Combining Kieker with Gephi for Performance Analysis and Interactive Trace Visualization

Christian Zirkelbach
Software Engineering Group
Kiel University, Germany
czi@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University, Germany
wha@informatik.uni-kiel.de

Leslie Carr
Electronics and Computer Science
University of Southampton, UK
lac@ecs.soton.ac.uk

ABSTRACT

Performing an analysis of established software usually is challenging. Based on reverse engineering through dynamic analysis, it is possible to perform a software performance analysis, in order to detect performance bottlenecks or issues. This process is often divided into two consecutive tasks. The first task concerns monitoring the software, and the second task covers analysing and visualizing the results.

In this paper, we report on our performance analysis of the Perl-based open repository software EPrints, which has now been continuously developed for more than fifteen years. We analyse and evaluate the software using Kieker, and employ the visualization tool Gephi for performance analysis and interactive trace visualization. This allows us, in collaboration with the EPrints development team, to reverse engineer their software EPrints, to give new and unexpected insights, and to detect potential bottlenecks.

1. Introduction

Reverse engineering is often employed to understand legacy software systems. One option is employing static analysis of a program's source code. Unlike static analysis, which focuses on examining the source code, dynamic analysis methods operate on the system execution. This provides valuable insights into a software system and its behaviour during a program's execution [4]. But even if an instrumentation is possible, the visualization is often challenging. The latter problem is often addressed via trace visualization, but finding an appropriate representation for an specific case is difficult. In this paper, our approach to reverse engineering of legacy software systems via analysing monitoring data of a program's operational use, based on dynamic analysis, is presented. We report on the performance analysis of the Perl-based software EPrints [2] with focus on analysing and evaluating it using the monitoring framework Kieker [12]. EPrints has been continuously developed for more than fifteen years. In order to aid the process of program comprehension, we analyse our monitoring results with two types of trace visualization and use their advantages to address different purposes and phases within our project. Therefore, we combine the batch-oriented visualization tool Graphviz [8] with the interactive visualization tool Gephi [1]. One of the main goals of the project presented in this paper was to detect potential bottlenecks in the architecture of Version 3.3.12 of EPrints in order to gather useful information to eliminate them in the planned release Version 4. For this forthcoming major release, a significant restructuring of the software architecture is planned. The rest of this paper is organized as follows. In Section 2, we will present the initial analysis results, visualized via Graphviz. We refine this analysis through interactive graph exploration with Gephi in Section 3. Section 4 reports on the detection of performance bottlenecks. In Section 5 we discuss related work regarding our approach. Finally, the conclusions are drawn and future work is described.

2. Batch Visualization via Graphviz

Initially, we applied a full instrumentation to EPrints, i.e. we monitored the complete software system by weaving monitoring probes around all Perl packages. By default, Kieker employs Graphviz[1] [8] to visualize the generated graphs as so-called component dependency graphs. Since the initial analysis of a complete system usually provides voluminous representations of the observed monitoring data, some complexity reduction is required. With reference to Kieker and Graphviz, it is possible to refine and reduce this representation either via modifying the aspect-oriented instrumentation, as mentioned before, or via configuring Kieker's analysis pipeline, as we did in previous projects [9, 10]. However, as we are interested in a performance analysis, the visualization of our graphs via Graphviz in form of dependency graphs turns out to be inappropriate. We need support to modify the graph and to filter for highly-frequented packages and exceptional response times. This requires an iterative approach.

3. Interactive Trace Visualization with Gephi

Our initial analysis with Kieker and visualization via Graphviz provides an overview of the software system. We try to reconstruct the EPrints architecture in order to identity packages that may contain potential bottlenecks. Since our visualization via Graphviz meets its limits for our purpose, as reported in Section 2, we employ another visualization tool, namely *Gephi*[1], an interactive visualization and exploration platform for handling graphs.[2] The iterative workflow, for our performance analysis, is illustrated

[1] `http://www.graphviz.org`
[2] `http://gephi.org`

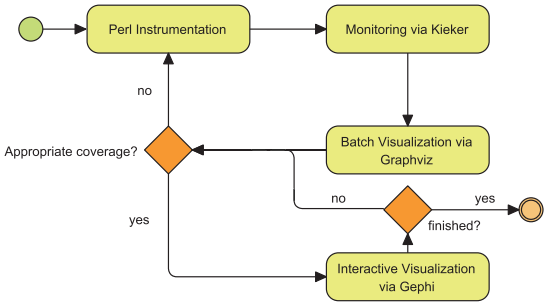

Figure 1: Our performance analysis workflow

in Figure 1.

Gephi is able to import the Graphviz graphs that are generated by *Kieker.Analysis*. Based on features such as dynamic filtering and layout algorithms it is possible to further process our initial graphs for improved program comprehension and additional analyses. Thus, the component dependency graph of our full instrumentation is interactively analysed and reduced with Gephi. Furthermore, we aggregate nodes based on their package hierarchy in order to obtain a more suitable overview with respect to a system architecture level. Aiming at further abstraction, we reduce our obtained graph with Gephi to display only first-level packages. As a result, we obtain the coloured dependency graph in Figure 2. The nodes represent Perl packages, including their sub packages. Edges express dependencies among them. Compared to the initial graph of our full instrumentation, the reduced graph is well-structured. The colors indicate the source nodes of the edges and the numbers represent the number of calls for this specific edge. At this stage of our performance analysis, we are able to focus our visualization on the identification of packages and their dependencies that may cause performance issues. Thereupon, we find suspicious calls from the *EPrints.Database* to the *EPrints.Repository* package. These calls are passing by the *EPrints.Obj* and *EPrints.Metafield* packages, which are supposed to handle database-related operations. Thus, we identify a violation of intended architecture rules. This observation leads us to a detailed analysis of dependencies regarding *EPrints.MetaField.** using Gephi.

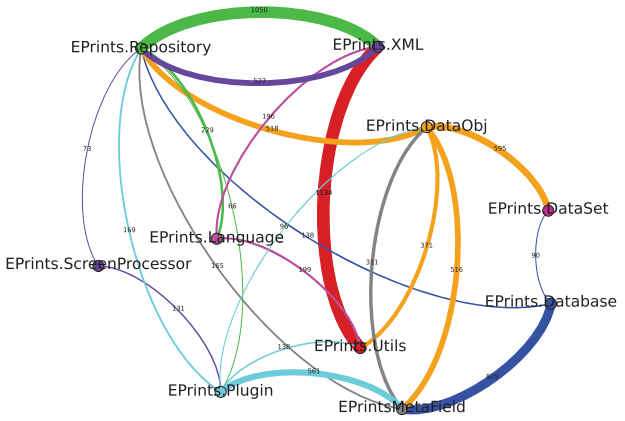

Figure 2: Colored component dependency graph for EPrints using Gephi

4. Performance Bottleneck Identification

As a result of obtaining detailed information on the dependencies among the Perl packages and the number of calls via Gephi, we are able to instrument the software at dedicated places in the source code to focus on potential performance bottlenecks. We start with our observations from the previous section and re-instrumented only a small subset of selected packages. Based on the analysis with Kieker and the subsequent visualization via Graphviz we are able to decide whether we reached a sufficient instrumentation level or not. In insufficient cases, we further refine our instrumentation, until we are satisfied with the obtained level of detail. This approach is based on the aforementioned workflow, which was illustrated in Figure 1. Subsequently, we used Gephi to interactively modify, and visualize the graphs.

Our first detailed instrumentation is applied to the package *Screen.Items.**. This is a central package with multiple sub packages, that handles processing display components. We analyze the dependencies among the operations within these packages and also their respective execution times. In this context, we ignore the number of calls and focus on the two most suspicious operations (based on the response time), namely *render()* and *render_items()*, which take a large share of the overall response time. After drilling down the monitoring to this specific area, the visualization via Graphviz is sufficient for our first analysis purposes. However, to find causes for the high response times within these two operations, it is necessary to locate the related outgoing calls (edges) within the component dependency graph, based on the initial full instrumentation using filtering techniques. This results in monitoring the package *EPrints.Database.**, allowing us to find the database-related operations, which may cause high response times. With respect to the obtained maximum response times, two operations, namely *do()* and *create_table()*, are suspicious. In comparison to the total response time, they take up to a third of the total, which is a significant share. Therefore, we further analyse these operations with respect to possible performance issues.

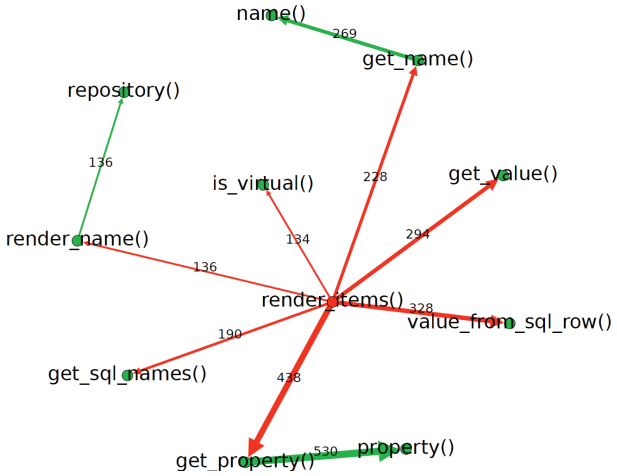

Figure 3: Detailed analysis of dependencies for EPrints.MetaField.* using Gephi

Additionally, we examine the *EPrints.MetaField.** packages. Again, we refine the instrumentation and generate a dependency graph on the operation-level. This leads us to a detailed analysis of the related dependencies using Gephi, as shown in Figure 3. The graph shows related calls annotated with the number of calls. We focus on the top ten operations, based on the number of calls. The most interesting operations within the graph were *value_from_sql_row()* (328 calls) with a maximum response time of 110 ms and *get_property()* (438 calls) with 212 ms. Both are handling database-related transactions.

5. Related Work

In this section, we discuss some related work in the area of trace visualization. Lange et al. [11] report on their software *Program Explorer*, which visualizes a program's interaction, for a given execution trace. In comparison to our approach they are limited to C++ software. In this project we employ an instrumentation for Perl, but we also support other programming languages like Java or COBOL. The tool *Web Services Navigator* [5] is a plug-in feature for the Eclipse platform and provides 2D graph visualizations of the communication of web services. It offers five different views for various purposes. Compared to our approach, they are limited to SOAP messages and reconstruct service transaction flows instead of dependency graphs. Cornelissen et. al present *ExtraVis* [3], which visualizes a program trace in two synergistic views, namely a circular bundle view and a massive sequence view. The first view utilizes hierarchical elements, including their call relationships to display the interaction of trace. The latter view provides a global overview of the trace. Another approach which visualizes program traces is *ExplorViz* [7], which monitors traces for large software landscapes and offers the visualization of a landscape and system level perspective. While the landscape perspective, which provides an overview of the software, employs a notation similar to UML, the system level perspective utilizes the city metaphor. In contrast to ExtraVis and ExplorViz, we combine two different kinds of visualization tools, utilize interactive graph exploration and focus on the detection of performance bottlenecks.

6. Conclusions

In this paper, we report on our approach of conducting a performance analysis of the long-term developed software system EPrints combining two visualization tools. We employ Kieker to reconstruct architectural models based on the monitored data, and Graphviz respectively Gephi to visualize the results. As Graphviz turns out to be insufficient for detailed analysis purposes, an interactive reduction of the visualized data through Gephi is required. We detected some performance bottlenecks within the software and could advise some changes for the next release. In addition to our work related to Version 3.3.12 of EPrints, the EPrints team already used Kieker for their current development release of Version 4 and debugged and fixed an infinite loop within the *MetaField* package. In the future, Kieker can be integrated upfront to establish automated quality management procedures such as continuous integration. Furthermore, we plan to perform additional performance analyses with other application performance management tools. As a first step we employed ExplorViz, which is able to import our generated Perl monitoring logs, and conducted a performance analysis [6]. For details please refer to the extended version of our paper [13].

7. References

[1] M. Bastian, S. Heymann, M. Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. In *Proc. of the Third Int. AAAI Conf. on Weblogs and Social Media*, pages 361–362, 2009.

[2] M. R. Beazley. EPrints institutional repository software: A review. *Partnership: the Canadian Journal of Library and Information Practice and Research*, 5(2), 2010.

[3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. In *Program Comprehension, 2007. ICPC '07. 15th IEEE Int. Conf. on*, June 2007.

[4] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[5] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. Morar. Web Services Navigator: Visualizing the execution of Web Services. *IBM Sys. Journal*, 44(4), 2005.

[6] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. In *Proc. of the 23rd European Conf. on Inf. Sys. (ECIS 2015)*, May 2015.

[7] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proc. of the 1st IEEE Int. Working Conf. on Soft. Vis. (VISSOFT 2013)*, September 2013.

[8] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[9] W. Hasselbring. Reverse engineering of dependency graphs via dynamic analysis. In *Proc. of the 5th European Conf. on Software Architecture*, ECSA '11. ACM, 2011.

[10] H. Knoche, A. van Hoorn, W. Goerigk, and W. Hasselbring. Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL systems. In *Proc. of the 14. Workshop Software-Reengineering (WSR '12)*, pages 45–46, May 2012.

[11] D. B. Lange and Y. Nakamura. Program Explorer: A Program Visualizer for C++. In *Proc. of the USENIX Conf. on Object-Oriented Techn. (COOTS'95)*, 1995.

[12] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. of the 3rd ACM/SPEC Int. Conf. on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, 2012.

[13] C. Zirkelbach, W. Hasselbring, and L. Carr. Performance Analysis of Legacy Perl Software via Batch and Interactive Trace Visualization. Technical Report TR-1509, Department of Computer Science, Kiel University, Germany, Aug. 2015.