Debugging Performance Failures

Juan Pablo Sandoval, Alexandre Bergel

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

ABSTRACT

An application execution profile has a meaning only when it is compared to another profile obtained from a slightly different executing context. Unfortunately, current profilers do not efficiently support performance comparison across multiple profiles. As a consequence, profiling multiple executions is often realized in an ad-hoc fashion, often resulting in missing opportunities for caching.

We propose multidimensional profiling as a way to repeatedly profile a software execution by varying some variables of the execution context. Having explicit execution variation points is key to precisely understand how a particular feature performance evolves along the version history of the software.

1. PROFILING EVOLUTION

Measuring the execution performance of an application is essentially realized by varying some parameters and profiling the program execution for each variation. Identifying which method is slower, for which argument and on which object is crucial to precisely understand the reason of a slow or fast execution. Moreover, an optimal execution is often used as a target for not-so-optimal executions. Caches are inserted and optimizations are implemented until the performance of a not-so-optimal execution is close enough to the optimal one.

Unfortunately, this work is essentially realized by software engineers in an ad-hoc manner. Set of benchmarks are manually constructed to measure the application performance for each slight variation. Typical variations includes the size of the data input, version of an algorithm or a particular sequence of function executions. As surprising as it may seem, current profilers are either unable to compare multiple executions or offers superficial comparison facilities.

Before going into detail about the existing profilers, consider the following situation that was faced during the development of Mondrian[1], an agile visualization engine. Mondrian displays an arbitrary set of data as a graph, in which each node and edge has a graphical representation shaped with metrics and properties computed from the data.

About two years ago, an optimization was implemented and made Mondrian 30% faster. The optimization was carefully measured with a set of benchmarks.

During the last two years, Mondrian has been in a continuous development. As Mondrian has gained new users, new requirements have been implemented to satisfy user wishes. Whereas the range of offered features has gotten wider, the performance of Mondrian have slowly decreased for some of the benchmarks. The optimization that made Mondrian 30% faster seems to have somehow vanished.

Tracking down the software changes that are responsible for this loose of performance is not easy, essentially because of the lack of adequate tools. Consider the commonly-used Java profilers[2]. xprof[3] is built in the Java virtual machine and is essentially used by the Just-in-time compiler. hprof[4] is the profiler promoted by Oracle. Both xprof and hprof report the CPU time consumption for each method for an application execution. The comparison of two profiles to identify the difference of execution has to be done manually, which is a tedious and laborious task.

JProfile[5] and YourKit[6] are two popular commercial profilers. Both support a comparison of profiles by indicating the difference in absolute and relative CPU consumption time of each method. Although useful to keep track of the overall performance, knowing the difference of method execution time is insufficient to understand the reasons of the performance variation. In addition, the call graph may significantly differ from two profiles, which seriously complicate the analysis.

Understanding the reason of a slow or fast execution, caused by the software evolution, the following questions are relevant:

- *How the performance has evolved over different software revisions?*
- *Which software version is the cause of a drop of performance?*

When applied to our example with Mondrian, xprof, hprof, JProfiler and YourKit are helpless to answer any of these

[1] `http://moosetechnology.org/tools/mondrian`

[2] We have conducted all our experiments in the Pharo programming language. It is however easy to guess how it would have been perform with Java profilers.

[3] `http://bit.ly/xprofiler`

[4] `http://bit.ly/hprofiler`

[5] `http://www.ej-technologies.com`

[6] `http://www.yourkit.com`

DYLA'12, June 12, 2012, Beijing, China

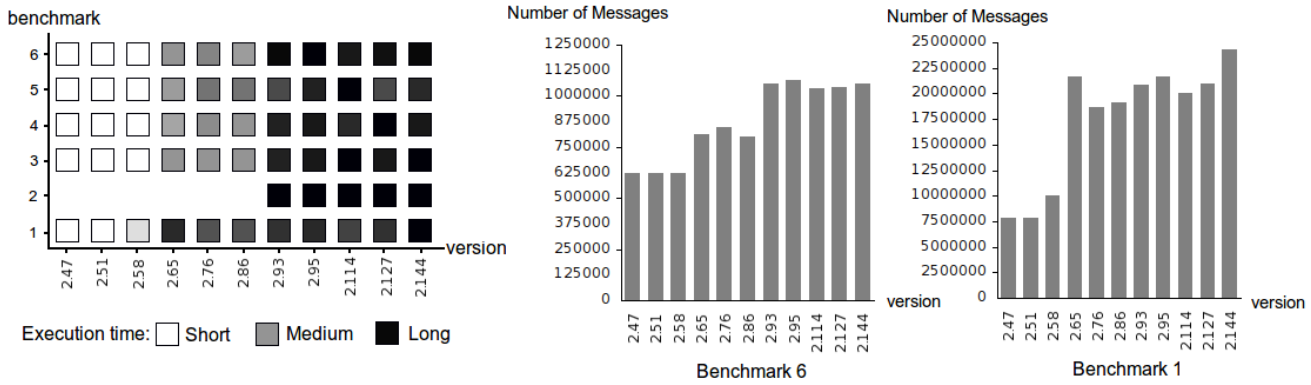

Figure 1: Multidimensional profiling of Mondrian (6 benchmarks are run for 11 software versions).

questions. The reason is that the profile comparison exercised by JProfiler and YourKit does not capture all the variables that these questions refer to, such as the benchmarks and software versions. Being able to profile an application along several variables is the topic of our work.

2. MULTIDIMENSIONAL PROFILING

We define *multidimensional profiling* the activity to reason about a software execution by varying multiple variables related to its execution. Typical variables are benchmarks and software versions. Our objective is to gain a better understanding of a software execution by relating different profiles obtained from slightly different conditions. Opportunities for optimization and ways to minimize resource consumption are then easier to find.

The rational behind multidimensional profiling is that if a software execution is particularly fast or slow for an identified situation (*i.e.,* particular values for the variables), then the situation can be exploited to improve the overall execution.

In a nutshell.

The ingredients to accurately exercise multidimensional profiling are:

- *Define the variation points of the executing environment.* Variation points are defined with a set of variables $(V_1, ..., V_n)$. Each of these variables is associated to a particular aspect of the execution environment, such as software version, benchmark, parameters of a particular methods, instances of a particular class.
- *Specify the variation of the executing environment.* Each variable may either be set to a fixed value, or may iterate over a range of values. Each iteration produces a new profile. To better measure the impact of a variable evolution, it is preferable to have all but one variable fixed. These executions result in a set of profiles $P_1, ...P_m$.
- *Having stable profiles.* Each execution has to be repeatable and isolated from other executions. This means that two profiles P_j and P'_j produced by two identical executions have to be "close enough" to be meaningful.
- *Presenting the results.* Data must be presented for analyze to emphasize the variation of performance. The evolution of V_i has to be unambiguously represented to be able to draw a conclusion about the performance evolution.

Implementation.

We have prototyped Rizel, a multidimensional profiler. Rizel is implemented for the Pharo Smalltalk language. The set of variables that Rizel currently consider are benchmarks and software versions. This means that for a given software, Rizel can:

- run a particular benchmark b for each of the software versions $s_1, ..., s_k$
- run different benchmark $b_1, ..., b_l$ for a particular software version s

Our profiler measures the number of messages sent by each method. It has been shown [3] that counting messages has many advantages over estimating the execution time. For example, counting messages is significantly more stable than directly measuring the time: profiling twice a same execution result in two very close profiles. Counting messages produce stable profiles.

Case study.

We have measured the performance of Mondrian for 6 benchmarks over 11 representative versions. The left-most diagram shows the evolution of the benchmarks against the versions of Mondrian. We see that each benchmark indicates a progressive degradation of the performance of Mondrian. Each of these benchmarks corresponds to a particular feature. Each feature is getting slower, not at the same pace however (*e.g.,* Benchmarks 3-6 are consuming much more time after Version 2.93. Execution time of Benchmark 1 increases after Version 2.65.)

We detail the evolution of Benchmarks 6 and 1 on the right hand side of Figure 1. Both histograms describes an increase of the execution time, which represents a gradual degradation of Mondrian performance.

In our situation, isolating each feature and measuring its performance evolution is key to have a clear understanding of Mondrian performance.

3. RELATED WORK

Comparing program elements between two versions of a program is essential in many areas, including regression testing and software version merging. There are a number of techniques that compare two program versions statically [4, 2]. There are also several techniques for dynamic analysis, specifically using calling context trees (CCT) [6, 1, 5].

Zhuang *et al.* [6] propose a framework for analyzing performance across multiple runs of a program, possibly in a dramatically different execution environment. Their framework is based on lightweight instrumentation technique for building a calling context tree (CCT) of methods at runtime. Adamoli *et al.* present Trevis, an extensible framework for visualizing, comparing, clustering, and intersecting CCTs [1]. On the other hand, our approach compares multiple profiles obtained even from slightly different conditions, counting messages to estimate the execution time and compare the profiles.

Mostafa and Krints [5] present PARCS, an offline analysis tool that automatically identifies differences between the execution behavior of two revisions of an application. PARCS collects program behavior and performance characteristics via profiling and generation of calling context trees.

In our case multidimensional profiling helps to determine which versions and in which execution context should be executed to reproduce the performance failure. It finally compares the executions to detect the possible cause of a drop performance.

4. CONCLUSION & FUTURE WORK

Multidimensional profiling is an innovative approach to measure software performance: crystalizing the performance of each software feature into a set of dedicated benchmarks makes it possible to precisely monitor the global performance of a software against different versions.

As a future work, in addition to the execution time we plan to extract additional metrics such as the distribution of the CPU time consumption over classes and methods. We will then affine our analysis by drilling down to the cause of a slowdown by identifying the method revision responsible of a slower performance.

We will then concentrate on identifying pattern to describe the evolution of feature performance across multiple software versions. As far as we are aware of, all these points have not been considered by the research community on software performance.

5. REFERENCES

[1] Andrea Adamoli and Matthias Hauswirth. Trevis: a context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 73–82, New York, NY, USA, 2010. ACM.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Alexandre Bergel. Counting messages as a proxy for average execution time in pharo. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11)*, LNCS, pages 533–557. Springer-Verlag, July 2011.

[4] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[5] Nagy Mostafa and Chandra Krintz. Tracking performance across software revisions. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 162–171, New York, NY, USA, 2009. ACM.

[6] Xiaotong Zhuang, Suhyun Kim, Mauri io Serrano, and Jong-Deok Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 4–13, New York, NY, USA, 2008. ACM.