Multidimensional Profiling

Juan Pablo Sandoval Alcocer
Department of Computer Science (DCC), University of Chile

I. 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 Mondrian performance variation. In addition, the call graph may significantly differ from two profiles, which seriously complicate the analysis.

Understanding what happened with Mondrian, the following questions are relevant:

- *Has the performance decreased for all the different benchmarks of Mondrian? Or only a few of them?* (Mondrian has dozen of different benchmarks to monitor each feature performance)
- *Is there a particular version of Mondrian that initiated the performance decrease?* (Mondrian has an history long of nearly 1,000 source code versions)

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

II. 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.

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

[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

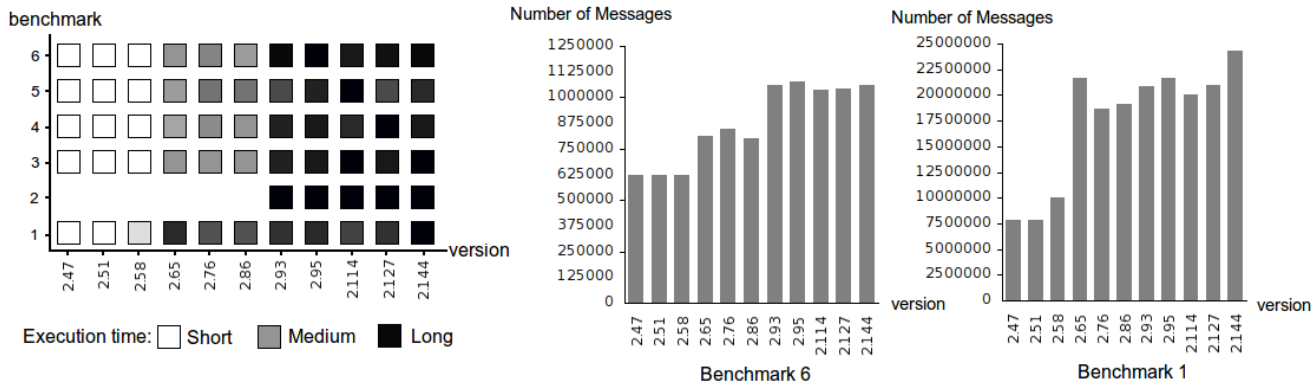

Figure 1. Multidimensional profiling of Mondrian (6 benchmarks are run for 11 software versions).

$(V_1, ..., V_n)$. 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 [1] 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.

III. 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.

References

[1] 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.