Performance Evolution Matrix: Visualizing Performance Variations along Software Versions

Juan Pablo Sandoval Alcocer
Departamento de Ciencias Exactas e Ingenierias
Universidad Católica Boliviana San Pablo
Cochabamba, Bolivia
sandoval@ucbcba.edu.bo

Fabian Beck
paluno
Universität Duisburg-Essen
Essen, Germany
fabian.beck@paluno.uni-due.de

Alexandre Bergel
ISCLab, DCC
University of Chile
Santiago, Chile
abergel@dcc.uchile.cl

***Abstract*—Software performance may be significantly affected by source code modifications. Understanding the effect of these changes along different software versions is a challenging and necessary activity to debug performance failures. It is not sufficiently supported by existing profiling tools and visualization approaches. Practitioners would need to manually compare calling context trees and call graphs. We aim at better supporting the comparison of benchmark executions along multiple software versions. We propose *Performance Evolution Matrix*, an interactive visualization technique that contrasts runtime metrics to source code changes. It combines a comparison of time series data and execution graphs in a matrix layout, showing performance and source code metrics at different levels of granularity. The approach guides practitioners from the high-level identification of a performance regression to the changes that might have caused the issue. We conducted a controlled experiment with 12 participants to provide empirical evidence of the viability of our method. The results indicate that our approach can reduce the effort for identifying sources of performance regressions compared to traditional profiling visualizations.**

***Index Terms*—Performance, software visualization, software evolution, performance regression.**

Video companion – https://vimeo.com/350953456

Artifact – https://doi.org/10.5281/zenodo.3355414

I. Introduction

Software performance may be affected by multiple source code changes along source code evolution [8], [22]. Comparing software executions is commonly applied to isolate the code changes that produce an unexpected behavior at runtime [7]. Developers compare and contrast performance variations in order to understand how and why the program performance varied unexpectedly. However, state of the art code execution profilers do not properly consider multiple software versions to carry out their analysis. Profilers commonly used are limited to comparing only two executions [26]. In addition, the profile information is visually represented using call context trees and call graphs, forcing developers to manually map performance information to source code changes. As a consequence, performance debugging along multiple software versions is a costly activity [17].

In this paper, we propose *Performance Evolution Matrix*, an interactive visualization technique that contrasts performance variations and source code changes at package, class, and method level. It uses a matrix layout (Figure 1), where each cell shows source code and runtime metrics for the execution of a given software component (a row) in a given version (a column). It allows practitioners to analyze performance of software components along multiple software versions at once. The matrix also supports multiple interactions to display detailed information on demand.

We present a real-life example to introduce all the features of our visualization technique. We also conducted a controlled experiment to provide empirical evidence of the viability of our approach. Our experiment with 12 participants compares the Performance Evolution Matrix and two alternative representations commonly used in software performance debugging: a calling context tree (as offered by TPTP[1]) and the merger of two calling context trees (as offered by JProfiler[2]). The results indicate that our matrix reduces the time to solve a realistic software performance problem. These reductions are statistically significant for some tasks (*e.g.,* spotting a method having a performance and source code variation).

II. Related Work

Execution comparison is used to understand the effect of different runs [28], [33], inputs, or program versions [23]. In addition, several tools have been proposed to help developers to spot and categorize performance regressions [7], [14], [17], [18], [20], [22]. In contrast to these works, our goal is to provide an interactive visualization to assist developers when analyzing performance variations caused by source code changes across software versions. This section positions our work against a number of graphical representations to compare executions of different software versions.

Our visualization approach is inspired by different techniques used in other applications; we combined, tailored, and extended these to a novel and application-specific visual representation. The basic structure of showing evolving quantities in a hierarchy goes back to Timeline Trees [9], which we augmented by arcs to show dynamic dependencies, similar to TimeArcTrees [13]. Integrating small additional timelines for each execution is a form of sparkline visualization [29]. This paper employs a cartesian layout to relate software artifacts metrics with

[1]https://projects.eclipse.org/projects/tptp.platform

[2]https://www.ej-technologies.com/products/jprofiler/overview.html

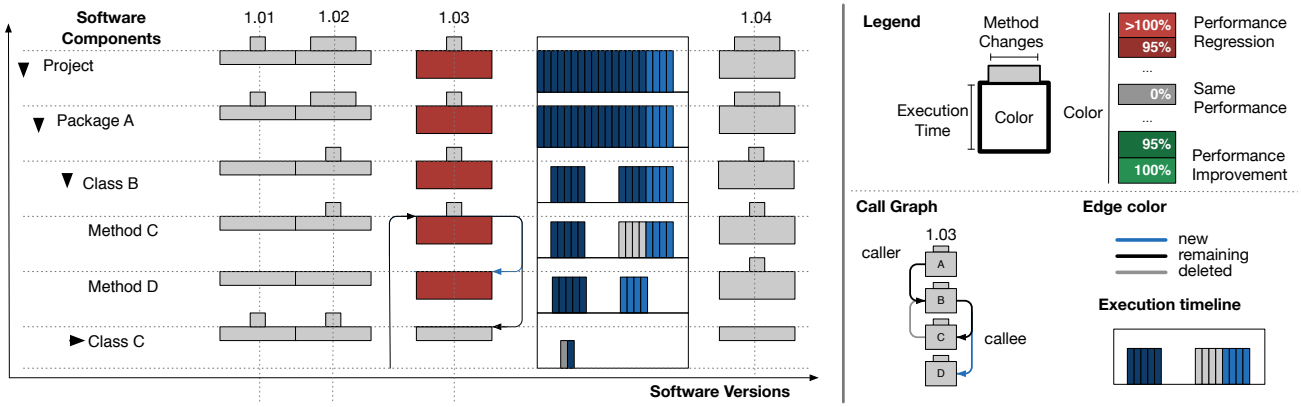

Fig. 1. Performance Evolution Matrix - Illustrating example. The left part shows software components (packages, classes, and methods); the columns represent the versions of these components. Each cell represents a software component in a given version and it is composed of two boxes (top and bottom box). The width of the top box is proportional to the method modifications made in the component in a given version. The height of bottom box is proportional to the components performance in that version. The color of the bottom box varies from red to green indicating if the performance becomes slower or faster in a given version. The execution timeline shown on demand for selected versions detail the time consumption during the execution of the benchmark. Edges show the callee/caller relations between software components (caller = a method that calls another, callee = a method called by another).

software versions [19], [21], [24], [30]. Our visualization also shares similarities with Parallel Edge Splatting [10], which can be used to show evolving static call graphs [10] or dynamic call graphs [2], but not the combination of evolving and dynamic information.

Profiling tools commonly display runtime information in a calling context tree structure. Prior research proposes to visually compare calling context trees to analyze the difference between executions using text tree widgets (as offered by JProfiler), ring charts [1], flame graphs [8], and polymetric views [26]. Holten *et al.* [15] proposed an approach to compare hierarchically organized data, which may be also applicable to call context trees. Trümper *et al.* [28] present a visualization based on icicle plots and edge bundles to compare execution traces. Bergel *et al.* [6] proposed a *Behavioral Evolution Blueprint* to compare execution from two software versions; the blueprint shows the runtime information in a call graph; this blueprint considers whether or not a method has been changed and if this method consumes more or less execution time. However, most of these visualizations are limited to comparing two executions and providing few source code metrics, forcing developers to manually map performance information to source code changes. Our approach shows a call graph of multiple software versions and groups the nodes by classes and packages.

A number of tools show profiling information in the source code view. Waddell *et al.* [31] propose the use of code line color to indicate the execution frequency. Beck *et al.* [3] propose in situ visualizations to display the consumed runtime and threads accessing for each method. The visualization also shows the consumed runtime among callers and callees of each method. However, these approaches are not prepared to compare different versions.

III. Performance Evolution Matrix

The Performance Evolution Matrix is an interactive visualization technique that contrasts performance variations against source code changes. Information is structured along both the static and dynamic structure of an application.

A. Data Model

Our technique is designed to visualize the performance and source code evolution of software components along n software versions $V := \{v_1, \ldots, v_n\}$, where each v_i corresponds to a version name. Our model considers that a project has s methods $M := \{m_1, \ldots, m_s\}$, t classes $C := \{c_1, \ldots, c_t\}$, and u packages $P := \{p_1, \ldots, p_u\}$. Then $W := M \cup C \cup P \cup \{Project\}$ is the set of components in a software project, and $Q \subseteq W \times V$ are the versioned components. Here, a tuple $q = (w, v) \in Q$ denotes component w in version v.

Components in our approach have a hierarchical structure $H = \{h_1, \ldots, h_k\}$, where each hierarchical element $h = (w, \{w_1, ..., w_l\}) \in H$ associates a parent component w with l subcomponents. If the parent component is a package $p \in P$ the hierarchy $(p, \{c_1, \ldots, c_l\})$ defines the containment relation between the package p and the classes $\{c_1, \ldots, c_l\}$. A hierarchy $(c, \{m_1, \ldots, m_l\})$ binds a class $c \in C$ with its methods. All components, except for the root package p_0, are exactly once listed in a list of subcomponents in the hierarchy H. Only packages and classes can be used as parent components, at maximum one time per each package or class.

Our approach also visualizes a sequence of dynamic call graphs $\vartheta = (G_1, \ldots, G_n)$. There is a call graph G_i for each software version $v_i \in V$. A graph G_i is defined by a tuple (W_i, E_i), where $W_i \subseteq W$ form the vertices of the graph and $E_i \subseteq W_i \times W_i$ is the set of directed edges.

B. Layout

Our visualization uses a matrix layout where each column in the matrix represents a software version $v_i \in V$ (*e.g.,* Version 1.01, 1.02) and each row is a component $w_i \in W$. Figure 1 gives an overview of the matrix layout and the components of our visualization. The first column lists the

analyzed software components. Subsequent columns indicate the performance for each version. The column corresponding Version 1.03 has been expanded to show charts indicating execution time variations during the benchmark execution (described in Section III-D). Each row of the matrix shows the performance evolution of a software component (*e.g.,* Package A, Method D). Performance information located at the position $(w, v) \in Q$ uses a performance glyph and execution time frames (only when expanded, as in Version 1.03).

The hierarchy H of the software components is represented using an indented tree layout. Components at each hierarchical level are sorted based on their execution time share. By using this criterion, the components with greater execution time are above components with lesser execution time, thus reducing the effort for practitioners to search for costly components.

Edges between classes and packages are the result of aggregating edges at the method level. For example, a class c_1 invokes another class c_2 if at least one method m_1 of c_1 invokes at least one method m_2 of c_2. Similarly, invocations between packages are deduced from invocations between classes contained in those packages.

Our visualization provides a number of interactions to filter the amount of information shown in the visualization. The software components in a hierarchy $h \in H$ can be hidden or displayed by clicking on the name of the software component (*e.g.,* clicking on Package A hides classes B and C, clicking once more on the package shows the classes again). In addition, we hide software components that: i) do not participate in the benchmark execution or ii) have an execution time below 2%. Note that this threshold may be manually adjusted.

C. Software Component Version Glyph

We represent each software component version with a glyph consisting of two rectangles. The shape and color of a glyph reflect the variation in both the source code and the performance (Figure 1, top right side).

Number of Modified Methods. The width of the top box indicates the number of modified methods in a class, package and project (*modifications*(w, v_i)). We consider that a method has been modified if it has different source code than in its previous version (v_{i-1}). In case of methods, the number of modified methods is zero or one (*modifications*$(m, v) \in \{0, 1\}$), indicating if the method has been modified or not. Visually identifying the number of changes in each component is useful for navigating along the matrix by using interactions and filtering components.

The width can have four sizes: zero, small, medium, and large to represent: i) zero method changes, ii) less than 5 method modifications, iii) less than 10 method modifications and iv) greater than or equal to 10 method modifications (these thresholds are configurable). We use four categories instead of a linear scale for different reasons: versions with no changes should be clearly discernible from those with at least a single change; precise numbers are not important because the number of changed methods is only a rough estimate of quantity of change; widths of boxes are hard to compare if the boxes are scattered across the screen space (but for four different sizes, it should still be possible); there would not be enough space to show an axis with labels to read the precise values. For instance, if the component is a class, then the width is related to the number of modified methods in that class for that version, *modifications*(c, v). In Figure 1, Class B in Version 1.03 has one modified method, which is C. In Version 1.04, Package A has more than 5 modified methods, the top box is therefore larger.

Execution Time. Let us define *time*(w, v) as the accumulated execution time associated to a component w in a version v. By accumulated execution time we refer to the execution time directly consumed by the component w and all its direct and indirect subcomponents. For a method, the height of the bottom box is linear to the accumulated execution time.

The highest box is typically assigned to the execution entry points (*e.g.,* the main method). In the case of classes, packages, and the overall project, the height is related to the accumulated execution time of the most expensive method of the class, package, or project respectively. Note that there is no formal metric to estimate the execution time of a class or a package. For instance, if we consider the execution time of a class as the sum of the execution times of the contained methods, it may poorly represent the class execution time, because it assumes that each method execution time is independent, which is not always true. A class' methods can be directly or indirectly called by each other, making the sum or the average execution time a misleading estimation. For this reason, we use the execution time of most expensive method of a class (or package) to determine the glyph height. For example, in Figure 1 Method C is modified in Version 1.03. The bottom box of this method is high, which means the method takes a great share of the execution time. Note that the width of the bottom box is constant and reflects the matrix column width.

Execution Time Variation. Let be $\Delta\, time(w, v_i)$ the execution time variation between version v_i and v_{i-1} as follows:

$$\Delta\, time(w, v_i) = (time(w, v_i) - time(w, v_{i-1}))/\, time(w, v_{i-1}) \quad (1)$$

The color of the bottom box denotes the execution time variation of a component w. Red is associated with a slower execution and green with a faster execution in v_i. Colors are normalized to represent different variation values, as shown in Figure 1 (right side). We use a divergent colormap: the color scale uses gray as an intermediate color instead of directly blending from red to green. This is a form of hue-preserving color blending [12], which avoids potentially misleading mixed colors for interpolated values. Despite the use of color, our visualization is readable for users with a color vision deficiency such as red-green blindness: color only acts as an amplifier for change, but is redundant to the encoding of the performance changes in the heights of neighboring boxes.

In Figure 1, the execution time of Version 1.03 is $\Delta\, time(\text{Project,Version 1.03}) \geq 100\%$ longer than the one of Version 1.02 since it is red. Method C is likely to be the cause of the performance regression since it is modified in Version 1.03 and painted in red.

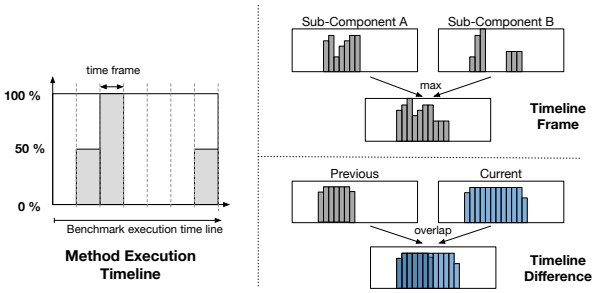

Fig. 2. Method execution timeline, merged execution time frame and execution time difference

Glyph Absence. In case a software component does not exist for a given version (not part of a version or excluded from the execution in a particular version), no glyph is shown. For instance, Figure 4 illustrates this situation where several classes are used in the benchmark only since Version 1.06.

D. Execution Timeline

The execution timeline panel (Figure 2 left side) indicates on a temporal axis for each software component its time consumption share during a benchmark execution for a particular version. Execution timelines are obtained by clicking on the version name (*e.g.,* the label "1.03" was clicked in Figure 1 to obtain the timelines).

Method Execution Timeline. The code execution profiler estimates the execution time share of each method for a time frame of fixed size. This information is used to render the method execution timeline. This diagram shows the evolution of the time share of a method within one execution from left to right. It highlights the time frames where a component participates in the execution. The height ranges from 0% (the method is not used in the time frame) to 100% (the method is used throughout the full time frame). For instance, Figure 2 (left side) shows that a method active half the second time frame (50%), all the third time frame (100%) and half the last time frame (50%).

Merged Timelines. For component containers, such as classes and packages, timelines of the contained components are merged. The merge operation considers the highest time frame of the merged set of time frames as illustrated in Figure 2 (top-right side).

Execution Timeline Difference. For a given software version, the execution timelines show the time share during the execution in that version (in blue) and indicate differences from the previous software version (in gray).

For instance, consider the timeline in Figure 2 (bottom-right side). The top-left bar chart displays the execution timeline of the previous version of the method, the top-right bar chart displays the timeline of the selected version. The two execution timelines are then superposed. In particular, Figure 2 shows that this method spent more time executing in the new version than in the previous version since the gray portion is horizontally shorter than the blue portion. Moreover, both executions are initiated at around the same time.

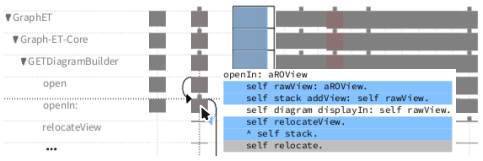

Fig. 3. Popup: Source code differences

E. Call Graph and Source Code Differences

Edges between components $w \in W$ are deduced from the method calls. Clicking on a software component glyph makes the dynamic call graph appear for the component. Incoming calls are located on the left hand side of the selected glyph and outgoing calls are located on the right hand side. A call addition is marked with blue while a deletion is marked with gray. Remaining calls are marked with black. Figure 1 (bottom right) is obtained by clicking on the glyph B. In Version 1.03 and the previous version (not represented in the figure), Component A invokes Component B, itself invoking Component C. The call from B to D is new in Version 1.03. In the previous version, C was calling B, but that call has been deleted in Version 1.03.

Source code differences are shown as a pop-up window when hovering over a method glyph (Figure 3). We use the same color convention as for the call differences to indicate source code line addition and deletion.

F. Implementation

Our visualization is implemented in the Pharo programming language, using the *Roassal* visualization engine [5]. We use a code instrumentation profiling technique to collect the method callees/callers. For measuring the execution time we use *Compteur*, a tool that uses the number of messages sent to estimate the execution time. Compared with other profilers tools, *Compteur* has more replicable and deterministic execution time estimations which is useful to perform comparisons across software versions [4].

IV. Examples

This section describes an application of our visualization approach to address two performance failure issues. The first example focuses on identifying performance and source code evolution while the second example is related to performance variation understanding. The two considered applications are written in the Pharo programming language.

Example 1. Figure 4 offers an overview of the performance evolution of the XML parsing library *XMLSupport*. The benchmark parses a large XML file. It is executed over 19 consecutive versions (from Version 1.03 to Version 1.22). *XMLSupport* has 76 classes and 974 methods.

Figure 4 shows two packages, XML−Parser and XML−Parser −Parser. The library contains other packages, however they are not listed since they have a total execution time share below 2%.

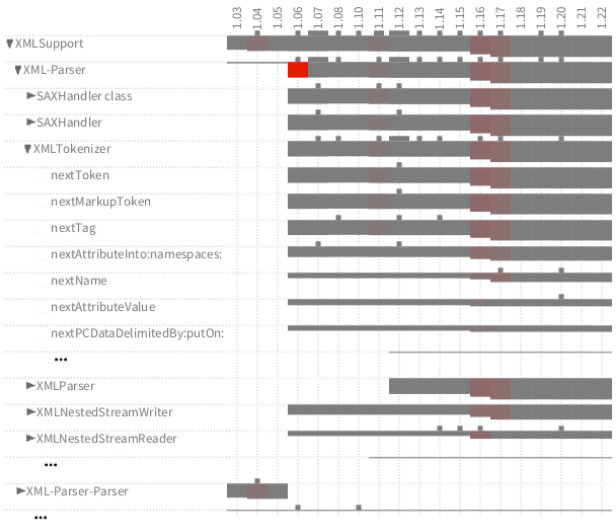

Fig. 4. XMLSupport – versions 1.03 to 1.22

The matrix indicates a relevant fact from the source code evolution: package XML−Parser−Parser is deleted in Version 1.06. According to the execution time of the packages, the visualization suggests that the code from XML−Parser−Parser has been moved to XML−Parser. This code migration leads to a significant increase of execution time of the package XML−Parser in Version 1.06, indicated in red. A mouse click has expanded the package XML−Parser to list 8 classes belonging to that package. The class XMLTokenizer is further expanded.

Some glyphs are missing which indicates that the component is not used. For example, the class XMLParser is introduced in the application or used by the benchmark only starting from Version 1.12.

Figure 4 shows two performance regressions, introduced by Version 1.16 and Version 1.17. This is indicated by two visual cues: the height of the glyphs of the project *XMLSupport* are increasing in Version 1.16 and 1.17 and the glyphs are painted red. Package XML−Parser is changed in Versions 1.16 and 1.17, which causes the performance regressions. Version 1.16 modifies classes XMLTokenizer and XMLNestedStreamReader, as indicated by their glyphs since both have a small top box.

Figure 5 shows the execution timelines of all components in Version 1.16 and Version 1.17. Version 1.16 adds a new method call from method basicNext to the method next of the class XMLPeekableStreamAdapter, causing a performance regression. This method call is highlighted with blue in Figure 5.

Version 1.17 modifies only the method nextName of the class XMLTokenizer. By analyzing the source code difference of this method (by passing the mouse over the method glyph), we conclude that the modification adds a loop to check if the characters of the XML tag name are valid, which also forces multiple calls to the method atEnd. This situation is illustrated by the execution timelines of the method atEnd, depicted in Figure 5. The timelines give an overview on how the

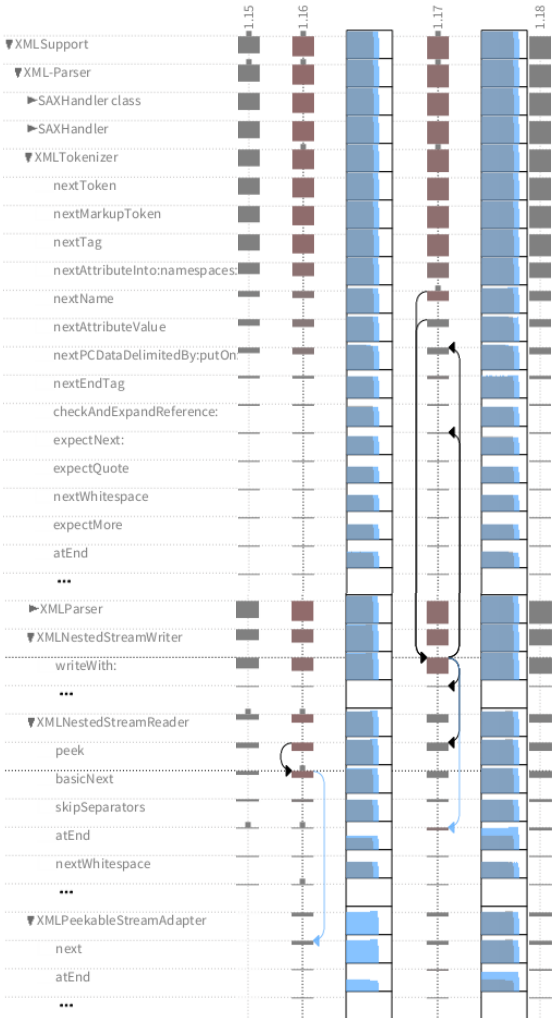

Fig. 5. XMLSupport – Analyzing version 1.16 and version 1.17

method atEnd spends more execution time along the benchmark execution.

Example 2. Figure 6 shows the performance evolution of *GraphET*, a graph drawing library. *GraphET* has 36 classes and 247 methods. As represented in the figure, Version 59 and 60 have been expanded. Overall, Version 59 is slower than Version 58, as denoted with the red taint of the glyph of *GraphET*. The class GETCompositeDiagram is modified as indicated with the top box. Expanding the class reveals that the method transElements has modified and is slightly slower (the glyph is red and has a top box). We have clicked on that method to show the call graph. Method transElements is called by operateElements both in Version 58 and 59, since the incoming arrow is black. In Version 59, transElements calls getPixelsFromValue and heightOfPositiveArea, as indicated with the two blue arrows. These new calls lead to a performance degradation. A close look at the execution time frames of these methods reveals two facts about Version 59: the new call of getPixelsFromValue is costly as indicated by the time frame panel; and the call to heightOfPositiveArea is relatively long. In

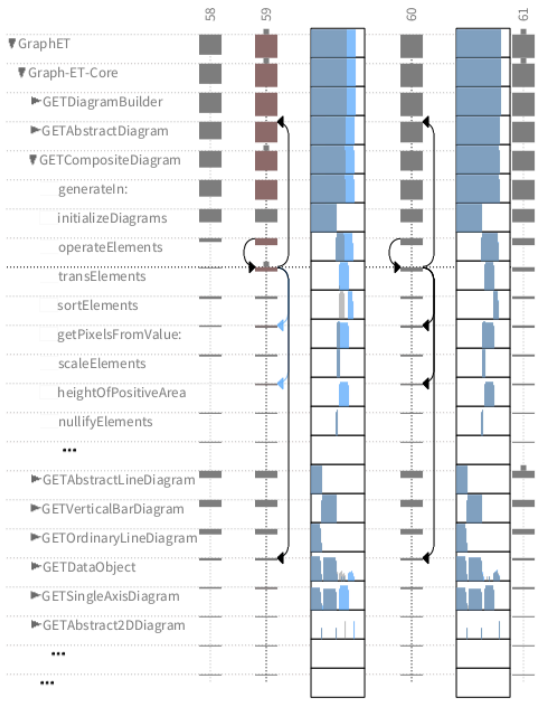

Fig. 6. GraphET – Analyzing versions 58 and 59

addition, note that there is no previous version of this method; therefore we conclude that this method was created in the same version. According to our previous study about profiling software versions and mining performance regressions [25], a considerable number of performance bugs and regressions are related to loops and method call additions. Therefore, we believe that the examples we presented above are representative and meaningful.

V. Experimental Design

We have designed an experiment to provide empirical evidence of the benefits of our visualization for practitioners.

A. Baseline for Comparison

Our experiment compares the effect of the matrix of two realistic tasks against a baseline. This baseline therefore defines the control group of our experiment. We considered two controls in our experiment: CCT and Merged-CCT.

Calling Context Tree (CCT). Many code execution profilers across programming languages structure profiling information as a calling context tree [34]. Each node of the tree represents a method call in a particular calling context. A calling context corresponds to the methods on the call stack just before to execute the call: it is the chain of invoked methods from the root method to the leaf method.

Merged Calling Context Tree (Merged-CCT). A few profilers, including JProfiler and YourKit, are able to compare two executions. A merged calling context tree (Merged-CCT) is used for that purpose [1], [23]. Each node of the first tree is merged with its equivalent node in the second tree. There are a number of equivalence definitions [23]. For our experiment, we used a common equivalence relation between CCT nodes which is: two nodes are equivalent if they correspond to the same method and have the same path to the root node.

B. Research Questions

Identifying performance variations. Our matrix has been designed to reduce the effort in identifying performance variations along a software history. In our experiment, we formalize the notion of "effort" by measuring the correctness and the time taken to identify performance variations. Our first research question is therefore:

RQ1: Does the use of Performance Evolution Matrix increase the correctness and reduce the time to identify performance variations along a software history, compared to source code and CCT-based profiler exploration tools?

Understanding performance variations. Once these variations have been identified, understanding the exact source code change causing it is a relevant aspect to measure. The time taken to correctly understand a variation is therefore a reliable metric to measure the comprehension effort. Our second research question is:

RQ2: Does the use of Performance Evolution Matrix increase the correctness and reduce the time needed to identify the source code changes that cause a performance variation, compared to source code and CCT-based profiler exploration tools?

C. Experimental Setup

Our experiment consists therefore in comparing three representations of the performance history evolution: Performance Evolution Matrix, CCT, and Merged-CCT.

Reducing bias and avoiding confounding factors are essential, albeit difficult. For example, CCT is offered by JVisualVM, and Merged-CCT by JProfiler. JVisualVM and JProfiler are two large profiling platforms with many options. Both platforms have a large scope in which monitoring performance evolution is just a narrow slice of the offered features. For example, both offer extended tool suites to monitor the memory consumption or thread scheduling, which is outside the scope of our experiment. To reduce bias while providing an experiment setting that is practical and realistic, we have taken the following decisions:

Pharo programming environment. We use a programming environment, in which the three treatments (Matrix, CCT, and Merged-CCT) are equally considered. The Pharo programming environment is very light, when compared with Eclipse for example. The tooling offered by Pharo has been trimmed down to let the experiment participants solely use the tools necessary to complete the tasks. We therefore hope to reduce bias introduced by parts of the environment that are irrelevant for our experiment.

Profiling tools. The standard Pharo profiler uses a CCT to structure the profiling information. We will therefore use it

TABLE I
PROJECT UNDER STUDY (ANON= AVERAGE NUMBER OF NODES IN THE CALLING CONTEXT TREE, MD= MAXIMUM TREE DEPTH)

Project	Packages	Classes	Methods	LOC	ANON	MD
Roassal	22	698	3,492	31,949	504	36
Grapher	6	128	801	7,584	989	38
XML-Parser	27	596	4,901	37,454	259	22

in our experiment. The standard Pharo profiler ecosystem originally does not provide a Merged-CCT. We therefore implemented one for our experiment. Our implementation of Merged-CCT is very similar to the one provided by JProfiler. Each node of the Merged-CCT displays three metrics: absolute time difference, method context (local) variation, and global variation.

Supplementary tools. We also provide participants two standard tools: a *code browser*, which allows them review the source code of a particular software version, and a *source code difference*, which indicates source code differences at the method level.

D. Projects and Versions Under Study

We picked three Pharo applications for which performance is seriously considered by the community behind these applications: *Roassal*, *Grapher*, and *XML-Parser*. Table I gives the number of packages, classes, methods and lines of code of these projects. The projects used in our study are considered by the Pharo community as being medium sized. In the case of XML-Parser and Roassal, they have a relatively similar number packages, classes, methods and lines of code, while the Grapher application is smaller.

Project versions. Five software versions were artificially and manually created for each of the three applications of our study, three versions introduce a performance regression and one a performance improvement. The changes are inspired from real code changes causing performance variations [27]. The changes we introduced are equivalent in their complexity across the applications. Each version contains five modified methods causing either a performance improvement or a regression. Note that the number of versions, the source code changes, and their size have been tuned in such a way that the tasks do not take too long for the participants to complete.

Benchmarks. For each application we identified a representative benchmark from those provided by the application authors. Each benchmark reflects a typical usage of the program using a large input set. The depth of CCTs generated by these benchmarks ranges from 22 to 38, and the number of nodes of CCTs in all five versions vary from 259 to 989 on average (ANON, Table I). This depth is comparable with other average stack depths of Java systems [23].

E. Tasks

We designed two tasks to answer the two research questions as described in Table II. Task T1 emphasizes relating source code and performance evolution. For a given application and visualization, each participant identifies in the five software versions methods that have both (i) their source code modified and (ii) execution time changed. Task T2 focuses on a single version. For a given application version and analyzing tool, each participant explains and details the root cause of a performance variation. The result of T1 and T2 will be used to answer the research questions RQ1 and RQ2, respectively. Both tasks are executed using the three treatments, each one with a different application. Each execution uses a distinct and random application and treatment. Overall, these two variables are equally distributed. Randomizing the task execution is relevant to reduce any possible learning effect.

TABLE II
TASKS

T1	**Concern:** Contrasting multiple performance variations across multiple software versions **Description:** The participant identifies and lists the methods that were (i) modified in more than one version and (ii) changed its performance after each modification (greater than 2% regarding the total execution time). **Rationale:** It has been shown that great portion of source code changes that cause a performance variation have direct impact in the performance of the modified method [27]. Therefore, identifying and contrasting methods that change source code and performance in the same version are promising candidates for further analysis. Comparing multiple performance variations is useful for understanding how source code changes are affecting performance and which components are involved along software evolution.
T2	**Concern:** Understanding the roots of performance variations **Description:** The participant identifies the source code changes that cause a performance variation (greater than 2% regarding the total execution time) and explains how these changes affect software performance. For each source code change the participant indicates which methods and classes were involved and in which version the change was introduced. **Rationale:** Identifying the root cause of a performance variation, practitioners may avoid unnecessary performance regressions, improve their self awareness about changes that affect program performance, and make more informed change decisions.

TABLE III
PARTICIPANTS' EXPERIENCE AND TASK ASSIGNMENT
(DEV.=DEVELOPMENT, P.I.= PERF. ISSUES, P.T.= PERF. TOOLS)

ID	Occupation	Task		Experience (years)			
		T1	T2	Dev.	P.I.	P.T.	Pharo
P1	PhD. Student	✓	✓	12	11	8	8
P2	Master Student	✓	✓	5	4	4	4
P3	PhD. Student	✓	✓	5	0.5	0.5	3
P4	Soft. Developer	✓	✓	10	4	1	4
P5	Master Student	✓		2	2	1	0.5
P6	PhD. Student	✓		7	0	0	0
P7	PhD. Student	✓	✓	3	1	4	5
P8	PhD. Student	✓		10	1	1	4
P9	Soft. Developer	✓		6	0	0	0
P10	PhD. Student	✓	✓	6	5	6	3
P11	Soft. Developer	✓		10	10	10	5
P12	Undergrad Student	✓		2	0	0	0

F. Participants

In total, 12 developers participated in our experiment, including 8 postgraduate students, 3 software engineers, and 1 undergraduate student (Table III). We picked these participants because they had experience Pharo, since the projects under study are implemented in this programming language. Their experience in software development ranges from 2 to 12 years, with a median of 6 years.

G. Work Session

The session of each participant is structured as follows:

1) *Preliminary questions* – We first ask the participants to indicate their current occupation and their Pharo experience.
2) *Learning material* – Since we cannot make any assumption about the knowledge of each participant, we provide a description of the three treatments (CCT, Merged-CCT, Matrix) and the supplementary tools. Each item of our material is illustrated with an example. During the reading of the material, participants are encouraged to experiment with a small toy application in order to get familiar with the tools and ask clarifying questions.
3) *T1* – The core of the experiment consists of completing T1, three times, each time using a different treatment (Matrix, CCT, and Merged-CCT) and a different application (Roassal, Grapher, XML-Parser)
4) *T2* – Since this task requires a deeper knowledge, only participants with a solid background in the Pharo programming language and self-confidence in understanding performance-related issues were requested to solve Task T2. Only six participants perform Task T2, each one having more than 3 years of experience in Pharo. Similarly to T1, each participant completed T2 three times, across treatments and applications.
5) *Feedback* – We request feedback about their experience using the three tools.

H. Data Collection

The following data is collected:

- *Completion Time.* We monitor the time needed for the participants to complete each task and treatment.
- *Interaction Events.* We collect the number of mouse clicks and mouse movements. Monitoring the interaction between a participant with the mouse is a reliable proxy of how much the participant is interacting with the visualization in general [11], [16].
- *Behavior.* We observe the participant interaction during the working sessions.
- *Correctness.* We know the correct answers of each task for each project. Note that the result is a set of methods for Task T1 and a set of source code changes for Task T2. We measure the precision and recall of these sets.

I. Pilot Study

We conducted a pilot study with a software developer with strong experience in software engineering, software performance, and source code evolution. We asked him to resolve the two tasks using the three tools: CCT, Merged-CCT, and Matrix. The pilot helped us to improve the tutorial and the task descriptions. The pilot also indicated that the tasks may be completed in a reasonable amount of time. Note that in the experiment, we do not enforce any time restriction.

VI. Results

A. RQ1: Identifying Performance Variations

Research Question 1 (Section V-B) is about measuring performance variations along multiple software versions and relates to Task T1. Table IV reports participants' results.

Precision and recall. Overall, participants perform well. With the CCT representation (*i.e.,* what most code execution profilers output), participants have an average precision of 89% and a recall of 83%. With the Merged-CCT representation (*i.e.,* as JProfiler and YourKit output), participants perform better, with an average precision of 95% and a recall of 91%. With the Matrix, participants obtained 100% of precisions and 98% of recall. However, the difference in the recall and precision to complete Task T1 is not statistical significant (Kruskal-Wallis test[3], $p = 0.3365$ for precision and $p = 0.1318$ for recall).

Completion time. On average, participants were about 3 times faster using Matrix than the two other treatments. The Kruskal-Wallis test coupled with the Post-Hoc Dunn's Multiple Comparison Test indicate that the time to complete T1 using our Matrix is significantly shorter than using CCT or Merged-CCT ($p < 0.0001$); and there is no significant difference between CCT and Merged-CCT.

Mouse interaction. Participants made fewer clicks with the matrix than the two other treatments. On average, participants clicked 126 times using the Matrix, 523 times with the CCT, and 396 times with Merged-CCT. Similarly, participants moved the mouse about 6 times less when performing the task with the Matrix. The Kruskal-Wallis test coupled with the Post-Hoc Dunn's Multiple Comparison Test indicate that the number of clicks and the mouse movement is significantly less using Matrix than using CCT and Merge-CCT ($p < 0.0001$, for both the movement and clicks); and there is no significant difference between CCT and Merge-CCT.

B. RQ2: Understanding Performance Variations

RQ2 is about identifying the method modification causing a performance variation in each software version and relates to Task T2. Table V gives the results.

Precision and recall. Similar to Task T1, participants performed well, with a precision greater than 92% and a recall greater than 83%. From the three treatments, Merged-CCT produced the best results with a precision of 97% and a recall of 100%, however these differences are not statistically significant. (Kruskal-Wallis test, $p > 0.9999$ for precision and $p = 0.1841$ for recall).

Completion time. Merged-CCT performed slightly better than the two other treatments, however, this difference is not significant (Kruskal-Wallis test, $p = 0.6680$).

Mouse interaction. Similarly to Task T1, participants had to use the mouse much less with the Matrix than with the two other treatments: 255 mouse clicks on average for the Matrix

[3]Kruskal-Wallis is a rank-based nonparametric test useful to determine if there are statistically significant differences. It is suitable since we have less than 30 points in our dataset.

TABLE IV
PARTICIPANTS RESULTS FOR TASK T1 (PREC. = PRECISION, MC = MOUSE CLICKS, MM = MOUSE MOVEMENTS, TIME = MINUTES)

ID	Calling Context Tree					Merged Calling Context Tree					Performance Evolution Matrix				
	Prec.	Recall	MC	MM	Time	Prec.	Recall	MC	MM	Time	Prec.	Recall	MC	MM	Time
P1	100	100	358	10,766	16	100	100	308	10,054	14	100	100	52	2,693	5
P2	100	100	678	19,209	20	100	100	478	16,020	18	100	100	86	2,002	7
P3	100	67	378	8,906	12	100	100	338	9,920	16	100	100	100	2,888	5
P4	100	75	430	10,903	14	100	100	480	13,668	20	100	100	112	3,093	8
P5	100	100	358	12,324	14	100	100	232	8,623	15	100	100	144	2,600	4
P6	100	100	444	12,950	19	100	80	238	7,508	13	100	100	136	3,301	4
P7	100	100	504	15,428	14	100	100	306	10,313	11	100	100	126	4,131	8
P8	25	33	418	14,057	17	40	50	234	9,503	17	100	80	158	3,554	6
P9	100	50	562	16,639	27	100	100	902	28,249	42	100	100	184	4,957	6
P10	100	100	988	15,500	21	100	60	232	7,790	16	100	100	108	3,138	5
P11	40	67	732	21,721	29	100	100	776	22,081	28	100	100	184	3,224	8
P12	100	100	424	12,764	13	100	100	226	8,368	13	100	100	126	17	5
median	**100**	**100**	**437**	**13,504**	**17**	**100**	**100**	**307**	**9,987**	**16**	**100**	**100**	**126**	**3,116**	**6**
mean	**89**	**83**	**523**	**14,264**	**18**	**95**	**91**	**396**	**12,675**	**19**	**100**	**98**	**126**	**2,967**	**6**

TABLE V
PARTICIPANTS RESULTS FOR TASK T2 (PREC. = PRECISION, MC = MOUSE CLICKS, MM = MOUSE MOVEMENTS, TIME = MINUTES)

ID	Calling Context Tree					Merged Calling Context Tree					Performance Evolution Matrix				
	Prec.	Recall	MC	MM	Time	Prec.	Recall	MC	MM	Time	Prec.	Recall	MC	MM	Time
P1	100	100	1,054	30,596	39	100	100	416	15,416	26	100	100	138	2,124	21
P2	100	100	1,264	29,897	48	100	100	578	16,271	44	100	100	414	6,005	45
P3	100	100	492	11,563	23	80	100	670	14,610	28	100	100	138	3,647	20
P4	100	50	488	12,210	20	100	100	392	16,266	26	50	50	282	5,379	26
P7	100	75	826	28,120	26	100	100	386	12,452	16	100	75	238	5,052	28
P10	60	75	884	23,949	31	100	100	402	13,224	17	100	75	318	3,493	28
median	**100**	**88**	**855**	**62,035**	**29**	**100**	**100**	**409**	**15,013**	**26**	**100**	**88**	**260**	**4,350**	**27**
mean	**93**	**83**	**835**	**22,723**	**31**	**97**	**100**	**474**	**14,707**	**26**	**92**	**83**	**255**	**4,283**	**28**

versus 835 clicks and 474 clicks, for CCT and Merged-CCT, respectively. These differences are significant (Kruskal-Wallis test, $p < 0.0003$).

C. Participant Feedback

Matrix Layout. All participants argue that the layout and the glyph components are useful for comparing multiple executions across software versions, which is beneficial to solve Task T1.

Calling Context and Participants. Since most of them are used to use calling context based tools, participants concern was that the Matrix does not provide the calling context of method calls. However, this does not affect the ability of participants in identifying and understanding performance variations. In addition, participants find that the color mapping for highlighting the call graph difference was useful to compare versions.

Execution Time Frames. Participants manifest that the callee and caller information together with the execution time frame is useful to understand the roots of the performance variations. However, they also mention that the execution time frame panels were difficult to understand at the beginning, mostly because it visualizes the participation of a method during the execution in a different way than traditional profiling tools. However, participants use this panel mainly to confirm some hypotheses they have about the roots of a performance variation. For instance, when a new method call is added at some point in the execution, this fact is reflected in the execution time frame panel, as we can see in Figure 2.

VII. DISCUSSION

Like all experimental designs, our experiment is subject to some threats to validity [32].

A. Construct Validity

Our experiment has a within-subject design, *i.e.,* participants have three exercises both for Task T1 and Task T2. A within-subject setting helps in coping with a reduced amount of participants, as it is frequently needed when assessing a software engineering population (due to the cost in time and resources to carry out the experiment). We took care to randomize each task to avoid learning effects.

We conducted our experiment in the Pharo programming languages. We used this language for practical reasons: the Pharo reflective API allows one to easily run multiple benchmarks over a large set of software revisions, without incurring a measurable bias [25], [27]. Our visualization is not tied to Pharo: for example, it does not rely on a language construct only provided by Pharo. We do not see any technical obstacles in conducting our experiment in another programming language.

B. Internal Validity

This threat concerns the factors that may influence the treatments effects on the dependent variables.

Tasks. Although we carefully justify the rationale of each program comprehension task, the choice of tasks may bias the results in favor or against the Performance Evolution Matrix. We believe that the matrix layout may represent an advantage in Task T1; however participants were more familiar with the CCT-based tools, which represents an advantage when performing both tasks. The Performance Evolution Matrix removes part of the calling context information by showing only the calls and callers of a particular method at a time. Although participants missed the CCT to solve the Task T2, they were able to resolve this task comparable time, precision, and recall.

Learning effect. There is a potential learning effect in between working sessions; the experience of participant about addressing performance issues increases along each session. To alleviate this threat, the order in which the tools and projects were evaluated was assigned randomly.

Projects Size. The size of the projects and the number of versions could influence the results. We addressed this threat by randomly assigning the projects for each working session. Each combination of *(treatment, application)* has been used the same number of times.

Experimental fatigue. Analyzing the source code changes between two software versions could be a tedious and time-consuming activity. Therefore, participants may get tired along the session. We addressed this threat by controlling the number of versions and the number of source code changes between versions. We considered five different versions for each application and each versions contains five method modifications. This apparently small amount of code modification is key to keeping time below two hours. In the experiment, we do not enforce any time restriction, besides that all participants could finish the tasks in a reasonable time, no greater than 2 hours.

Pharo Experience. Code profiling is an activity that requires a solid understanding of the underlying program infrastructure at runtime. Therefore, the degree of experience in the Pharo programming language could bias the results of Task T2. To mitigate this threat, we exclude novice programmers from our experiment since a profiling activity is likely to be considered by experienced practitioners. Since Task T2 required a deeper knowledge, only participants with a solid background in the Pharo programming language and self-confidence in understanding performance related issues were requested to solve Task T1 and T2. All these participants had more than 3 years of experience in Pharo.

C. External Validity

This refers to the generalizability of the experiment results.

Participants. This concerns that, of all software developers, participants are not representative of the population. We tried to mitigate this threat by having a pool of different participants, which includes graduate students and professional engineers.

Software applications. We used three software applications for our experiment. These applications are well-known within the Pharo community and are considered as relatively large (when compared to the majority of Pharo applications). Moreover, these applications are central to several sub-communities. Although it may be that our results are tied to the software applications we have chosen in our experiment, we have not spotted any indicator that would support this.

Scalability. Like other visualization techniques, our approach may be subject to scalability issues for some scenarios. The screen size may affect the navigability of our visualization approach. The height and width of our approach depend on the number of versions and software components.

Programming Language. Our experiment focused on Pharo applications. We cannot be sure of how much the results generalize to other software projects in other programming languages.

Precision and recall. Table IV and Table V report high precision and recall. Two reasons are possibles: either our tasks are not representative of the difficulties encountered by practitioners, or the participants carefully checked for their answers. Our setting did not impose a time limit, so participants where careful when reporting their answers.

D. Conclusion Validity

This threat concerns the statistical analysis of the results. The results of the 12 participants for Task T1 indicate that Matrix leads to a better precision and recall. However, this difference is not significant. Similarly, participants are slightly slower at completing Task T2. Having a larger pool of participants may result in significant differences.

VIII. Conclusion

In this paper, we propose an interactive visualization to compare multiple performance variations caused along a set of software versions. We present a controlled experiment to show viability of our approach when identifying and understanding performance regressions and improvements. Our results show that participants are statistically significant faster using our visualization at spotting methods with a performance and source code variation compared to other traditional tools. In addition, participants perform about as well as traditional tools when studying the sources of performance problems. Our visualization technique is not meant to replace traditional representations, but to provide a specialized tool when comparing performance variations along multiple software versions.

Acknowledgment

We would like to thank the German Research Foundation (DFG) for financial support within project A01 of SFB/Transregio 161 and research grant 288909335. Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers. We thank Daniel Weiskopf for his comments on an early draft of this paper. We are deeply grateful to Lam Research, who has partially sponsored the work presented in this article.

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 2010, pages 73–82. ACM, 2010.

[2] Fabian Beck, Michael Burch, Corinna Vehlow, Stephan Diehl, and Daniel Weiskopf. Rapid serial visual presentation in dynamic graph visualization. In *Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC 2012, pages 185–192. IEEE, 2012.

[3] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter D. Rey. In situ understanding of performance bottlenecks through visually augmented code. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC 2013, pages 63–72. IEEE, 2013.

[4] 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 2011, pages 533–557. Springer-Verlag, July 2011.

[5] Alexandre Bergel. *Agile Visualization*. LULU Press, 2016.

[6] Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution Profiling Blueprints. *Software: Practice and Experience*, 42(9):1165–1192, 2012.

[7] Cor-Paul Bezemer, E. Milon, Andy Zaidman, and Johan A Pouwelse. Detecting and analyzing I/O performance regressions. *Journal of Software: Evolution and Process*, 26(12):1193–1212, 2014.

[8] Cor-Paul Bezemer, Johan Pouwelse, and Brendan Gregg. Understanding software performance regressions using differential Flame Graphs. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER 2015, pages 535–539. IEEE, 2015.

[9] Michael Burch, Fabian Beck, and Stephan Diehl. Timeline Trees: Visualizing sequences of transactions in information hierarchies. In *Proceedings of 9th International Working Conference on Advanced Visual Interfaces*, AVI 2008, pages 75–82. ACM, 2008.

[10] Michael Burch, Corinna Vehlow, Fabian Beck, Stephan Diehl, and Daniel Weiskopf. Parallel Edge Splatting for scalable dynamic graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2344–2353, 2011.

[11] Stuart K. Card, William K. English, and Betty J. Burr. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics*, 21(8):601–613, 1978.

[12] Johnson Chuang, Daniel Weiskopf, and Torsten Möller. Hue-preserving color blending. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1275–1282, 2009.

[13] Martin Greilich, Michael Burch, and Stephan Diehl. Visualizing the evolution of compound digraphs with TimeArcTrees. *Computer Graphics Forum*, 28(3):975–982, 2009.

[14] Christoph Heger, Jens Happe, and Roozbeh Farahbod. Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE 2013, pages 27–38. ACM, 2013.

[15] Danny Holten and Jarke J. van Wijk. Visual comparison of hierarchically organized data. In *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*, EuroVis'08, pages 759–766, Chichester, UK, 2008. The Eurographs Association; John Wiley; Sons, Ltd.

[16] Kasper Hornbæk. Some whys and hows of experiments in human–computer interaction. *Foundations and Trends in Human-Computer Interaction*, 5(4):299–373, June 2013.

[17] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71. ACM, 2014.

[18] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. *SIGPLAN Notices*, 47(6):77–88, June 2012.

[19] Michele Lanza. The Evolution Matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE 2011, pages 37–42. ACM, 2001.

[20] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024. ACM, 2014.

[21] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: an interactive toolset for source code investigation. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pages 24–31, Oct 2005.

[22] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR 2016, pages 25–36. ACM, 2016.

[23] 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 2009, pages 162–171. ACM, 2009.

[24] Dennie Reniers, Lucian Voinea, Ozan Ersoy, and Alexandru Telea. The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Science of Computer Programming*, 79:224 – 240, 2014.

[25] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139. ACM, 2015.

[26] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. Performance Evolution Blueprint: Understanding the impact of software evolution on performance. In *Proceedings of the First IEEE Working Conference on Software Visualization, VISSOFT 2013*, pages 1–9. IEEE, 2013.

[27] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE 2016, pages 37–48. ACM, 2016.

[28] J. Trümper, J. Döllner, and A. Telea. Multiscale visual comparison of execution traces. In *In Proceedings of 21st International Conference on Program Comprehension, ICPC 2013*, pages 53–62, May 2013.

[29] Edward R. Tufte. *Beautiful Evidence*. Graphics Press, 1st edition, 2006.

[30] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: Visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 47–56, New York, NY, USA, 2005. ACM.

[31] Oscar Waddell and J. Michael Ashley. Visualizing the performance of higher-order programs. *SIGPLAN Notices*, 33(7):75–82, July 1998.

[32] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000.

[33] 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 2008, pages 4–13. ACM, 2008.

[34] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2006, pages 263–271. ACM, 2006.