A Study of the Impact of Callbacks in Staged Static+Dynamic Partial Analysis

Aditya Anand
Indian Institute of Technology Mandi
Mandi, Himachal Pradesh, India
s20007@students.iitmandi.ac.in

Abstract

Partial analysis is a program analysis technique used in compilation systems when the whole program is not available. Many recent promising approaches perform partial analysis statically that involves identifying the interprocedural dependencies across program elements. These generated dependencies further get evaluated during runtime while generating the final analysis result. However, as the application and library methods are analyzed independently during static analysis, these approaches do not account for the effect of dynamic features such as callbacks. Consequently, in such scenarios, the runtime (say the Java Virtual Machine) needs to discard the static-analysis results and use the existing imprecise builtin analyses. The primary goal of this work is to find out the percentage of objects and methods that may get affected by callbacks, and to propose possible techniques to enhance the generation of dependencies in their presence.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Dynamic analysis**.

Keywords: Partial analysis, callbacks, static analysis

ACM Reference Format:
Aditya Anand. 2022. A Study of the Impact of Callbacks in Staged Static+Dynamic Partial Analysis. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3563768.3563957

1 Motivation and Background

Programs written in modern languages such as Java and C# are compiled *just-in-time* (JIT) to native code during execution. As the JIT-compilation time affects the execution-time of the program under consideration, typical JIT compilers perform imprecise (e.g., intraprocedural) program analyses. One of the promising ways to generate precise analysis results is to perform partial analysis of the available program statically, and record the dependencies on the unavailable portions as conditional values [1, 5, 7]. These dependencies get resolved at runtime to generate final analysis results. Though this approach generates precise program-analysis results efficiently during JIT compilation, it does not take dynamic features such as callbacks into consideration. This motivated us to study and find out the proportion of objects and methods that actually may get impacted due to callbacks, and to explore strategies to mitigate the same. We expect that our findings will help us derive sound and robust partial analyses for Java-like languages, as well as promote the adaptability of such static+dynamic approaches in future.

```
class A {
  void foo(X p) {
    Y m = p.f;
    this.bar(m);}
  void bar(Y q) {
    Z r = new Z();
    q.g = r;
  }} /* class A */
```

Figure 1. Example to demonstrate dependencies.

We now illustrate the generation of dependencies in existing static+dynamic analysis schemes, along with the problem in presence of callbacks, using an example.

1. Dependencies in static+dynamic partial analysis. In order to address the absence of libraries while statically analyzing Java applications (and vice-versa), existing approaches [1, 7] generate results for various elements of a program (e.g. abstract objects in Java) in terms of dependencies on other program elements. As an example, consider the code snippet shown in Figure 1. Here, the variable `m` points to an object from the caller, in absence of whose code the analysis would record a dependency `<caller,arg1>` for the object pointed to by `m`, denoting the first argument passed to `foo` by its caller(s). Similarly, the object pointed to by `r` in the method `bar` depends on the argument passed to `bar` and also (transitively) on the argument passed to `foo`.

2. Partial analysis in presence of callbacks. Java offers several dynamic features such as callbacks, reflection, etc. Performing static analysis without considering these features may generate unsound results. For example, callbacks are a popular dynamic feature where the library methods can also invoke an application method. Thus, if the method

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9901-2/22/12...$15.00
https://doi.org/10.1145/3563768.3563957

```
1  Procedure callbackObjectsAndMethods()
2      foreach application class c do
3          if c overrides or implements a library method m then
4              Mark m and all its parameters as callback_affected.
5      repeat
6          foreach callback_affected method mtd do
7              foreach object obj ∈ mtd do
8                  if obj has dependency on a callback_affected
                     formal parameter then
9                      Mark obj as a callback_affected object.
10         foreach callback_affected object obj do
11             if obj is passed as the kth argument to another method
                 n then
12                 Mark n and its kth parameter as
                     callback_affected.
13     until fixpoint;
```

Figure 2. Finding objects and methods dependent on callbacks.

foo is a library-overridden method, the statically unknown argument passed to foo from the library may affect the analysis result of the objects pointed-to by m in foo as well as by r in bar. However, existing staged approaches [1, 4, 7, 9] for performing partial static+dynamic analysis ignore callbacks and may generate unsound results in their presence.

2 Impact of Callbacks

To handle callbacks while performing partial analysis for a Java application, we must first identify the program element that may get affected by possible callbacks from libraries. We now describe our approach to estimate the number of objects and methods that may depend on such callbacks; see Figure 2.

The procedure callbackObjectsAndMethods maintains a data structure named callback_affected (overloaded, for brevity) to record program elements that may be affected by callbacks. To begin with, lines 2-4 identify library-overridden or implemented methods in the application classes, and mark all their parameters as potentially callback-affected. Next, for all the callback-affected methods, if any of their local objects depends on the existing callback-affected methods and objects, lines 6-9 mark those objects also as callback-affected. Finally, if any of the callback-affected objects is passed to another method, say as the k^{th} argument, we mark the callee as well as the corresponding parameter as callback-affected (lines 10-12). We perform the previous two operations till a fixed point, in order to get all the callback-affected methods and objects in the application classes.

3 Study Setup and Preliminary Results

1. Static analysis. We extend an existing static program-analysis tool called Stava [6] that generates dependencies for each program element. Stava is written in Soot [8], a popular Java Bytecode analysis framework, and uses TamiFlex [3] to resolve reflective calls while constructing the call-graph.

Bench-mark	Callback methods	Total methods	Callback objects	Total objects
avrora	43	3181	147	13344
batik	896	6934	1865	34137
lusearch	123	1971	338	9054
luindex	69	1998	171	10260
pmd	595	5941	1301	30016
sunflow	49	2428	104	14136
h2	639	4777	1315	27610
xalan	821	6396	2131	31488
fop	968	11470	2387	74590
eclipse	1515	29419	3505	79443

Figure 3. No. of methods and objects affected by callbacks.

2. Benchmarks and system configuration. We considered 10 benchmarks from the DaCapo suite[2] version 9.12 using the "default" input size. The benchmarks excluded from the suite could not be analyzed – either by Soot or by TamiFlex. Our experiments have been performed on a 3.00 GHz Intel(R) Xeon(R) system with 48 cores and 100 GB of memory, running Ubuntu 20.04.4 LTS.

Figure 3 shows the number of methods and objects that may be affected by potential callbacks, along with the total number of methods and objects in the corresponding benchmark. We find that on an average, the 7.7% of methods can be affected by callbacks from libraries, either directly or transitively through parameters. On the other hand, we found that 4.1% of objects contain dependencies related to the parameters involved in potential callbacks. Thus, we can observe that the static-analysis results computed for these many program elements, by prior static+dynamic schemes, may be unsound in presence of corresponding callbacks at run-time. On one hand, an overall low percentage of objects affected by callbacks enhances the confidence on the potential of existing static+dynamic schemes, and on the other hand shows that it is important to develop schemes that maintain precision and soundness in their presence.

4 Conclusion and Future Work

In order to balance the trade-off between the efficiency and precision of generating whole-program analysis results in JIT-based runtimes, recent approaches propose partial analysis of the statically available program. However, these approaches either ignore dynamic features such as callbacks, or generate conservative results in their presence. In order to address this problem, we have first computed the percentage of application methods and objects that may potentially be affected by callbacks made by Java libraries. We found that the percentage is low enough to maintain confidence on staged static+dynamic schemes, and high enough to deserve a solution that enhances their practicality further. In future, we plan to extend the idea of staged partial analysis, by efficiently and precisely modeling the identification and generation of dependencies across program elements, in presence of dynamic features such as callbacks.

References

[1] Aditya Anand and Manas Thakur. 2022. Principles of Staged Static+Dynamic Partial Analysis. In *Proceedings of the 29th Static Analysis Symposium (SAS 2022)*. Springer International Publishing, 30 pages.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190. https://doi.org/10.1145/1167515.1167488

[3] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. https://github.com/secure-software-engineering/tamiflex. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[4] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. *SIGPLAN Not.* 43, 10 (Oct. 2008), 313–328. https://doi.org/10.1145/1449955.1449790

[5] Rishi Sharma, Shreyansh Kulshreshtha, and Manas Thakur. 2022. Can We Run in Parallel? Automating Loop Parallelization for TornadoVM. https://doi.org/10.48550/arXiv.2205.03590

[6] Nikhil T R, Dheeraj Yadav, and Manas Thakur. 2021. Stava. https://github.com/CompL-IITMandi/stava.

[7] Manas Thakur and V. Krishna Nandivada. 2019. PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 16 (July 2019), 37 pages. https://doi.org/10.1145/3337794

[8] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13–23. http://dl.acm.org/citation.cfm?id=781995.782008

[9] Hao Zhong and Xiaoyin Wang. 2017. Boosting Complete-Code Tool for Partial Program. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 671–681.