PerformanceHat – Augmenting Source Code with Runtime Performance Traces in the IDE

Jürgen Cito
University of Zurich
Zurich, Switzerland

Philipp Leitner
Chalmers | University of Gothenburg
Gothenburg, Sweden

Christian Bosshard
ergon Informatik AG
Zurich, Switzerland

Markus Knecht
University of Zurich
Zurich, Switzerland

Genc Mazlami
zühlke Engineering
Zurich, Switzerland

Harald C. Gall
University of Zurich
Zurich, Switzerland

ABSTRACT

Performance problems observed in production environments that have their origin in program code are immensely hard to localize and prevent. Data that can help solve such problems is usually found in external dashboards and is thus not integrated into the software development process. We propose an approach that augments source code with runtime traces to tightly integrate runtime performance traces into developer workflows. Our goal is to create operational awareness of performance problems in developers' code and contextualize this information to tasks they are currently working on. We implemented this approach as an Eclipse IDE plugin for Java applications that is available as an open source project on GitHub. A video of PerformanceHat in action is online: https://youtu.be/fTBBiylRhag

KEYWORDS

software performance engineering, program analysis, development workflow

ACM Reference format:
Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat – Augmenting Source Code with Runtime Performance Traces in the IDE. In *Proceedings of 40th International Conference on Software Engineering Companion, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18 Companion),* 4 pages.
https://doi.org/10.1145/3183440.3183481

1 INTRODUCTION

Software developers produce high volumes of code, usually in an IDE, to provide functionality that solves problems. Code is then often deployed in complex systems exhibiting distributed architectures and scalable infrastructure. Through observation and instrumentation, we collect logs and metrics (we collectively call them runtime traces) to enable comprehension of the inner workings of our deployed software. Performance problems that occur in production of these complex systems often originate in development [8]. Developers then have the challenging task to reason about runtime behavior and determine why certain parts of their code do not exhibit desired performance properties. For some performance problems, this reasoning can be supported by information provided by profilers executed in a local environment. However, there is a whole class of performance anomalies that occur in production that cannot be solved by the information provided by these local profilers. Tracing this class of problems back to their origin in the source code and debugging them requires the inspection of runtime traces from production. Since our deployment is distributed and complex, so are the traces collected at runtime. To mitigate this complexity, a large body of academic [1, 13] and industrial work, both in open source and commercial solutions attempt to provide insight into runtime behavior by introducing dashboards that visualize data distributions in the form of time series graphs and enable search capabilities for trace properties.

This particular design and level of abstraction of runtime information is largely designed to support the reactive workflow of software operations, whose responsibilities require them to think in systems, rather than the underlying source code. This traditional view of operations analytics does not fit into the workflow of developers, who struggle to incorporate information from traces into their development and debugging activities. This is in line with existing work that argues for program analysis to be effective, need to be smoothly integrated into the development workflow [11].

PerformanceHat. We argue that the abstraction level at which runtime performance traces are presented in the current state-of-the-art is not well-suited to aid software developers in understanding the operational complexities of their code. In this paper, we present *PerformanceHat*, an open source IDE plugin[1] that tightly integrates runtime performance aspects into the development workflow by augmenting the source code with performance traces collected in production environments. Distributed runtime traces are modeled in a way that they can be visually attached to source code elements (e.g., method calls and definitions, loop headers, collections).

All efforts going into the conception and implementation of our approach are guided by the following goals:

- *Operational Awareness:* By integrating runtime aspects into source code and, thus, into the development workflow, developers become more aware of the operational footprint of their source code.
- *Contextualization:* When runtime performance aspects are visible when working with source code, they are contextualized to

[1] http://sealuzh.github.io/PerformanceHat/

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3183481

the current task of the developer (e.g., feature development, debugging). Thus, developers do not have to establish their current context in external tools through search.

2 APPROACH OVERVIEW

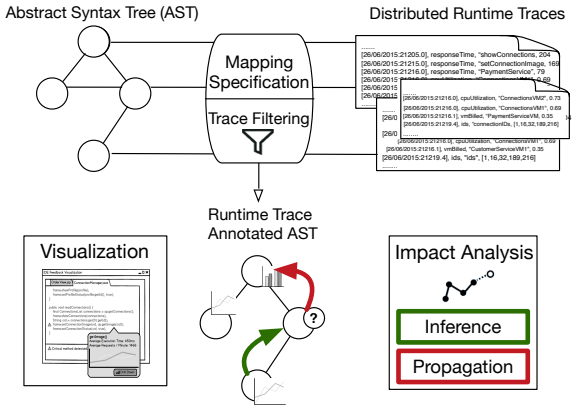


Figure 1: Conceptual Framework for the proposed approach. Nodes of source code, represented as a simplified Abstract Syntax Tree (AST), are annotated with runtime performance traces.

The basic theory underlying the approach is that static information available while working on maintenance tasks in software development does not suffice to properly diagnose problems that occur at runtime. The conjecture then is: Augmenting source code with dynamic information from the runtime yields enough information to the software developer to (1) make informed, data-driven decisions on how to perform code changes, (2) perceive code as it is experienced by its users. There are four particular abstractions that highlight the essence of our approach: *specification*, *trace filtering*, *impact analysis*, and *visualization*. We provide an overview in the following, details can be found in the full paper [**?**].

Mapping/Specification. In an initial step, the abstract syntax tree (AST) of the source code is combined with the dynamic view of runtime information in a process called specification or feedback mapping [4]. A set of traces can be mapped to different AST node types (e.g., method invocations, loop headers) based on different specification criteria in both node and trace. In the case of *PerformanceHat*, we map response time traces based on the fully-qualified method name in Java, that is both available as part of the AST node and in the trace data. While this is, in part, also done by regular profilers, we allow for more complex specification queries. Specifications define declarative queries about program behavior that establish this mapping relationship. In previous work, we demonstrate how we can establish approximative mappings between arbitrary traces from logs to source code through semantic similarity [5]. In the IDE, this results in a performance augmented source code view, that allows developers to examine their code artifacts annotated with performance data from runtime traces from production. Examples for such a mapping go from method calls with execution times, usage statistics for features or collections with size distribution.

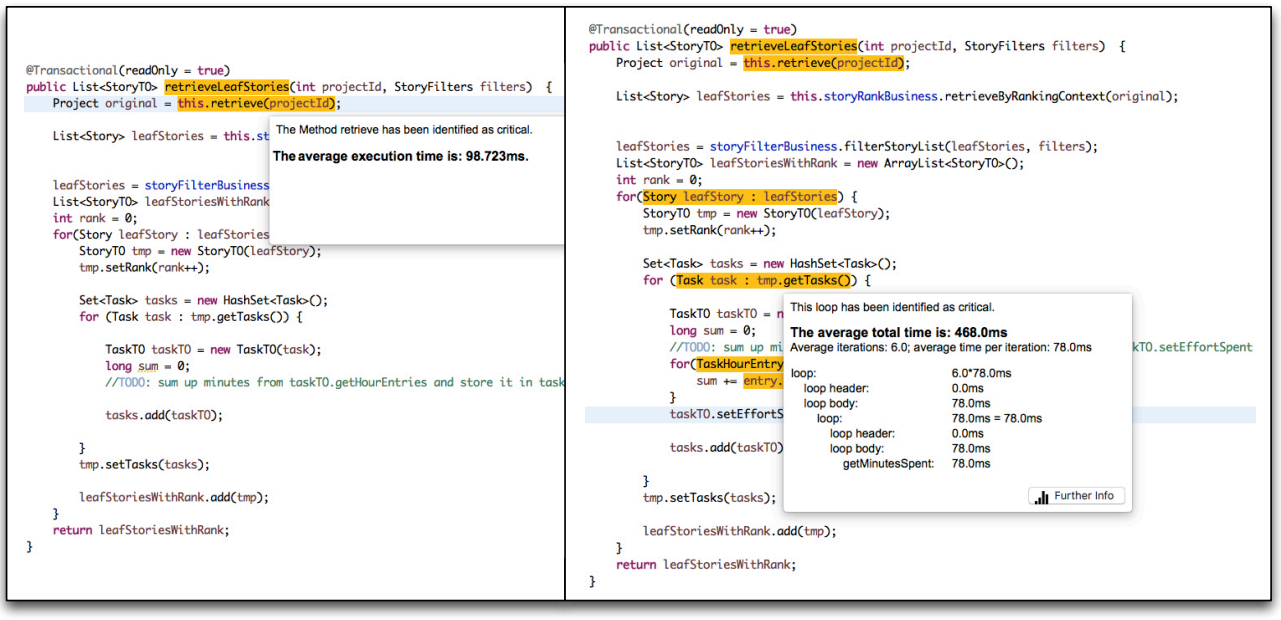
Trace Filtering. Traces are collected from distributed architectures with scalable instances. Data from these traces often exhibit a multi-modal distribution. Attaching all data that corresponds to a code element from the entire trace dataset might be misleading, as different subsets of the data might originate from multiple sources of distributions. Trace filtering enables sensitivity analysis of potential problematic properties of traces (e.g., from multiple data centers, or from users accessing data stores with different characteristics). Further, there are also other reasons to filter certain data points before displaying them to software developers (i.e., data cleaning). This can either be removing measurement errors through fixed thresholds to more complex filtering with interquartile ranges or dynamic filtering adjusted based on trends and seasonality.

Impact Analysis. When working on maintenance tasks, software developers often add or delete code. Nodes in the AST of newly added code does not yet have runtime traces that can be mapped. The goals of impact analysis are two-fold: (1) attempt to predict properties of unknown nodes (*inference*), and (2) given new information from inference, update the information of all dependent nodes (*propagation*). A prediction model is given the delta between the annotated AST, with all its attached operational data, and the current AST that includes the new changes as parameters to infer new information about the unknown nodes in the current AST. The prediction can be kept as simple as a "back-of-the-envelope" additive model that sums up all execution times within a method, to more complex machine learning models taking into account different load patterns to estimate latency of methods. Impact analysis gives early feedback and gives developers an outlook to the future of their code changes prior to running the application in production.

Visualization. Eventually, the trace data that was either mapped or inferred needs to be displayed to the developer in a meaningful context to become actionable. There are two dimensions to the visualization part: (1) how trace data and predictions are displayed, and (2) how developers are made aware that data for certain elements of source code exists in the first place. From previous steps in this framework, we are usually given a set of data points (distribution), rather than just a point estimate. A possible way to present runtime performance data could be to show a summary statistic as initial information and allow for more interaction with the data in a more detailed view. In terms of diverting attention that data exists, source code elements should be in some way highlighted in the IDE through warnings and overlays in the exact spot of the identified issue.

2.1 Implementation

We implemented an instantiation of the described framework as a proof-of-concept for runtime performance traces as a tight combination of components: An Eclipse IDE plugin for Java programs and local server components with a database (*feedback handler*) that deal with storing a local model of trace information and filtering. Performance traces (e.g., execution times, CPU utilization) are attached to method definition and method calls. Impact analysis



(a) Displaying Operational Footprint **(b) Inferring the Performance of Newly-Written Code**

Figure 2: Our tool *PerformanceHat* as viewed in the development workflow in the Eclipse IDE.

is currently supported for adding method calls and loops within a method body. Figure 2 provides a screenshot of *PerformanceHat* for two basic scenarios that are further elaborated in the demonstration video:

- (a) *Awareness and Comprehension for Performance Debugging:* We are displaying execution times observed in production, contextualized on method level. The box with initial information is displayed as developers hover over the marker on the "this.retrieve" method invocation. A button "Further info" opens a more detailed view (e.g., time series graph) from an existing dashboard.
- (b) *Problem Prevention through Impact Analysis:* After introducing a code change, the inference model attempts to predict the newly written code. Further, it is propagated over the blocks of foreach-loops. The box is displayed when hovering over the loop over the Task collection. It displays the prediction in this block and basic supporting information.

We depict a high-level architecture that enables scalable feedback of runtime traces in the IDE. In the following, we describe the main components of the architecture (*IDE Integration* and *Feedback Handler*) and discuss some considerations to achieve scalability.

IDE Integration. To achieve tight integration into the development workflow, we implemented the frontend of our approach as an Eclipse IDE plugin for Java. PerformanceHat hooks program analysis, specification/mapping, and inference into Eclipse's incremental builder. This means, whenever a file is saved, we start the analysis for that particular file. Both specification and inference function are designed as extension points and can be embedded by implementing a predefined interface. This allows us to implement different specification queries and inference functions based on the domain and application.

For PerformanceHat, we implemented specification queries that map execution times to method definitions and method invocations and, for certain use cases, instrumented collection sizes to for-loop headers. We also implemented inference functions for adding new method invocations and for-loops over collections within existing methods. The plugin handles the interactions between all sub-components and the *feedback handler*.

Feedback Handler. The feedback handler component is the interface to the data store holding the feedback in a model that the plugin understands. It is implemented as a Java application, exposing a REST API, with a MongoDB data store. Conceptually, we can think of two separate feedback handlers: (1) deployed feedback handler (remote), and (2) local feedback handler (on the developer's workstation). The deployed feedback handler is installed on the remote infrastructure, close to the deployed system and has an interface to receive or pull runtime information from monitoring systems. The local feedback handler runs as a separate process on the software developer's local workstation. The reasoning for this split is that displaying metrics and triggering inference in the IDE requires fast access to feedback. The local feedback handler acts as a replicated data store of potentially many remote traces. However, the implementation of any particular replication method is left to the particular user as requirements for consistency of local data vary between use cases. To achieve trace filtering, developers could register their own filters as MongoDB view expressions.

Scalability Considerations. For any kind of program analysis that is hooked into the incremental build process (i.e., it is executed with every file/project save), researchers and developer tool designers need to ensure immediacy of analysis results.

In initial versions of PerformanceHat, every build process triggered fetching new data from the *remote* feedback handler, introducing network latency as a bottleneck. Initial attempts to grow from smaller, toy examples to larger and more realistic applications showed that this simplistic one-to-one adoption of the conceptual framework does not scale sufficiently. In an iterative process, we arrived at the architecture depicted on a high level in Figure 3. We briefly summarize our efforts to enable scalability for runtime trace matching in development workflows:

- *Local Feedback Handler:* To decrease the time required to load all data required for the analysis it is moved as close as possible to the IDE, on the developer's workstation. This increases coordination between deployed and local instances, but is absolutely necessary to avoid interruptions to the developer workflow.
- *Local IDE Cache:* To further reduce the time for loading metrics from the feedback handler, we introduced a local IDE cache, such that each entry must be fetched only once per session. We used an LRU cache from the Guava[2] open source library with a size of maximal 10000 entries and a timeout of 10 minutes (after 10 minutes the cache entry is discarded, however both these entries are configurable in a configuration file). This reduced the build time significantly.
- *Bulk Fetching:* A significant improvement also occurred when, for an empty cache, we first registered all nodes that required information from the feedback handler and then loaded all information into the cache in bulk.

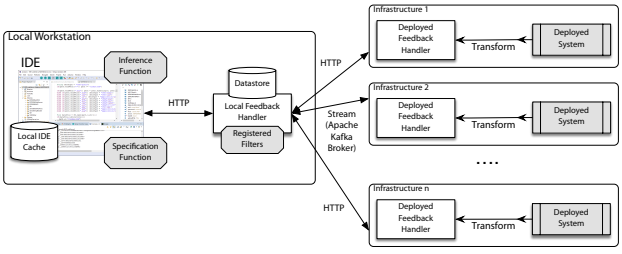


Figure 3: Scalable architecture for rapid feedback of runtime performance traces in the IDE

3 RELATED WORK

Our approach has parallels to work that investigates different ways to understand runtime behavior through visualization. Sandoval et al. looked at "performance evolution blueprints" to understand the impact of software evolution on performance [12]. Senseo is an approach embedded in the IDE that augments the static code view perspective with dynamic metrics of objects in Java [10]. Bezemer et al. investigated differential flame graphs to understand software performance regressions[3]. Cornelissen et al. conducted an empirical user study showing that trace visualization in the IDE can significantly improve program comprehension [6]. ExplorViz is an approach that provides live trace visualization in large software architectures [7]. Theseus augments JavaScript code in the debug view in the browser with runtime information on call count of functions' asynchronous call trees to display how they interact [9]. Similarly to our work, Beck et al. provide augmented code views with information retrieved from profilers that sample stack traces [2]. Our work differs first in the use of data retrieved through instrumentation in production systems. Further, our approach goes beyond displaying information on existing traces by providing live impact analysis on performance of newly written code. Our impact analysis approach is applied live, i.e. during software development, and leverages a mixed model consisting of the immediate software change and the existing runtime information to provide early feedback to software developers.

[2]https://github.com/google/guava

4 CONCLUSION

We presented PerformanceHat, a tool that integrates runtime performance traces into the development workflow by augmenting source code in the Eclipse IDE and providing live performance feedback for newly written code during software maintenance. Its goal is to create operational awareness of source code and contextualize runtime trace information to tasks in the development workflow.

REFERENCES

[1] *Aceto Giuseppe, Botta Alessio, De Donato Walter, Pescapè Antonio.* Cloud monitoring: A survey // Computer Networks. 2013. 57, 9. 2093–2115.
[2] *Beck Fabio, Moseler Oliver, Diehl Stephan, Rey Gunter Daniel.* In situ understanding of performance bottlenecks through visually augmented code // Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. 2013. 63–72.
[3] *Bezemer Cor-Paul, Pouwelse Johan, Gregg Brendan.* Understanding software performance regressions using differential flame graphs // Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. 2015. 535–539.
[4] *Cito Jürgen, Leitner Philipp, Gall Harald C, Dadashi Aryan, Keller Anne, Roth Andreas.* Runtime metric meets developer: building better cloud applications using feedback // 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). 2015.
[5] *Cito Jürgen, Oliveira Fabio, Leitner Philipp, Nagpurkar Priya, Gall Harald C.* Context-Based Analytics - Establishing Explicit Links between Runtime Traces and Source Code // Proceedings of the 39th International Conference on Software Engineering Companion. 2017.
[6] *Cornelissen Bas, Zaidman Andy, Van Deursen Arie.* A controlled experiment for program comprehension through trace visualization // Software Engineering, IEEE Transactions on. 2011. 37, 3. 341–355.
[7] *Fittkau Florian, Krause Alexander, Hasselbring Wilhelm.* Exploring software cities in virtual reality // Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on. 2015. 130–134.
[8] *Hamilton James.* On Designing and Deploying Internet-scale Services // Proceedings of the 21st Conference on Large Installation System Administration Conference. Berkeley, CA, USA: USENIX Association, 2007. 18:1–18:12. (LISA'07).
[9] *Lieber Tom, Brandt Joel R, Miller Rob C.* Addressing misconceptions about code with always-on programming visualizations // Proceedings of the 32nd annual ACM conference on Human factors in computing systems. 2014. 2481–2490.
[10] *Röthlisberger David, Härry Marcel, Villazón Alex, Ansaloni Danilo, Binder Walter, Nierstrasz Oscar, Moret Philippe.* Augmenting static source views in IDEs with dynamic metrics // Software Maintenance, 2009. ICSM 2009. IEEE International Conference on. 2009. 253–262.
[11] *Sadowski Caitlin, Gogh Jeffrey van, Jaspan Ciera, Soederberg Emma, Winter Collin.* Tricorder: Building a Program Analysis Ecosystem // International Conference on Software Engineering (ICSE). 2015.
[12] *Sandoval Alcocer Juan Pablo, Bergel Alexandre, Ducasse Stéphane, Denker Marcus.* Performance evolution blueprint: Understanding the impact of software evolution on performance // Software Visualization (VISSOFT), 2013 First IEEE Working Conference on. 2013. 1–9.
[13] *van Hoorn André, Waller Jan, Hasselbring Wilhelm.* Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis // Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE). New York, NY, USA: ACM, 2012. 247–248.