Rethinking Soot for Summary-Based Whole-Program Analysis

Dacong Yan
Ohio State University
yan@cse.ohio-state.edu

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Abstract

Whole-program static analysis has been extensively studied and widely used in the past few decades. For modern object-oriented programs, scalability has become an important issue for using whole-program analysis in real-world tools. In addition, the ever-increasing size of libraries (e.g., the JDK library) and frameworks exacerbates the scalability problems. To achieve the desired level of analysis performance, an effective approach could be to generate and apply analysis summary information for library methods. In this paper, such an approach is referred to as a *summary-based whole-program analysis*. The challenges for this technique are two-fold: (1) carefully designed abstractions and algorithms are needed to create and use client-independent and analysis-specific library summary information; and (2) support for summary generation and application should be effectively incorporated into existing analysis infrastructures.

This paper focuses on the second challenge. It uses Soot, a widely-used program analysis framework for Java, as a vehicle to explore some of the important issues in providing analysis infrastructures with capabilities for summary-based analysis. Experimental studies are presented to show that significant savings can potentially be achieved by making a whole-program alias analysis summary-based. To actually achieve these savings, the paper proposes to extend Soot to add support for summary-based analysis. Finally, a brief discussion of the required framework extensions is presented.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, measurement, experimentation

Keywords Summary, whole-program static analysis, library

1. Introduction

Whole-program static analysis is essential for a large variety of tasks in the fields of programming languages and software engineering. There has been a great deal of work on this topic over the past few decades. Modern object-oriented languages, with their complicated interprocedural flow of control, sophisticated data structures, and extensive use of reusable components, present a particularly important target for whole-program analysis.

The scalability of whole-program analysis for large programs can be a significant hurdle for practical use in real-world tools. Even for programs that appear to be relatively small, the underlying library code can contribute thousands of methods to the static call graph and to the scope of the whole-program analysis. For example, for each program analyzed in our experiments, more than 50% of the nodes in the call graph are methods in the Java standard libraries; for most of these programs the percentage exceeds 80%. Motivated by such observations, we argue that a worthwhile goal for designers/implementors of whole-program analyses (and analysis infrastructures such as Soot [32, 35]) is to provide support for preanalysis of reusable components. Examples of such components include (1) standard libraries which form an integral part of modern languages, (2) domain-specific libraries, and (3) powerful extensible frameworks.

These large libraries and frameworks present a big challenge for the scalability of whole-program static analyses. Analyzing libraries is an expensive and repetitive task. In order for whole-program analyses to work correctly, they should consider all code that is potentially reachable at run time, which, of course, includes code in the libraries. This means that library methods are analyzed repeatedly in the context of different client programs. Even for the same client program, it is often the case that the whole-program analysis has to be rerun on a slightly modified version of the program (e.g., to perform change impact analysis), which typically includes reanalyzing the same library methods.

It is not always necessary to repeatedly perform these expensive analyses on library methods. Depending on the underlying whole-program analysis and on the properties of the library code, it may be possible to perform a preanalysis of the library code once, independently of any client programs, in order to produce *library summary information*. This information can then be reused when analysis of a specific client program is needed. We will refer to such an approach as *summary-based whole-program analysis*. Such analysis could become more scalable, compared to traditional whole-program analysis. Furthermore, for certain categories of analyses that refine an initial imprecise solution until a client-defined time budget is exhausted (e.g., [33]), the reduced cost could result in a more refined (i.e., more precise) analysis solution under the same time budget.

Challenges Support for summary-based analysis involves two important challenges. First, the ability to create and use client-independent library summary information presents a number of technical challenges in terms of analysis abstractions and algorithms. There is a body of work on various approaches to address these challenges (e.g., [8, 11, 23, 27, 28, 39]). A second, very significant problem is the need for infrastructure support to seamlessly apply precomputed summary information. In this paper we focus our discussion on the second problem. The specific question we are interested in is the following: given an analysis infrastructure for whole-program analysis, and several particular analyses built on top of it, how should summary-based versions of these analyses be implemented? Typically, the solution is to add ad hoc extensions to the existing infrastructure. There are several problems with this approach: (1) code changes, spreading everywhere in the infrastructure, are difficult to maintain; (2) extensions from different analysis developers, even for the same kind of analysis, can differ significantly in key design components (e.g., related to encoding and storing of summary information); and (3) the achievable benefits by making the analysis summary-based are limited due to the lack of inherent support in the analysis infrastructure.

Goals This paper uses Soot [32, 35], a widely-used program analysis framework for Java, as a vehicle to explore some of the issues in providing frameworks with capabilities for summary-based analysis. First, we perform a case study on a whole-program alias analysis [38]. This analysis exhibits features typical for many other whole-program analyses—for example, traversals of intraprocedural and interprocedural program representations, as well as usage of intermediate data structures produced and consumed by different stages of the algorithm. We demonstrate the potential for savings that would be achieved by adding support for library summary information. Next, we briefly discuss the major Soot components that are important for obtaining benefits from summarization. In summary, the goals of this paper are to present:

- A case study of a Soot-based alias analysis for Java;
- An evaluation of the potential benefits of using library summaries in the alias analysis;
- A discussion of the components of Soot that need to be considered to make the framework more amenable to summary-based whole-program analysis.

2. Case Study: An Alias Analysis

Alias analysis is one of the most commonly used whole-program analyses. As such, it can benefit from careful design and implementation to handle large, complicated libraries and frameworks. As a case study, this section examines an alias analysis originally proposed in our previous work [38]. This analysis formulates a demand-driven analysis algorithm for Java as a CFL-reachability problem. There are two stages in this approach: (1) construction of an abstracted program representation, and (2) CFL-reachability computation on this representation.

2.1 Construction of an Abstracted Program Representation

The analysis uses the context-insensitive call graph constructed by Soot's Spark component [21] as a starting point to build a program representation referred to as an interprocedural symbolic points-to graph (ISPG). To build the ISPG, symbolic points-graphs (SPGs) for individual methods are first constructed through purely intraprocedural analysis. An SPG for a method is a locally-resolved points-to graph, containing all points-to facts that can be obtained within a method, and using placeholder nodes for unknown external objects. Then, the intraprocedural SPGs are connected together to form the ISPG. This is done by examining the method invocation statements in each method, and adding ISPG edges to represent calls and returns.

Each intraprocedural symbolic points-to graph has the following types of nodes and edges: (1) local variables $v \in \mathcal{V}$; (2) allocation nodes $o \in \mathcal{O}$; (3) symbolic nodes $s \in \mathcal{S}$ representing outside objects; (4) edges $v \rightarrow o_i \in \mathcal{V} \times \mathcal{O}$ representing that variable v may point to allocation node o_i; (5) edges $v \rightarrow s_i \in \mathcal{V} \times \mathcal{S}$ representing that variable v may point to an object defined outside of the current method, with the symbolic node s_i used as a placeholder for that object; (6) edges $o_i \xrightarrow{f} o_j \in (\mathcal{O} \cup \mathcal{S}) \times \mathit{Fields} \times (\mathcal{O} \cup \mathcal{S})$ representing that the field f of o_i may point to o_j.

The majority of the cost for building this abstracted program representation comes from building the intraprocedural SPGs. However, the local nature of this construction presents opportunities for optimization. For example, the intraprocedural SPGs for library methods can be precomputed only once and stored as summary information. For any whole-program alias analysis performed later, this summary information can be applied to avoid reconstructing SPGs for these library methods. Measurements of potential savings from this approach are reported in Section 3.

Of course, this technique is not restricted only to the particular alias analysis discussed here. Many other forms of intermediate data structures are used in various static analyses; typical examples are control-flow graphs and dependence graphs. In many cases these data structures have an intraprocedural component which, for a particular library method, does not depend on the rest of the program. Sometimes even interprocedural components of these data structures can be precomputed in advance for library classes (e.g., when call sites in the library can be fully and precisely resolved without any information about the client code [27]). It is natural to consider an approach that precomputes these data structures for library methods, stores the results in some compressed format in summary files, and retrieves them on demand when whole-program analyses are performed.

2.2 CFL-Reachability Computation

Once the ISPG for a whole program is built, alias queries can be answered by graph traversals to perform single-source-single-target CFL-reachability computations. For a query on two local variables $x, y \in \mathcal{V}$, the analysis considers any nodes $o_x, o_y \in (\mathcal{O} \cup \mathcal{S})$ that they point to, respectively. Starting from one of the nodes (e.g., o_x), the analysis traverses the graph to see whether there is a path to the other node (e.g., o_y). The edge labels along the path should have certain matching parentheses properties (as described by a context-free language over these labels), in order to model context sensitivity and field sensitivity.

Among all paths in the graph, consider the subset of paths consisting of only nodes from the intraprocedural SPG of the same method m or of m's callees (including transitive callees). Furthermore, suppose that the call and returns on these paths are balanced. These paths, representing flows starting from calls into a m and ending at exits from m, will be referred as *partial balanced paths* for m. For methods that are invoked many times (i.e., methods with many incoming edges in the static call graph), their partial balanced paths might be traversed repeatedly in the alias analysis, as part of the traversal of some other longer paths. To avoid such repeated work, the partial balanced paths can be computed in advance and saved as summary information. During graph traversals performed by a summary-based whole-program analysis, the summary information can be applied to eliminate redundant graph retraversal and reduce analysis running time. Experimental results related to this technique are reported in Section 3.

A similar approach has been (or could be) applied to many other whole-program analyses: partial analysis results are computed intraprocedurally, or in some cases even interprocedurally, and are stored for subsequent reuse during the analysis. Given such an analysis algorithm, that natural next step could be taken: partial results for library methods could be precomputed in advance, independently of any client code, and could be used for creating summary information.

3. Evaluation

The alias analysis described in Section 2 was implemented using the Soot 2.3.0 program analysis framework for Java. To be consistent with our previous work [38], the Sun JDK 1.3.1_01 library was used in all studies. Experiments were performed on a machine with

Benchmark	Methods	Libraries
compress	2352	2283 (97.1%)
db	2362	2296 (97.2%)
jack	2617	2299 (87.8%)
javac	3537	2382 (67.3%)
jess	2782	2317 (83.3%)
mpegaudio	2547	2291 (89.9%)
mtrt	3501	3297 (94.2%)
sootc	4616	2343 (50.8%)
sableccj	9013	7165 (79.5%)
jflex	4094	3598 (87.9%)
muffin	4419	3786 (85.7%)
jb	2453	2312 (94.3%)
jlex	2445	2314 (94.6%)
java_cup	2637	2328 (88.3%)
polyglot	2350	2309 (98.3%)
antlr	3043	2317 (76.1%)
bloat	5063	2566 (50.7%)
jython	4142	2475 (59.8%)
ps	5532	4688 (84.7%)

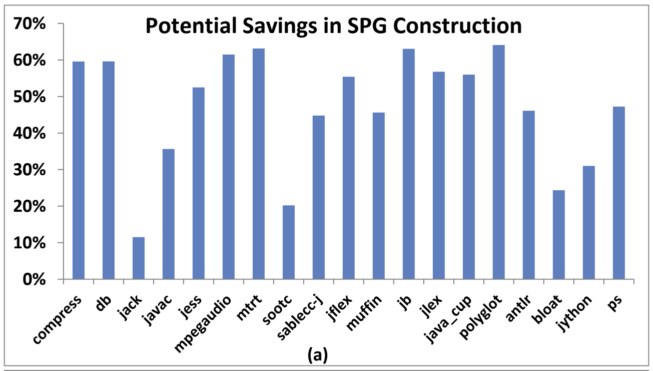
Table 1. Characterization of the context-insensitive call graph.

a 3.4GHz Intel i7-2600 processor. A total of 19 Java programs were studied in our experiments.

Methodology To simulate how a client analysis would use the alias analysis, we implemented a simple data dependence client that performs alias queries regarding variables x and y for all pairs of heap dereference statements that access the same field f (i.e., $x.f$ and $y.f$) where at least one statement writes to f. If x and y may alias, the two statements may access the same heap location and thus may have a data dependence. All possible pairs of heap dereference statements are considered so that the alias queries used for evaluation are not biased. Dependence analysis is an important component of many software analyses and tools, and presents an interesting client for an alias analysis.

Table 1 shows characteristics of the context-insensitive call graphs built by Spark [21] for the studied programs. Column "Methods" shows the total number of nodes in the call graph. Column "Libraries" shows the number of nodes representing methods from the Java standard library. The percentage of these nodes in the call graph is given in parentheses. This percentage is higher than 50% for each program, and exceeds 80% for 13 out of the 19 programs. These large numbers of library methods indicate that the benefits of using summary information for these library methods could be significant, and warrant more in-depth investigation.

Potential Savings in SPG Construction The first set of experiment measures the running time reduction that could potentially be achieved when the SPGs for library methods are precomputed as summary information. For each program, we measure (1) the time taken to compute the SPGs for all library methods in the call graph, and (2) the total time to build the ISPG. When the ratio between these two numbers is high, the savings from summary-based analysis in SPG construction could be substantial. Figure 1(a) shows these ratios for each program. These measurements provide upper bounds on the savings that could be achieved by an implementation of a summary-based analysis. In an actual analysis, additional costs will be present—e.g., to read summaries from disk and to create the necessary data structures in memory. As discussed later, efficient implementation of such enabling functionality is a desirable addition to an analysis infrastructure such as Soot. The measurements indicate significant potential for savings, and motivate further studies of summary-based analyses and the necessary enabling features of the underlying infrastructure.



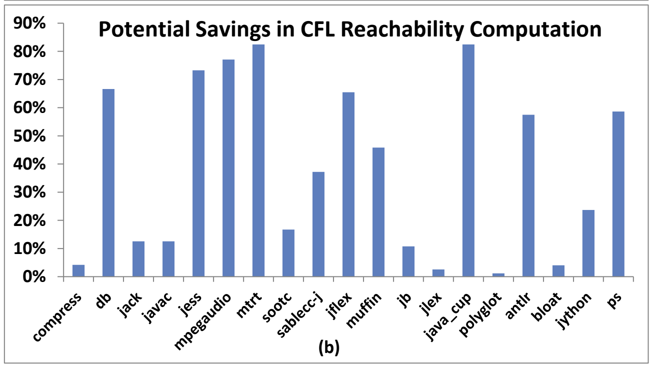


Figure 1. Potential for running time reduction: (a) SPG construction, and (b) CFL-reachability computation.

Potential Savings in CFL Reachability Computation In the second experiment, we measure the potential for time reduction when the balanced partial paths for library methods are precomputed and applied as summary information. These paths are first computed and remembered (in memory), based on the whole-program ISPG. The alias analysis is then run in "summary mode": whenever the graph traversal is about to enter a library method, the summary for that method is applied. As a result, repeated traversals of the same library method are avoided. Figure 1(b) compares the total running times of the alias analysis with and without the use of summary information. As with SPG construction, these measurements indicate promising upper bounds on potential improvements, and motivate additional investigation into the design and efficient implementation of a summary-based version of the analysis.

Although the average potential improvements are high, this is not uniform across programs. For programs that exhibit low potential savings, we manually examined some of the code and the alias queries. Multiple factors that could affect the analysis running time were considered. These factors include the percentage of alias queries that require analysis of library methods, the number of times each library method has to be analyzed when summary information is not available, the cost of analyzing each library method, and so on. The investigation indicated that there is not one single factor that strongly correlates with the measurements. It remains an open question to find a composite metric (i.e., a combination of several individual factors) that can provide a good estimation of the benefits of summarization. Such metrics can guide analysis developers when designing summary-based analyses.

Other Potential Savings In addition to SPG construction and traversals, other components of the overall analysis cost could potentially be reduced with the use of summary information. The first possible target is the generation of the Jimple intermediate representation in Soot. Both for this specific alias analysis and for other whole-program analyses, if summary information is already available for the library methods, is it absolutely necessary to pay the cost of constructing their complete Jimple representation? Is it possible to employ a simpler, more lightweight version of Jimple for such methods, or even to completely circumvent Jimple creation? In a complex system such as Soot, with a large number of interdependent analyses, the answers to these questions are far from trivial, and deserve careful consideration.

Similar questions can be asked for the whole-program call graph (and points-to) analysis performed by Spark, as well as for other call graph analyses available in Soot. A whole-program call graph is essential for almost any whole-program analysis. Is it possible to perform summary-based call graph construction in Soot? How can such an analysis be designed to easily support many different summary-based whole-program analyses built on top of it? Can Jimple generation be avoided in the context of this call graph analysis? Further investigation of these questions represents a valuable target for future work. The next section contains additional discussion of these and related issues.

4. Discussion

As demonstrated in Section 3, the running time of a whole-program analysis could potentially be reduced by making it summary-based. However, to achieve such savings, support is needed from the underlying analysis infrastructure. This section briefly discusses how to provide better support for summary-based analysis.

Configuration Mechanisms Plugins provide a popular way of extending large software systems with specialized functionality. Soot also takes this approach to allow users to define customized analysis phases and organize phases into packs. For example, the data dependence client described in Section 3 is implemented as a whole-program analysis phase in the `wjtp` pack, and executed immediately after Spark, another whole-program analysis phase. Customization of plugins needs to be generalized to allow summary-based analyses. For example, it may be possible to design analyses that completely avoid the generation of the Jimple representation for certain library methods. This would require easy-to-use customization for the `jb` (or `jj`) phase, without requiring the "average" analysis writer to understand the internals of Soot.

The dependences between different analyses also have to be managed carefully. For example, suppose that the output of one whole-program analysis is used as input to another whole-program analysis. If one of these analyses is modified to be summary-based, how would this affect the other analysis? This question is of particular importance when one of the analyses is a call graph construction algorithm such as Class Hierarchy Analysis or the subset-based points-to analysis used by Spark, since many other whole-program analyses depend on it.

To the best of our knowledge, there are no general mechanisms in Soot to achieve such configurability and to manage inter-analysis dependences of this nature. As a result, analysis developers need to rely on ad hoc solutions to modify the framework in order to achieve the desired effect. The design of general configuration mechanisms and interfaces is a challenging task for a complex system such as Soot.

Management of Summary Information Different analyses may require different kinds of summary information, and the most efficient ways to store this information may differ significantly from analysis to analysis. Even for the same analysis, different components or phases of the analysis may require different summary formats. For example, as discussed in Section 3, two different kinds of summary information are used in two phases of the alias analysis: method-level SPGs in ISPG construction, and balanced partial paths for CFL reachability computation.

The capabilities for summary information management should provide analysis developers with a set of unified summary APIs that allow them to (1) define customized ways of storing, retrieving, and verifying summary information, and (2) register/unregister summary information management callbacks. Developers should be free to define analysis-specific summary formats, and to store the information in a way that is best for the analysis. Existing techniques for efficient serialization and deserialization of data structures [1–4, 14, 17–19, 24, 34, 37] could prove to be a useful addition to the infrastructure services provided by Soot.

Verification of Summary Information To ensure correctness, the summary information should be verified before being used. First, the file containing the summary should remain unchanged compared to when it was written. Second, the summary information in the file should be consistent with the code being analyzed. A typical way to check consistency is to use checksums, but developers should be free to define any other verifying methods. As a default mechanism, the `md5` checksum of a method's bytecode can be computed and stored together with the method's summary. Before the method summary is applied, the checksum of the bytecode can be recomputed and compared with the one stored in the summary.

5. Related Work

Various techniques have been used to achieve a certain degree of modularity in static analysis. For example, several whole-program analyses construct summary information for a procedure, and then use it when analyzing the procedure's callers. A common approach is to construct the summary by analyzing not only the procedure, but also all procedures that are directly or transitively called by it. Several analyses [5–9, 22, 29, 36] employ algorithms based on this approach. Other techniques compute summary information independently of the callers and callees of the analyzed component (e.g., [12, 13, 15, 23, 26]). Other relevant work, focusing on general aspects of defining and computing summary information, is presented in [11, 25, 27, 28, 30, 39]. Our prior work on the alias analysis described earlier [38], as well as subsequent work of similar nature [31], compute and use method summaries for certain methods during the whole-program analysis, but do not make a distinction between library code and client code. When one-time summary generation is performed for the libraries in advance, before any subsequent whole-program analyses for different client programs, the potential for cost savings is substantial, as discussed in Section 3.

Several frameworks have been built for static analysis of Java programs, but none of them put a strong emphasis on using precomputed summary information. In the Chord analysis framework [10], online summarization is provided as part of its implementation of graph-reachability-based interprocedural dataflow analyses [25]. Summary support for other analyses is not available, and relies on developer's ad hoc extensions if needed. WALA [16] provides an interface for defining method summaries and saving them into XML files. The method summary in WALA is organized as a set of SSA instructions, so individual whole-program analyses still have to analyze and transform the summary information into their respective desired format. Expensive reanalysis of complicated methods is not entirely avoided using this approach. A recent retrospective on Soot [20] proposes to build extensions to allow serialization of generated Jimple code for library methods. This technique reduces the Soot startup time; however, specific analyses still have to analyze the Jimple code repeatedly.

Efficient serialization and deserialization of object graphs is an enabling technology that could simplify the storage and management of summary information. There are several approaches (e.g., [1–3, 14, 18, 34]) and tools (e.g., [4, 17, 19, 24, 37]) that could be used to provide such support.

6. Conclusions

This paper examines an interprocedural alias analysis to explore the issues related to extending Soot for summary-based whole-program analysis. Experimental studies show that applying precomputed library summaries could potentially achieve significant savings in analysis running time. Actual achievable savings are limited by the lack of infrastructure support. A brief discussion highlights some of the challenges in making Soot more amenable to summary-based whole-program analysis.

Acknowledgments

We thank the SOAP reviewers for their valuable comments. This material is based upon work supported by the National Science Foundation under grants CCF-0546040 and CCF-1017204.

References

[1] N. Abu-Ghazaleh and M. J. Lewis. Differential deserialization for optimized SOAP performance. In *ACM/IEEE Conference on Supercomputing*, pages 21–31, 2005.

[2] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential serialization for optimized SOAP performance. In *IEEE International Symposium on High Performance Distributed Computing*, pages 55–64, 2004.

[3] B. Aktemur, J. Jones, S. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *International Conference on Generative Programming and Component Engineering*, pages 221–236, 2005.

[4] Apache Avro. `avro.apache.org`.

[5] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[6] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[7] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[8] S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *International Conference on Compiler Construction*, pages 172–186, 2007.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, 1999.

[10] Chord: A Versatile Program Analysis Platform for Java. `pag.gatech.edu/chord`.

[11] P. Cousot and R. Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, pages 159–178, 2002.

[12] M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[13] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, Mar. 1999.

[14] M. Gligoric, D. Marinov, and S. Kamin. CoDeSe: fast deserialization via code generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 298–308, 2011.

[15] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[16] IBM T. J. Watson Libraries for Analysis. `sourceforge.net/projects/wala`.

[17] JSX. `jsx.org`.

[18] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: run-time code generation for Java and its applications. In *International Symposium on Code Generation and Optimization*, pages 48–56, 2003.

[19] L. B. Mesquita. Faster Java serialization. `jserial.sourceforge.net`.

[20] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[21] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

[22] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium*, pages 279–298, 2001.

[23] C. W. Probst. Modular control flow analysis for libraries. In *Static Analysis Symposium*, pages 165–179, 2002.

[24] Protocol Buffers. `code.google.com/apis/protocolbuffers`.

[25] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[26] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, pages 20–36, 2001.

[27] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, pages 2–16, 2006.

[28] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, pages 53–68, 2008.

[29] E. Ruf. Effective synchronization removal for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[30] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[31] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *International Symposium on Code Generation and Optimization*, 2012.

[32] Soot Framework. `www.sable.mcgill.ca/soot`.

[33] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[34] W. Tansey and E. Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[35] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, pages 18–34, 2000.

[36] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[37] XStream. `xstream.codehaus.org`.

[38] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 155–165, 2011.

[39] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–234, 2008.