Using Bayesian Learning to Estimate How Hot an Execution Path is

Karim Ali
David R. Cheriton School of Computer Science
University of Waterloo
karim@uwaterloo.ca

Abstract

1. Introduction

In the domain of program analysis, it is always advised to perform code optimizations to the paths that have higher likelihood of occurring in a given run of the program. This prioritization process can be tackled by identifying hot paths (i.e. paths of high frequency of execution) and cold paths (i.e. paths of low frequency of execution). The process of identifying hot paths is called *Program/Code Profiling*. The importance of program profiling arises from the empirical observation that most or all of the execution time of a typical program is spent along a small percentage of program paths (i.e. hot paths) [5]. Not only is hot path identification useful for determining cost/benefit ratios for certain compiler optimizations [4], it is also essential for maintenance [9], data-flow analysis [3], and debugging [7]. There are two approaches to perform program profiling: static, and dynamic.

Dynamic program profiling requires the instrumentation of the original source code. In other words, real-life program traces are recorded and gathered by the program to be analyzed by program developers/maintainers later. Due to the lack of of real-life program traces, the general technique that has been adapted to many domains [8, 12, 6, 10, 11] is to generate the workloads automatically. Such workloads might be useful in carrying out software stress tests, but have not been shown as indicative of actual usage [5].

In the context of static program profiling, no workloads are generated but rather execution paths are enumerated based on edge information from the control flow graph (e.g. 1) of various methods. Consequently, additional information is required to classify the execution frequency of a given enumerated execution path. Otherwise, one can only assume that all execution paths are equally likely to be followed in a given program trace. [5] suggests that more information about a static execution path can be obtained by analyzing the relationship between the relative frequency of some features of a path and its effect on program state. The authors show that common code blocks usually do not affect the program state much. Therefore, it is safe to assume that paths with small impact on program state are more likely to be hot paths.

In this project, we adopt the same approach suggested by Buse and Weimer in [5] to classify execution paths after applying two main modifications to it. Firstly, we statically enumerate execution paths based on the control flow graph of each method individually rather than the control flow graph of each individual class. We adjust the feature description model of the execution path accordingly. We then cluster the statically enumerated paths using a simple k-means clusterer into two clusters (hot and cold). The results from the clusterer are used as the ground truth. Afterwards, we train a Bayesian network Secondly, we evaluate our model using the more recent SPECjvm2008 [1] benchmarks as opposed to the obsolete SPECjvm98 [2] benchmarks.

The rest of the report is structured as follows. In Section 2, we present a motivating example. We then discuss related work in Section 3. In Section 4, we provide an in depth discussion for the implementation details of our model. We provide our the experimental results of our model in Section 5 followed by discussion and some proposed future work in Section 6. Finally, we conclude the report in Section 7.

2. Motivation

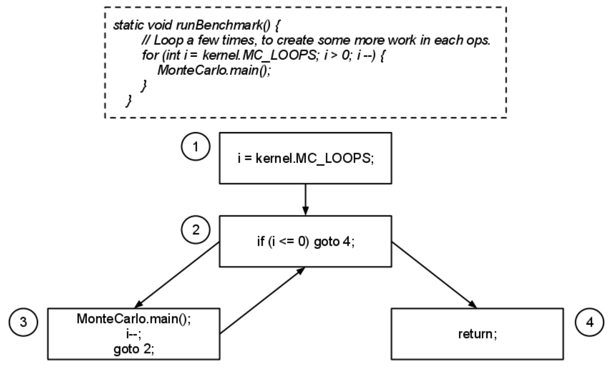
3. Related Work

4. Implementation

4.1. Path Description Model

4.2. Path Enumeration

assumptions: generate cfg for each method (useless to generate for a whole class, some classes do not have defined entry points), a path through the loop represents all paths that take the loop once or more (to prevent unbounded number of paths). generate cfg feature extraction



```
static void runBenchmark() {
    // Loop a few times, to create some more work in each ops.
    for (int i = kernel.MC_LOOPS; i > 0; i--) {
        MonteCarlo.main();
    }
}
```

Figure 1. An example of a control flow graph for a the *rubBenchmark* method of *SciMark* benchamrk from the *SPECjvm2008* benchmarks [1].

4.3. Dataset Generation

specjvm2008 training set, test set clustering training data to get estimates for hot, cold. show graph of 4 datasets labeling both training and test data to get their ground truth train bayes net with various training data and test the test data against it

5. Experimental Results

6. Discussion and Future Work

7. Conclusion

References

[1] SPECjvm2008 Benchmarks. "http://www.spec.org/jvm2008/". 1, 2

[2] SPECjvm98 Benchmarks. "http://www.spec.org/jvm98/". 1

[3] G. Ammons and J. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 72–84. ACM, 1998. 1

[4] C. Boogerd and L. Moonen. On the Use of Data Flow Analysis in Static Profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88. IEEE, 2008. 1

[5] R. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 144–154. IEEE Computer Society, 2009. 1

[6] G. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia. Automatically generating bursty benchmarks for multitier systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):32–37, 2010. 1

[7] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009. 1

[8] D. Krishnamurthy, J. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, pages 868–882, 2006. 1

[9] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *Software EngineeringESEC/FSE'97*, pages 432–449, 1997. 1

[10] K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007. 1

[11] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006. 1

[12] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for Web-based software systems. *Performance Evaluation: Metrics, Models and Benchmarks*, pages 124–143, 2008. 1