used for various other analyses and transformations. In this category, we briefly discuss some of the applications of pointer analysis, e.g., slicing and shape analysis.

11. **Evaluations and Quantifications** (Section 3.12): In this category, we focus on the work which are experimental in nature and deduce some interesting characteristics of benchmarks or of the underlying analysis, which help us understand the analysis behavior.

12. **Points-to Analysis for Other Languages** (Section 3.13): We discussed our work in the context of C programs. However, other languages pose different challenges and provide different opportunities to pointer analysis. For instance, Java is type-safe which enables us to design a polynomial time points-to analysis for the language, but it also poses challenges due to dynamic class loading and reflection. In this category, we discuss the work on pointer analysis that deals with different programming languages.

3.2 Surveys

In this section, we discuss various surveys on pointer analysis. The most cited survey is by Hind [57]. It identifies several dimensions which affect the trade-off between precision and scalability, and along which the existing work can be categorized. Some of the dimensions mentioned are flow-sensitivity, context-sensitivity, heap modeling which deals with how allocation sites are modeled, etc. The survey also discusses various issues in the terminology, metrics of precision evaluation and reproducibility of results. Finally, it discusses several directions for future research (in the year 2000): scalability, improving precision without affecting scalability, client-driven analysis, extending pointer analysis along the dimensions of path-sensitivity and context-sensitivity, modeling heap to improve precision without adversely affecting scalability, better modeling of aggregates, demand-driven and incremental analyses, handling features of object oriented languages, analyzing incomplete programs, providing annotations for improving precision, and pursuing and uncovering engineering insights for a scalable analysis. While mentioning the issues and the open problems, the survey also provides anecdotes from experts in the field.

Ryder [113] classifies various approaches to reference analysis for object oriented languages based on analysis dimensions. Some of the dimensions are also applicable to procedural languages. The dimensions are flow-sensitivity, context-sensitivity, program representation, object representation, field-sensitivity, reference representation and directionality of information flow in a constraint. For each of these dimensions, she discusses the related work. She also discusses various open issues in the context of object oriented languages, namely, reflection, native methods, exceptions, dynamic class loading, and incomplete programs.

A few instances of informal classification of pointer analysis algorithms exist. One of the most informative surveys on pointer analysis in the recent past is by Lhotak [76]. The survey deals with two aspects: various abstractions in the analysis which affect the precision and efficiency and algorithmic aspects which affect the analysis efficiency. The former discusses various abstractions like type filtering, and field-sensitivity, flow-sensitivity, context-sensitivity. The latter deals with points-to information propagation and implementation of sets to represent the points-to constraints and the points-to information.

Raman [104] surveys three methods: unification-based analysis (using a type system) [123], pointer analysis using BDDs [6] and an application of pointer analysis in bug detection [80]. He discusses various precision-scalability trade-offs involved in choosing an appropriate method for analyzing a program with pointers.

Wu [134] presents a survey of alias analysis. He discusses about various kinds of flow graphs used in literature, namely, inter-procedural control flow graph [70, 74], invocation graph [32, 132], procedural call graph [18, 10] and context-sensitive procedural call graph [13, 17]. He also discusses about abstract data representations used to model variables inaccessible to a procedure [70, 32], global variables [132] and access paths [70, 17]. He further discusses various approaches used to model recursive data structures, namely, 1-level [31, 132], `k`-limiting [70, 17], beyond `k`-limiting using symbolic access paths [27], and using shape analysis [41]. Wu also categorizes various points-to analysis algorithms based on flow-sensitivity and context-sensitivity and also based on their alias representation, flow graph, target language, benchmarks used and modularity. In a later part of the survey, Wu discusses related work on assembly level alias analysis.

Rayside [105] provides a five-page summary of various important aspects of pointer analysis. Derived from Whaley's talk slides [127], Rayside classifies several analyses based on flow-sensitivity and context-sensitivity. It also mentions incremental analysis, handling incomplete programs and demand-driven analysis as new research challenges.

3.3 Complexity Results

In this section, we discuss various results on the computability of pointer analysis. Landi [70] and Ramalingam [103] proved that in the presence of non-scalar variables and dynamic memory allocation, flow-sensitive alias analysis is undecidable. Chakaravarthy [12] proved the undecidability of flow-sensitive aliasing even when a program contains only scalar variables when dynamic memory allocation is allowed. Landi [71] and Muth and Debray [93] proved that flow-sensitive points-to analysis is PSPACE-Complete even when all the variables are scalars with well defined types and only two levels of dereferencing (only `p, *p` and `**p`) are allowed. For the case when only a single level of dereferencing is allowed, Landi [71] gave a polynomial-time algorithm.

We now discuss some results related to flow-insensitive points-to analysis. Horwitz [59] proved that in the absence of dynamic memory allocation, flow-insensitive points-to analysis is NP-Hard when all the variables are scalars and arbitrary number of dereferencing is allowed. When the problem is restricted to well-defined types, the flow-insensitive analysis can be solved in polynomial time, even when arbitrary number of dereferencing is allowed [12]. This positive result is important for type-safe languages like Java and proposes a non-trivial points-to analysis variant which is solvable in polynomial time. It also theoretically proves that flow-insensitive analysis is easier than its flow-sensitive counterpart, since the flow-sensitive version is PSPACE-Complete.

In the context of object-oriented languages, Chatterjee et al. [14] discuss the complexity of points-to analysis in the presence of exceptions. They present a polynomial-time algorithm for points-to analysis in the presence of exceptions for program without threads. They also prove that an interprocedural points-to analysis with single-level types and exceptions with subtyping, but without dynamic dispatch, is PSPACE-Hard. They also prove that the intraprocedural version of the above problem is PSPACE-Complete. They also improve the worst-case time complexity of points-to analysis in the absence of exceptions from $\mathtt{O}(\mathtt{n}^7)$ [72, 106] to $\mathtt{O}(\mathtt{n}^5)$.

Several open problems related to flow-insensitive analysis exist. It is unknown whether flow-insensitive points-to analysis in the absence of dynamic memory allocation with arbitrary number of dereferencing is in NP. Also, it is unknown whether, for a bounded number of dereferences, the problem can be solved in polynomial time. Further, it is not known whether the problem remains decidable when dynamic memory allocation is allowed.

3.4 Use of Novel Data Structures

In this section, we discuss the work that proposed innovative data structures for representing the points-to (or alias) information. Earlier approaches stored alias pairs explicitly [70]. Since this representation is storage-intensive, compact representation is proposed which stores only a few basic alias pairs explicitly and new alias pairs are derived based on dereference, transitivity and commutativity [18]. Later, a more crisp representation in the form of points-to pairs has been devised [31] which significantly reduces the storage requirement.

Heintze and Tardieu [54] propose the use of sparse bitmaps for storing points-to information. Since for most programs, the points-to information is actually sparse — i.e., only a few pointers have a large number of pointees while most others have very few pointees — and accessing a sparse bitmap is very efficient, sparse bitmap is a desirable data structure for storing points-to information. It is used in GCC 4.1 [48]. However, bitmaps cannot take advantage of the commonality across various points-to sets. Therefore, for a context-sensitive analysis, the use of bitmaps requires a large amount of memory.

Zhu [141] was the first one to observe that the vast amount of points-to information can be encoded in a space-efficient manner using binary decision diagrams (BDD) [8]. Until then, BDDs were used in symbolic model checking [9] and to represent large sets and maps [84]. Due to the storage efficiency, BDDs were quickly adapted for solving points-to analysis algorithms. Berndl et al. [6], Whaley and Lam [129] and Zhu and Calman [142] propose variants of points-to analysis algorithms using BDDs for Java.

Hardekopf and Lin [48] compared the performance of BDD-based and bitmap-based points-to analyses. They found that a BDD-based implementation is, on an average, 2× slower than a sparse bitmap-based implementation, but uses 5.5× less memory.

We propose to store points-to information in a bloom filter (Chapter 4). Bloom filters offer the best-of-both-the-worlds: it's access time is as low as that for bitmaps and its memory requirement is even below that of BDDs. Although it incurs a minimal amount of precision loss, a bloom filter based analysis is likely to scale well with program size both in terms of memory and analysis time, as demonstrated in Chapter 4.

3.5 Optimizations and Techniques

In this section, we discuss the work that exploits properties of the problem to improve the efficiency of points-to analysis. Since these techniques exploit the problem structure, they are typically applicable to a wide variety of points-to analysis algorithms. While several techniques and engineering artifacts are proposed in almost every algorithm, we restrict this discussion to broader techniques which stand out on their own.

One of the most important optimizations in scaling points-to analysis is online cycle elimination [34]. This optimization is an outcome of the formulation of points-to analysis as a graph problem. The points-to constraints are represented using a constraint graph. Since the edges are dynamically added to the graph, cycles may get introduced during the analysis. Due to the abundance of large cycles, a fixed-point computation of points-to information along the cycles requires considerable number of propagations. The online cycle elimination technique exploits the fact that all the pointers involved in the cycle have the same points-to information at the fixed-point. This allows the technique to collapse the cyclic component into a single representative node which enables tracking fewer pointers than before.

Choi and Choe [19] propose cycle elimination for invocation-graph based context-sensitive points-to analysis. Their method first models sets of contexts as annotations and eliminates cycles only when the annotations appear in all the contexts.

Since cycles are formed dynamically, the cycle detection needs to be performed repeatedly. It is important to choose a good frequency of cycle detection, since checking for cycles too often can be costly and may outweigh its benefits; whereas checking for cycles very infrequently may reduce the benefits obtained using cycle detection. Hardekopf and Lin [48] propose Lazy Cycle Detection which checks for cycles when there is a good chance of finding one. This is done based on the heuristic that when an edge gets formed between two nodes of a constraint graph and the two nodes have the same points-to information, the two nodes may be in a strongly connected component. Lazy Cycle Detection significantly reduces the overhead of online cycle detection [48].

Hardekopf and Lin [48] also propose a Hybrid Cycle Detection which combines an offline analysis with an online cycle detection to improve the running time of the online analysis. The authors apply Hybrid Cycle Detection to three state-of-the-art solvers and illustrate significant benefits in the analysis performance.

To optimize the set of points-to constraints even prior to running the analysis, Rountev and Chandra propose offline variable substitution [108]. The technique identifies pointer equivalent variables (i.e., pointers with the same points-to sets at the fixed-point) from the points-to constraints by building a subset graph without running the analysis. It has been established that offline variable substitution can reduce the number of constraints by a large amount. For instance, Hardekopf and Lin found that the technique reduced the number of constraints by 60 – 77% [48].

Hardekopf and Lin [50] improve upon offline variable substitution to find more pointer equivalent variables. They also propose location equivalence to reduce the number of variables tracked during an analysis. Due to these offline optimizations, the authors found that sparse bitmaps actually require lesser memory than BDDs [50].

All the above optimizations reduce analysis time without affecting precision. Approximate pointer analysis [94] identifies pointers with *mostly similar* points-to sets as pointer equivalent and identifies pointees with *mostly similar* pointed-by sets as location equivalent. By suitably altering the similarity threshold, a client can obtain varying levels of analysis precision.

Several optimizations address efficient propagation of points-to information in the constraint graph. Pearce et al. [99] propose difference propagation to propagate only the difference in the points-to information across nodes. Wave and Deep Propagation [100] propagate points-to information in breadth-first and depth-first manner respectively for efficient analysis. Kanamori and Weise [67] propose several heuristics for choosing a node-ordering for points-to information propagation, e.g., Greatest Input Rise, Greatest Output Rise and Least Recently Fired. Pearce et al. [99] find that the Least Recently Fired strategy works very well in practice over other heuristics especially for large programs. Our work on prioritizing constraint evaluation (Chapter 6) is related, but deals with constraint evaluation ordering rather than points-to information propagation. It can be easily combined with any propagation related optimization for enjoying joint benefits.

Kahlon [66] proposes bootstrapping an analysis with the result of a prior analysis to improve scalability. He illustrates his technique by running a fast, imprecise analysis like Steensgaard's [123] to find disjoint alias sets and then running a slow but more precise analysis like Andersen's [3] on each of these alias sets to regain precision. The technique has been shown to scale well for moderately sized programs [66].

3.6 Exact Methods

In this section, we discuss the work which uses an inclusion-based approach for its analysis.

Inclusion-based points-to analysis is introduced by Andersen [3]. It is a flow-insensitive and context-insensitive points-to analysis that approximates the realizable points-to information based on inclusion of points-to sets. Approximations are inevitable since precise points-to analysis is NP-Hard [12]. For a pointer assignment $\mathtt{p}_{\mathtt{expr}} = \mathtt{q}_{\mathtt{expr}}$, it adds the points-to information of $\mathtt{q}_{\mathtt{expr}}$ into that of $\mathtt{p}_{\mathtt{expr}}$, achieving $\mathtt{pointsto}(\mathtt{p}_{\mathtt{expr}}) \supseteq \mathtt{pointsto}(\mathtt{q}_{\mathtt{expr}})$, which is essentially a set-inclusion constraint. For various pointer manipulating statements, Andersen's analysis adds such set-inclusion constraints and finally solves those constraints to achieve a fixed-point which represents a sound approximation to the points-to information.

Hind et al. [55] propose an approximation algorithm for interprocedural alias analysis. They also propose a technique for function pointer analysis that constructs a program call graph during alias analysis. They find that a flow-insensitive analysis with *kill* information does not improve precision over a flow-insensitive analysis without the *kill* information.

Liang and Harrold [78] find that Andersen's analysis [3] and Landi and Ryder's analysis [73] may not scale to large programs. Therefore, they propose a flow-insensitive, context-sensitive points-to analysis. Salient features of their analysis are that, similar to Steensgaard's analysis [123], it processes each pointer-related statement only once, computes a separate points-to graph for each procedure, and is modular. They demonstrate that their algorithm is almost as precise as Andersen's analysis, and its running time is within six times that of Steensgaard's analysis.

Yong et al. [135] propose a points-to analysis to handle structures and type-casting. They observe that supporting field-sensitivity can significantly improve the analysis precision. They also illustrate that making conservative assumptions when casting is involved usually does not cost much in terms of analysis time, space or precision.

Cheng and Mei [17] propose a modular interprocedural pointer analysis based on access-paths for pointers. They illustrate that access-paths can reduce the overhead of representing context-sensitive transfer functions .

Martena and Pietro [85] apply model checker SPIN to compute precise alias analysis information. They show that in the case of intra-procedural alias analysis, a model checking tool can enhance precision as well as efficiency.

Whaley and Lam [128] use Heintze and Tardieu's points-to analysis [54] and Cheng and Mei's access paths [17] to develop an efficient reference analysis for Java. They show the effectiveness of their field-sensitive and intra-procedural flow-sensitive method by computing precise static call-graphs for very large Java programs.

Pearce et al. [98, 97] propose a field-sensitive points-to analysis for modeling aggregates and function pointers. They find that a field-sensitive analysis is more expensive to compute, but yields significantly better precision over a field-insensitive analysis.

Lattner et al. [75] propose a heap-cloning based context-sensitive points-to analysis. For achieving a scalable implementation, they illustrate several algorithmic and engineering design choices such as, using a flow-insensitive and unification-based analysis, and sacrificing context-sensitivity within strongly connected components.

Sotin and Jeannet [120] address the problem of interprocedural analysis in the presence of pointers to the stack. They use abstract interpretation to define local semantics for their language and then apply relational interprocedural analysis to the local semantics to generate a forward semantics manipulating sets of activation records. Finally, they apply their interprocedural analysis for verifying relational properties on program variables.

Rountev et al. [109] propose points-to analysis for Java using annotated constraints. They use the annotated inclusion constraints to precisely and efficiently model the semantics of virtual calls and the flow of values via object fields.

Fahndrich et al. [35] propose a context-sensitive flow-analysis using instantiation constraints. They apply their analysis to develop a points-to analysis algorithm. They show that flow information can be computed efficiently while considering only the paths with well-defined call-return sequences, even for higher-order programs.

Foster et al. [38] propose the use of annotated type-qualifier *restrict* to specify that certain pointers are not aliased to other pointers within a lexical scope. Aiken et al. [1] extend it to support another annotation called *confine* for restricting expressions, rather than single variables. They also give algorithms to infer restricted variables and confined expressions. They find that the use of annotation can significantly improve the analysis precision and can help in finding several real-world bugs.

Heintze and Tardieu [54] propose a database-centric analysis architecture called compile-link-analyze (CLA) and an algorithm for computing dynamic transitive closure to develop a very fast points-to analysis. Their system is able to analyze about a million lines of unprocessed C code in less than a second without using more than 10 MB of memory.

Hardekopf and Lin [50] propose a semi-sparse flow-sensitive analysis for efficient handling of strong updates. They convert non-address-taken or top-level variables to Static Single Assignment (SSA) form to improve analysis efficiency.

Yu et al. [136] propose a field-sensitive, flow-sensitive and context-sensitive pointer analysis by analyzing pointers according to their levels. Their analysis is fully sparse flow-sensitive which is a generalization of Hardekopf and Lin's semi-sparse flow-sensitivity [50].

Our work on solving points-to analysis as a set of linear equation is an exact (inclusion-based) analysis (Chapter 5).

3.7 Methods Achieving Explicit Trade-offs

In this section, we discuss those works which offer an explicit control over the trade-off between precision and scalability (in terms of analysis time and/or memory requirement).

Ryder [113] discusses several analysis dimensions that affect precision, including flow-sensitivity, context-sensitivity, field-sensitivity, program representation, directionality of information flow. Various choices for implementing these dimensions lead to different trade-offs between precision and scalability. For instance, a flow-sensitive, context-sensitive analysis is more precise and requires more time than its flow-insensitive, context-insensitive counterpart.

In the context of object oriented languages, Milanova et al. [88] propose parameterized *object sensitivity*, which analyzes a method separately for each object name on which that method is invoked. Their parameterization framework offers an explicit control over the trade-off between precision and analysis time by changing the parameters.

Buss et al. [11] propose an analysis space for pointer analysis based on ordering of program statements, modeling of conditionals and handling of strong updates. By choosing various values for these three configurable parameters, they show that one can design an analysis with the desired precision and scale.

In Steensgaard's analysis [123], every pointer has a single outgoing points-to edge for all its pointees, whereas in case of Andersen's analysis [3], a pointer has one outgoing points-to edge for each of its pointees. By choosing `k` outgoing edges between these two extremes, Shapiro and Horwitz [117] propose an algorithm which can be tuned so that its worst-case time and space requirements and its precision range from those of Steensgaard's analysis to those of Andersen's analysis.

Hasti and Horwitz [52] propose an iterative algorithm to make the results of flow-insensitive analysis more and more precise using Static Single Assignment (SSA) form. Depending upon a client requirement, the algorithm can iterate only for a limited number of times to achieve the desired trade-off between analysis time and precision, still guaranteeing a sound result.

The concept of offline variable substitution [108] is widely known for maintaining the precision of the original analysis. However, as suggested by the authors in their paper, it can also be used to offer a trade-off between scalability and precision. This can be done by identifying a set of pointer variables and substituting them with a representative. If all the variables in the set are pointer equivalent, there is no loss in analysis precision. However, by adding pointers to this set which are not pointer equivalent, the number of variables tracked during the analysis can be reduced resulting in a more efficient analysis. This *imprecise* substitution results in some loss of precision. A client, depending upon its requirement, can select an appropriate precision-scalability trade-off.

Approximate pointer analysis [94] identifies pointers with *mostly similar* points-to sets as pointer equivalent and identifies pointees with *mostly similar* pointed-by sets as location equivalent. A client may choose an appropriate similarity threshold to achieve a desired trade-off between analysis precision and scalability.

The work on program decomposition [138] helps an analysis choose different parts of a program to be analyzed with varying levels of precision. Zhang et al. [138] present a program decomposition technique that partitions program statements to allow separate pointer analyses to be used on independent parts of the program. This decomposition enables exploration of trade-off between algorithm efficiency and precision.

Various configuration parameters in our work on points-to analysis using bloom filters (Chapter 4) offer a client an explicit control over the precision-scalability trade-off. By choosing different values for the number of bits in each bucket `B`, the number of context bits `C`, the number of hash functions `D`, etc., the analysis time, memory and precision can be tuned as per the requirement. We also show that with minimal precision loss, which can be probabilistically bounded, our bloom filter based points-to analysis achieves significant reductions in analysis time and memory requirement.

Selection probability or the degree of randomization in our randomized points-to analysis (Chapter 7) also allows for explicitly controlling the scalability-precision trade-off. Further, a client can choose the number of randomized runs as an additional control parameter. Depending upon the values of these configuration parameters, our randomized analysis is able to achieve a significant reduction in analysis time at the cost of a small amount of precision.

Several client-driven analyses adjust their analysis time and precision based on the client needs. We review the work on client-driven analysis in the next section.

3.8 Client-driven and Demand-driven Methods

In this section, we discuss client-driven and demand-driven points-to analyses. A client-driven analysis adjusts its cost and the achieved precision according to the needs of the client analyses (or programs). Thus, if a client $\texttt{C}_1$ requires a high precision, a client-driven points-to analysis can tune its configurable parameters or its algorithm to extract precise points-to information from the program. In contrast, if the goal of another client $\texttt{C}_2$ is a scalable analysis, the same client-driven points-to analysis can configure itself to extract sound points-to information with as less analysis time as possible, at the cost of some precision.

Guyer and Lin [45, 46] propose client-driven pointer analysis for C programs. Their analysis has two passes. The first pass is a fast, low-precision points-to analysis to discover the precision demands of various parts of the program. The second pass uses this information along with the feedback from the client to run a customized precision policy on different parts of the program. Their analysis treats data objects in a flow-sensitive or flow-insensitive manner and procedures in a context-sensitive or context-insensitive manner depending upon the precision policy.

Shapiro and Horwitz's algorithm [117] is an instance of client driven pointer analysis. They propose a points-to analysis whose worst-case time and space requirements and its precision range from those of Steensgaard's analysis [123] to those of Andersen's analysis [3]. This is based on the requirement of a client to choose an appropriate, `k` number of outgoing points-to edges for a pointer. In Steensgaard's analysis [123], every pointer has a single outgoing points-to edge for all its pointees, whereas in case of Andersen's analysis [3], a pointer has one outgoing points-to edge for each of its pointees. By choosing `k` outgoing edges between these two extremes, a client can use their algorithm suitable to its needs.

A client-driven points-to analysis, like a regular points-to analysis, is exhaustive; i.e., it computes the points-to sets for all the pointers in a program. However, often a client is interested only in a subset of the points-to information. A demand-driven points-to analysis computes only the required amount of points-to information to answer *a* particular query of the client. As more queries are processed, a demand-driven analysis computes more and more information on-the-fly. We discuss the work on demand-driven points-to analysis next.

Heintze and Tardieu [53] introduce demand-driven context-insensitive and flow-insensitive points-to analysis. They use deductive reachability formulation to propose a provably optimal demand-driven analysis for C. They observe that the performance of their demand-driven analysis depends heavily on the amount of points-to information that needs to be computed to answer an alias query. Thus, if a query requires only a small amount of points-to information to be computed, then the analysis is very fast. However, if a query requires a large amount of points-to information, then the analysis can be slower than an exhaustive analysis.

Sridharan et al. [122] propose demand-driven points-to analysis for Java. They formulate Andersen's analysis [3] as a control flow language (CFL) reachability problem [106] and show that Andersen's analysis for Java is a balanced-parentheses problem. By exploiting this balanced parentheses structure, they obtain an asymptotically faster analysis. Their analysis allows a client to set a time-budget for answering a query, terminating the query once the time-budget is exceeded. They show that their algorithm yields much higher precision than previous techniques within small time-budgets.

Sridharan and Bodik [121] build upon their previous work [122] to propose a demand-driven, client-driven refinement-based context-sensitive points-to analysis for Java. Their technique simultaneously refines handling of method calls and heap accesses allowing the analysis to precisely analyze important code, skipping irrelevant code. One of the major contributions of their work is to develop an inclusion-based context-sensitive points-to analysis that has context-sensitive call-graph and context-sensitive heap abstraction, and is shown to scale for large programs.

Zheng and Rugina [140] propose a demand-driven alias analysis for C. Similar to the work on demand-driven points-to analysis for Java [122, 121], they also formulate the computation of (alias) queries as a CFL-reachability problem. The aliasing relations in their analysis can be described using two, mutually dependent, hierarchical state machines, one for memory aliases and the other for value aliases. A useful aspect of their approach is that it does not require building or intersecting points-to sets. Their technique has been shown to be very efficient in practice, which makes it a good candidate for interactive tools.

3.9 Incremental and Probabilistic Methods

In this section, we discuss the work on incremental points-to analysis and probabilistic points-to analysis. An incremental analysis allows for dynamic addition and/or deletion of a set of statements to the already analyzed program and processes the statements without having to analyze the complete program (original program plus new statements) from scratch. An incremental points-to analysis allows for on-the-fly addition or deletion of a points-to constraint over existing constraints with pre-computed points-to information for the existing constraints.

Yur et al. [137] propose an incremental flow-sensitive and context-sensitive points-to analysis algorithm to handle addition and deletion of single statements in a C program. For an incremental change, their worklist-based method identifies the affected region and updates the interprocedural control flow graph to reflect the change. As a next step, their method adds the relevant aliases onto the worklist which is reiterated to find the final aliasing solution.

Saha and Ramakrishnan [114] describe a framework based on logic programming for implementing various incremental and demand-driven program analyses formulated using deductive rules. They instantiate their framework for an incremental and demand-driven points-to analysis.

A definite or a non-probabilistic points-to analysis computes points-to information which may or must hold at various program points during the execution of the program. Such an analysis does not quantify the certainty with which a points-to fact would hold at a program point. A probabilistic points-to analysis, in contrast, assigns a probability with each points-to fact computed. The result of such an analysis can help optimize the runtime execution of a program, e.g., speculative execution of such a program can make intelligent decisions based on the likelihood of a points-to fact.

Hwang et al. [60, 16] introduce probabilistic points-to analysis. To identify the probabilities with which each points-to fact is generated and preserved, their approach first computes the transfer functions for probabilistic data-flow analysis. The probability of each points-to fact is then computed using the transfer functions. Chen et al. [15] use the above probabilistic points-to analysis for speculative multithreading.

Silva and Steffan [23] propose a one-level flow-sensitive and context-sensitive probabilistic points-to analysis by encoding linear transfer functions as sparse matrices. They demonstrate that, even without edge-profiling information, their analysis can provide accurate probabilities for the points-to facts.

In the context of object-oriented programs, Sun et al. [124] propose probabilistic points-to analysis for Java. In contrast to the former work in the context of C programs, their work handles object-oriented features such as inheritance and polymorphism.

3.10 Analysis of Parallel Programs and Parallel Analyses

In this section, we discuss the work related to pointer analysis of multithreaded programs and parallel versions of pointer analysis itself. Due to numerous thread-interleavings possible in a multithreaded program, the analysis of such programs poses severe challenges from precision and scalability perspectives.

Rugina and Rinard [111, 112] propose an interprocedural, context-sensitive and flow-sensitive pointer analysis for multithreaded programs. Their method extracts thread interference to take into account the shared pointers accessed by parallel threads.

Salcianu and Rinard [116] propose a combined pointer and escape analysis for multithreaded programs. Their algorithm uses interaction graphs to analyze the interactions between threads and is compositional, i.e., it analyzes each method or thread once to extract a parameterized analysis result that can be specialized in a context.

There has been some work on parallelizing the pointer analysis algorithm itself. While some of the former approaches simply mention that their algorithms could be parallelized [66, 138, 110], the first parallel pointer analysis is proposed by Mendez-Lojo et al. [86]. They illustrate that inclusion-based points-to analysis can be formulated entirely in terms of graphs and graph-rewrite rules. Their algorithm exposes the amorphous data-parallelism in irregular applications.

Edvinsson et al. [30] propose parallel points-to analysis for object oriented programs. It deals with different target methods of polymorphic function calls and independent control flow branches.

3.11 Application of Points-to Analysis

Pointer analysis is not an optimization; a client needs to use the computed points-to information for performing an optimization over the program. In this section, we discuss various clients which have been shown to make use of pointer analysis.

Livshits and Lam [80] propose an extended form of SSA, called IPSSA, to track pointers and apply it for finding buffer overruns and format string violations in C programs.

Milanova et al. [87] apply a pointer analysis [138] for precise call-graph construction. They find that for call-graph construction as a client, an inexpensive pointer analysis may provide precise enough information.

Wu et al. [133] propose element-wise points-to mapping for loop-based dependence analysis. An element-wise points-to mapping summarizes the relation between a pointer and the heap object it points to, for every instance of the pointer inside a loop and for every array element directly accessible through the pointer. They demonstrate that element-wise points-to information can significantly improve the precision of loop-based dependence analysis.

Avots et al. [4] use a context-sensitive, field-sensitive points-to analysis to detect security vulnerabilities in C programs. By assuming a restricted, but common usage C syntax, they improve the pointer analysis precision. They show that their optimistic pointer analysis can be used to reduce the overhead of a dynamic string-buffer overflow detector.

Buss et al. [11] propose an analysis space for pointer analysis based on ordering of program statements, modeling of conditionals and handling of strong updates. By choosing various values for these three configuration parameters, they show that one can design an analysis with the desired precision and scale. They apply the developed points-to analyses for bug finding and show that the precision of the underlying points-to analysis directly affects the precision of the bug finding tool.

Mock et al. [90] apply dynamic points-to information to improve the precision of static program slicing for C. They find that programs with many call sites that make calls through function pointers experience a significant reduction in slice size when dynamic points-to information is used. However, for other programs which do not make much use of function pointers for calling functions, there is little reduction in slice size.

Ghiya and Hendren [41] use pointer information to develop a shape analysis for C programs. Their analysis can detect if a data structure has a tree-like structure, a directed acyclic graph or whether it is cyclic.

Tonella et al. [125] apply their flow-insensitive and context-insensitive points-to analysis for C++ to reaching definitions analysis and slicing.

Guyer and Lin [45, 46] apply a client-driven pointer analysis for C programs to several error detection problems. Their analysis has two passes. The first pass is a fast, low-precision points-to analysis to discover the precision demands of various parts of the program. The second pass uses this information along with the feedback from the client to run a customized precision policy on different parts of the program. Their analysis treats data objects in a flow-sensitive or a flow-insensitive manner and procedures in a context-sensitive or a context-insensitive manner depending upon the precision policy. They claim that typical clients need a small amount of extra precision applied to selected parts of each program and one can trade off precision for scalability for the remaining parts.

Silva and Steffan [23] propose a one-level flow-sensitive and context-sensitive probabilistic points-to analysis for speculative optimizations.

Orso et al. [95] classify data dependences in the presence of pointers and then make use of the classification for data-flow testing and to develop an incremental slicing algorithm.

Hind and Pioli [57] compare the effect of various pointer analyses on different clients including Mod/Ref analysis, live variable analysis, reaching definitions analysis, conditional constant propagation and dead code elimination.

In Chapter 4, we apply our bloom filter-based points-to analysis for Mod/Ref analysis. We find that the effect of false positives incurred by our approximate representation is very less on the client.

3.12 Evaluations and Quantifications

In this section, we discuss the work which focus on the experiments to obtain insights about the usage of pointers in programs and properties of a pointer analysis.

Hackett and Aiken [47] study four applications to identify common aliasing patterns that arise in practice. They find that almost all pointers are used as one of the following nine use-cases: *parent pointers* are references to data closer to the root of a data structure, *child pointers* are references to data stored deeper in a data structure, *shared immutable pointers* are multiple references to the data and all are used only for reading, *shared I/O pointers* are two references to the data where one is used only for writing and the other only for reading, *global pointers* are references to the globals, *index cursors* to support an additional index for a structure, *tail cursors* to hold the end point of an index, *query cursors* to read data internal to an index, and *update cursors* to write data internal to an index.

Lhotak and Hendren [77] present a framework of BDD-based context-sensitive points-to analyses for Java. Using their framework, they evaluate the precision of various context-sensitive analyses. Two of their main findings are (i) object-sensitive analyses are more precise than comparable variations of other approaches, and (ii) context-sensitive heap-abstraction improves precision more than extending the length of the context string.

Hind and Pioli [56, 57] compare different points-to analyses on C programs. The analyses vary in their use of control-flow information and their work quantifies the effect of varying flow-sensitivity on the analysis performance in terms of analysis time and precision. They also report their findings on how the points-to information computed by each of the analyses affects different clients, namely, Mod/Ref analysis, live variable analysis, unreachable code identification, reaching definitions analysis, dependence analysis, and conditional constant propagation. One of the main findings of their study is that a flow-sensitive pointer analysis offers only a small amount of additional precision over a flow-insensitive analysis [57]. They also find that the time and space efficiency of a client analysis improves as the pointer analysis precision is increased.

Das et al. [25] estimate the impact of scalable flow-insensitive and context-insensitive analyses on compiler optimizations. Their major finding is that limited forms of context-sensitivity and subtyping provide the same precision as algorithms with full context-sensitivity and subtyping.

Zhang et al. [139] present results of their combined analysis which uses program decomposition to apply different aliasing to independent program segments. They find that combined analysis allows application of a flow-sensitive analysis to segments of a program which is too large to be analyzed by a flow-sensitive analysis as a whole. They also find that points-to analysis is more efficient than an alias analysis.

Mock et al. [91] evaluate the degree of imprecision caused by static pointer analysis for C programs with respect to the actual behavior of pointers at run-time. They find that the pointer information produced by existing scalable static pointer analyses is far worse than the actual behavior observed at run-time. They advocate usage of profile data on pointer values to improve analysis precision.

Mock et al. [90] apply dynamic points-to information to improve the precision of static program slicing for C. They find that programs with many call sites that make calls through function pointers experience a significant reduction in slice size when dynamic points-to information is used. However, for other programs which do not make much use of function pointers for calling functions, there is little reduction in slice size.

Diwan et al. [28] evaluate three alias analyses based on programming language types for Modula-3. They find that type-compatibility alone yields a very imprecise alias analysis. However, if it is coupled with field-sensitivity, it significantly improves precision.

In the context of Java, Liang et al. [79] evaluate the precision of static reference analysis using profiling information. They demonstrate that modeling heap allocated memory with the allocation site may be sufficiently precise for most allocation sites. They also find that static Andersen's analysis [3] can compute very precise information for some allocation sites, but can also compute very imprecise information for many allocation sites. Further, they illustrate that existing approaches may compute very imprecise points-to information for programs that use sophisticated data structures.

Ribeiro and Cintra [107] also investigate the sources of uncertainty in the points-to information computed by a static analysis. Their approach also makes use of the profiling information, but in contrast to the other works in this direction, they wish to quantify the amount of uncertainty that is intrinsic to the applications. They find that often static pointer analysis is very accurate, but for some benchmarks a significant fraction, up to 25%, of their accesses via pointer-dereferences cannot be statically disambiguated. They claim that the main reasons behind this behavior is the use of pointer arithmetic and the fact that some control paths are not taken.

3.13 Points-to Analysis for Other Languages

In this section, we discuss various points-to analysis algorithms for other languages such as Java and Python.

Apart from C, one of the languages for which pointer analysis is developed is Java [63]. Since the pointers in Java are called references, the analysis is often termed as reference analysis. We mention the work in the context of Java below. Berndl et al. [6], Whaley and Lam [129] and Zhu and Calman [142] propose variants of points-to analysis algorithms using BDDs for Java. Sridharan et al. [122] propose demand-driven points-to analysis. Sridharan and Bodik [121] build upon their previous work [122] to propose a demand-driven, client-driven refinement-based context-sensitive points-to analysis. Sun et al. [124] propose probabilistic points-to analysis for object-oriented programs. Lhotak and Hendren [77] present a framework of BDD-based context-sensitive points-to analyses. Liang et al. [79] evaluate the precision of static reference analysis using profiling information.

Gorbovitski et al. [43] propose alias analysis for a dynamic object-oriented language, for program optimization by incrementalization and specialization. Incrementalization is a language optimization for reducing the cost of expensive queries. Specialization is an optimization technique for generic code. They instantiate their flow-sensitive and context-sensitive analysis for Python [102].

Tonella et al. [125] propose a flow-insensitive and context-insensitive points-to analysis for C++, which handles various object-oriented features like polymorphism, templates and dynamic binding. They show its effectiveness by applying it to reaching definitions analysis and slicing.

In contrast to traditional languages like C and Java, JavaScript [64] has dynamic features such as run-time modification of objects through addition of properties and updating of methods. Jang and Choe [62] propose the first points-to analysis for JavaScript. Their analysis can identify the use of a structure's field directly or via a property, to improve precision over a traditional Andersen's analysis [3].

Jovanovic et al. [65] propose a precise alias analysis for server-side scripting languages like PHP [101]. They target their analysis towards the unique reference semantics commonly found in scripting languages and apply it to detect web application vulnerabilities.

While the above work is related to dynamic languages, there has also been some work on dynamic pointer analysis, i.e., performing pointer analysis while the program is running. Salami and Valero [115] propose a dynamic interprocedural pointer analysis for multimedia applications using a memory disambiguation technique called dynamic memory interval test. Hirzel et al. [58] propose online pointer analysis for Java.

Since most alias analyses are formulated in terms of high-level language features, they cannot easily handle features such as pointer arithmetic and out-of-bound array references. To handle these issues, Debray et al. [26] propose alias analysis of executable code. In order to be practical, their algorithm trades off precision for memory requirement. They show that their analysis is able to provide non-trivial information about 30% to 60% of the memory references.

3.14 Chapter Summary

In this chapter we presented a survey of various pointer analysis methods. We classified the analyses in 12 categories and briefly discussed the work in each category.

References

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 129–140, New York, NY, USA, 2003. ACM.

[2] N. Alon, B. Awerbuch, and Y. Azar. The online set cover problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 100–105, New York, NY, USA, 2003. ACM.

[3] L. O. Andersen. Program analysis and specialization for the C programming language, PhD Thesis, DIKU, University of Copenhagen, 1994.

[4] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a c pointer analysis. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 332–341, New York, NY, USA, 2005. ACM.

[5] B. Awerbuch, Y. Azar, A. Fiat, and T. Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time (extended abstract). In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 519–530, New York, NY, USA, 1996. ACM.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[9] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(4):401 –424, apr 1994.

[10] M. G. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 234–250, London, UK, 1995. Springer-Verlag.

[11] M. Buss, D. Brand, V. Sreedhar, and S. A. Edwards. A novel analysis space for pointer analysis and its application for bug finding. *Sci. Comput. Program.*, 75:921–942, November 2010.

[12] V. T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 115–125, New York, NY, USA, 2003. ACM.

[13] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. ACM.

[14] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of points-to analysis of java in the presence of exceptions. *IEEE Trans. Softw. Eng.*, 27:481–512, June 2001.

[15] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *SIGPLAN Not.*, 38:25–36, June 2003.

[16] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Interprocedural probabilistic pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15:893–907, October 2004.

[17] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN*

2000 conference on Programming language design and implementation, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[18] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[19] W. Choi and K.-M. Choe. Cycle elimination for invocation graph-based context-sensitive pointer analysis. *Inf. Softw. Technol.*, 53:818–833, August 2011.

[20] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms, McGraw Hill, 2001.

[22] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[23] J. Da Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 416–425, New York, NY, USA, 2006. ACM.

[24] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.

[25] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 260–278, London, UK, 2001. Springer-Verlag.

[26] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of*

the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[27] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 230–241, New York, NY, USA, 1994. ACM.

[28] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 106–117, New York, NY, USA, 1998. ACM.

[29] N. Dor, M. Rodeh, and M. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 155–167, New York, NY, USA, 2003. ACM.

[30] M. Edvinsson, J. Lundberg, and W. Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM.

[31] M. Emami. A practical inter-procedural alias analysis for an optimizing/paralleling c compiler, Master thesis, School of Computer Science, McGill University, 1993.

[32] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[33] J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *J. ACM*, 57:33:1–33:47, November 2010.

[34] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[35] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.

[36] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8:281–293, June 2000.

[37] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proceedings of the Third International Symposium on Static Analysis*, pages 189–204, London, UK, 1996. Springer-Verlag.

[38] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM.

[39] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 345–354, New York, NY, USA, 2003. ACM.

[40] GCC, http://gcc.gnu.org/.

[41] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 1–15, New York, NY, USA, 1996. ACM.

[42] Gnu mp integer library, http://gmplib.org/.

[43] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM.

[44] L. L. Gremillion. Designing a bloom filter for differential file access. *Commun. ACM*, 25:600–604, September 1982.

[45] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[46] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58:83–114, October 2005.

[47] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 69–80, New York, NY, USA, 2006. ACM.

[48] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[49] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2007.

[50] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[51] Ben hardekopf, http://www.cs.utexas.edu/users/benh/.

[52] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 97–105, New York, NY, USA, 1998. ACM.

[53] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 24–34, New York, NY, USA, 2001. ACM.

[54] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming*

language design and implementation, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[55] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21:848–894, July 1999.

[56] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the 5th International Symposium on Static Analysis*, SAS '98, pages 57–81, London, UK, 1998. Springer-Verlag.

[57] M. Hind and A. Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[58] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29, April 2007.

[59] S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19:1–6, January 1997.

[60] Y.-S. Hwang, P.-S. Chen, J. K. Lee, and R. D.-C. Ju. Probabilistic points-to analysis. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 290–305, Berlin, Heidelberg, 2003. Springer-Verlag.

[61] Ilog toolkit, http://www.ilog.com/.

[62] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1930–1937, New York, NY, USA, 2009. ACM.

[63] Java programming language, http://www.java.com/.

[64] Javascript programming language, http://www.javascript.com/.

[65] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[66] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[67] A. Kanamori and D. Weise. Worklist management strategies for dataflow analysis, MSR Technical Report, MSR-TR-94-12, 1994.

[68] U. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78791-4_15.

[69] D. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1997.

[70] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1:323–337, December 1992.

[71] W. Landi. Interprocedural aliasing in the presence of pointers, PhD thesis, Rutgers University, 1992.

[72] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[73] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[74] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 56–67, New York, NY, USA, 1993. ACM.

[75] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN*

conference on Programming language design and implementation, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[76] O. Lhotak. A tour of pointer analysis, Summer School on Theory and Practice of Language Implementation, University of Oregon, 2009.

[77] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18:3:1–3:53, October 2008.

[78] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. *SIGSOFT Softw. Eng. Notes*, 24:199–215, October 1999.

[79] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 22–32, New York, NY, USA, 2002. ACM.

[80] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 317–326, New York, NY, USA, 2003. ACM.

[81] The LLVM compiler infrastructure, http://llvm.org.

[82] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 149–159, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[83] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett.*, 50:191–197, May 1994.

[84] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 196–212, London, UK, 2002. Springer-Verlag.

[85] V. Martena and P. S. Pietro. Alias analysis by means of a model checker. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 3–19, London, UK, 2001. Springer-Verlag.

[86] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM.

[87] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11:7–26, January 2004.

[88] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005.

[89] M. Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pages 144–150, New York, NY, USA, 2001. ACM.

[90] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27:71–80, November 2002.

[91] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 66–72, New York, NY, USA, 2001. ACM.

[92] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.

[93] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 67–80, New York, NY, USA, 2000. ACM.

[94] R. Nasre. Approximating inclusion-based points-to analysis. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 66–73, New York, NY, USA, 2011. ACM.

[95] A. Orso, S. Sinha, and M. J. Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. *ACM Trans. Softw. Eng. Methodol.*, 13:199–239, April 2004.

[96] A. Partow. General purpose hash function algorithms, http://www.partow.net/programming/hashfunctions/.

[97] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30, November 2007.

[98] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for c. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 37–42, New York, NY, USA, 2004. ACM.

[99] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Control*, 12:311–337, December 2004.

[100] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 126–135, Washington, DC, USA, 2009. IEEE Computer Society.

[101] Php: Hypertext preprocessor, http://www.php.net/.

[102] Python programming language, http://www.python.org/.

[103] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16:1467–1471, September 1994.

[104] V. Raman. Pointer analysis – a survey, CS203 UC Santa Cruz, 2004. http://www.soe.ucsc.edu/ vishwa/publications/Pointers.pdf.

[105] D. Rayside. Points-to analysis, 2005. http://www.cs.washington.edu/homes/mernst/teaching/6.883/lectures/points-to.pdf.

[106] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[107] C. G. Ribeiro and M. Cintra. Quantifying uncertainty in points-to relations. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 190–204, Berlin, Heidelberg, 2007. Springer-Verlag.

[108] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.

[109] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. *SIGPLAN Not.*, 36:43–55, October 2001.

[110] E. Ruf. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 15–26, New York, NY, USA, 1997. ACM.

[111] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM.

[112] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25:70–116, January 2003.

[113] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

[114] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis

using logic programming. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005. ACM.

[115] E. Salamí and M. Valero. Dynamic memory interval test vs. interprocedural pointer analysis in multimedia applications. *ACM Trans. Archit. Code Optim.*, 2:199–219, June 2005.

[116] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[117] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[118] O. G. Shivers. Control-flow analysis of higher-order languages, PhD Thesis, Carnegie Mellon University, 1991.

[119] SML, http://en.wikipedia.org/wiki/Standard_ML.

[120] P. Sotin and B. Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, pages 459–479, Berlin, Heidelberg, 2011. Springer-Verlag.

[121] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[122] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[123] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[124] Q. Sun, J. Zhao, and Y. Chen. Probabilistic points-to analysis for java. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 62–81, Berlin, Heidelberg, 2011. Springer-Verlag.

[125] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 433–443, New York, NY, USA, 1997. ACM.

[126] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[127] J. Whaley. Program Analysis using BDDs, Talk at MIT, 2005.

[128] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 180–195, London, UK, 2002. Springer-Verlag.

[129] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[130] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[131] Static program analysis, http://en.wikipedia.org/wiki/Static_program_analysis.

[132] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[133] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 262–273, New York, NY, USA, 2002. ACM.

[134] Q. Wu. Survey of alias analysis, http://www.cs.princeton.edu/ jqwu/Memory/survey.html.

[135] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.

[136] H. Yu, 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 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 218–229, New York, NY, USA, 2010. ACM.

[137] J.-s. Yur, B. G. Ryder, and W. A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM.

[138] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: a step toward practical analyses. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '96, pages 81–92, New York, NY, USA, 1996. ACM.

[139] S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with combined analysis for pointer aliasing. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '98, pages 11–18, New York, NY, USA, 1998. ACM.

[140] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[141] J. Zhu. Symbolic pointer analysis. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 150 – 157, nov. 2002.

[142] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.