Exploring Visual Comparison of Multivariate Runtime Statistics

Hagen Tarner[1], Veit Frick[2], Martin Pinzger[2], and Fabian Beck[1]

[1]paluno, University of Duisburg-Essen, Germany
[2]Alpen-Adria-Universität Klagenfurt, Austria

Abstract

To understand program behavior or find performance bottlenecks in their software, developers need tools to efficiently compare runtime statistics collected across multiple executions. As there is a variety of useful metrics, a good visualization needs to be able to handle multivariate data and highlight the most important differences between multiple versions. We identify three scenarios for the comparison of execution-relevant changes, and explore possible visualizations of the gathered multivariate runtime statistics.

1 Introduction

When trying to understand and optimize how fast a program executes, time-based metrics are essential. With modern profiling software it is possible to measure how long the execution of a single method takes, and how often it is executed. Typical metrics for this are *self time* (excluding time waiting for calls), *time* (including time waiting for calls), and the number of method invocations. Gathering these across multiple executions, grants insights into the evolution of the software or allows differential analysis and debugging. However, most modern profiling tools only support the visualization of a single execution or the aggregated, non-differential analysis of multiple executions.

As a basis for such comparison, we consider the visualization of multivariate runtime statistics, which a few works have already discussed. The *Execution Profiling Blueprints* [2] technique visualizes up to four metrics in a glyph-based call or inheritance tree representation. Another glyph-based representation [3, 5], summarizing runtime and call information, embeds the visualizations in the code next to the methods that the respective glyphs refer to. Most related to our approach but restricted to a specific set of metrics, the *Performance Evolution Blueprints* [4] technique extends *Execution Profiling Blueprints* to represent metric changes across software evolution.

2 Use Cases of Execution Comparison

The comparison of runtime behavior can have various applications within software development, for instance, load testing, finding performance bugs, or analyzing computational scalability. We categorize the factors that potentially could influence the execution of a program by the source that the variation of runtime behavior comes from.

Environment Changes: A program needs to run in various environments, such as, different hardware, operating systems, browser versions, or while executing under load.

Input Changes: Every program relies on inputs. This is particularly important for evaluating the practical scalability of a program.

Code Changes: Code modifications, except for refactorings, change the runtime behavior of the program. Comparing different versions of the program would allow for a detection of performance regressions.

3 Visual Design Space

For the visual comparison of the method-level runtime statistics, we can leverage standard visualization techniques for multivariate data. We discuss some of the most frequently used approaches and how they can be extended toward comparison of two program executions. Three basic operations are available for visual comparison: (i) *juxtaposition* (i.e., placing visual elements next to each other), (ii) *superposition* (i.e., placing visual elements on top of each other), and (iii) *explicit encoding* (i.e., computing an explicit difference or aggregated value and visualizing this result) [1].

Grid-based charts use a tabular layout to present the data in a series of similar diagrams. Adding visual comparison capabilities is done by extending the diagram in each cell with one of the visual comparison approaches. Figure 1 demonstrates the approach for juxtaposed bar charts. The resulting visualizations allow for an easy intra-cell comparison, as well as contrasting rows and columns. This approach is useful for visualizing small datasets (or subsets of larger datasets), but by default does not scale well as each method in the dataset creates one row in the grid.

Glyph-based techniques map data values to visual properties of a geometric object to create a unique glyph for an entity. This allows visual comparison via juxtaposition or superposition, but also explicit encoding like demonstrated in *Performance Evolution Blueprints* [4]. For example, Figure 2 shows a superposition of two star glyphs, where the metric val-
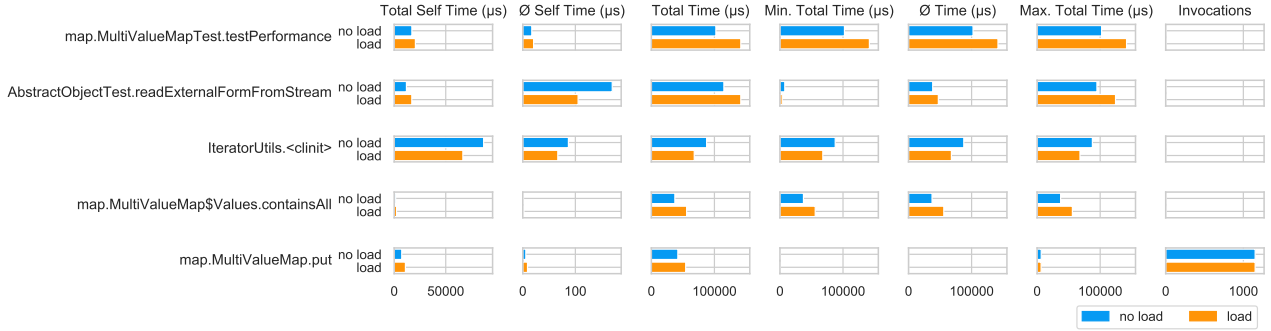
Figure 1: *Environment Changes* scenario: juxtaposed bar charts in a grid-based layout, showing the top 5 methods by Δ *Total Time*.

ues define the distance to a center point of a star-like polygon. Every method in the dataset creates one glyph. Similar to grid-based charts, this approach is able to visualize a few dozens methods, but it has limited scalability as the number of glyphs increases with the number of methods.

Parallel coordinate plots represent the data points as lines connecting the different dimensions that are plotted as parallel axes. For enabling visual comparison, the plot in Figure 3 overlays a set of lines for two color-coded executions. Parallel coordinate plots scale well with the number of methods, but two representatives of the same method are hard to find because there is no visual indication which two lines belong together; juxtaposing two plots would have similar problems.

Scatterplots show entities as points in a usually two-dimensional space and visual comparison can be implemented with different color (superposition) or in separate diagrams (juxtaposition). For visualizing multivariate data, scatterplot matrices show all pairwise combinations of variables. Alternatively, we can leverage dimension-reducing projections to represent the data in a single scatterplot. While being useful to find clusters (projection) or relationships between variables (scatterplot matrices), scatterplots share similar properties with respect to visual comparison as parallel coordinates plots: they scale well, but connecting the representatives of the same method across the two executions is difficult.

4 Application Examples

Next, we discuss examples for each of the three scenarios defined in Section 2. Inspired by Baltes et al. [5], we use the *Apache Commons Collections*[1] project. For this we created a static application that can analyze *JProfiler* output files post-mortem. The resulting data consists of the invocation count, mean, and total *self time*, as well as min/mean/max/total *time* per method.

While we systematically tested each scenario with each visualization approach, we found that the following produced the most legible results. For the sake of brevity and the problems mentionend in Section 3, we included only the top 5 methods for Figures 1 and 2.

Environment Changes: To simulate a changing environment in between consecutive runs of the same program, we used *memtest*[2] to add load to the system. We recorded executions of the program with normal load and afterwards recorded with added full load to CPU and memory. Figure 1 shows a matrix of bar charts with the recorded method as rows and the recorded metrics as columns. Each single plot contains two colored bars: the value recorded without load (blue) and under load (orange). The top row shows high values for all total-time-related metrics (column-wise observation), with each having a visible difference between the two versions (intra-cell observation). When compared to other methods, the difference in total time is highly visible (row-wise observation).

Input Changes: For this scenario, we scaled the number of input data by factor 100 compared to the original run. Figure 2 shows five examples of methods as star plots contrasting the two runs. Each axis of the star plot represent one single runtime statistic. The first and second plot show a recurring pattern for the metrics related to total time and self time. This indicates a caller–callee relationship. This assumption was later verified by examining the code of the corresponding methods.

Code Changes: For evaluating changes made to the program code, we used a performance bug that was reported to the issue tracker[3] of the project. We compared the buggy version with one version that contains a fix. The resulting parallel coordinates plot (Figure 3) contains two lines per recorded method: blue lines indicate the fixed system, and orange lines the buggy version. The plot gives an overview of all methods across all statistics. The impact of the bug-

---

[1] https://github.com/apache/commons-collections

[2] http://www.memtest.org/
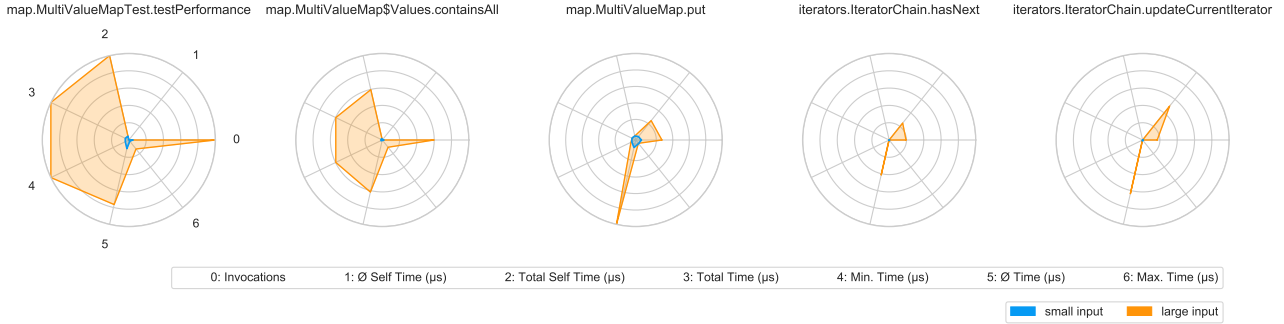[3] https://issues.apache.org/jira/browse/COLLECTIONS-429

Figure 2: *Input Changes* scenario: star glyph charts showing the top 5 methods by Δ *Total Time.*
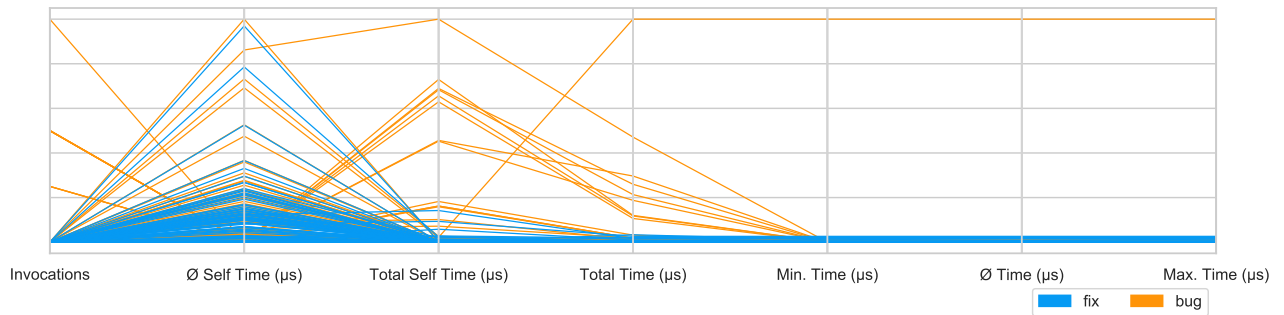

Figure 3: *Code Changes* scenario: a parallel coordinates plot showing a version with a performance bug (orange) and one with a fix for that (blue).

fix is evident: self time and total time have improved significantly after applying the fix.

While each of the proposed scenario/visualization-combinations has unique advantages (easy comparison capabilities of the grid-based layout, newly occurring patterns in the glyph-based visualization, and a general overview in the parallel coordinates plot), the predicted disadvantages (see Section 3) also come into play. Addressing these with a combination of multiple visualizations might be the subject of further research.

5 Conclusion

In this paper, we identified three scenarios to compare runtime statistics: (i) environment changes, (ii) input changes, and (iii) code changes. We analyzed how current standard representations for multivariate data can visualize differences in the presented metrics. We observed that the visualizations complement each other with respect to what features they reveal in the data and which scenarios they are most suitable for. With a focus on highlighting differences, this approach has potential to improve the developers' workflow of understanding runtime behavior and fixing performance bottlenecks in their software.

Acknowledgements

References

[1] M. Gleicher et al. "Visual comparison for information visualization". In: *Information Visualization* 10.4 (2011), pp. 289–309.

[2] A. Bergel et al. "Execution Profiling Blueprints". In: *Software: Practice and Experience* 42.9 (2012), pp. 1165–1192.

[3] F. Beck et al. "In Situ Understanding of Performance Bottlenecks through Visually Augmented Code". In: *Proceedings of the 21st IEEE International Conference on Program Comprehension.* ICPC. IEEE, 2013, pp. 63–72.

[4] J. P. Sandoval Alcocer et al. "Performance Evolution Blueprint: Understanding the impact of software evolution on performance". In: *Proceedings of the 1st IEEE Working Conference on Software Visualization.* VISSOFT. IEEE, 2013, pp. 1–9.

[5] S. Baltes et al. "Navigate, Understand, Communicate: How Developers Locate Performance Bugs". In: *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement.* ESEM. IEEE, 2015, pp. 1–10.