# Enhancing Commit Graphs with Visual Runtime Clues

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

Harold Camacho Jaimes
*Departamento de Ciencias Exactas e Ingenierías*
*Universidad Católica Boliviana San Pablo*
Cochabamba, Bolivia
hcj1@ucbcba.edu.bo

Diego Costa
*Institute of Computer Science*
*Heidelberg University*
Germany
diego.costa@informatik.uni-heidelberg.de

Alexandre Bergel
*ISCLab*
*Department of Computer Science*
*University of Chile*
Chile
bergel@dcc.uchile.cl

Fabian Beck
*University of Duisburg-Essen*
Germany
fabian.beck@paluno.uni-due.de

***Abstract*—Monitoring software performance evolution is a daunting and challenging task. This paper proposes a lightweight visualization technique that contrasts source code variation with the memory consumption and execution time of a particular benchmark. The visualization fully integrates with the commit graph as common in many software repository managers. We illustrate the usefulness of our approach with two application examples. We expect our technique to be beneficial for practitioners who wish to easily review the impact of source code commits on software performance.**

***Index Terms*—Software Evolution, Software Visualization**

## I. Introduction

Changes in the source code of a software may have a significant impact in the overall performance of a program [1], [2]. Most of the current repository hosting services, including GitHub and GitLab, offer rich exploration tools to compare different software versions. However, this comparison is mostly performed on the source code, thus, discarding non-functional properties, such as software performance (*e.g.,* time and memory).

This paper proposes a lightweight visualization technique, called *Spark Circle*, to compare two software commits along three metrics: the number of modified methods, a benchmark execution time, and the amount of allocated objects. We employ *Spark Circle* to visually represent static and dynamic properties of a software across multiple versions. A *Spark Circle* is obtained after automatically analyzing the source code and running benchmarks on two consecutive software revisions. Our technique is (i) paradigm/language-independent and (ii) easily embeddable within a visual graph of source code commits, as our preliminary application examples illustrate.

Our technique is meant to provide visual support that summarizes differences between consecutive source code revisions in terms of added, removed, and modified methods as well as number of created objects and CPU consumption.

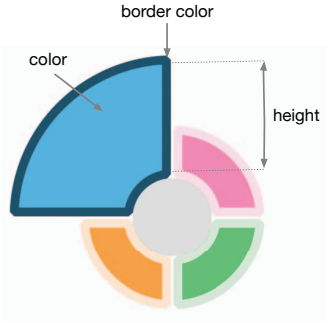


Fig. 1. Illustrating example of *Spark Circle*

The paper makes the following contributions: (i) it describes *Spark Circle*, a new lightweight visualization technique to highlight variations in software metrics, and (ii) illustrates its applicability embedded commit graphs.

Several techniques have been already proposed to monitor the variation of dynamic properties across multiple software versions [3]–[8]. However, none of the existing work proposes a glyph that is easily embeddable within a commit graph of software changes, such as the one offered by popular versioning client systems.

The paper is outlined as follows: Section II describes the *Spark Circle* technique; Section III applies *Spark Circle* to visualize static and dynamic properties; Section IV presents two application examples; Section VI concludes and outlines our future work.

## II. Spark Circle

We propose a glyph to visually highlight variations of metric values. We call this glyph *Spark Circle*; as the name suggests, it is inspired by a sparkline chart. Tufte [9] introduced sparklines as "data-intense, design-simple, word-sized graphics".

978-1-7281-4939-4/19/$31.00 ©2019 IEEE
DOI 10.1109/VISSOFT.2019.00012

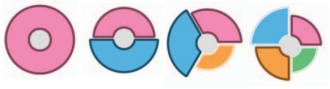

Fig. 2. Spark circles with different number of metrics

*Spark Circle* is a small bar chart drawn in a circular fashion, visually representing metrics through ring sectors. Each ring sector has a unique color. For instance, consider the example in Figure 1, it is a *Spark Circle* with four ring sectors. Each sector has two additional properties: height and border color; where the height is used as main encoding and the border to highlight additional related information. To illustrate this point, Figure 2 displays four spark circles with one, two, three and four ring sectors respectively. Since spark circles are designed to be small, it is necessary to consider that a large number of ring sectors may affect the readability.

## III. Runtime Visual Clues

We use *Spark Circle* to highlight variations between consecutive commits in a commit graph visualization. In particular, we focus on three metrics: number of changed methods, variation of the number of allocated objects, and execution time variation.

Consider that the commit graph visualizes $n$ commits, where $C = \{c_1, \ldots, c_n\}$ is a set of commits.

***Number of Changed Methods.*** Let be $ma_i$, $md_i$, $mm_i$ be the number of added, deleted, and modified methods in version $c_i$, which are computed regarding a previous commit $c_{i-1}$ $(1 < i \leq n)$. Let $mc_i$ be the number of changed methods in version $c_i$, defined as:

$$mc_i = ma_i + md_i + mm_i \tag{1}$$

We map these metrics to a spark circle bar as follows:

- *Color* – Pink.
- *Border Color* – If $mc_i > 0$, the bar has a dark pink border, and if $mc_i = 0$, it has a light pink border.
- *Height* – We categorize the number of changed methods in six categories and assign a static height to each one of these. Figure 3 illustrates each one of these categories.

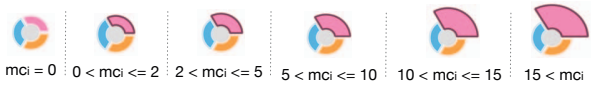


Fig. 3. Number of changed methods ($mc_i$ in commit $c_i$)

We prefer the discrete categorization instead of using a linear scale. Due to the small size of a spark circle, it is difficult to estimate the exact number of changed methods, but for few different sizes and categories, it should be possible. Note that our visualization is not tied to these categories; end users may personalize those according to their needs.

***Object Allocation Variation.*** Memory consumption may be characterized in many different ways (*e.g.,* collection characterization [10], object clusters [11]). We use the number of created objects as a simple metric to summarize the memory footprint. Let $o_i$ and $o_{i-1}$ be the number of allocated objects during the benchmark execution with the software version at commits $c_i$ and $c_{i-1}$, respectively $(1 < i \leq n)$. We define the object allocation variation in a commit $c_i$ as:

$$\Delta(o_{i-1}, o_i) = \frac{o_i - o_{i-1}}{o_{i-1}} \tag{2}$$

We map this metric to a spark circle bar as follows:

- *Color* – Orange.
- *Border Color* – If $\Delta(o_{i-1}, o_i) > 0$, the bar has a dark orange border, and if $\Delta(o_{i-1}, o_i) \leq 0$, it has a light orange border.
- *Height* – The height is proportional to the absolute value of the object allocation variation $abs(\Delta(o_{i-1}, o_i))$.

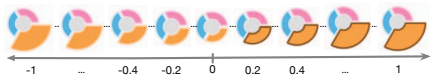


Fig. 4. Object allocation variation ($\Delta(o_{i-1}, o_i)$ in commit $c_i$)

Figure 4 illustrates how the height is related to object allocation variation. The figure illustrates, from left to right, a range from -100% (the new commit reduces the number of create objects by 100%) to $\geq$ 100% (the new commit doubles, at least, the number of created objects). The spark circle bars have a maximum height, which is assigned if $abs(\Delta(o_{i-1}, o_i)) \geq 1$.

***Execution Time Variation.*** Let $t_i$ and $t_{i-1}$ be the average execution time of the benchmark execution with the software version at commit $c_i$ and $c_{i-1}$, respectively $(1 < i \leq n)$. We define an execution time variation in the commit $c_i$ as follows:

$$\Delta(t_{i-1}, t_i) = \frac{t_i - t_{i-1}}{t_{i-1}} \tag{3}$$

We map this metric to a spark circle as follows:

- *Color* – Blue.
- *Border Color* – If $\Delta(t_{i-1}, t_i) > 0$, the bar has a dark blue border, and if $\Delta(t_{i-1}, t_i) \leq 0$ it has a light blue border.
- *Height* – The height is proportional to the absolute value of the execution time variation $abs(\Delta(t_{i-1}, t_i))$.

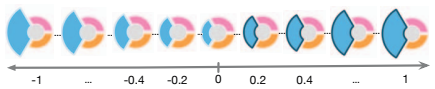


Fig. 5. Execution time variation ($\Delta(t_{i-1}, t_i)$ in commit $c_i$)

Figure 5 illustrates how the height is related to execution time variation. The figure illustrates, from left to right, a range of variations of execution time from $-100\%$ (the new commit reduces the benchmark execution time by 100%) to $\geq 100\%$ (the new commit doubles, at least, the execution time of the benchmark). Similar to the previous metric, to control the bar height boundaries, if the $abs(\Delta(t_{i-1}, t_i)) \geq 1$, the height is maximal.

## IV. Application Examples

We developed an initial prototype to analyze the evolution of two open source projects: *XMLSupport* and *Roassal*. Roassal is a visualization engine [12], [13] and XMLSupport is a XML parser, both are written in the *Pharo* programming language [14].

***Data Collection.*** We developed a script to collect the source code and runtime metrics automatically.

- *Number of Changed Methods*. We use static analysis to automatically compare two consecutive commits $c_i$ and $c_{i-1}$. We use the method signature to detect if a method was added, deleted, or modified. A method is added if it exists at $c_i$ but not at $c_{i-1}$; a method is deleted if it exists at $c_{i-1}$ but not at $c_i$; a method is modified if a method with the same signature exists in both $c_i$ and $c_{i-1}$ but their source code is different.
- *Execution Time*. We used benchmarks produced by the developers of *Roassal* and *XMLSupport*. To measure steady-state performance we first execute a warm up session where we run the benchmark twice; then, we run the benchmarks 25 times while measuring the execution time. As a result, we get 25 time measurements and average them to compare the execution times.
- *Object Allocations*. We use instrumentation to count how many objects are created during the benchmark execution. Since the instrumentation may affect the benchmark execution time, we measure this metric on a separate run, apart from the execution time measurements.

***XMLSupport.*** Figure 6 renders 13 commit versions in the *XMLSupport* main branch. Commits tagged with letters $A$, $B$, and $D$ suffer performance regressions while having a small number of changes only. Due to the size of the blue bar, we conclude that the regression is relatively small. Figure 6 also shows that the commit tagged with $C$ reduces the number of allocated objects, by 66.5%. The pink bar in this commit reveals also that a few changes were done.

***Roassal.*** Figure 7 renders commits done in *Roassal*. Glyphs tagged with A, B, D, and E show that these commits introduce a small performance regression in the project. Pink bars give an overview of how many method changes were done in such versions. Figure 7 shows two branches. Commits tagged with A and C merge left branches with the main branch (the one on the right side). In case of commits that merge two branches, the execution time and object variations is computed using the previous version of the main branch. For instance, the glyph at A shows that the program is slower regarding commit B, and the glyph at C shows that the execution time of the program remains similar to commit D. Note that commit D introduces a performance regression, which remains in commit C.

## V. Related Work

A diverse body of research work focuses on helping developers understand the evolution of source code through the use

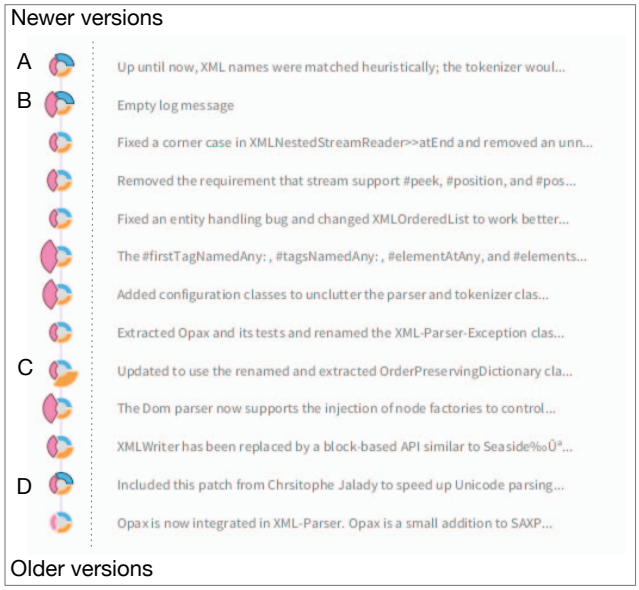


Fig. 6. XMLSupport commit graph visualization. Commits are sorted chronologically, where the newest commit is at the top, and the oldest one at the bottom


Fig. 7. Roassal commit graph visualization. Commits are sorted chronologically, where the newest commit is at the top, and the oldest one at the bottom

of visualizations [15]–[18]. Uquillas *et al.* [16] combines text-based diff information with structural visual representations into a dashboard, that enables developers to explore and understand the changes in their source code. Wilde and German [15] make use of Merge-Trees to enhance the capabilities of conventional graphs on tracking source code changes, and ChronoTwigger [17] focuses on visualizing the co-evolution of test and source code. In such studies, the visualization is focused on structural aspects of code changes, while our approach focuses on performance metrics and source code variation.

Studies have also attempted to bridge the gap between workflows in software development and performance analysis. Beck *et al.* [19] propose the embeddings of run-time metrics in the source code, and Cito *et al.* [20] build upon this concept by developing an approach that provides live performance feedback from production systems. In both studies, the embedded run-time metrics can be used (during development) to highlight potential performance issues at code level. Our approach focuses on visualizing performance metrics across multiple versions and can be combined with above mentioned approaches to increase performance awareness of developers.

Our closest related works are studies that combine the visualization of software evolution with tracking of performance attributes.

Sandoval *et al.* [5] introduce the performance evolution blueprint, a visualization that contrasts source code changes with performance and call context trees. Similarly, Bezemer *et al.* [4] propose the use of flame graphs to compare performance and call context trees of two software versions. In the same direction, Sandoval *et al.* [21] an interactive visualization to compare multiple performance variations caused along a set of software versions. It shows the evolution of source code and performance metrics at different level of granularity based on a matrix layout. Tarner *et al.* [3] discuss various visualization approaches to compare runtime statistics on different executions, including scenarios with source code changes. While also tackling the problem of visualizing software evolution, our work proposes a visualization better coupled with clients of popular versioning systems.

## VI. Conclusion

The Git version-control system plays an essential role in the way software are nowadays built and most of git clients offer a representation of commit graphs. In this work, we propose to enrich commit graphs with a small, focused, and expressive glyphs indicating relevant performance and source code metrics. In particular, memory consumption, execution time, and source code changes. We present two examples to illustrate the usability of our visual glyph.

We believe that the metrics we used and their visual mapping are useful to spot versions with performance anomalies. However, the design may change depending on the objective of the analysis. There is a large range of metrics that can be encoded as spark circles. For instance, focusing on code quality rather than performance, developers could use our glyphs to visualize changes, code size, complexity, coupling, and cohesion metrics as part of the commit graph.

As a future work, we plan to evaluate the impact of *Spark Circle* on software development. We plan to conduct a controlled experiment (measuring the causality between the presence of spark circles and the efficiency of some maintenance tasks) and a field study (measuring the impact on software engineering in an organization and physical environment).

## Acknowledgment

We are deeply grateful to Lam Research and SEMANTICS S.R.L. for their continued support. This work has been partly funded by Deutsche Forschungsgemeinschaft (DFG, research grant 288909335) and Baden-Württemberg Stiftung (project "Visual Reporting of Performance and Resilience Flaws in Software Systems"). We also thank Schloss Dagstuhl and GI for sponsoring the GI-Dagstuhl seminar on "Visualizing Systems and Software Performance (July 2018); the seminar influenced this work and collaboration.

## References

[1] J. P. Sandoval Alcocer and A. Bergel, "Tracking down performance variation against source code evolution," in *Proceedings of the 11th Symposium on Dynamic Languages*. ACM, 2015, pp. 129–139.

[2] 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.

[3] H. Tarner, V. Frick, M. Pinzger, and F. Beck, "Exploring visual comparison of multivariate runtime statistics," in *Proceedings of the 9th Symposium on Software Performance*, 2018.

[4] C. P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 535–539.

[5] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance Evolution Blueprint: Understanding the impact of software evolution on performance," in *In Proceedings of the First IEEE Working Conference on Software Visualization*. IEEE, 2013, pp. 1–9.

[6] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente, "Learning from source code history to identify performance failures," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 37–48.

[7] X. Zhuang, S. Kim, M. i. Serrano, and J.-D. 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*. ACM, 2008, pp. 4–13.

[8] A. Bergel, F. B. nados, R. Robbes, and W. Binder, "Execution profiling blueprints," *Software: Practice and Experience*, vol. 42, no. 9, pp. 1165–1192, 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1120

[9] E. Tufte, *Beautiful Evidence*. Graphics Press, 2006.

[10] S. M. Alexandre Bergel, Alejandro Infante and J. P. S. Alcocer, "Reducing resource consumption of expandable collections: The pharo case," *Science of Computer Programming*, vol. 161, pp. 34–56, 2018, advances in Dynamic Languages.

[11] A. Infante and A. Bergel, "Object equivalence: Revisiting object equality profiling (an experience report)," in *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, ser. DLS 2017. New York, NY, USA: ACM, 2017, pp. 27–38.

[12] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Agile visualization with Roassal," in *Deep Into Pharo*. Square Bracket Associates, 2013, pp. 209–239.

[13] A. Bergel, *Agile Visualization*. LULU Press, 2016. [Online]. Available: http://AgileVisualization.com

[14] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Square Bracket Associates, 2013. [Online]. Available: http://books.pharo.org/deep-into-pharo/

[15] E. Wilde and D. German, "Merge-Tree: Visualizing the integration of commits into Linux," in *Proceedings of the 4th Working Conference on Software Visualization*. IEEE, 2016, pp. 1–10.

[16] V. U. Gómez, S. Ducasse, and T. D'Hondt, "Visually characterizing source code changes," *Science of Computer Programming*, vol. 98, pp. 376–393, 2015, Special Issue on Advances in Dynamic Languages.

[17] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "Chrono-Twigger: A visual analytics tool for understanding source and test co-evolution," in *Proceedings of the 2th IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 117–126.

[18] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind, "System evolution tracking through execution trace analysis," in *13th International Workshop on Program Comprehension*, May 2005, pp. 237–246.

[19] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *Proceedings of the 21st International Conference on Program Comprehension*. IEEE, 2013, pp. 63–72.

[20] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall, "PerformanceHat: Augmenting source code with runtime performance traces in the IDE," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 41–44.

[21] J. P. Sandoval Alcocer, A. Bergel, , and F. Beck, "Performance Evolution Matrix: Visualizing performance variations along software versions," in *7th IEEE Working Conference on Software Visualization*. IEEE, 2019.