Tracking Performance Failures with Rizel

Juan Pablo Sandoval Alcocer, Alexandre Bergel

Department of Computer Science (DCC), University of Chile
http://users.dcc.uchile.cl/~jsandova http://bergel.eu

ABSTRACT

Understanding and minimizing the impact of software changes on performance are both challenging and essential when developing software. Unfortunately, current code execution profilers do not offer efficient abstractions and adequate representations to keep track of performance across multiple versions. Consequently, understanding the cause of a slow execution stemming from a software evolution is often realized in an *ad hoc* fashion.

We have designed *Rizel*, a code profiler that identifies the root of performance variations thanks to an expressive and intuitive visual representation. Rizel highlights variation of executions and time distribution between multiple software versions. Rizel is available for the Pharo programming language under the MIT License.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.7 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Performance

Keywords

performance, profiling, software visualization, software evolution

1. INTRODUCTION

Software programs inevitably change to meet new requirements [8]. Unfortunately, changes made to source code may cause unexpected behavior at run-time. It is not uncommon to experience a drop in performance when a new software version is released.

IWPSE '13, August 19-20, 2013, Saint Petersburg, Russia

Consider the following situation that has been faced during the development of Roassal, an agile visualization engine[1]. Roassal displays an arbitrary set of data as a graph in which each node and edge has a graphical representation shaped with metrics and properties. Roassal has 218 different versions for which most of them were implemented to either satisfy new user requirements or fix malfunctions. Whereas the range of offered features has grown and Roassal is now stable, the performance of Roassal has slowly decreased. This loss of performance is globally experienced by end users and measured by the benchmarks of Roassal.

Unfortunately, *identifying which of the changes contained in these versions are responsible for this performance drop is difficult.* The reason stems from the fact that state-of-the-art code execution profilers (*e.g.,* JProfiler[2] and YourKit[3]) are simply inappropriate to address our performance drop in Roassal due to[4]:

- *Variations have to be manually tracked* – Exploration of the space *(Benchmarks, Versions)* is realized manually. A set of benchmarks is manually constructed to measure the application performance in each software version. Manually iterating over a large number of benchmarks and/or software versions is highly tedious.
- *Relevant metrics are missing* – The few profilers that are able to compare execution do not consider source code variation metrics. As a consequence, slowdown that occurs in unmodified code may distract the programmer from identifying code changes that introduced the slowdown.
- *Poor visual representation* – Visual support used by profilers cannot adequately represent variation of a dynamic structure and multiple metrics. Concluding what is the cause of the loss of performance requires a significant effort from the programmer.

2. RIZEL

Rizel is a code execution profiler that uses advanced profiling techniques and visual representations to easily identify the cause of a performance drop.

[1] http://objectprofile.com/roassal-home.html
[2] http://www.ej-technologies.com/products/jprofiler/overview.html
[3] http://www.yourkit.com
[4] Our work has been carried out in Pharo. It is however easy to figure out how JProfiler and YourKit would be used if they were written in Pharo.

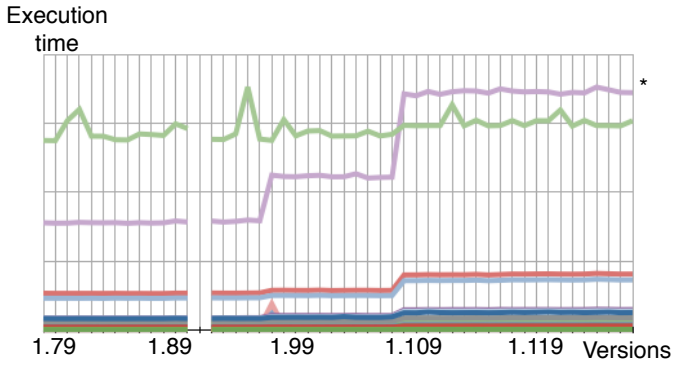

Figure 1: Illustration of a performance degradation: each line describes the execution time of a benchmark across the versions of Roassal.

In particular, *Rizel* innovates by supporting the following two features:

- *run a set of benchmarks over software versions automatically* – By exploring the two dimensional space of software versions and benchmarks, *Rizel* identifies which versions introduce a performance drop and which benchmarks capture performance variations.
- *compare the execution of a benchmark over two software versions* – *Rizel* offers a navigation browser using a polymetric view [7] to highlight differences in the number of method executions and time distribution along the call context tree.

The following subsections elaborate on these two features.

2.1 Tracking performance failure across software versions

Rizel offers an API to script the run of some particular benchmarks over software versions. Consider the following script example:

```
1 data := Rizel new
2       setTestsAsBenchmarks;
3       trackLast: 50 versionsOf: 'Roassal';
4       run.
5 data exportAsCSVFile.
```

Rizel analyzes the last fifty versions of *Roassal*. For each of these versions, *Rizel* profiles the execution of the unit tests. Only the tests that are not modified during these 50 versions are compared. In this particular case, unit tests are appealing since they represent common execution scenarios for Roassal.

Figure 1 shows the evolution of the execution time of Roassal benchmarks. The X-axis indicates the incrementing software versions and the Y-axis indicates the amount of time for a benchmark to execute. Each graph indicates the execution times of a particular test. The graph gives the execution time evolution of a dozen benchmarks. The graph marked with a `*` indicates two jumps of the execution time, which occurred at Version 1.97 and Version 1.108. This graph corresponds to test method `testFixedSize`. Each of these jumps is a drop in performance.

This evolution of benchmarks over multiple software versions gives a global overview of the performance variation of Roassal. From that graph, an analysis of the two software versions may be carried out.

2.2 Comparing two executions

To analyze the two performance drops, we need to compare four executions of the test method `testFixedSize`, versions 1.97 - 1.98 for the first performance drop and versions 1.107-1.108 for the second drop. Comparing two software executions implies monitoring the evolution of several metrics. We use a polymetric view [6, 7] for that purpose.

Rizel offers a second API to compare the profiles of multiple executions. To understand where the first performance drop stems from, the following script compares the execution of `textFixedSize` method in version `1.97` and `1.98`:

```
1 Rizel new
2   compareVersions: '1.97' and: '1.98' of: 'Roassal';
3   usingBenchmark:
4     [ ROMondrianViewBuilderTest run:#testFixedSize. ];
5   visualize.
```

Performance Evolution Blueprint.

Figure 2 shows the difference between version 1.97 and 1.98 of *Roassal*, which corresponds to the first drop of performance mentioned earlier. Each box represents a node of the call context tree (CCT). A node in a CCT represents a method and the context the method was invoked in [2]. Note that two nodes may represent the same method if it was executed in two different contexts.

The color and the shape of a box tells about whether the method in that particular context is slower or faster, or whether it has been executed more often or not.

The color of a node indicates whether the node is slower or whether it had been added/deleted in the new version. *Red* means the node is slower and its method source code has changed in the new version. *Light red* means the node is slower but its method has not been modified. A *yellow* node is new. *Green* means the node is faster and its method source code has changed. *Light green* means the node is faster but its method has not been modified. *Gray* indicates the node has been removed in the new version.

Each node is associated to the following metrics:

- The height of a box represents redistribution of execution time from the first software version with the second version. This redistribution is the difference of the percentage of execution time between versions. A red tall node corresponds to a method that consume more the CPU than in the first version. A green tall node less percentage in the new version.
- The width of a node indicates the difference in the number of times it has been executed. Knowing whether a method is executed more or less often is key to understand the root of a performance variation. The width is proportional to the absolute value of difference between number of executions. A logarithmic scale is used to cope with large variations.

Edges between nodes indicate method invocations. Since we show the difference of two context calling trees, we always have a tree for which the root call is the the top of the tree.

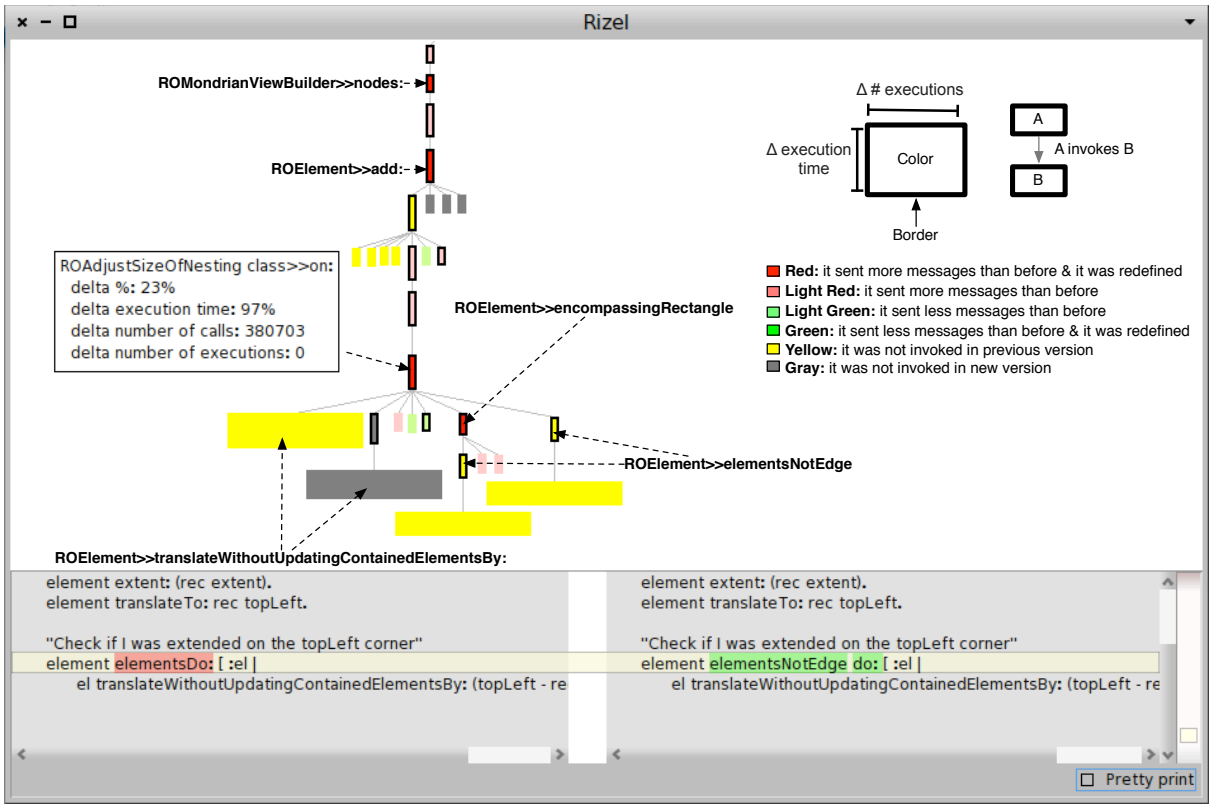

Figure 2: Comparing the execution of testFixedSize test method in version 1.97 and 1.98

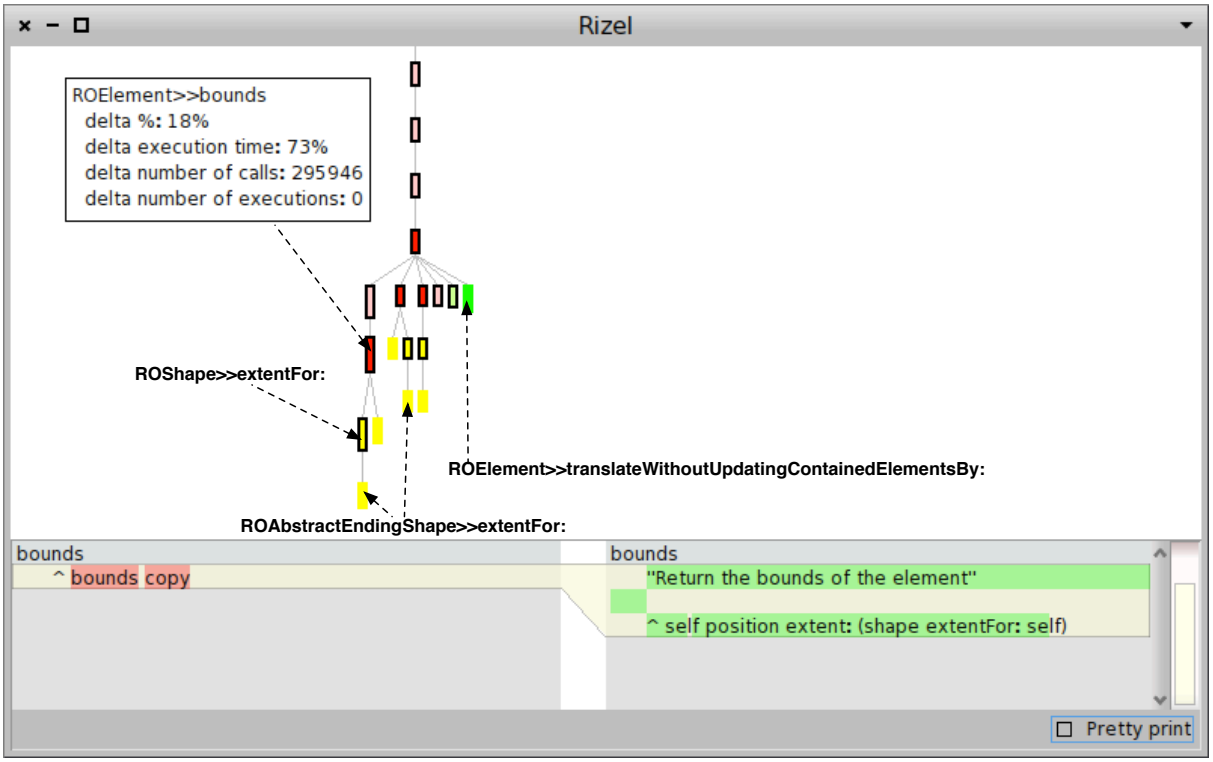

Figure 3: Comparing the execution of testFixedSize test method in version 1.107 and 1.108

Consider the blueprint example in Figure 2 which compares two executions of the benchmark `B` during the first drop in performance. The node `ROAdjustSizeOfNesting class>>on:` is the tallest red box. This means that this method spends more time than the previous version in that context-call and it has been modified in the new version. Figure 2 also shows the difference between the source code of the old version and the new version of method `ROAdjustSizeOfNesting class >>on:`.

One of the reasons for the slow down in `ROAdjustSizeOfNesting class>>on:` is because it executes the method `ROElement>>elementsNotEdge` twice in new version (yellow boxes). One invocation is made directly for it and the other one is through `ROElement>>encompassingRectangle` method, that was also modified.

Calling the method `elementsNotEdge` twice is the root of the first performance drop.

Note that the `ROElement >>translateWithoutUpdating ContainedElementsBy:` method in previous version (gray one) was invoked in different context that in new version (yellow one).

Figure 3 compares the execution of `textFixedSize` test methods for Versions `1.107` and `1.108`.

In Figure 3 the tallest and red node corresponds to the method `ROElement>>bounds`, which means `bounds` takes longer to execute in Version 1.108 and its definition is modified. The lower pane of the *Rizel* browser shows the changes made in this particular method.

The new version of `bounds` is the root of the second performance drop of Roassal.

Note that part of this performance loss is compensated with an optimization made in the method `translateWithout UpdatingContainedElementsBy:` (the only one green box). This method is faster in Version `1.108`.

Interactions.

A common problem in visualising call context trees is the scalability. *Rizel* uses an expandable tree layout in which only relevant nodes are expanded. *Rizel* provides a number of interactions actionable by the end-user to reduce the amount of information in the visualization, in particular: (i) show delta hot path (ii) show subnodes (iii) show all subnodes (iv) hide all subnodes. This interactions are displayed as a popup-list by right-clicking on a node. The thick black border of a box indicates that the node has undisplayed children.

The lower part of the *Rizel* interface compares the source code of both version of the selected method.

Rizel indicates via a contextual popup some data about the variation of the performance for that particular node.

3. IMPLEMENTATION

Rizel uses Spy [5], a flexible and open instrumentation-based profiler framework. Rizel retrieves from a code execution a complete call context tree [9]. Instead of estimating the amount of time in each method (as most code profilers do), *Rizel* counts for each method the number of messages the method has sent. It has been shown that the number of messages a method directly and indirectly sends is linear with the amount of execution time of the method in average [3]. Counting messages accurately estimates the execution time without the inconveniences usually associated with time measurement and execution sampling. For example, counting messages is significantly more stable than directly measuring the time: profiling the same execution twice results in two very close profiles.

4. RELATED WORK

There are a number of techniques for dynamically comparing two program executions using call context trees.

Zhuang *et al.* [12] propose *PerfDiff* a framework for analyzing performance across multiple runs of a program, possibly in a dramatically different execution environment (*i.e.,* different platforms). Their framework is based on a 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]. To scale their tree visualization, they use calling context tree ring charts (CCRC) and just reduce the thickness of a ring segment, which leads to a reduction of the diameter of the visualization [10]. Neither approach considers source code changes metrics to comparing the executions.

Mostafa and Krints [11] 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. It compares CCTs by identifying performance-attribute differences (e.g execution time, invocation count) and topological differences (e.g. added, deleted, modified, and renamed methods). However, the visual support used by *PARCS* cannot adequately represent variation of a dynamic structure and multiple metrics. In our case, we use a polymetric view to visually represent multiple difference metrics between call context trees.

Performance Evolution Blueprint is based on a behavioral evolution blueprint presented by Bergel [4]. They use a call graph to compare executions. The main difference with our blueprint is that we use a call context tree with different metrics, in particular, we use the number of sent messages to estimate the execution time in order to get replicable results. Another difference with this approach is that they only use colors to indicate which method is slower or faster. In our case, we visualize the difference between execution time and number of execution as height and width respectively. This is key to knowing whether or not a method spends more time and if it is executed more times than before.

5. CONCLUSION

This short paper presents *Rizel*, a code execution profiler that offers an effective visual and interactive support to track performance variation across software versions. It visually presents a number of run-time metrics in order to understand the reason for slow execution at a fine grained level. *Rizel* has been successfully used to understand and remove the cause of negative performance variations.

Acknowledgement. Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT (Comisión Nacional de Investigación Científica y Tecnológica de Chile). CONICYT-PCHA/Doctorado Nacional/2013-63130199. This work has been partially funded by Program U-INICIA 11/06

VID 2011, grant U -INICIA 11/06, University of Chile, and FONDECYT project 1120094.

6. 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] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[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] Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Software: Practice and Experience*, August 2011.

[5] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. In *Smalltalks 2010*, 2010.

[6] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[7] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[8] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[9] Philippe Moret, Walter Binder, and Alex Villazón. CCCP: Complete calling context profiling in virtual execution environments. In Germán Puebla and Germán Vidal, editors, *PEPM*, pages 151–160. ACM, 2009.

[10] Philippe Moret, Walter Binder, Alex Villazón, and Danilo Ansaloni. Exploring large profiles with calling context ring charts. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, WOSP/SIPEW '10, pages 63–68, New York, NY, USA, 2010. ACM.

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

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