# Comparing Call Graphs

Ondřej Lhoták

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
olhotak@uwaterloo.ca

## Abstract

Comparing program analysis results from different static and dynamic analysis tools is difficult and therefore too rare, especially when it comes to qualitative comparison. Analysis results can be strongly affected by specific details of programs being analyzed, so quantitative evaluation should be supplemented by qualitative identification of those details. Our general aim is to develop tools to reduce the difficulty of qualitative comparison. In this paper, we focus on comparison of call graphs in particular.

We present two complementary tools for comparing call graphs. Our main contribution is a call graph difference search tool that ranks call graph edges by their likelihood of causing large differences in the call graphs. This is complemented by a simple interactive call graph viewer that highlights specific differences between call graphs, and allows a user to browse through them. In a search for the causes of call graph differences, a user first uses the search tool to identify which of the thousands of spurious edges to look at more closely, and then uses the interactive viewer to determine in detail the root cause of a difference.

We present the ranking algorithm used in the difference search tool. We also report on a case study using the comparison tools to determine the most important sources of imprecision in a typical static call graph by comparing it to a dynamic call graph of the same benchmark.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Procedures, functions, and subroutines

***General Terms*** Algorithms, Design, Languages, Measurement

***Keywords*** call graph, comparison tools, static analysis, dynamic analysis, interpreting analysis results

## 1. Introduction

Program analysis researchers have developed a wide range of analysis tools, both static and dynamic. Naturally, we would like to compare the results of these analyses for several reasons: for example, to determine which analyses to select for specific applications, to compare the effectiveness of different analyses, or to determine whether the result of an analysis conservatively approximates actual runtime behaviour. In this paper, we focus specifically on comparing call graphs, a key prerequisite of many interprocedural analyses.

Although the need for comparisons of program analysis results is a recurring suggestion [10, 22], the results of different program analysis tools are usually not easily comparable. In many cases, a tool is not publicly available to researchers, and published results about it lack the detail necessary for detailed comparison. Even when a tool is publicly available for testing, differences in the program representation and abstractions of program behaviour make the results produced by different tools incomparable. To make such comparison feasible, we have defined common data formats for describing program analysis results, and made available a library to make it easy for existing analysis tools to read and write the formats. We call the project PROBE, because we intend it to be the beginning of a collection of data formats for describing **pro**gram **be**haviour. The PROBE framework is freely available from `http://plg.uwaterloo.ca/~olhotak/probe/`. Of the program analysis result formats currently defined in PROBE, the format for describing call graphs is most fully developed, so this paper focuses specifically on call graphs. In addition to the file format, we have developed tools to help analysis designers compare call graphs and search for sources of imprecision; we present these tools in this paper.

Because the comparison tools understand the PROBE call graph format, they can be used with any analysis, static or dynamic, that uses the PROBE to output call graphs. We have implemented PROBE call graph output for Soot [3], a static analysis framework, and *J [2, 7], a dynamic analysis framework.

The call graph comparison tools presented in this paper were used extensively in our earlier evaluation of the effects of context sensitivity on analysis precision [15].

In this paper,

- we identify key difficulties in comparing call graphs,
- we present an algorithm for finding the root causes of differences in call graphs, which is implemented in one of the comparison tools, and
- we report on a case study using the tools to compare static and dynamic call graphs of the Jess benchmark from the SPEC JVM 98 suite [26].

The rest of this paper is organized as follows. In Section 2, we provide background information and definitions about call graphs, motivate call graph comparison with several applications, and discuss some of the reasons why it is difficult to meaningfully compare call graphs without appropriate tools. In Section 3, we present the algorithm for finding causes of call graph differences, and describe the call graph comparison tools developed as part of the PROBE library. In Section 4, we show how the tools are used to determine



*PASTE'07* June 13–14, 2007, San Diego, California, USA.

the causes of differences between static and dynamic call graphs of an example program. In Section 5, we survey related work, and in Section 6, we conclude.

## 2. Background

### 2.1 Call graph terminology

A call graph is typically defined as a set of directed edges, each connecting a *call site* (i.e. a function invocation statement) to a *target* function invoked from the call site. In languages with function pointers (e.g. C) or dynamic dispatch (e.g. Java), the target of a call site is selected at run time depending on a run-time value, so a single call site may have multiple call targets. Depending on the application, a call graph often also specifies the program entry point. From a given execution of the program, a *dynamic* call graph is constructed by recording which call sites are executed and all the target functions that are called from each of them. A theoretical *ideal* call graph can be defined as the union of the dynamic call graphs over all possible executions of the program. A conservative static call graph is a superset of the ideal call graph; it over-approximates the dynamic call graph of every possible execution. Conversely, every dynamic call graph is a subset of the ideal call graph.

### 2.2 Why compare call graphs?

The following are some of the key reasons to compare call graphs.

- Interprocedural static analyses rely on conservative static call graphs. Because call graphs for realistic programs are large and complicated, it is difficult to check that a static call graph is conservative. A good way to detect missing edges in a static call graph is by comparing it to dynamic call graphs.
- The precision of client analyses often depends on the precision of a static call graph. To select the best call graph construction algorithm for a particular client analysis, we need to compare the precision of call graphs produced by different construction algorithms.
- Specific programming idioms can cause quantitatively large imprecisions in static call graphs. Qualitative comparisons of static call graphs to actual run-time behaviour can be used to identify these idioms, so that static call graph construction algorithms can be improved to model them more precisely.
- Although the ideal call graph for a program is generally uncomputable, it is bounded from below by dynamic and from above by static call graphs. Comparison of these bounds provides a bound on the level of imprecision in these approximations.

### 2.3 Issues in comparing call graphs

Although quantitative comparison of call graphs is straightforward if they are produced by the same tool, it is difficult to determine the qualitative causes of the differences. For example, it is common to count the total number of call edges or the number of functions reachable through the graph from the program entry point. For qualitative analysis, real call graphs are too large to examine by hand: even a dynamic call graph of a Java "Hello, World!" program contains 498 functions.[1]

Many programs contain call sites that are never executed, such as those in an unused portion of a library. There may be a single function that, if executed, would call (transitively) a large unused module of the program, but no actual execution ever reaches the function. In a conservative call graph, a single spurious call edge to the function forces the entire unused module to be included in the call graph. To uncover the source of the discrepancy, a comparison technique must identify the specific spurious call edge, rather than just give summary statistics about the overall graph.

In our experience, the differences between call graphs of real applications are still more complicated. Often, a module is included due to multiple spurious call edges. In a static call graph constructed using an approximation of run-time values to resolve dynamic call site targets, many spurious edges are caused by spurious run-time values, which may in turn be caused by (other) spurious call edges. Thus, identifying the root cause of a call graph discrepancy is generally difficult, and a given discrepancy often has several root causes. Our goal is to devise tools to ease this task.

## 3. Call Graph Comparison Tools

Call graphs are usually much too large to be explored by hand. For example, a statically constructed call graph (constructed using the default configuration of SPARK [12, 14], an interprocedural analysis framework included in SOOT [3]) for the trivial "Hello, World!" program mentioned above contains 3204 methods. There is a significant research community devoted to graph drawing – presenting graphs in a way that exhibits their structure – and this research has resulted in ready-to-use tools such as GraphViz [1], yEd [4], and many others. When comparing call graphs, however, we are interested less in the structure of the graph, and more in finding, out of the many functions and call edges, the few that are important.

We have developed two complementary tools to help compare call graphs. The first tool, our main contribution, searches a pair of large call graphs for differences, and produces a ranked list of call graph edges most likely to be the root cause of a difference. We present the difference search tool in Section 3.1. The search tool is complemented by a simple interactive call graph viewer, which allows the user to closely examine specific areas of the call graph identified by the search tool. We briefly present the interactive viewer in Section 3.2. Other graph exploration tools could be substituted for the interactive viewer. Typically, a user first uses the search tool to find candidate edges to examine more closely, and then browses the surroundings of those edges using the interactive viewer.

### 3.1 Call Graph Difference Search Tool

We have developed in PROBE a call graph comparison tool to aid in finding the root causes of call graph differences. Like the UNIX `diff` utility, the call graph comparison tool compares two call graphs and reports their differences. Unlike `diff`, which does a symmetric comparison, the call graph comparison tool is unidirectional: it reports only methods and edges present in the first graph and absent in the second graph. When comparing call graphs, we are usually looking for imprecisions in one "imprecise" call graph not present in a second, "precise" call graph. It is possible to find all differences between the two graphs by running the tool twice and switching the order of the input graphs between the two runs. In the rest of this presentation, we will call the two graphs given as input to the tool the *subgraph* and the *supergraph*. The call graph comparison tool reports methods and edges present in the supergraph but absent from the subgraph. Despite the names, it is not required that the subgraph actually be a subgraph of the supergraph, although it often will be.

When we compare the static and dynamic call graphs of the "Hello, World!" program, the tool finds 2706 spurious methods and 15993 spurious call edges in the static graph that are not present in the dynamic graph — much too many to examine manually. A spurious call edge that originates from a spuriously reachable method is not very useful in finding root causes of imprecision (because if the originating method were actually called, the edge

[1] All analysis results reported in this paper were collected on the Sun JDK 1.3.1_12 running on Linux.

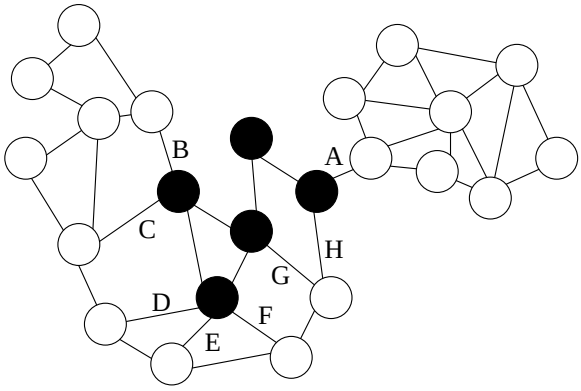


**Figure 1.** Sketch of example call graph

might also be traversed, and would no longer be spurious). Therefore, in its default configuration, our call graph comparison tool reports only spurious edges originating at a method reachable in both graphs. Using this configuration, the number of spurious edges in our "Hello, World!" is reduced to 2370. Although this is an improvement over 15993, it is still too many to easily find the root causes of imprecision.

To better help find causes of imprecision, we experimented with ways to automatically rank the spurious call edges in some way that would identify the edges most likely to be causing a lot of imprecision.

Ideally, we would like to find edges like the one labeled A in Figure 1. In this sketch of a fictional call graph, black nodes indicate truly reachable methods, and white nodes indicate spuriously reachable methods. The call edges labelled with the letters A through H are spurious edges each originating at a truly reachable method; therefore, these would be the edges reported in the default configuration of our call graph comparison tool. In searching for root causes of imprecision and seeking to improve the imprecise analysis that constructed this call graph, it would appear to be most fruitful to first look at the edge labelled A, for two reasons. First, it connects a large number of spurious methods to the call graph (all of the white nodes in the upper right of the figure). Second, because it is the *only* edge that connects them, its removal would immediately eliminate the many spurious methods from the graph.

Notice that an edge such as B also leads to a large number of spurious methods, but removing B would not eliminate them, since they are also reachable by other call edges. Yet, it appears more fruitful to pursue the removal of B than H, for example, because B is more closely connected to more spurious methods than H. Therefore, informally, our ranking should put edges like A near the top, and it should put edges like B before edges like H.

To produce such a ranking, we first experimented with graph-theoretic algorithms such as min-cut and max-flow and related algorithms, but did not find them to work well for this problem. While these algorithms look for *optimal* solutions, we need to find solutions that are merely *good*. For example, min-cut would find the smallest set of edges whose removal would disconnect the graph into multiple connected components. In actual call graphs, a min-cut was often a single edge, and it usually disconnected only one or very few spurious methods from the graph. Instead of the *minimal* cut, we are interested in a merely a *small* cut, but one which cuts off a *large* component of spurious methods.

The algorithm that we have implemented in PROBE is shown in Figure 2. Intuitively, the algorithm is similar to simulating the flow of water through a system of lakes and rivers; the fast-flowing rivers indicate large watersheds with few alternative drainage paths. Each

```
double level(method m)
    if m reachable in subgraph
        return 0
    else
        return l[m]
    endif

for each method m in supergraph
    set l[m] = 1.0
endfor
loop
    for each call edge e = m → m′ in supergraph
        diff = α·(level(m′) - level(m))
        if diff > 0
            total[e] += diff
            l[m′] −= diff
            l[m] += diff
        endif
    endfor
until (max_m l[m]) < ε
output edges in order of decreasing total[e]
```

**Figure 2.** Call graph flow algorithm

method is modelled as a lake, and each call edge as a connecting river. Initially, all spuriously reachable methods are assigned a uniform level of fluid. Truly reachable methods are sinks through which the fluid drains (i.e. their fluid level is always kept at zero). The algorithm simulates the draining of the fluid from the spurious methods, along call edges in the direction opposite to the calling direction, eventually reaching the sinks. As the fluid drains, the amount that has flowed through each edge is recorded. Once all of the fluid has drained, the call edges are ranked according to how much has flowed through each of them. An edge (such as A in Figure 1) connecting many spurious methods to the graph will have the fluid from those many methods flowing through it, giving it a high rank. However, if a region of spurious methods is connected to the graph by many independently spurious edges, the fluid of those methods will be divided among these edges, so each edge will rank lower.

The quality of the simulation is affected by two configurable parameters in the algorithm, $\alpha$ and $\varepsilon$. The parameter $\alpha$ controls the granularity of each simulation step. In each iteration, for each call edge from method $m$ to $m'$, if the level at $m'$ is higher than at $m$, the difference in levels is multiplied by $\alpha$, and the levels of $m$ and $m'$ are raised and lowered, respectively, by the resulting amount. Thus, smaller values of $\alpha$ result in a more faithful simulation of fluid flow, while larger values result in fewer iterations and therefore faster computation. The parameter $\varepsilon$ controls how much of the fluid must drain before the algorithm terminates. Because the flow in each iteration is proportional to the differences in levels between methods, and these are in turn proportional to the amount of fluid left, the amount left asymptotically approaches but never reaches zero. A larger $\varepsilon$ allows the algorithm to terminate sooner than a smaller $\varepsilon$. After some tuning, we settled on values of $\alpha = 0.125$ and $\varepsilon = 0.001$ for these parameters. With these values, when comparing a static and dynamic call graph for the SPEC JVM 98 benchmarks [26], the algorithm completed within $10^5$ iterations (each processing on the order of $10^4$ edges), thus completing within 30 seconds on a 2GHz AMD Athlon 64. This is in line with the time required to generate the call graphs being compared. When using smaller values of these parameters, we did not observe noticeable differences in the ranking of call edges.

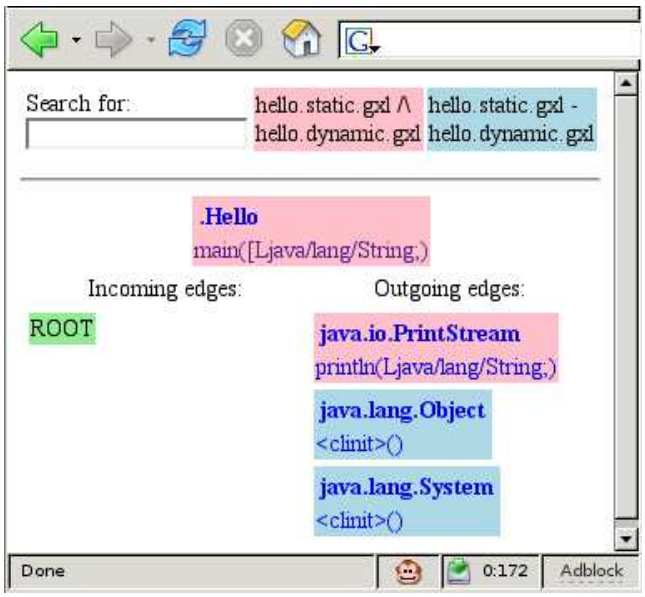


**Figure 3.** Screenshot of call graph viewer

### 3.2 Interactive Call Graph Viewer

Once the difference search tool has pinpointed differences likely to be important, the relevant code must be inspected by hand to determine the exact cause of the difference. We complement the difference search tool with a simple interactive call graph viewer, to help a person manually explore the functions and call edges near an important difference. Because it is intended for local exploration, the interactive call graph viewer implemented in PROBE presents only a single method at a time.

The viewer acts as an HTTP server to which the user connects with a web browser. Each web page generated by the viewer represents one method, and contains clickable links to all methods calling and called by the current method. A search box allows the user to search for methods or classes by name. A sample screenshot of the `main` method of the "Hello, World!" program is shown in Figure 3. The method has an incoming edge from the special `Root` node (indicating that it is an entry point), and three outgoing edges, to the `println` method in `java.io.PrintStream`, and to two static initializers.

The viewer uses colour to allow two call graphs to be browsed simultaneously and compared. The example screenshot in Figure 3 shows the static and dynamic call graphs of the "Hello, World!" program being compared. Although it is not apparent in a black and white version of this paper, the targets of call edges are colour-coded to show whether they appear in both call graphs or only in the static one. The call to the `println` method is pink, to show that it appears in both call graphs. However, the calls to the static initializers are blue, meaning that they appear only in the conservative static call graph. That is, they were not actually called from `main` in the run of the program from which the dynamic call graph was produced, but the static analysis conservatively estimates that they could have been, had `Object` and `System` not been initialized earlier. Thus, these two call edges are a potential source of imprecision in the static call graph.

## 4. Case Study: the Jess benchmark

Conservative static call graphs for Java are imprecise, mainly due to the large standard library provided with Java, whose use can rarely be statically ruled out even in very small programs. Even when running a trivial "Hello, World!" program, 498 distinct methods are actually executed, but a conservative static call graph contains over six times as many: 3204. Using the Jess benchmark from the SPEC JVM 98 suite [26] as an example, we used the call graph comparison tools to determine the main causes of differences between the static and dynamic call graphs.

In any evaluation of a static analysis result, the first step should be to check that it is a conservative approximation of dynamic behaviour. We ran the difference search tool with the dynamic call graph (generated using *J) as the supergraph and a static call graph (generated using the default settings of SPARK) as the subgraph, expecting to see no dynamic edges or methods missing from the static graph. However, the tool found 143 methods in the dynamic call graph but not the static call graph. The Jess language implemented by the benchmark allows external functions to be written as Java classes, which the benchmark loads using reflection; we had not told SPARK about these classes. In the top ten missing call edges reported by the difference search tool, five were calls to these external functions, and three were calls from the `newInstance0` method of `java.lang.Class` instantiating classes using reflection. By searching for all call edges from `newInstance0`, we composed a list of classes loaded using reflection to give to SPARK, so it could construct a conservative call graph.

At this point, the static call graph contained 3427 reachable methods, compared to only 986 in the dynamic call graph (recall that the true ideal call graph is somewhere between these two approximations). We ran the difference search tool, this time with the static call graph as supergraph and the dynamic call graph as subgraph. The most important spurious edge identified by the comparison algorithm was from the `openStream()` method in `spec.io.FileInputStream` to `getInputStream()` in `sun.net.www.protocol.file.FileURLConnection`. The SPEC harness includes code to download benchmark input data directly from the SPEC website, instead of reading it from disk; this forces a conservative static call graph to include the Java networking and HTTP library classes. Since we do not use this feature in any of our test runs, we commented out the part of the `openStream()` method that accesses the network, leaving only the part that reads input from a local file. Although this reduced the static call graph by only 10 methods, the difference search tool showed why: the top-ranked spurious call edge was to the `getInputStream()` method in `sun.misc.URLClassPath$3`, and other highly-ranked spurious edges also represented calls from class loading code to networking code. The networking code is used not only by the SPEC harness, but also by the JVM to load Java classes from the network. After advising the static call graph analysis to ignore these spurious calls,[2] which are not used when the JVM is running only class files found on the local filesystem, the static call graph shrank to 3186 methods.

After we removed networking code from the static call graph, eight of the top ten spurious call edges reported by the comparison algorithm were evidently related to either Jar file loading and verification, or were calls into cryptography code in the `sun.security` package. We used the interactive browser to explore around the spurious edges, and inspected bytecode of several rele-

[2] The purpose of the case study is to use the tools to determine the root causes of differences between the static and dynamic call graphs, rather than to improve the static analysis while maintaining soundness. We do not claim that it is sound to allow the analysis to ignore code that only executes when classes are loaded from the network — the soundness of this optimization depends on the environment in which the JVM is run (i.e. whether the classpath points to network classes), and we have not defined a specific environment. We remove the code from the analysis only to show that no other code contributes to these differences between the static and dynamic call graph.

vant methods. When the JVM loads classes from a signed Jar file, the JarVerifier class checks the signature. Although none of the Jar files in our environment were signed, the static call graph analysis had no way to know that, so it conservatively assumed that the JarVerifier (and all the cryptography code on which it depends) could be called. Advising the static analysis to ignore calls to the JarVerifier reduced the number of reachable methods in the static call graph by more than 30%, to 2194.

We again ran the difference search tool, and found that the highest-ranked spurious call edge was to the `run()` method of `sun.security.provider.PolicyFile$1`. This class interprets the `security.policy` file that can be provided to the JVM to restrict the external operations that the JVM will perform, such as accessing disks or the network. Since we do not use this feature, we removed it from the static analysis, reducing the static call graph to 2070 methods.

So far, we have determined that 1357 of the 2441 spurious methods appearing in the static but not the dynamic call graph were caused solely by several unused features of the SPEC harness and of the JVM. We could continue in this way, using the call graph difference search tool and the interactive call graph viewer to find the root causes of the remaining differences.

## 5. Related Work

The need for the program analysis community to agree on benchmarks and generate reproducible analysis results that can be compared has been identified before, for example by Hind [10] and by Ryder [24]. We echo this call, and provide the PROBE infrastructure in the hope that it will help the program analysis community to address this need.

Rountev [22] discussed how static and dynamic analysis results can be evaluated together to identify sources of imprecision (in the static results) and incomplete coverage (in the dynamic results). The two approaches to analysis over- and under-approximate the true program behaviour, and by considering them together, the amount of imprecision can be measured. A similar argument was made by Ernst [8]. Rountev *et al*. [23] applied the proposed approach to quantify the imprecision in an analysis of call chains. Researchers can use the qualitative call graph comparison tools we have presented to point to specific areas in a program where these approximations can be improved to converge closer to the true call graph. Qualitative differences indicate opportunities for either increasing the coverage of test inputs used to construct the dynamic call graph (making it bigger), or improving precision of the analysis used to construct the static call graph (making it smaller).

In the area of call graph construction, Murphy, Notkin, and Lan [21] compared the call graphs produced by five static tools on three C programs both quantitatively, and by qualitative examination of a sample of differences. They found that the call graphs all had important differences: no tool produced a call graph that was a supergraph of any other. Tip and Palsberg [28] quantitatively compared several low-cost call graph construction algorithms for Java. Grove and Chambers [9] defined a unified framework for expressing call graph construction algorithms, and performed a quantitative study comparing different instantiations of the framework. Our comparison tools could be used to repeat these experiments and provide additional insight into the specific program features behind the primarily quantitative differences observed in these studies.

Another area with existing empirical comparisons is points-to analysis. Hind and Pioli [11] studied a collection of points-to analyses for C, and compared their efficiency and precision with respect to several client analyses. Mock *et al*. [20] used dynamic points-to sets to measure the precision of static analysis results. Like call graphs, points-to analysis results are typically very large; qualitative comparison tools for points-to information would be very helpful for developing and tuning future points-to analyses.

The past several years have seen a strong interest in efficient implementation of context-sensitive program analyses, so it is fitting to ask how context sensitivity contributes to analysis precision. Stocks *et al*. [27] studied the effects of flow sensitivity and context sensitivity on side-effect analysis of C programs. Liang *et al*. [16] evaluated the effect of object sensitivity [17–19] and call site string context sensitivity on the size of pointed-to-by sets computed by subset-based points-to analyses an Java. Lhoták and Hendren [13, 15] compared object sensitivity, call site string context sensitivity, and the Zhu/Calman/Whaley/Lam algorithm [29, 30] in terms of the precision of call graph construction, call devirtualization, points-to analysis, and cast safety analysis. Sridharan and Bodík [25] evaluated a demand-driven adaptively-context-sensitive points-to analysis using a cast safety client analysis. All of these studies reported mainly quantitative results, with qualitative observations playing a secondary role. To continue to improve precision of static analyses, researchers will increasingly have to consider the specific program constructs behind these numbers, and tailor analyses to these constructs. Identifying these constructs will require qualitative comparison tools not only for call graphs, but also for other program analysis results, such as points-to information.

## 6. Conclusions and Future Work

We have motivated the need for qualitative comparison of program analysis results, and of call graphs in particular, and presented tools to help make such comparison practical. Design, tuning, and evaluation of future program analyses can benefit from qualitative examination of analysis results.

The comparison tools we presented are part of PROBE, a collection of data formats for program analysis results, a library for generating output in those formats from different analysis tools, and tools for evaluating and comparing analysis results. We have already used PROBE for both qualitative and quantitative evaluations [13, 15] of static analyses implemented within the SPARK [12, 14] and PADDLE [13] frameworks in SOOT [3], and compared their results with dynamic information collected using *J [5, 6].

In our comparison of static and dynamic call graphs of the Jess benchmark, we observed that most of the difference was caused by JVM features that were not enabled in our runtime environment. In a solely quantitative study, it would have been tempting to blame the difference on imprecision of the static analysis, and perhaps senselessly attempt to improve its precision. The qualitative comparison that we performed using the tools showed the true causes of the difference; only by removing these specific fragments of code from consideration by the static analysis could we quantify their effect on call graph size.

Because call graphs are a key prerequisite for nearly all interprocedural analysis, we have focused first on comparing call graphs. In languages such as Java in which objects are always accessed indirectly through references, points-to information is also an important prerequisite for many analyses, and points-to analysis is often interdependent with call graph construction. Thus, tools for comparing points-to analysis results should also be developed, perhaps combined with the call graph comparison tools.

More generally, qualitative comparison and appropriate tools to support it can be applied to arbitrary program analyses. Specific, localized program constructs often have considerable effects on quantitative measurements of program analysis results, so summary numbers reported on a handful of benchmarks, as is so common, may be strongly biased by specific coding styles. Quantitative evaluations should by no means be suppressed, but they should be augmented with more qualitative insight identifying specific program features behind the numbers.

## 7. Acknowledgements

This work, and the PROBE framework in particular, was motivated by discussions by the participants at the PASTE 2004 workshop. In particular, Atanas Rountev and David J. Pearce provided suggestions both at the workshop, and in subsequent e-mail discussions. We are also grateful for suggestions from Gordon Cormack. This work was partly funded by NSERC and an IBM PhD Fellowship. Part of this work was done in the Sable lab at McGill University, led by Laurie Hendren.

## References

[1] Graphviz - graph visualization software. `http://www.graphviz.org/`.

[2] *J: A tool for dynamic analysis of java programs. `http://www.sable.mcgill.ca/starj/`.

[3] Soot: a Java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[4] yEd - Java graph editor. `http://www.yworks.com/en/products_yed_about.htm`.

[5] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 149–168. ACM Press, 2003.

[7] B. Dufour, L. Hendren, and C. Verbrugge. *J: a tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[8] M. D. Ernst. Static and dynamic analysis: synergy and duality. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–35. ACM Press, 2004.

[9] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM Press, 2001.

[11] 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*, pages 113–123. ACM Press, 2000.

[12] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Dec. 2002.

[13] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.

[14] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, Apr. 2003. Springer.

[15] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna, Mar. 2006. Springer.

[16] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12, New York, NY, USA, 2005. ACM Press.

[17] A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, Aug. 2003.

[18] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. ACM Press, 2002.

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

[20] 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*, pages 66–72. ACM Press, 2001.

[21] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, Washington, DC, USA, 1996. IEEE Computer Society.

[22] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 14–16. ACM Press, 2004.

[23] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. ACM Press, 2004.

[24] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003.

[25] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press.

[26] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.

[27] P. A. Stocks, B. G. Ryder, W. A. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 21–31. ACM Press, 1998.

[28] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293. ACM Press, 2000.

[29] 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*, pages 131–144. ACM Press, 2004.

[30] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157. ACM Press, 2004.