GraalSP Profiles Logger: A Tool for Analyzing and Interpreting Predictions of the ML-Based Static Profilers

Milan Čugurović[2,1], Milena Vujošević Janičić[1,2]
[1] Faculty of Mathematics, University of Belgrade
[2] Oracle Labs, Belgrade, Serbia
milan.cugurovic@matf.bg.ac.rs,
milena.vujosevic.janicic@matf.bg.ac.rs

Abstract. Program profiles provide information about the program's behavior during execution. Profiles include the number of branch executions in branching statements, context and number of function calls, context and number of virtual method calls, information on monitor locks and unlocks, and more. Profiles enable profile-guided optimizations including aggressive inlining, cache memory optimizations, and similar techniques which generate highly efficient code.

Instrumentation-based profiling is a common technique for collecting precise profiles. However, the profile collection represents a significant overhead in software development, and in some environments, it is unfeasible (for example, in memory-constrained environments such as embedded systems or in real-time systems where extended execution time due to the additional instrumentation code is not acceptable). An alternative to dynamic profiling is static profiling. Static profilers predict profiles instead of collecting them. The current development of static profilers is mainly focused on using machine learning techniques for predicting branch execution probabilities. Modern static profilers [1, 2, 3] use ensembles of random forests and deep neural networks for regression to predict the probabilities of the taken branch of a branching statement. The best current static profilers achieve execution time speedups of 5 to 7 percent.

Static profilers can make mispredictions as similar code segments may execute differently depending on inputs and various conditions. The impact of individual predictions can significantly affect the performance of the program. For instance, an incorrect prediction that the branch would not lead into a frequent loop body can result in a substantially slower program. To maximize performance, state-of-the-art static profilers employ instance weighting to prioritize critical cases. However, misprediction can still happen and it is necessary to efficiently identify and analyze them, to adjust and improve the ML model driving the static profiler.

We have developed the GraalSP Profiles Logger (GraalSP-PLog), a tool for efficiently analyzing ML-based predictions made by the GraalSP static profiler [1] which is a part of the Oracle GraalVM Native Image compiler [4]. GraalSP-PLog runs the input program with GraalSP to log the profile predictions. As GraalSP employs a set of static heuristics that correct predictions of the ML model, GraalSP-PLog also captures this information. To collect ground truth values, GraalSP-PLog performs instrumentation profiling to dynamically collect program profiles. Combining static predictions and collected profiles, GraalSP-PLog generates a prediction report. The report facilitates the identification of performance-critical predictions, both accurate and inaccurate, and sorts nodes by their impact on overall program performance. The tool provides results in *.csv* format so that they can be easily opened on any operating system and sorted by any criteria.

Additionally, we have incorporated the Ideal Graph Visualizer (IGV) [5] into GraalSP-PLog to evaluate the impact of branch execution probabilities on optimization choices. IGV offers a visual depiction of the transformations in intermediate representation graphs throughout the compilation process, enabling users to understand how optimizations influence method compilation. This visualization capability greatly enhances the human assessment of GraalSP.

GraalSP-PLog is a command-line application. When run, it launches the IGV visualizer and generates a graph of the target method. Figure 1 shows part of the IGV graph corresponding to the loop in the binary search method, with its source code displayed in Figure 1. The *13If* node corresponds to the exit from the *while* loop. Regarding exiting the while loop, dynamic profiling predicts a value very close to zero, while static profiling estimates a probability of 0.07. In this case, the model made an accurate prediction.

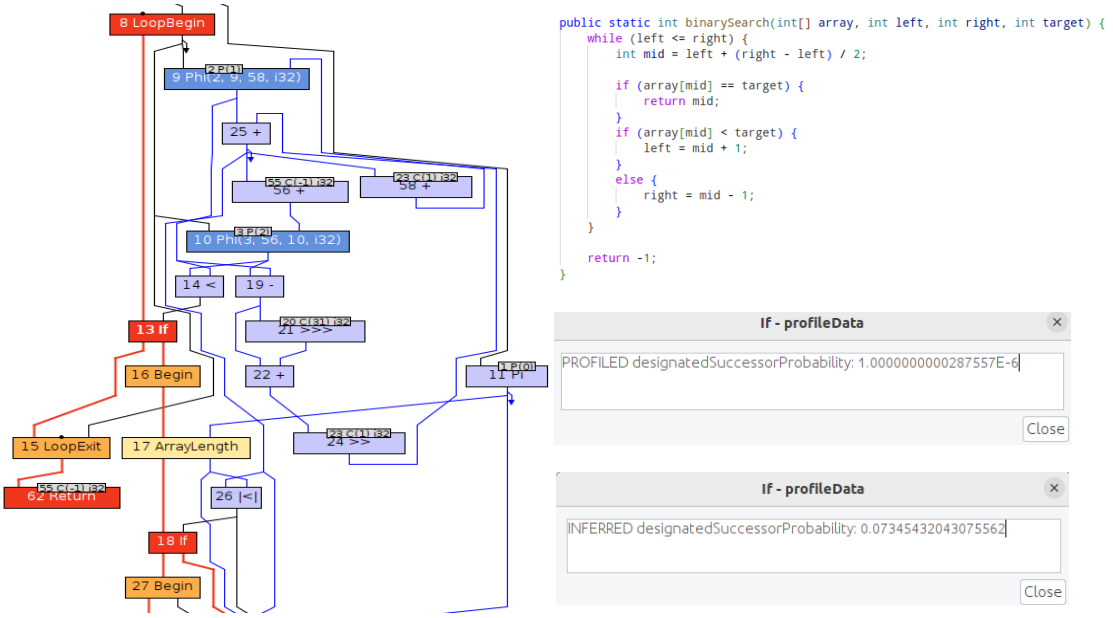

Fig. 1.: Source code of the binary search algorithm (top right), the corresponding IGV graph (left), and the profiled and predicted probability of executing the true branch of the *13If* node (bottom right).

The GraalSP-PLog tool for analyzing static profiler predictions is crucial for understanding the results of profile-guided optimizations based on profiles predicted by ML models. By analyzing model errors, the set of attributes describing branching instructions can be expanded, or manually defined heuristics can be adjusted to improve the ML model's predictions. Additionally, this approach allows for comparing different machine learning models and their predictions on individual instances. The generated report can also offer insights into which methods need more focus, highlighting where the model performs poorly and where it excels.

This work is being carried out as part of a collaboration project between the Faculty of Mathematics and Oracle Labs. It is partially supported by project 451-03-47/2023-01/200104 of the Ministry of Education, Science, and Technological Development.

Keywords: Oracle GraalVM Native Image, Static profilers, GraalSP, Program Profile

References:

[1] Čugurović, Milan, et al. "GraalSP: Polyglot, efficient, and robust machine learning-based static profiler." *Journal of Systems and Software* 213 (2024).
[2] Moreira, Angélica Aparecida, et al. "Vespa: static profiling for binary optimization." *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021): 1-28.
[3] Rotem, Nadav, and Chris Cummins. "Profile guided optimization without profiles: A machine learning approach." *arXiv preprint arXiv:2112.14679* (2021).
[4] Wimmer, Christian, et al. "Initialize once, start fast: application initialization at build time." *Proceedings of the ACM on Programming Languages* 3. OOPSLA (2019): 1-29.
[5] Würthinger, Thomas. "Extending the Graal compiler to optimize libraries." In *Proceedings of the ACM international conference companion on Object-oriented programming systems languages and applications companion*, (2011): 41-42.