Static Resource Analysis at Scale (Extended Abstract)

Ezgi Çiçek[1], Mehdi Bouaziz[2], Sungkeun Cho[1], and Dino Distefano[1]

[1] Facebook Inc. {ezgi,scho,ddino}@fb.com
[2] Nomadic Labs mehdi@nomadic-labs.com

1 Introduction

Programs inevitably contain bugs. Fortunately, recent research and engineering efforts across the industry and academia made significant advances in static analysis techniques allowing automatic detection of bugs that cause a program to crash or to produce an unintended result. In many settings, it is not enough for a program to execute without errors. Programs must also finish executing within expected resource bounds and adhere to a sensible resource usage. At the very least, we expect the resource usage of programs to not deteriorate significantly as the source code evolves, hurting the experience of the users or even making the program unusable.

There are many static analysis techniques for estimating and verifying the resource usage of a program, ranging from static worst-case execution time (WCET) analyses (see [13] for a detailed survey) to typed-based approaches and program logics [6, 2, 12, 10, 11, 7, 5]. Research in static WCET analysis has been widely applied to validation and certification of embedded systems in safety critical systems. To estimate hard real-time bounds, these analyses must be tuned carefully to take into account abstract models of caching, scheduling and pipeline behavior of the embedded system. On the other hand, type based analyses and program logics are often more abstract but require sophisticated type checking/inference algorithms or specialized tools like proof assistants which make them unsuitable to be used on big codebases without specialist proof engineers.

In our work, we turn our attention to big codebases for mobile applications. We observe that although many static analysis techniques have been deployed to detect functional correctness bugs, not much attention is given to statically detecting performance regressions in industrial codebases. Most often, developers in such codebases deal with performance regressions through dynamic analysis techniques by relying on a combination of performance tests and profilers. Considering that these applications are developed in a continuous way where developers regularly add new features or modify existing code, only a limited amount of testing and monitoring can effectively be done before the code runs in production. Moreover, once a performance regression is introduced, it may take several days or even weeks for it to be detected by production monitoring systems. Once the regression is observed, tracking it back to its root cause is also a very time consuming task: The release of an application has normally

thousands of code changes and singling out the changes responsible for the performance regression is like finding a "needle in the haystack". This whole process of *identifying* and *fixing* performance regressions is costly not only for the application and its users, but also in terms of engineering time. In fact it requires multiple developers to interact, coordinate, and finally verify that fix improves the performance.

2 Static Complexity Analysis with Infer

Motivated by these issues, we have developed an inter-procedural static analysis technique to automatically detect a class of performance regressions early in the development cycle. Our analysis is based on an abstract-interpretation technique [3, 9] which computes symbolic upper bounds on the resource usage of programs—execution cost being the main resource we consider. These costs are expressed in terms of polynomials describing the asymptotic complexity of procedures with respect to their input sizes. The main input of the analysis is the source file which is then translated to an intermediate language along with the control-flow graph of the program. The analysis then operates on this intermediate language in several phases: 1) a numerical value analysis based on InferBo [1] computes value ranges for instructions accessing memory, 2) a loop bound analysis determines upper bounds for the number of iterations of loops and generates constraints for nodes in the control-flow graph, and 3) a constraint solving step resolves the constraints generated in the second step and computes an upper bound on the execution cost. The analysis assumes a simple sequential model with an abstract cost semantics: each primitive instruction in the intermediate language is assumed to incur a unit execution cost. The analysis is not limited to inferring bounds for just execution cost. In order to statically detect regressions in other types of resource usage, we have generalized the analysis to account costs for different types of resources such as memory allocations.

3 Diff-time Deployment at Scale

We implemented the analysis on top of the Infer Static Analyser [8], which is used at Facebook to detect various errors related to memory safety, concurrency, and many more specialized errors suggested by Facebook developers. Infer hooks up to the continuous integration mechanism with the internal code review system where it is run on any code change (diff) over Facebook's Android codebase [4, 8]. For our diff-based analysis, we rely on this mechanism and infer polynomial bounds for the original and the updated procedures. Whenever there is an increase in the degree of the complexity from the original to the modified version (e.g. from constant to linear or from linear to quadratic), we report a warning to the developer with a trace explaining where and how the complexity increase occurred.

Since the tool was deployed, thousands of complexity increase warnings were issued in Facebook's Android codebase where hundreds of these were fixed before

the code was committed. Unlike functional correctness bugs where fix-rate is a good indicator of whether the issues found by the analyser are useful to the developer, we do not solely rely on fix-rate as a metric to measure the effectiveness of asymptotic complexity increase signal. This is because, unsurprisingly, not all complexity increase warnings point to an actual performance regression: a) the complexity increase could be intended or the input sizes used in production could be small enough to have no effect on the performance and b) the warning could also be a false positive due to limitations of the analyzer. To alleviate these, we follow a two-pronged approach. First, we ask developers to provide feedback on whether a warning is good-catch, expected, or wrong (potentially pointing to a false-positive). Only a small fraction of developers provide such feedback but they are still useful: the most frequent feedback is that the warning was expected. Wrong warnings are very rare (a few times a week) and we follow up these warnings closely to fix weaknesses of the analyzer. Secondly, to help developers evaluate the severity of the warning, we incorporate different types of contextual information that surface e.g. whether the procedure with the complexity increase runs on the critical path or main (UI) thread, which critical user interactions the procedure occurs on, and some dynamic profiling info (e.g. avg CPU time of the original procedure) when available. We observe that warnings with such contextual information are fixed (and marked as good-catch) more frequently in comparison to vanilla complexity increase warnings.

Thanks to the compositional nature of the analysis that enables us to generate execution costs of procedures independently of calling contexts, it can scale to large codebases and work incrementally on frequent code modifications. We believe that there is much unlocked potential and future work opportunities for applying this type of static performance analysis. Although not all complexity increase signal could be considered an actual performance regression, we observed that surfacing them to developers is still useful for code quality and regression prevention.

We are currently working on extending the analysis to detect out-of-memory errors, combining static analysis with dynamic techniques, and adding support for handling other languages such as C++ and Objective-C.

References

1. Inferbo: Infer-based numerical buffer overrun analyzer (2017), https://research.fb.com/blog/2017/02/inferbo-infer-based-buffer-overrun-analyzer/
2. Atkey, R.: Amortised resource analysis with separation logic. Log. Methods Comput. Sci. (2011)
3. Bygde, S.: Static WCET Analysis Based on Abstract Interpretation and Counting of Elements. Ph.D. thesis (2010)
4. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NASA Formal Methods. LNCS, vol. 6617, pp. 459–465. Springer (2011)

5. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational cost analysis. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017)
6. Crary, K., Weirich, S.: Resource bound certification. POPL '00, Association for Computing Machinery, New York, NY, USA (2000)
7. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '08, New York, NY, USA (2008)
8. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Commun. ACM (2019)
9. Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. Proc. 7th International Workshop on Worst-Case Execution Time Analysis(WCET'2007) **6** (01 2007)
10. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for ocaml. SIGPLAN Not. (2017)
11. Knoth, T., Wang, D., Reynolds, A., Hoffmann, J., Polikarpova, N.: Liquid resource types. Proc. ACM Program. Lang. (ICFP) (2020)
12. Wang, P., Wang, D., Chlipala, A.: Timl: A functional language for practical complexity analysis with invariants **1**(OOPSLA) (2017)
13. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (2008)