Understanding Software Performance Regressions using Differential Flame Graphs

Cor-Paul Bezemer, Johan Pouwelse
Delft University of Technology, the Netherlands
{c.bezemer, j.a.pouwelse}@tudelft.nl

Brendan Gregg
Netflix, USA
bgregg@netflix.com

***Abstract*—Flame graphs are gaining rapidly in popularity in industry to visualize performance profiles collected by stack-trace based profilers. In some cases, for example, during performance regression detection, profiles of different software versions have to be compared. Doing this manually using two or more flame graphs or textual profiles is tedious and error-prone.**

In this 'Early Research Achievements'-track paper, we present our preliminary results on using differential flame graphs instead. Differential flame graphs visualize the differences between two performance profiles. In addition, we discuss which research fields we expect to benefit from using differential flame graphs. We have implemented our approach in an open source prototype called FLAMEGRAPHDIFF, which is available on GitHub. FLAMEGRAPHDIFF makes it easy to generate interactive differential flame graphs from two existing performance profiles. These graphs facilitate easy tracing of elements in the different graphs to ease the understanding of the (d)evolution of the performance of an application.

I. Introduction

One of the major challenges in performance analysis is understanding the large amounts of data collected. Several visualization methods, such as heat maps [1] and icicle plots [2], have been introduced to assist with this understanding. Over the past few years, the flame graph [3], [4], a visualization based on the icicle plot, has gained rapidly in popularity in industry. A flame graph is a visualization of hierarchical data. More specifically, it visualizes a collection of (*stack trace*[1], *value*)-pairs in which *value* represents a metric monitored or calculated for that specific *stack trace*. These pairs can, for example, be obtained by using a stack trace-based profiler: such a profiler records and aggregates metrics per executed stack trace.

In a flame graph, elements are shown in a stacked fashion. Stack traces are shown from bottom to top, where the top element represents the function called latest in the stack trace. As a result, the height of the graphed stack represents the depth of the stack trace. The width of the stack represents the relative size of the monitored values compared to the other values. Hence, the element with the largest value (e.g., the stack trace in which most time is spent) in the profiled data is identified by finding the widest element in the graph.

Figure 1 depicts a sample program and its corresponding CPU profile. In this profile, the CPU time spent within each

[1]Throughout this paper we consider a stack trace a report of the active stack frames at a certain point in time during the execution of a program.

```
def a():
  if some_condition:
    b()
  ...

def b():
  if some_condition:
    c()
  ...

def c():
  ...

def d():
  if some_condition:
    c()
  ...

a()
d()
```

Stack trace	*CPU time*
a()	100
a()→b()	25
a()→b()→c()	10
d()	50
d()→c()	10

Fig. 1. A sample program and its corresponding CPU profile

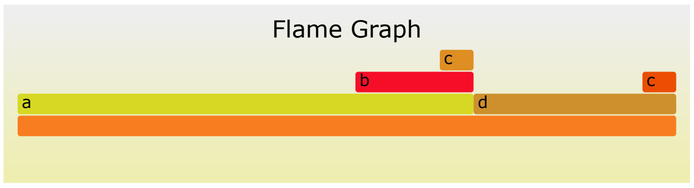

Fig. 2. Flame graph for profile in Figure 1

stack trace is recorded during execution of the program. Note that the times represent the total time spent within a stack trace and that the profile does not contain information about the execution order of these stack traces or about the time spent in a single call. Because of this, the x-axis of a flame graph does not and cannot show the passage of time, but instead spans the sample population, with the ordering of stacks sorted alphabetically to maximize merging of profiled elements. Figure 2 depicts the corresponding flame graph[2]. From this figure, it is easy to spot that most time is spent within stack trace *a()*.

In various cases, it is desirable to compare the performance profiles of two or more different software versions. One of these cases is performance regression analysis. The goal of performance regression analysis is to find out whether and why the performance of software degraded after an update to the code [5], [6]. Current practice is to manually compare profiles which is time consuming and tedious. In this paper, we

[2]Note that a random color palette is used to generate flame graphs, i.e., the colors do not have a meaning.

propose a method for doing this comparison using *differential flame graphs (DFGs)*. In a DFG, the differences between two performance profiles are depicted using a flame graph.

In Section II, we present the DFG-set, explain its components and give examples on how they can be used for performance analysis. In Section III, we describe scenarios in which we expect DFGs to be useful. We discuss the challenges and the open source implementation of our approach in Section IV. We discuss related work in Section V and we conclude our paper in Section VI.

II. Using DFGs for Performance Analysis

Our main proposed application for DFGs is in the performance analysis process. We propose to use DFGs in the three following cases:

- To detect performance regressions (Section II-B)
- To validate the effect of a performance fix (Section II-C)
- To compare the performance of an application on different systems (Section II-D)

Our approach is based on a combination of DFGs, the DFG-set. In this section, we will first elaborate on the components of a DFG-set and after that, explain how to use them in the cases above.

A. Differential Flame Graph-Sets (DFG-sets)

For any two software versions v_1 and a newer version v_2, we can record stack trace-based performance profiles p_1 (for v_1) and p_2 (for v_2). A DFG-set visualizes the differences between p_1 and p_2 using three components:

1) DFG_1: A comparison of p_1 and p_2 with p_1 as base
2) DFG_2: A comparison of p_1 and p_2 with p_2 as base
3) DFG_{diff}: A flame graph based on the differences of DFG_2

To generate a flame graph with p_1 as base we draw the base profile as a flame graph, so that the frames and their widths reflect the base. We then add color to show the profile differences. Note that we must compare the profiles both with p_1 and p_2 as base to deal with stack traces that may be added or removed in v_2. DFG_1 and DFG_2 visualize the two options for the performance engineer: either stay with v_1 instead of v_2 (DFG_1), or move from v_1 to v_2 (DFG_2). The DFG-set of three standard flame graphs does visualize all the necessary profile data for regression analysis. However, we introduce color as a dimension to show profile differences within each flame graph. The colors of the components within the flame graph depicting the comparison of p_1 and p_2 with p_2 as base (DFG_2) represent the following:

- *White* – profile value unchanged in version v_2
- *Blue* – profile value reduced in version v_2
- *Red* – profile value grew in version v_2

The interpretation of these colors depends upon the type of profile: for a metric such as time spent, red represents regression, while for a metric such as throughput, it represents an improvement. In addition, we add intensity to the color - a darker shade of blue or red indicates a reduction or growth that is relatively large compared to the other elements that have changed in v_2. In DFG_1, p_1 is used as the base, and the colors show the profile difference if we revert the changes from v_2 back to v_1.

To further highlight the differences in performance when using software version v_2 instead of v_1, we draw DFG_{diff} which contains only the differences. This allows us to draw the element width size relative to the size of the difference, making it easy to spot the largest differences. We have chosen to draw the differences of DFG_2 in DFG_{diff} only as these are in our opinion the most often investigated ones when searching for performance regressions.

B. Detecting Performance Regressions

Performance regression can occur for various performance metrics, such as CPU time and I/O traffic [6]. Below we give examples on how to use a DFG-set to detect such regressions.

1) CPU Time Regression: We demonstrate the applicability of a DFG-set on finding CPU time regressions using the rsync[3] test suite as an example. Rsync is a widely-used utility software for synchronizing files and directories from one location to another while minimizing data transfer by using delta encoding. We used perf[4] to record the number of CPU cycles spent in each function during an execution of the test suite of rsync, resulting in performance profile p_1. To generate profile p_2, we have altered p_1 to simulate a performance regression which increases the cycles spent within the `main` and `md5_process` functions.

Figure 3, 4 and 5 depict the DFG-set that results from comparing p_1 and p_2. Figure 3 and 4 show that a regression occurred in a function called by `main` and in the `md5_process` function, which are the regressions seeded by us. Figure 5 further highlights these regressions. This can especially be helpful in flame graphs with a large number of elements or differences.

2) I/O Writes Regression: In earlier work [6], we have proposed an approach for detecting regressions in the amount of I/O write traffic. This approach generates a report when comparing two code revisions, which contains a ranking of the stack traces which were the most likely to have increased their write traffic (*impact*). Table I depicts (part of) such a ranking. The actual ranking contains 204 records. Because this ranking is textual, it can be tedious to see relations between the stack traces such as relative size and overlap. The interpretation of the textual ranking can be made easier[5] by using a DFG-set instead, because a DFG-set allows us to see the relative size and groups stack traces that share common elements together. Figure 6 depicts the DFG-set corresponding to the (full) ranking of Table I. An advantage is that minor increases are easily ignored in the DFG-set. One can clearly see that the database commit in the `store_update_forward` function causes the largest part of the regression and that the increase in writes caused by other functions is negligible in comparison. In

[3]http://rsync.samba.org/

[4]https://perf.wiki.kernel.org/index.php/Main_Page

[5]Although one can argue that in this specific case the textual ranking is already easy to read due to the large increase.

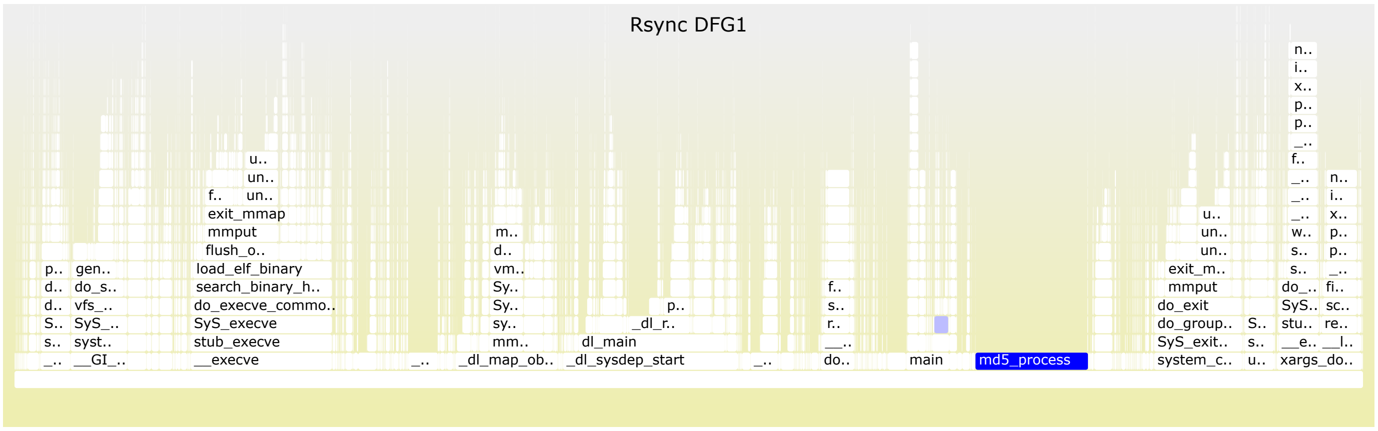

Fig. 3. DFG_1 for rsync test suite (unmodified version)

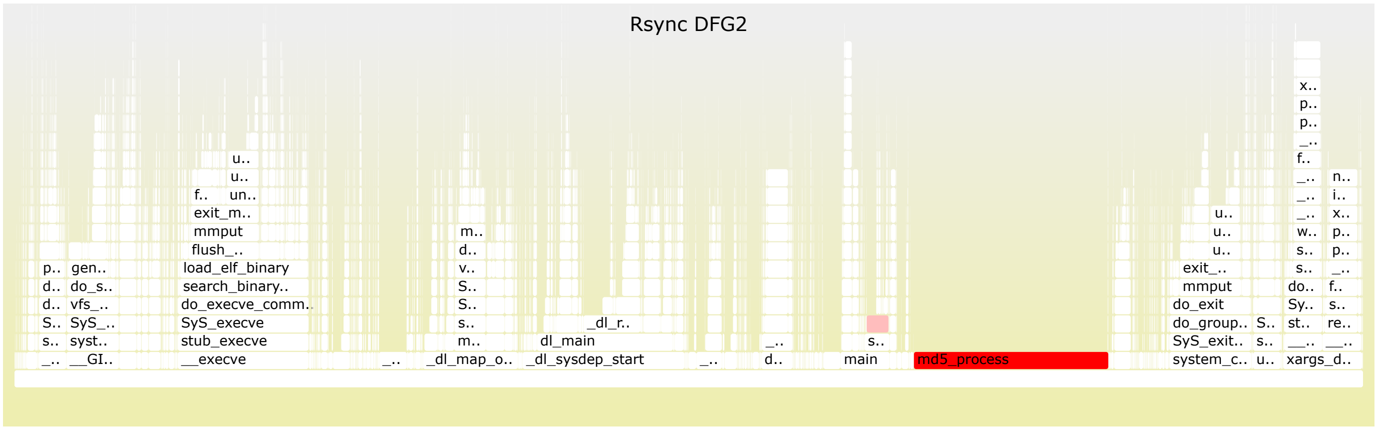

Fig. 4. DFG_2 for rsync test suite (version with seeded regression)

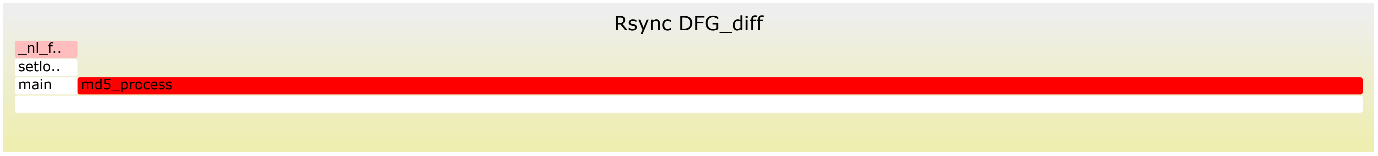

Fig. 5. DFG_{diff} for rsync test suite

addition, we would not have been able to display both profiles and the ranking on one page in a textual form, while using flame graphs this is possible.

C. Validating a Performance Fix

After a regression occurred, or another performance issue was found, the responsible code should be fixed. After applying this fix, a DFG-set can be used to validate the effect of the fix. DFG_1 in Figure 6 shows us what happens to the performance of our application if we revert from the new version of the code to the old version. Hence, we can validate the effect of the 'fix' of undoing these changes. Likewise, we can use blue-colored elements in a DFG to analyze whether a performance fix had the desired effect.

D. Comparing Performance on Different Systems

The DFG-set is a representation of the absolute difference between two profiles. In some cases, it may be more useful to compare their relative difference, i.e., the proportion that each stack trace takes up in the profile. An example of such a case is when we want to compare two profiles recorded on different systems, for example, when analyzing debug data which was submitted through a number of crash reports by client users. In this case, it may not be useful to compare the absolute values in this data but to use normalized values instead. The differences may help to learn more about the limitations and system requirements of an application.

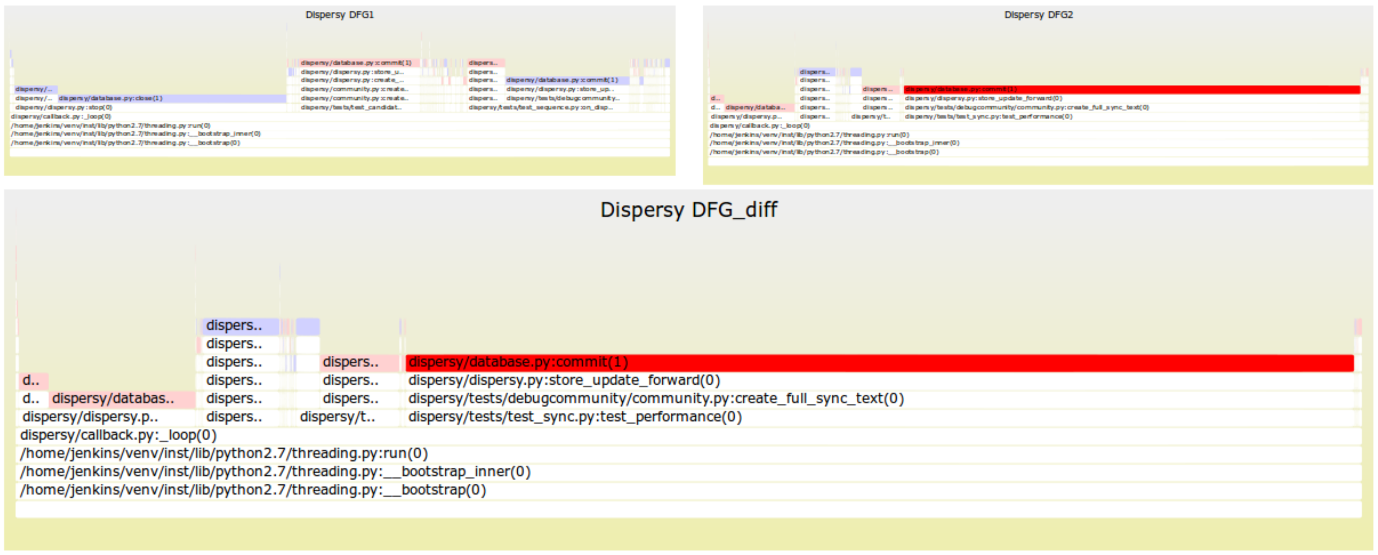

Fig. 6. DFG-set for Dispersy test suite

#	Impact	Stack trace
1	435 MB	dispersy/database.py:commit dispersy/dispersy.py:store_update_forward dispersy/tests/debugcommunity/community.py:create_full_sync_text ...
2	0.4 MB	dispersy/database.py:commit dispersy/database.py:close dispersy/dispersy.py:stop ...
3	0.2 MB	dispersy/community.py: dispersy/tool/tracker.py: dispersy/tests/test_candidates.py: ...

TABLE I
PARTIAL TEXTUAL RANKING OF THE APPROACH PRESENTED IN [6]

III. Different Applications

In this section, we present various different scenarios we expect to be suitable for analysis with DFGs. These scenarios can be used to guide the formation of new research questions regarding DFGs.

A. Parallel/Distributed Computing

An important challenge in parallel and distributed computing is to divide a large task into several smaller subtasks. We expect that DFGs can assist with the validation of this division as DFGs allow easier analysis of differences between profiles. Hence, DFGs make it easier to analyze the difference in workload between various nodes performing similar tasks.

Another scenario in which we expect DFGs to help out, is with the analysis of distributed algorithms, such as those used in peer-to-peer networks. In such algorithms, the goal is often to distribute the workload evenly over the available peers. If such an algorithm contains a bug, it is difficult to debug because the bug may only be exhibited when a large number of peers is in the network. We expect that DFGs can assist in debugging scenarios by allowing a quick comparison of the profile of a large number of nodes. Likewise, we expect DFGs can assist with the debugging process of load balancers.

B. GUI and Website Analysis

We expect that DFGs can be applied in fields other than software performance analysis as well. GUI analysis exhibits similarities with software performance: GUI usage can be monitored by counting click-paths [7], which can be considered a stack trace of the actions performed in the GUI. After adding a new option to the GUI, the DFG-set can be used to investigate how the new click-path affects usage of existing functionality. Note that these ideas apply to website analysis as well.

IV. Discussion

A. Challenges

The most important challenge of DFGs is data collection. Because DFGs require full stack traces, profiles must be recorded using profilers that can generate such traces. In practice collecting such data appeared to be difficult for some languages (e.g., Java and Python), due to the inavailability of suitable profilers.

In large flame graphs it can be difficult to locate targets. In future work, we will add a keyword search to FLAMEGRAPHDIFF to make this easier.

B. Implementation

The prototype implementation of our approach is available as an open source project called FLAMEGRAPHDIFF[6]. FLAMEGRAPHDIFF takes two files containing *(stack, value)*-pairs as input and generates the corresponding DFG-set. Optionally, values can be normalized before they are being graphed. To generate the DFG-set, first the flame graphs are generated[7]. Then, the profiles are compared and the elements of the flame graphs are colored accordingly. Finally, DFG_{diff} is generated

[6]http://corpaul.github.io/flamegraphdiff/

[7]For more information on flame graph generation see the original FlameGraph repository: https://github.com/brendangregg/FlameGraph

by hiding all stack traces from DFG_2 of which the value of the last function on the stack did not change.

FLAMEGRAPHDIFF generates the three DFGs in the DFG-set as interactive SVGs. When the user hovers the mouse over an element in any of the graphs in the set, the corresponding elements are highlighted in the other graphs and their values are displayed. This allows for easy tracing of elements over the various graphs. A demonstration of several scenarios can be found at the project website.

V. Related Work

Performance regression analysis through visualization has received surprisingly little attention in research. The widely-used profiler OProfile [8] implements a technique known as differential profiles, which expresses differences between profiles in percentage. However, this is a textual approach and does not offer a visualization.

Bergel et al. [9] have proposed a profiler for Pharo which compares profiles using visualization. In their visualization, the size of an element describes the execution time and number of calls. Alcocer [10] extends Bergel's approach by proposing a method for reducing the generated callgraph. Additionally, Alcocer et al. [11] propose Performance Evolution Blueprints (PEBs), which show the evolution of an application. The data graphed by PEBs is similar to the data graphed by DFGs, however, DFGs appear to do this in a more compact fashion. In future work, we will do a thorough comparison of the opportunities and limitations of PEBs and DFGs.

Nguyen et al. [12] propose an approach for detecting performance regressions using statistical process control techniques. Nguyen et al. use control charts to decide whether a monitored value is outside an accepted range. The violation ratio defines the relative number of times a value is outside this range. The main difference in the approach used by Nguyen and our approach is the granularity. Their approach identifies performance regressions in system-level metrics, while our approach identifies regressions on the function-level, making analysis of the regression easier. In future work, we will investigate how our approach and Nguyen's approach can complement each other.

Trumper et al. [13] use icicle plots and edge bundles to visualize differences between execution traces. They focus on the functional aspects of an application, while our approach focuses on a non-functional aspect (performance). In addition, they focus on ordered sequences, while for our visualization, the order of events is not important as we work with aggregated data. Finally, the use of a color scheme to represent differences rather than colored edge bundles results in a clearer graph, which is beneficial for graphs with many elements.

Other visualizations have been proposed for large amounts of performance data, such as heat maps [1], [14], but these have not been applied to performance regression detection.

VI. Conclusion

In this paper, we have presented the differential flame graph (DFG) for visualizing differences between performance profiles. A DFG is a flame graph depicting the differences of two performance profiles, using one of those profiles as a base. A DFG-set combines three DFGs in one figure: one using the first profile as a base, one using the second profile as a base, and one in which the differences in the second DFG are emphasized to facilitate easier analysis. Without a DFG-set, comparing performance profiles is tedious and error-prone. In this ERA-track paper, we have indicated and given examples of how DFGs can be used for detecting performance regression, validating performance fixes and comparing performance profiles recorded on different systems.

In addition, we present the prototype open source implementation of our approach, FLAMEGRAPHDIFF, which makes it easier to generate and analyze DFGs and to trace elements in multiple graphs. We expect this implementation to be useful in several research areas, such as performance analysis, parallel and distributed computing and GUI and website analysis. Hence, we invite researchers from other fields to use our prototype in their research or to contact us for research collaborations. In future work, we will focus on thoroughly evaluating DFGs in large research and industrial projects.

References

[1] C.-P. Bezemer, A. Zaidman, A. van der Hoeven, A. van de Graaf, M. Wiertz, and R. Weijers, "Locating performance improvement opportunities in an industrial software-as-a-service application," in *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*. IEEE C.S., 2012, pp. 1–10.

[2] J. B. Kruskal and J. M. Landwehr, "Icicle plots: Better displays for hierarchical clustering," *The American Statistician*, vol. 37, no. 2, pp. 162–168, 1983.

[3] B. Gregg. (2012) Flame graphs. [Online]. Available: http://www.brendangregg.com/flamegraphs.html

[4] ——, *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2013.

[5] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *Quality Software (QSIC), 2010 10th International Conference on*, July 2010, pp. 32–41.

[6] C. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse, "Detecting and analyzing I/O performance regressions," *Journal of Software: Evolution and Process*, pp. n/a–n/a, 2014.

[7] T. Srivastava, P. Desikan, and V. Kumar, "Web mining – concepts, applications and research directions," in *Foundations and Advances in Data Mining*, ser. Studies in Fuzziness and Soft Computing, W. Chu and T. Lin, Eds. Springer Berlin Heidelberg, 2005, vol. 180, pp. 275–307.

[8] J. Levon, P. Elie *et al.* (2007) Oprofile: A system-wide profiler for linux systems. [Online]. Available: http://oprofile.sourceforge.net

[9] A. Bergel, F. Bañados, R. Robbes, and W. Binder, "Execution profiling blueprints," *Softw., Pract. Exper.*, vol. 42, no. 9, pp. 1165–1192, 2012.

[10] J. P. S. Alcocer, "Tracking down software changes responsible for performance loss," in *Proc. Int'l Workshop on Smalltalk Technologies (IWST)*. ACM, 2012, pp. 3:1–3:7.

[11] J. P. S. Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance," *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, vol. 0, pp. 1–9, 2013.

[12] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proc. ACM/SPEC Int'l Conf. on Performance Engineering (ICPE)*, 2012, pp. 299–310.

[13] J. Trumper, J. Dollner, and A. Telea, "Multiscale visual comparison of execution traces," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 53–62.

[14] B. Gregg, "Visualizing system latency," *Commun. ACM*, vol. 53, no. 7, pp. 48–54, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1785414.1785435