Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance

Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, Marcus Denker

Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance

Accepted at VISSOFT'13

Juan Pablo Sandoval Alcocer, Alexandre Bergel
Department of Computer Science (DCC)
University of Chile, Chile
jsandoval@dcc.uchile.cl, abergel@dcc.uchile.cl

Stéphane Ducasse, Marcus Denker
RMoD team
INRIA- Lille Nord Europe, France
stephane.ducasse@inria.fr, marcus.denker@inria.fr

***Abstract*—Understanding the root of a performance drop or improvement requires analyzing different program executions at a fine grain level. Such an analysis involves dedicated profiling and representation techniques. JProfiler and YourKit, two recognized code profilers fail, on both providing adequate metrics and visual representations, conveying a false sense of the performance variation root.**

We propose *performance evolution blueprint*, a visual support to precisely compare multiple software executions. Our blueprint is offered by Rizel, a code profiler to efficiently explore performance of a set of benchmarks against multiple software revisions.

***Keywords*- visualization, profiling, software evolution, software execution**

I. Introduction

Software programs inevitably change to meet new requirements [1]. Unfortunately, changes made on 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.

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.

This situation is not anecdotal. A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment [1], [2]. And most of these changes could introduce a loss in performance.

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 at addressing our performance drop in Roassal due to[4] performance variations having to be manually tracked. These profilers do not offer relevant metrics, for instance, they do not consider whether source code has been modified or not. And a poor visualization is used to represent the profile. Dedicated visualizations are efficient support to analyze the execution [3], [4].

This paper proposes *Performance Evolution Blueprint*, a new visualization to visualize the performance difference between (i) one benchmark and two software versions or (ii) two benchmarks and one software version. In addition, we present Rizel, a code execution profiler that automatically explores a two dimensional space *(benchmark, software version)*.

The performance evolution blueprint is summarized in Figure 1. A blueprint is obtained after running two executions. Each box is a method context. Edges are invocations between methods (a calling method is above the called methods). Height of a method is the difference of execution time between the two executions. If the difference is positive (*i.e.,* the method is slower), then the method is shaded in red, otherwise it is green. The width of a method is the absolute difference in the number of executions, thus always positive. Light red / pink color means the method is slower, but its source code has not changed between the two executions. If red the method is slower and the source code has changed. Light green indicates a faster non-modified method. Green indicates a faster modified method. Yellow indicates new methods and gray indicates removed methods. Tooltip gives an extended list of data for the particular methods, including its name, its defining class and the numerical values of the differences.

Our blueprint is offered by Rizel, a code execution profiler for the Pharo programming language [5]. We have successfully used the blueprint and Rizel for understanding the cause of

[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.

[5]http://pharo-project.org

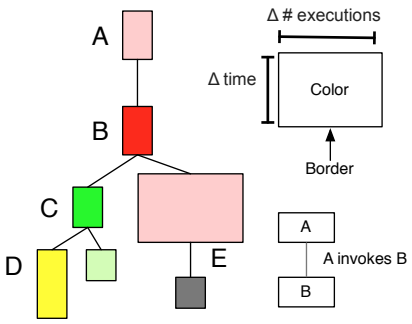

Figure 1. Performance evolution blueprint

a major slow downs in Roassal, a non-trivial software.

The paper is structured as follows. Section II analyzes two commercial profilers and discusses their limitations. Section III details the performance evolution blueprint. Section IV gives the principal characteristics of the Rizel code execution profiler. Section V gives an overview of the related work. Section VI concludes and presents future work.

II. Measuring and Representing Difference of Profiles

Consider JProfiler and YourKit, two popular commercial Java profilers. Both support a comparison of profiles by indicating the difference in absolute and relative CPU consumption time of each method. Although useful in keeping track of the overall performance, knowing the difference between method execution times is often insufficient to understand the reasons for the performance variation.

To illustrate the limitation of current profilers, we pick a contrived, but representative example. The following expression parses a large XML file:

```
new SAXBuilder().build(new File("catalog.xml"));
```

Profiling this expression essentially outputs the following:

```
942{100%}BenchMark.main(String[])
 942{100%}SAXBuilder.build(File)
  942{100%}SAXBuilder.build(URL)
   942{100%}SAXBuilder.build(InputSource)
    495{53%}AbstractSAXParser.parse(InputSource)
    404{43%}SAXBuilder.createParser()
    34{4%}SAXBuilder.createContentHandler()
...
```

The execution time is essentially distributed between the method `parse(...)`, `createParser()`, and `createContentHandler()`.

We artificially introduce a performance drop by slowing down the method `build(InputSource)`. Profiling the new expression produces the following call tree:

```
1905{100%}BenchMark.main(String[])
 1905{100%}SAXBuilder.build(File)
  1905{100%}SAXBuilder.build(URL)
   1905{100%}SAXBuilder.build(InputSource)
    1085{57%}SAXBuilder.parse_proxy(...)*
     699{36%}SAXBuilder.new_method() *
     385{20%}AbstractSAXParser.parse(InputSource)
    784{41%}SAXBuilder.createParser()
    36{2%}SAXBuilder.createContentHandler()
...
```

The new nodes of the call context tree are designed with a "$\star$". We see that total execution times went from 942 ms to 1905 ms. We measured the slowdown using JProfiler and YourKit.

Figure 2 shows the performance degradation of our expression using JProfiler. This figure is obtained by successively executing the parsing expression with and without the slowdown (in the following, we refer to these executions as first and second execution).

Each line represents a method that has been executed in both executions. The indentation represents the control flow. The values before each method name indicate the differences between the two executions. For example, `main(...)` is 104% slower in the second execution and `parse(InputSource)` is 100% faster. The percentage indicates the variation of execution time. Some nodes have a red or green bar to visually indicate this difference.

Figure 3 shows the performance degradation using YourKit. The represented information is essentially the same as with JProfiler. Each method comes with three values: the difference of executions times, the old execution time and the new one. When a method has been added or removed, a 0 is put in place.

JProfiler and YourKit presents a number of limitations, listed below.

Performance variations have to be manually tracked. Exploration of the space *(benchmarks, software versions)* is manually carried out. For each execution, the profiler has to be manually configured to run a particular version. After the run, the profile has to be saved on the file system. After a second profiled run, the two profiles can be compared. Each run comes with a fair load of manual actions. Manually iterating over a large number of benchmarks and/or software versions is tedious.

Relevant metrics are missing. JProfiler and YourKit do not consider whether source code has been modified or not. As a consequence, slowdown that occurs in unmodified methods may distract the programmer from identifying code changes that actually introduce the slowdown. Furthermore, profilers that support profile comparison do not provide information about which method is not called in the new/old version.

Current difference run-time metrics used by profilers are not enough for this kind of analysis. The reason is that variations of the execution time are meaningful when the compared methods are present in both versions. They are useless when a node simply appears in one of the two executions, but not in both. For instance, the variation of the

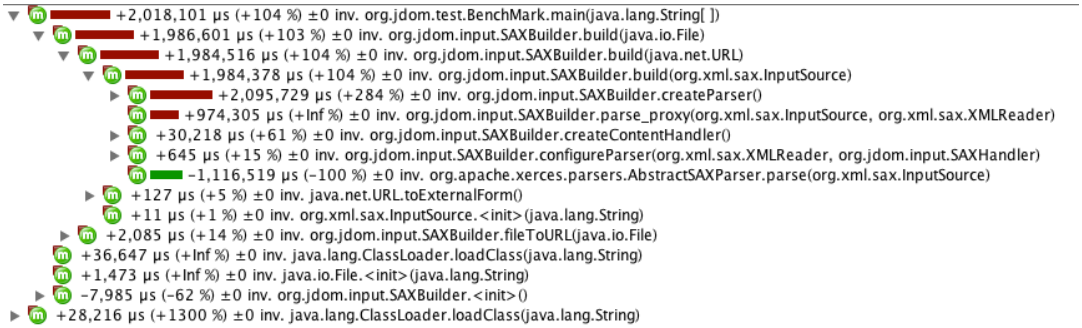

Figure 2. Performance comparison with JProfiler

Name	Time Diff (ms)	Old Time (ms)	New Time (ms)
<All threads>	+1,101	856	1,957
org.jdom.test.**BenchMark.main**(String[])	+1,112	840	1,952
org.jdom.input.**SAXBuilder.build**(File)	+1,114	828	1,942
org.jdom.input.**SAXBuilder.build**(URL)	+1,134	808	1,942
org.jdom.input.**SAXBuilder.build**(InputSource)	+1,155	787	1,942
org.jdom.input.**SAXBuilder.parse_proxy**(InputSource, XMLReader)	+971	0	971
org.jdom.input.**SAXBuilder.new_method**()	+570	0	570
org.apache.xerces.parsers.**AbstractSAXParser.parse**(InputSource)	+400	0	400
org.jdom.input.**SAXBuilder.createParser**()	+589	325	914
org.jdom.input.**SAXBuilder.configureParser**(XMLReader, SAXHandler)	+25	0	25
org.jdom.input.**SAXBuilder.createContentHandler**()	+14	16	31
org.apache.xerces.parsers.**AbstractSAXParser.parse**(InputSource)	-445	445	0
java.net.**URL.toExternalForm**()	-20	20	0
org.jdom.input.**SAXBuilder.fileToURL**(File)	-20	20	0
org.jdom.input.**SAXBuilder.<init>**()	+2	0	2
java.lang.**ClassLoader.loadClass**(String)	-4	11	7
java.lang.ref.**Finalizer$FinalizerThread.run**()	+4	0	4
java.lang.**ClassLoader.loadClass**(String)	-15	15	0

Figure 3. Performance comparison with Yourkit

execution time given by *JProfiler* uses an infinite value, *+Inf* for a new node. Using such an infinite value means that a new method is "infinitely slower" in the second execution than in the first execution, which obviously has little meaning.

Inefficient visual representation. Both JProfiler and YourKit use a textual table augmented with some icons to indicate variations. It is known that textual listing are suboptimal in identifying particular values or patterns [5]. Understanding which performance drop stems from software changes requires significant effort by the programmer. The visual support used by profilers does not adequately represent variation of a dynamic structure and multiple metrics.

III. Performance Evolution Blueprint

Performance Evolution Blueprint visually associates software changes with variation in execution performance. Figure 4 illustrates the same performance drop used earlier. The figure compares two call context trees, each corresponding to an execution of the same benchmark for a particular software version. The method `parseDocument` is slower in the second execution and its definition has been changed. The artificially introduced method `newMethod` is responsible for the slowdown.

This blueprint is a polymetric view [6], meaning shapes and colors of visual elements indicate metric values and properties of the software system considered.

Tree of nodes. Each box represents a node of the call context tree (CCT). A node in a CCT represents a method and the context in which the method is invoked [7]. Two nodes may represent the same method if the method is executed in two different contexts.

An edge between two nodes represents one or more invocations between these two context methods: a calling context method is above the called context method. Since we show the difference between two context calling trees, we always have a tree for which the root call is the the top of the tree.

Colors and shapes. The color and the shape of a box tells whether or not the method in that particular context is slower or faster, whether it has been executed more often or not; or whether it has been added or removed in the second execution. The colors compare two executions that we refer to as first and second executions. *Red* means the node is slower in the second software execution and its method source code is also different. *Light red* means the node is slower, but its method has not been modified. A *yellow* node means the corresponding method call is new in the second execution (*i.e.,* during the first execution the method has not been executed at all or it has, in a different execution context). *Green* means the node is faster and its method source code has changed. *Light green* means the node is faster and its

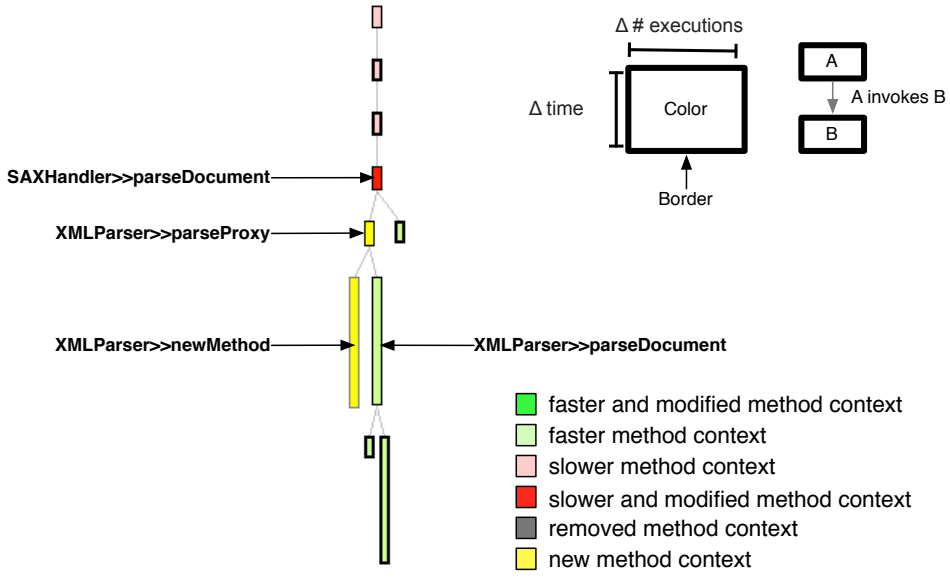

Figure 4. Using Performance Evolution Blueprint to understand the root cause of performance slow down

method has not been modified. *Gray* indicates that the node has been removed in the new version (*i.e.,* the method was executed in that particular context in the first execution, but not in the second).

The shape of a node is given by two metrics, one for the node's height and another for its width.

The *height* of a box represents the redistribution of execution time between two software versions. This redistribution is the percentage difference of execution time between versions. The height is calculated as $H_n = E_n^2 - E_n^1$, where E_n^2 is the relative execution time of the method context n in the second execution and E_n^1 from the first execution. We further have $E_n^2 = T_n^2/T^2$ and $E_n^1 = T_n^1/T^1$, where T_n^2 is the execution time of node n in the second execution; T_n^1 the execution time in the first execution; T^2 the execution time of the whole benchmark in the second execution; T^1 the execution time of the whole benchmark in the first execution.

The height of a box is the absolute value (*i.e.,* always positive therefore) of H_n. A red tall node corresponds to a method that consumes more of the CPU than in the first version (*i.e.,* H_n is negative). And a green tall node less percentage in the new version (*i.e.,* H_n is positive).

The *width* of a node indicates the difference of the number of times in which it has been executed. It is given by $W_n = log(abs(Ex_n^2 - Ex_n^1 + 1))$, where Ex_n^2 is the amount of times n was executed in the second execution; Ex_n^1 is the amount of times n was executed in the first execution. Knowing whether a method is executed more or less often is key in understanding the root of a performance variation. A logarithmic scale is used to cope with large variations. Adding a value 1 is useful so as to not obtain an infinite value due to the log.

In practice, a speedup is represented as a green node. A green node is often associated with a reduction in its amount of executions: Ex_n^2 is often less than Ex_n^1 when the method is faster. Visually, one cannot see whether $Ex_n^2 - Ex_n^1$ is positive or not. A tooltip gives all the numerical data.

The case of new method context. The width and the height of a node is computed by comparing the metrics from the second execution with the first execution. Comparing metric value differences cannot be directly applied with a new or removed method context since only values for one execution are available. The naive approach is to use the value *0* for missing metric values. However, as we have seen for JProfiler and YourKit, this may convey a false sense of variation (*e.g.,* a new method context is indicated as infinitely slower and a removed method context as infinitely faster). Such odd situations are avoided by determining the height and width of a method context based on its children.

The height of a new node is obtained with $H_n = E_n^2 - \sum_{n' \in children(n)} E_{n'}^1$, where

- E_n^2 is the relative execution time of the method context n in the second execution;
- $children(n)$ is the set of children nodes of n;
- $E_{n'}^1$ is the relative execution time of the method context n' in the first execution.

If the node does not have any children, then the height is simply $H_n = E_n^2$. Similarly, a method context that is present only in the first execution (*i.e.,* removed in the second execution) is given by $H_n = E_n^1 - \sum_{n' \in children(n)} E_{n'}^2$. This definition of height has the property of representing a method context as tall if it is *directly* responsible of a slowdown.

For instance, consider Figure 5 that shows two call context trees a) and b) corresponding to first and second execution

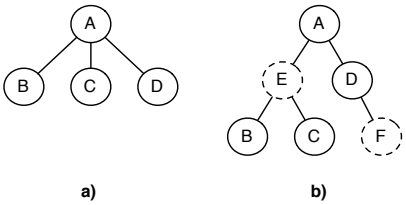

Figure 5. a) CCT of first execution b) CCT of second execution. Nodes E and F are new nodes

respectively. In this figure the node `E` and `F` are new members (added in the second execution), because they just appear in the second call context tree. In the case of node `E` (an inner new node) the height is the differences between the execution time of this node in the second execution and the sum of execution time of nodes `B` and `C` of first execution. Because, if we imagine that node `E` exist in the first execution; we are assuming that the execution time of `E` would be at least the sum of the execution time of the nodes children of `E`. In the case of the leaf node `F` the difference is made with `0`.

The width of a new method context is given by $W_n = log(abs(Ex_n^2 - Ex_{parent(n)}^1 + 1))$, where $parent(n)$ is the node parent of n. Since the amount of execution may greatly vary among methods, a logarithmic scale conveys a visual sense of the difference that remains exploitable. In case a method has not been executed in both executions, the logarithm should remain meaningful. This is the reason for adding 1. A method with a width of 0 is visually represented as 5-pixels wide and means that the method remains constant in the amount of time it has been executed in both executions.

The thick black border of a box indicates that the node has undisplayed children. A thin black border of a box indicates that all children nodes have been displayed.

Consider our previous XML parsing example, given in Figure 4. We applied the same artificial slowdown as with the Java case by adding a slow method and a method that simply calls it. The two methods are indicated in yellow in the figure. The method `parseDocument` has been modified accordingly, to introduce the call to `parseProxy`. This last method calls `newMethod`, the slow method.

The method `XMLParser>>parseDocument` is indicated in green since in the second execution its share of CPU consumption is less than in the first execution.

Figure 4 shows that the method `parseDocument` is slower and it has been modified due to the red color. The method context `parseProxy` and `newMethod` are new in the second execution.

IV. Rizel

Rizel is a code execution profiler that uses advanced profiling techniques and the performance evolution blueprint to easily identify the cause of a performance drop. In particular, *Rizel* innovates by supporting the following two features:

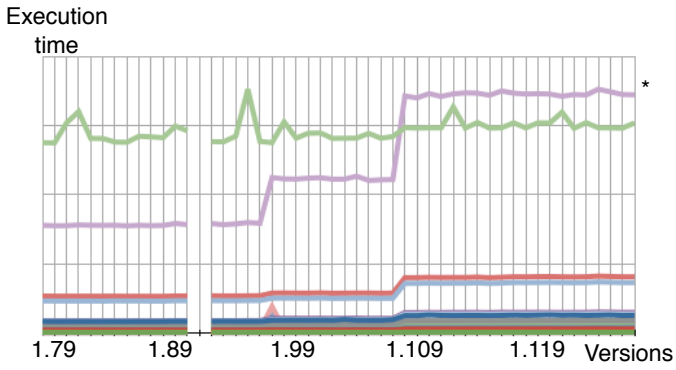

Figure 6. Illustration of a performance degradation: each line describes the execution time of a benchmark across the versions of Roassal.

- *automatically run a set of benchmarks over software versions* – By exploring the two dimensional space of software versions and benchmarks, *Rizel* identifies which versions introduce a performance drop.
- *compare the execution of a benchmark over two software versions* – Based on the performance evolution blueprint, *Rizel* offers a browser to navigate between execution call flow, compare source code, highlight difference in the amount of method executions and time distribution along the call context tree.

The following sections elaborate on these two features.

A. Tracking performance failure across software versions

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

```
1 Rizel new
2     setTestsAsBenchmarks;
3     trackLast: 100 versionsOf: 'Roassal';
4     run.
```

Rizel analyzes the last one hundred versions of *Roassal* against a particular benchmark. Such a benchmark can be externally provided. In this particular scenario, we use unit tests as benchmarks. For each of these 100 versions, *Rizel* profiles the execution of the unit tests. Only the tests that are not modified during these 100 versions are compared. In this particular case, unit tests are appealing since they represent common execution scenarios of Roassal that are commonly performed by end-users.

The result of the profiles may be handled in various forms, including kept in memory to be processed later on using the blueprint or exported to a CSV file.

Figure 6 shows the result of the test executions against the last 100 versions of Roassal. The X-axis indicates the incrementing software versions and the Y-axis indicates the amount of time for a particular benchmark (*i.e.,* execution scenario in this particular case) to execute. Each graph indicates the execution times of a benchmark. The graph gives

the execution time evolution a dozen of 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 the 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 Roassal performance variations. From that graph, an analysis of the two software versions using the performance evolution blueprint may be carried on.

B. Comparing two executions

To analyze the two performance drops, we need to compare four executions of the benchmark `testFixedSize`, versions 1.97 - 1.98 for the first performance drop and versions 1.107-1.108 for the second drop.

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

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

Performance Evolution Blueprint. Figure 7 shows the difference between version 1.97 and 1.98 of *Roassal*, which corresponds to the first drop of performance mentioned earlier.

Consider the blueprint example in Figure 7 which compares two executions of the benchmark `testFixedSize` during the first drop in performance. The node `ROAdjustSizeOfNesting class>>on:` is the tallest box and red, meaning this method spends more time than its previous version in that context-call and it has been modified in the new version. Figure 7 also shows the difference between the source code of the old version and the new version of method `ROAdjustSizeOfNesting class>>on:`.

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

We now focus on the second performance drop, the one that occurred between Version 1.107 and 1.108. Figure 8 compares the execution of the benchmark `testFixedSize` method for these two versions.

In Figure 8, the tallest red node corresponds to the method `ROElement>>bounds`. This color and shape indicates that `bounds` takes longer to execute in Version 1.108 and its definition is modified. The lower text pane of the *Rizel* browser gives the changes made on the source code of 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 green box). This method is faster in Version `1.108`.

Interactions. A common problem to visualize 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: show delta hot paths (*i.e.,* the path from the root node to the leaves that has the greater execution time variation); showing up children nodes; recursively showing up all children nodes; recursively hiding all children nodes. These interactions are displayed as a popup-list by right-clicking on a node. The lower part of the *Rizel* window compares the source code of the two versions of the selected method.

Rizel indicates via a contextual flyby help some data about the variation of the performance for that particular method context node.

The visualizations presented in this section use the interactions described above to highlight relevant call paths. A Rizel visualization solely shows the path from the root node to the leaves that has the greatest execution time variation.

At the beginning *Rizel* shows the path from the root node to the leaves that has the greater execution time variation, as default. We expand the children nodes of red and yellow nodes by right-clicking on these nodes.

V. Related Work

Offering a visual support to compare calling-context trees eases the understanding of a low-level aspect of the execution. Zimmer *et al.* [8] compare the effect of the recompilation effort on the performance: the color of a node indicates whether the method has not been recompiled at all (blue), or whether its recompilation paid off (green) or not (red). By clicking at a node of the graph the user can get detail information about the corresponding method.

Many approaches have been proposed to compare software versions from a syntactical, semantic, and structural point of views [9], [10], [11], [12]. However, these approaches differ from our work since our purpose is to compare performances obtained from executions. Comparing execution is highly relevant since execution comparison is commonly applied

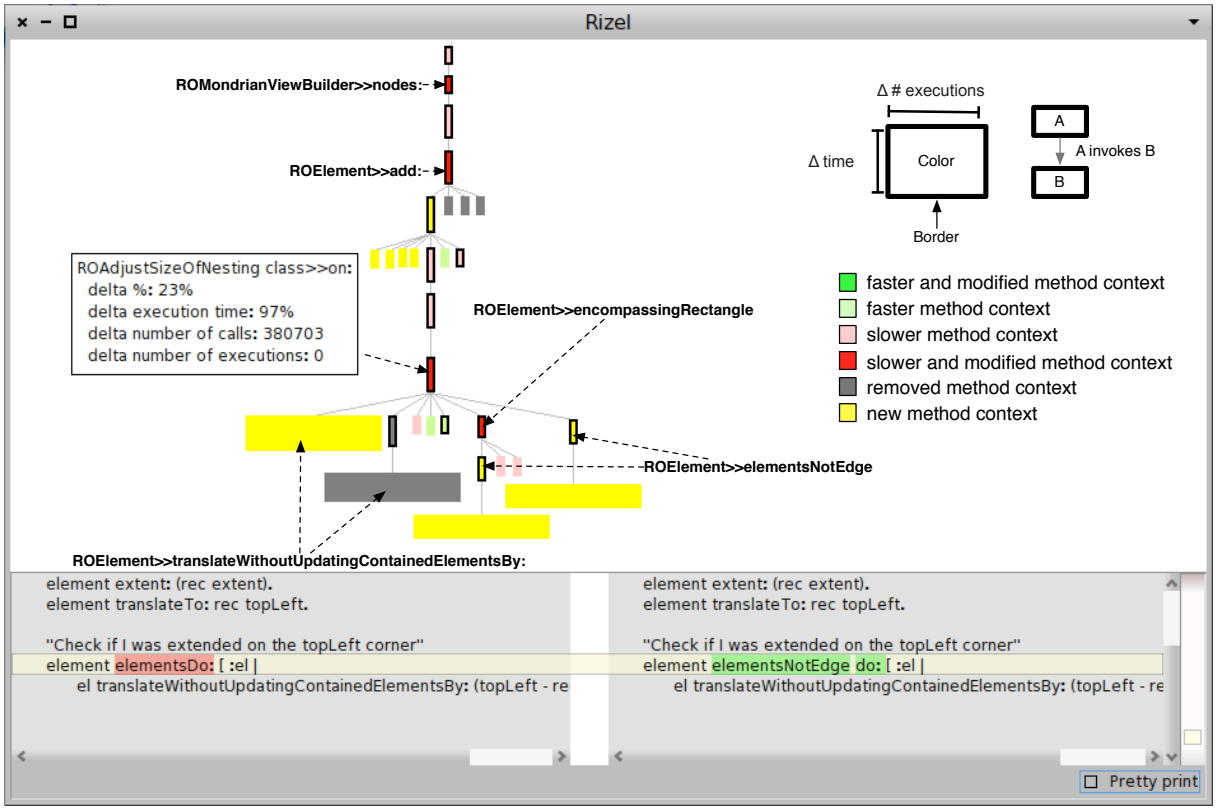

Figure 7. Comparing the execution of testFixedSize test method in version 1.97 and 1.98

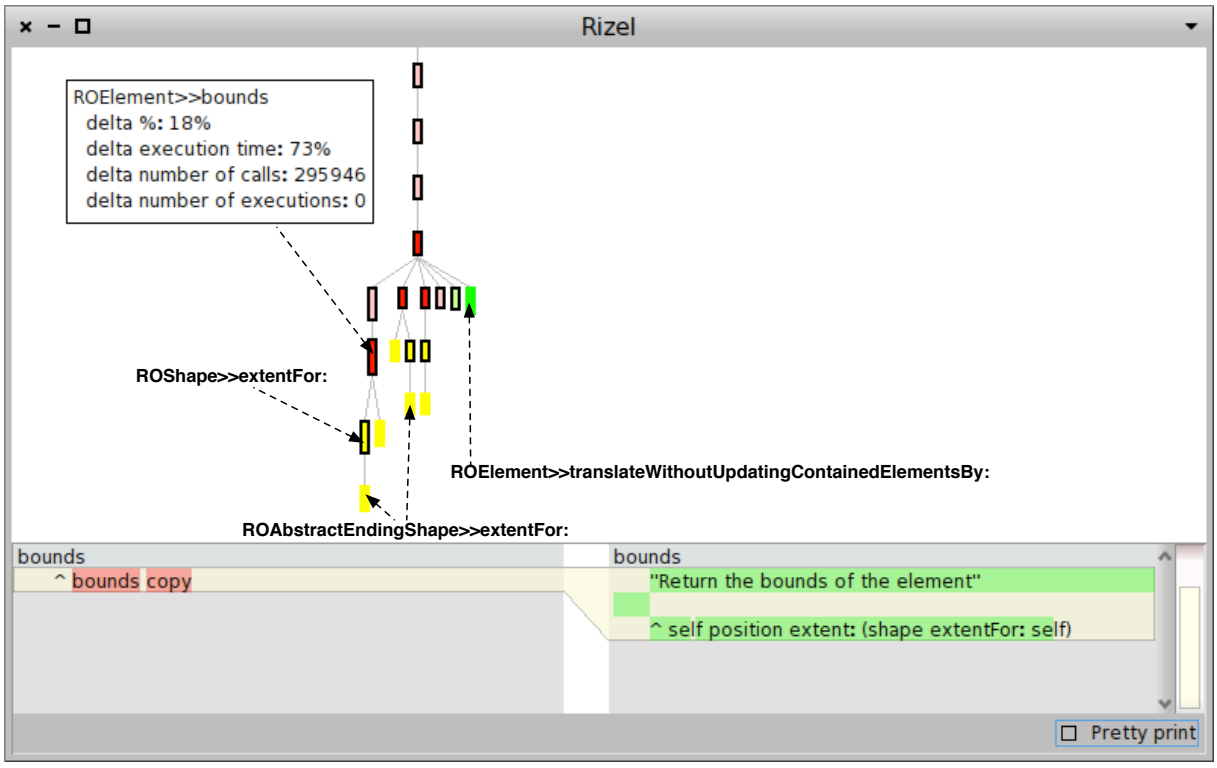

Figure 8. Comparing the execution of testFixedSize test method in version 1.107 and 1.108

to isolate performance failures induced by a program input or a program update [13], [14]. Given two executions of a program, a developer may compare and contrast the variation in order to better understand how and why the program behaved differently [15]. This section positions our work against a number of techniques that compare executions using call context trees and call graphs.

Shasha *et al.* propose the *Tree Transformation* algorithm to compare two ordered trees. An ordered tree is a tree in which the children of each node have total order. Given two trees, the algorithm finds a sequence of insert, delete and rename operations that, when applied to one tree, transforms it to the other. This algorithm was originally designed for abstract trees. However, it was used by Zhuang [16] to compare two call context trees of the execution of a same benchmark in two different platforms or with different inputs. They use the number of operations required to transform one call context tree (CCT) to another, as difference metric. The main disadvantage is the way that the algorithm matches nodes; relying solely on the node label and its post-order in the tree. It ignores the context of the nodes (path from root to the node).

Mostafa *et al.* [17] proposed a technique to compare two call context trees, each obtained from a particular software version. They present *PARCS*, an offline analysis tool that automatically identifies differences between the execution behavior of two revisions of an application. They use as the base the *common tree matching algorithm* to compare two call context trees. This algorithm uses a *node equivalence definition* that considers the method, the context and the order to make sure that two nodes of both trees are equivalent. The problem with the definition is strongly constraining with respect to the node ordering. Mostafa *et al.* propose a variation of this definition called *relaxed node equivalence definition*. This definition is more flexible with regard to ordering. They use the *call-site* information of each invocation as a metric to determine if two nodes are equivalents instead of the order of the invocations.

However, this approach can not detect new inner nodes. Consider a CCT corresponding to first execution and the only difference with the second CCT is a new node added at the root of the tree. This approach will consider that both trees are totally different since all children nodes of the second execution will have different contexts regarding nodes of the first execution. Our approach has a more flexible node equivalence definition regarding call context nodes. Two nodes are equivalent if the call context of one of them (the list of nodes from root to the node) is a sub-sequence of the call context of the other one. It made possible to detect added or deleted inner nodes. Detecting possible added or deleted nodes is useful in understanding the real impact in performance of a new node.

Adamoli *et al.* [18] present *Trevis*, an extensible framework for visualizing, comparing, clustering, and intersecting call context trees. They use call context ring charts to handle large trees [19]. Trevis provides an approach to compute a representative exemplar of a set of context trees. They have implemented two mechanisms to compute such an exemplar: context tree union and context tree intersection. They base the corresponding operations for context trees on the notion of common tree matching. However, their approach does not consider source code metrics.

Bergel *et al.* proposed *Behavioral Evolution Blueprint*, with its goal of comparing the profiles over two software versions. The blueprint shows runtime information as a call graph, where nodes are methods and edges are method invocations [20]. Each node in the call graph is rendered as a box and an invocation is a line that joins two boxes (upper methods invoke lower ones). However, this blueprint only considers whether or not a method spent more or less execution time than its previous version. We improve on this blueprint by adding more metrics, highlighting difference execution amounts and time distribution along the call graph as well as structural differences. This blueprint also does not consider methods call context, making the analysis difficult. For instance, if a method is slower and it is invoked from different changed methods, it is difficult to know which change is responsible for the slow down.

VI. Conclusion

This paper presents *performance evolution blueprint*, a visual approach to understanding the root of a performance slowdown. The blueprint compares the performance of two executions. The blueprint uses a simple metaphor: large red methods are methods that consume the CPU the most, whereas large green methods indicate where the improvements lie.

Our blueprint is a visual and interactive support to track performance variation across software versions. It visually presents a number of run-time metrics to determine the reason for slow execution at a fine grained level. *Rizel*, a code execution profiler that implements the performance evolution blueprint, has been successfully used to understand and remove the cause of negative performance variations.

As future work, we plan to categorize execution patterns that represent the relationship between source code changes, call context tree variations and performance.

Acknowledgment

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT and AGCI, 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. This work has been partially funded by ESUG, the European Smalltalk User Group. We thank the support of Inria for the associated Team Plomo.

References

[1] M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*. London: London Academic Press, 1985.

[2] F. P. Brooks, "No silver bullet," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[3] J. Trümper, J. Döllner, and A. Telea, "Multiscale visual comparison of execution traces," in *Proceedings of the 21st International Conference on Program Comprehension*. IEEE Computer Society, 2013.

[4] C. Zhao, K. Zhang, J. Hao, and W. E. Wong, "Visualizing multiple program executions to assist behavior verification," in *Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, ser. SSIRI '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 113–122.

[5] E. R. Tufte, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press, 2001.

[6] M. Lanza and S. Ducasse, "Polymetric views—a lightweight visual approach to reverse engineering," *Transactions on Software Engineering (TSE)*, vol. 29, no. 9, pp. 782–795, Sep. 2003.

[7] G. Ammons, T. Ball, and J. 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*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 85–96.

[8] S. Zimmer and S. Diehl, "Visual amortization analysis of recompilation strategies," *2010 14th International Conference Information Visualisation*, vol. 0, pp. 509–514, 2010.

[9] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE international conference on Automated software engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.

[10] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 243–252.

[11] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *Software Maintenance, 1992. Proceerdings., Conference on*, 1992, pp. 282–290.

[12] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[13] H. Cleve and A. Zeller, "Locating causes of program failures," in *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 342–351.

[14] A. Zeller, "Isolating cause-effect chains from computer programs," in *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2002, pp. 1–10.

[15] W. N. Sumner, T. Bao, and X. Zhang, "Selecting peers for execution comparison," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 309–319.

[16] X. Zhuang, S. Kim, M. i. Serrano, and J.-D. 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*. New York, NY, USA: ACM, 2008, pp. 4–13.

[17] N. Mostafa and C. Krintz, "Tracking performance across software revisions," in *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. New York, NY, USA: ACM, 2009, pp. 162–171.

[18] A. Adamoli and M. 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*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 73–82.

[19] P. Moret, W. Binder, A. Villazón, and D. Ansaloni, "Exploring large profiles with calling context ring charts," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, ser. WOSP/SIPEW '10. New York, NY, USA: ACM, 2010, pp. 63–68.

[20] A. Bergel, R. Robbes, and W. Binder, "Visualizing dynamic metrics with profiling blueprints," in *Objects, Models, Components, Patterns*, ser. Lecture Notes in Computer Science, J. Vitek, Ed., vol. 6141. Springer Berlin / Heidelberg, 2010, pp. 291–309.