INSTITUT FÜR INFORMATIK

Performance Analysis of Legacy Perl Software via Batch and Interactive Trace Visualization

Christian Zirkelbach, Wilhelm Hasselbring,
Florian Fittkau, and Leslie Carr

Bericht Nr. 1509
August 2015
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

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 the monitoring of the legacy software, and the second task covers analysing and visualizing the results. Dynamic analysis is usually addressed via trace visualization, but finding an appropriate representation for a specific issue still remains a great challenge.

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 the Kieker monitoring framework, and apply and combine two types of visualization tools, namely Graphviz and Gephi. More precisely, we employ Kieker to reconstruct architectural models from recorded monitoring data, based on dynamic analysis, and Graphviz respectively Gephi for further analysis and visualization of our monitoring results. We acquired knowledge of the software through our instrumentation and analysis via Kieker and the combined visualization of the two aforementioned tools. This allowed 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 [Cornelissen et al. 2009]. 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 [Harnad et al. 2004; Beazley 2010] with focus on analysing and evaluating it using the monitoring framework Kieker [van Hoorn et al. 2012]. EPrints has been continuously developed for more than fifteen years with Perl, a family of high-level, general-purpose, interpreted, dynamic programming languages [Srinivasan 1997]. In order to aid the process of program comprehension, we analyze our monitoring results with two types of trace visualization and used their advantages to address different purposes and phases within our project. Therefore we combined the batch-oriented visualization tool Graphviz [Gansner and North 2000] with the interactive visualization tool Gephi [Bastian et al. 2009].

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. To support this process, we reverse engineered Version 3.3.12 to provide useful information for restructuring the new release. In parallel, the new version was instrumented to continuously observe and analyze the new software during the restructuring phase. Dynamic analysis serves for the reverse engineering and performance analysis to support a major restructuring of the studied software. In the past, we analysed Java-based software and non-Java software implemented in programming languages such as C++, C#, Visual Basic and COBOL. Kieker employs Graphviz for the visualization [Knoche et al. 2012]. The project presented in this paper is the first to combine Graphviz with Gephi for visualizing monitoring data that was captured with Kieker. We do not report on a new tool, but a new combination of tools for visualization a

reverse engineering process.

The rest of this paper is organized as follows. Section 2 describes the instrumentation with Kieker and shows how probes are integrated into the Perl source code. In Section 3, we will present the initial analysis results, visualized via Graphviz. We refine this analysis through interactive graph exploration with Gephi in Section 4. Section 5 reports on the detection of performance bottlenecks. In Section 6 we discuss related work regarding our approach. Finally in Section 7, we summarize our paper and indicate areas for future work.

2 Perl Instrumentation using Kieker

In this section we present our instrumentation with Kieker and describe how monitoring probes are integrated into the Perl source code.

2.1 Instrumentation

For our monitoring we employ a non-intrusive instrumentation technique: aspect-oriented programming (AOP) [Kiczales 1996]. Listing 1 shows our Perl instru-

```
1  use Sub :: WrapPackages
2          packages = > [qw(EPrints EPrints ::*)] ,
3          pre = > sub {
4                  use Kieker ;
5                  my $kieker = Kieker -> new () ;
6                  my $packageName = $_ [0];
7                  $packageName =~ s /::/./ g ;
8                  $packageName =~ /^(.*) \..*? $ /;
9                  $kieker -> EntryEvent ($packageName,$1);
10         },
11         post = > sub {
12                 use Kieker ;
13                 my $kieker = Kieker -> new () ;
14                 my $packageName = $_ [0];
15                 $packageName =~ s /::/./ g ;
16                 $packageName =~ /^(.*) \..*? $ /;
17                 $kieker -> ExitEvent ($packageName,$1);
18         };
```

Listing 1. Instrumentation – weaving monitoring probes into Perl

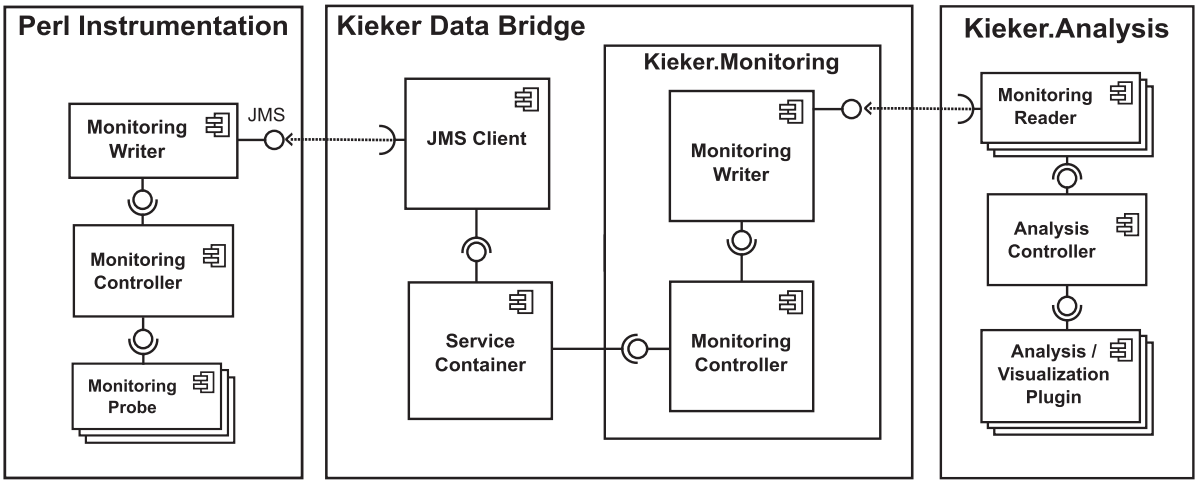

Figure 1. Instrumenting Perl with Kieker: components and assembly

mentation code. Based on the *WrapPackages* module,[1] it is possible to weave the instrumentation code, defined through *pre*, which initializes a starting monitoring probe, and *post*, an ending monitoring probe, around selected Perl packages [Wechselberg 2013]. Which Perl packages of EPrints are monitored, is indicated via the option *packages* using a regular expression. In principle, this mechanism corresponds to the so-called *around advices* in AspectJ, which provides aspect-oriented programming to Java [Kiczales et al. 2001].

2.2 Monitoring Architecture

Figure 1 depicts the core components and their assembly for our monitoring instrumentation. The Perl instrumentation component and the Kieker framework are also displayed. As mentioned before, we instrument the Perl software through weaving of monitoring probes into the application code. In order to analyse the gathered information with Kieker, it is necessary to convert the recorded data from Perl to Java. For this purpose we employ the Kieker Data Bridge (KDB).[2] The KDB is a tool to convert non-Java recorded data into well-structured Kieker Java records, hence they can be processed by Kieker to perform the analysis. Figure 1 illustrates the component structure of the Kieker Data Bridge with its interfaces. We are using JMS (Java Message Service) to send our Perl records from the *Monitoring Writer* to the KDB, which then employs *Kieker.Monitoring*, which provides an infrastructure for obtaining and logging measurements from software systems, to create valid Kieker records

[1] http://search.cpan.org/~dcantrell/Sub-WrapPackages-2.0/lib/Sub/WrapPackages.pm

[2] http://kieker.uni-kiel.de/trac/wiki/DevGuide/kdb

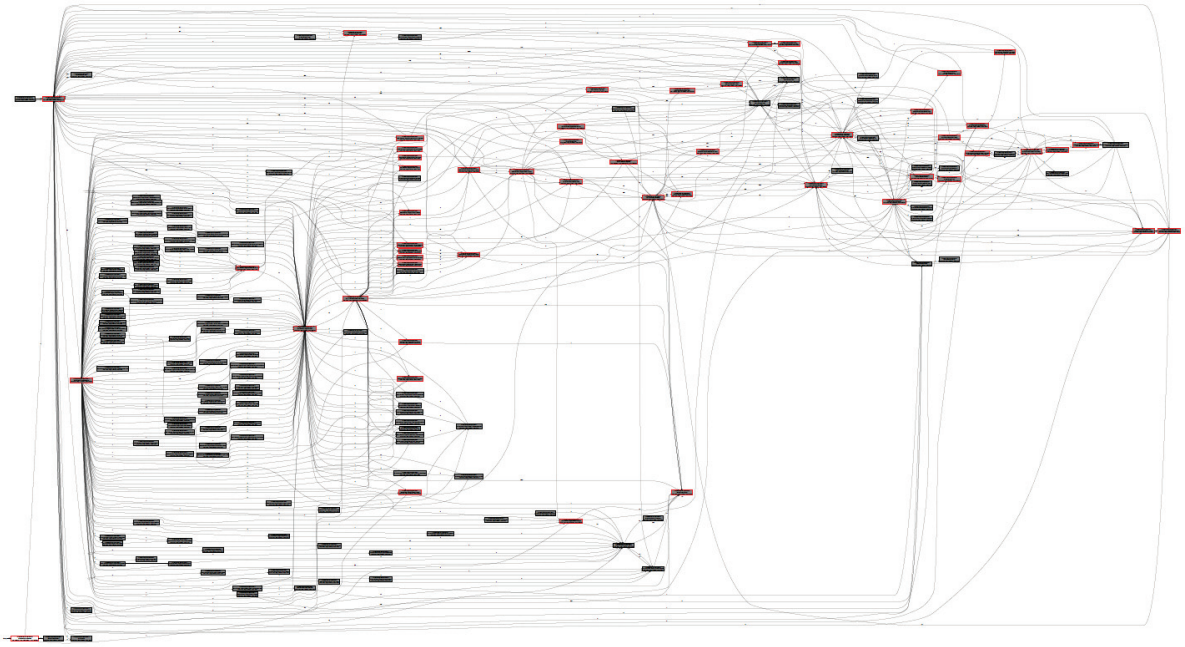

Figure 2. Initial component dependency graph for EPrints using Graphviz

[van Hoorn et al. 2009]. Within the Kieker framework, the *Kieker.Analysis* component is employed for the analysis. *Kieker.Analysis* employs Graphviz for visualization, as presented in the following section.

3 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 (*EPrints::** in Listing 1). By default, Kieker employs Graphviz [Gansner and North 2000],[3] a package of open-source tools, to visualize the generated graphs as so-called component dependency graphs. To indicate the complexity of an initial instrumentation, we present the result of monitoring in our use case in Figure 2. Since the initial analysis of a complete system usually provides such 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

[3] `http://www.graphviz.org`

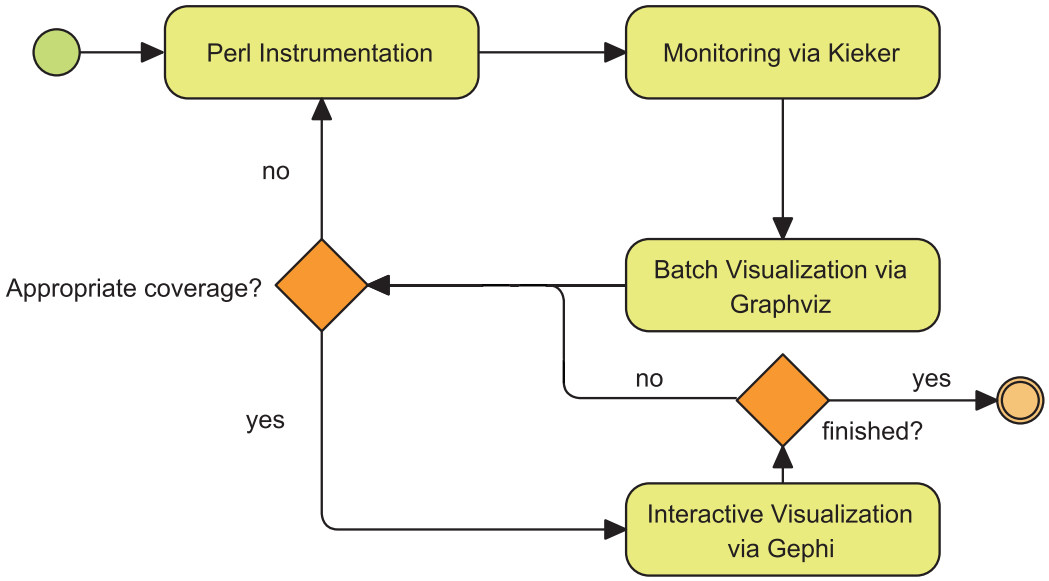

Figure 3. Performance Analysis Workflow

before, or via configuring Kieker's analysis pipeline, as we did in previous projects [Knoche et al. 2012; Hasselbring 2011]. Furthermore we can proceed and generate architectural diagrams and dependency graphs – on a component and operation level – to visualize the generated model [van Hoorn et al. 2009]. However, as we are interested in a performance analysis, the visualization of our graphs via Graphviz turned out as inappropriate. We needed support to modify the graph and to filter for highly-frequented packages and exceptional response times. This requires an iterative approach. Therefore we selected a tool that supports interactive graph exploration, as presented in the following section.

4 Interactive Graph Exploration with Gephi

Our initial analysis with Kieker and visualization via Graphviz provided an overview of the software system. We tried to reconstruct the EPrints architecture in order to identity packages that may contain potential bottlenecks. Since our visualization via Graphviz met its limits for our purpose, as reported in Section 3, we employed another visualization tool, namely *Gephi* [Bastian et al. 2009], an interactive visualization and exploration platform for handling graphs.[4] The iterative workflow, we followed for our performance analysis, is illustrated in Figure 3.

[4] http://gephi.org

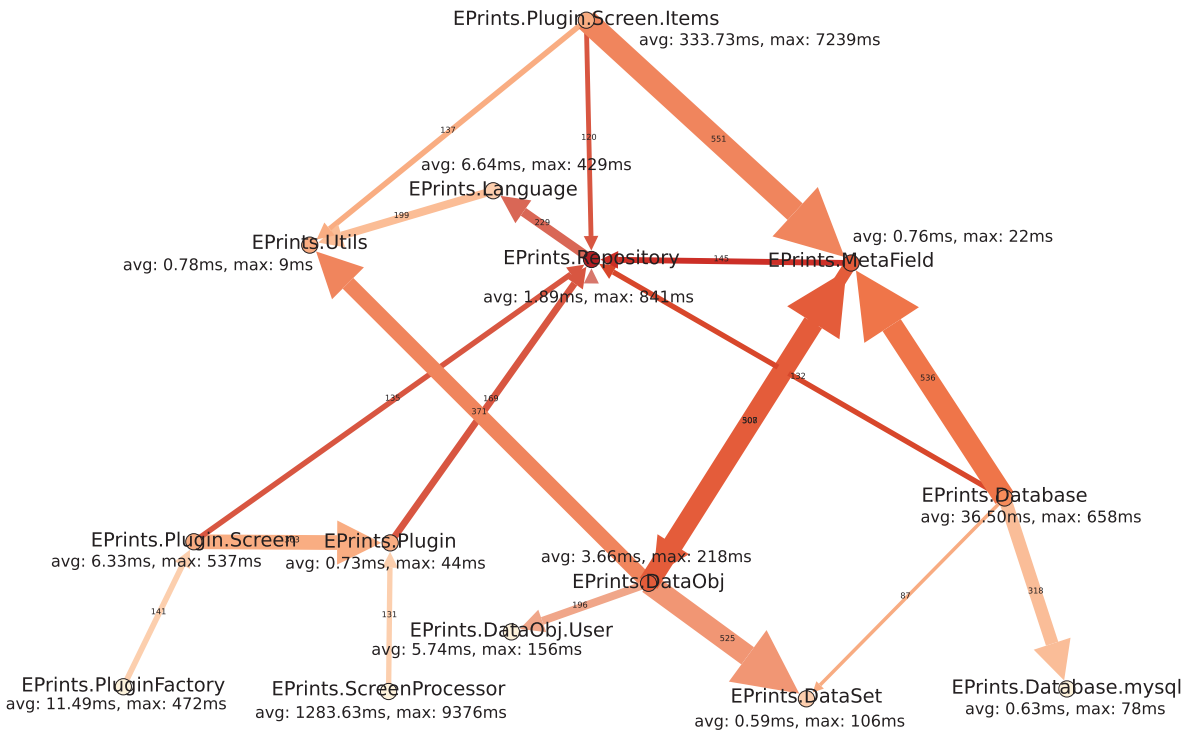

Figure 4. Reduced component dependency graph for EPrints using Gephi

Gephi is able to import the Graphviz graphs that were 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 in Figure 2 was interactively analyzed and reduced with Gephi. Furthermore, some nodes were aggregated based on their package hierarchy in order to obtain a more suitable overview with respect to a system architecture level.

Figure 4 shows the reduced component dependency graph, based on the full instrumentation. The Nodes represent Perl packages, including their sub packages. Edges express dependencies among them. Compared with our initial graph in Figure 2, the reduced graph is well-structured. We can immediately see the dependencies between the packages, displayed as directed edges, followed by the number of calls, and their average and maximum response times. Aiming at further abstraction, we further reduced the graph with Gephi to display only first-level packages. As a result we obtained the coloured dependency graph in Figure 5. The colors indicate the source nodes of the edges and the numbers represent the number of calls for this specific

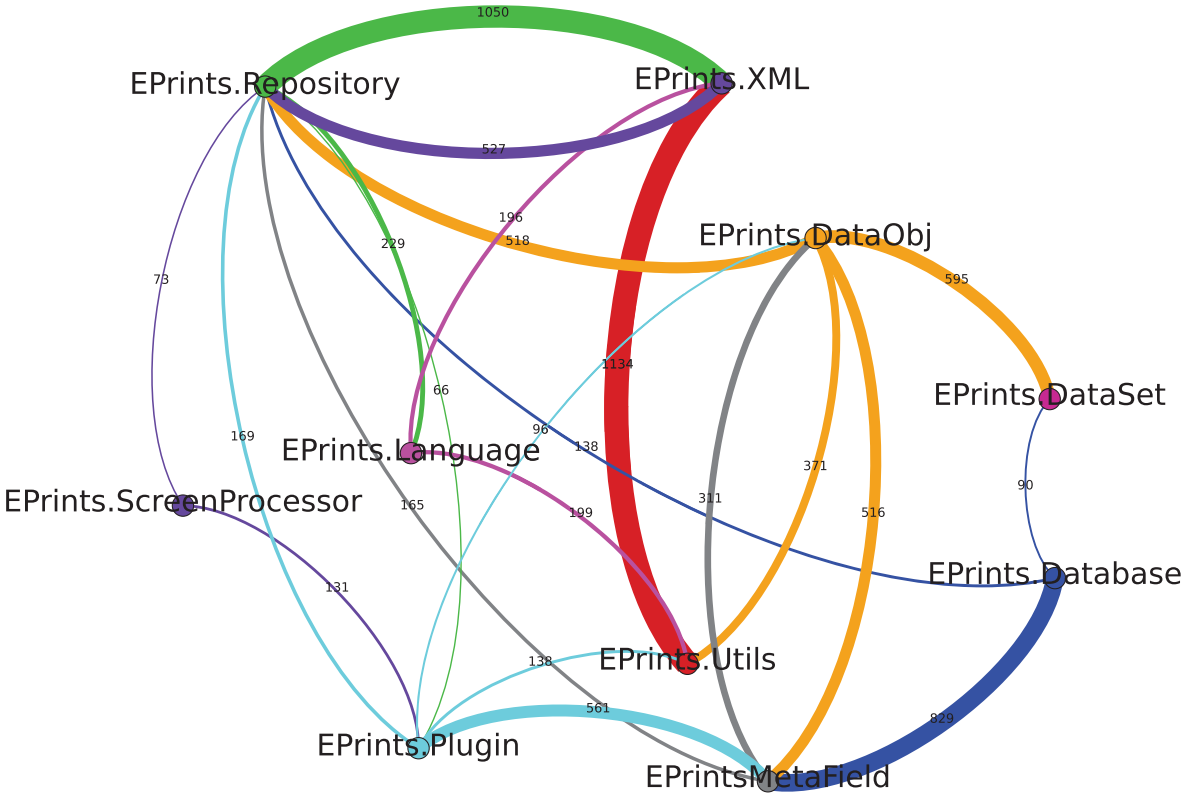

Figure 5. Colored component dependency graph for EPrints using Gephi

edge.

At this stage of our performance analysis, we were able to focus our visualization on the identification of packages and their dependencies that might cause performance issues. Thereupon, we found 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 identified a violation of intended architecture rules. This observation led us to a detailed analysis of dependencies regarding *EPrints.MetaField.** using Gephi. We report on this performance analysis in the next section.

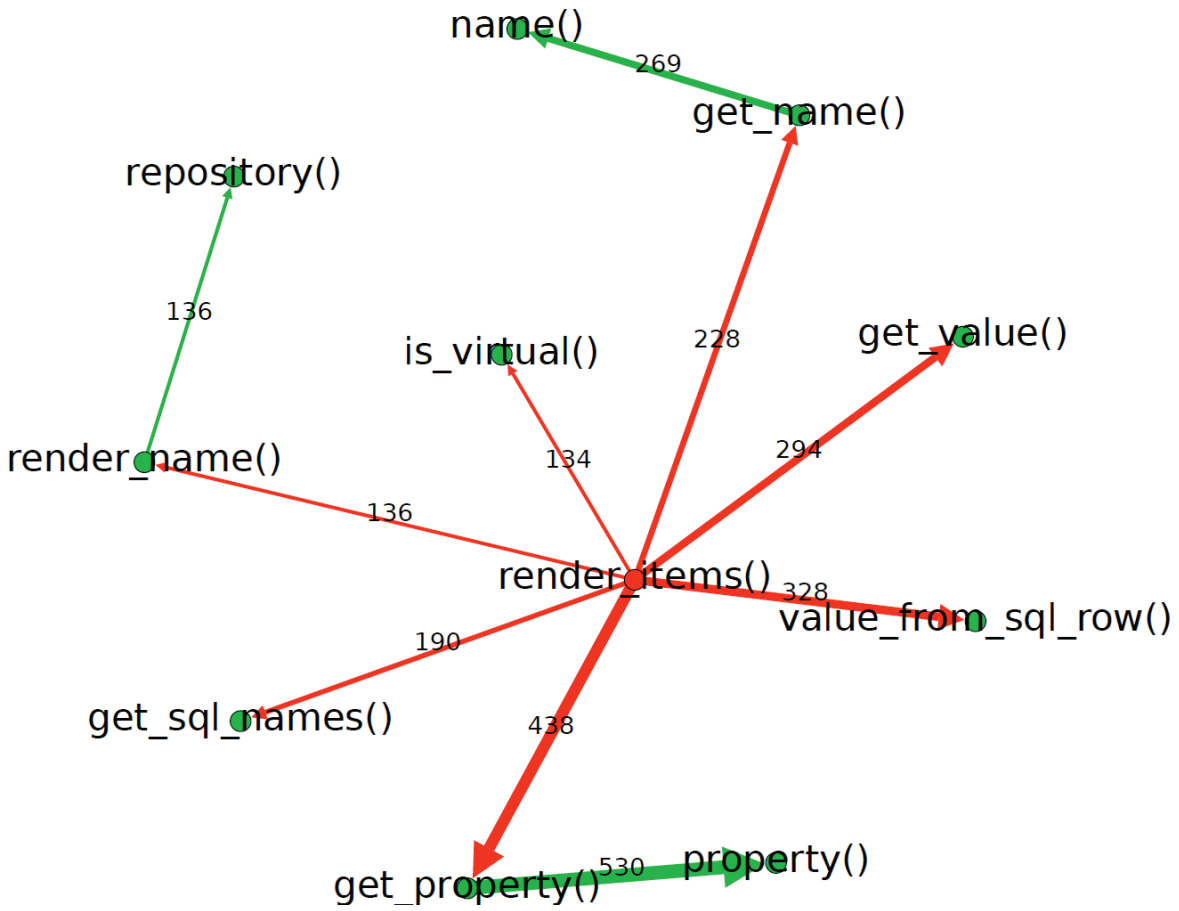

Figure 6. Detailed analysis of dependencies for EPrints.MetaField.* using Gephi

5 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 were able to instrument the software at dedicated places in the source code to focus on potential performance bottlenecks. We started 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 were able to decide whether we reached a sufficient instrumentation level or not. In insufficient cases, we further refined our instrumentation, until we were satisfied with the obtained level of detail. This approach is based on the aforementioned workflow, which was illustrated in Figure 3. Subsequently, we used Gephi to interactively modify, and visualize the graphs.

Our first detailed instrumentation was applied to the package *Screen.Items.**. This is a central package with multiple sub packages, that handles processing display components. We analysed the dependencies among the operations within these packages and also their respective execution times. Additionally, we measured operations with a response time greater than 100 ms as potential performance issue candidates. In this context we ignored the number of calls and focused on the two most suspicious operations, namely *render()* and *render_items()*, which took a large share of the overall response time. So we proceeded with an adapted instrumentation [Ehlers et al. 2011]. After drilling down the monitoring instrumentation to this specific area within the analysed software system, the visualization via Graphviz was sufficient for our first analysis purposes.

However, to find causes for the high response times within these two operations, it was necessary to locate the corresponding outgoing calls (edges) within the component dependency graph, based on the initial full instrumentation using filtering techniques. This resulted in monitoring the package *EPrints.Database.**, allowing us to find the database-related operations that may cause high response times. The outcome was a more detailed instrumentation, caused by a set of operations, that took a large share of the total response time.

With respect to the obtained maximum response times, two operations, namely *do()* and *_create_table()*, were suspicious. In comparison to the total response time, they took up to a third of the total. This is a significant share and thereupon we further analyzed these operations with respect to possible performance issues.

Additionally, we examined the *EPrints.MetaField.** packages, as a potential performance issue. Again, we refined the instrumentation and generated an operation-level dependency graph. This lead us to a detailed analysis of the dependencies for*EPrints.MetaField* using Gephi, as shown in Figure 6. The graph shows related calls annotated with the number of calls. We focused 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. Based on this analysis, we were able to identify specific source code locations in order to remove performance bottlenecks.

6 Related Work

In this section, we discuss some related work in the area of trace visualization. Lange et al. [Lange and Nakamura 1995] 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* [De Pauw et al. 2005] 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* [Cornelissen et al. 2007], 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* [Fittkau et al. 2013], 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.

7 Summary and Future Work

In this paper, we reported on our approach of conducting a performance analysis of the long-term developed software system (the Perl-based repository software EPrints) combining two visualization tools. We employed Kieker to reconstruct architectural models, using reverse engineering based on the monitored data, and Graphviz respectively Gephi to visualize the results. The first visualization tool Graphviz performs batch-style graph processing, which is sufficient for initial monitoring purposes, such as getting an overview of the software or serving as an orientation of a specific instrumentation. However, for a detailed analysis a reduction of the visualized data is required. Therefore, we performed an interactive graph processing via Gephi, based on an initial Graphviz graph representation. The gained knowledge was used for tasks such as identifying possible bottlenecks, reverse engineering, and

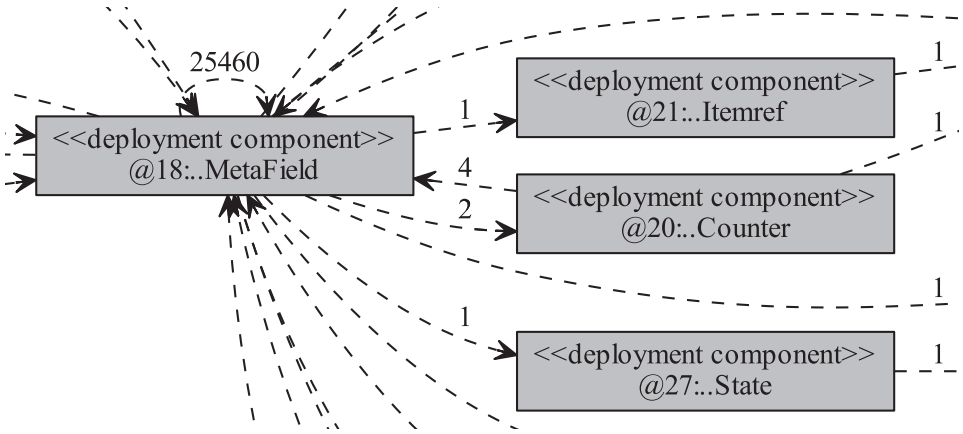

Figure 7. Excerpt of the component dependency graph of the new EPrints Version 4, containing an infinite loop (stopped after 25.460 executions)

restructuring the software system.

At the beginning of the project, the EPrints development team already suspected some performance issues within their current version. However, there were also new and unexpected insights gained through the instrumentation and analysis based on Kieker in combination with the visualization via Graphviz and Gephi. We detected some performance bottlenecks within the software and could advise some changes for the next release. We also used the new Kieker Data Bridge tool and improved our Perl instrumentation module during this project. 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 an infinite loop within the *MetaField* package. The loop is illustrated in Figure 7. In order to generate this graph the corresponding request was aborted after a certain time. Based on this discovery, the EPrints team was able to fix this bug at an early stage.

There is still room for further analysis. During our instrumentation we covered just a small part of EPrints. There will be other parts of the source code and other issues that could be interesting to take a closer look at. For the forthcoming new release of EPrints Version 4, Kieker, combined with the aforementioned two visualization tools, can be integrated upfront to establish automated quality management procedures such as continuous integration [Duvall et al. 2007]. This way, a new reverse engineering project for EPrints can be a avoided in a few years. Furthermore, we plan to perform additional performance analyses with other application performance management tools.

Figure 8. Application-level perspective: full-instrumented software run of EPrints

As a first step we employed ExplorViz, which is able to import our generated Perl monitoring logs, and conducted a performance analysis [Fittkau et al. 2015b] based on our initial full instrumentation. Figure 8 shows the resulting 3D software model within the system level perspective, which applies a visualization based on the city metaphor. The related screenshot shows the underlying architecture of EPrints and illustrates the dependencies between inherited packages. Additionally, we created a physical 3D model, based on the generated model, in order to further support the program comprehension within the major restructuring development phase of the upcoming release version 4. We used a 3D printer in combination with polylactic acid (PLA) to achieve a physical artifact of our 3D model. More information on how physical 3D models can be used to supporting software engineering is available in [Florian Fittkau and Erik Koppenhagen and Wilhelm Hasselbring 2015; Fittkau et al. 2015c]. Our physical printed 3D model is presented in Figure 9.

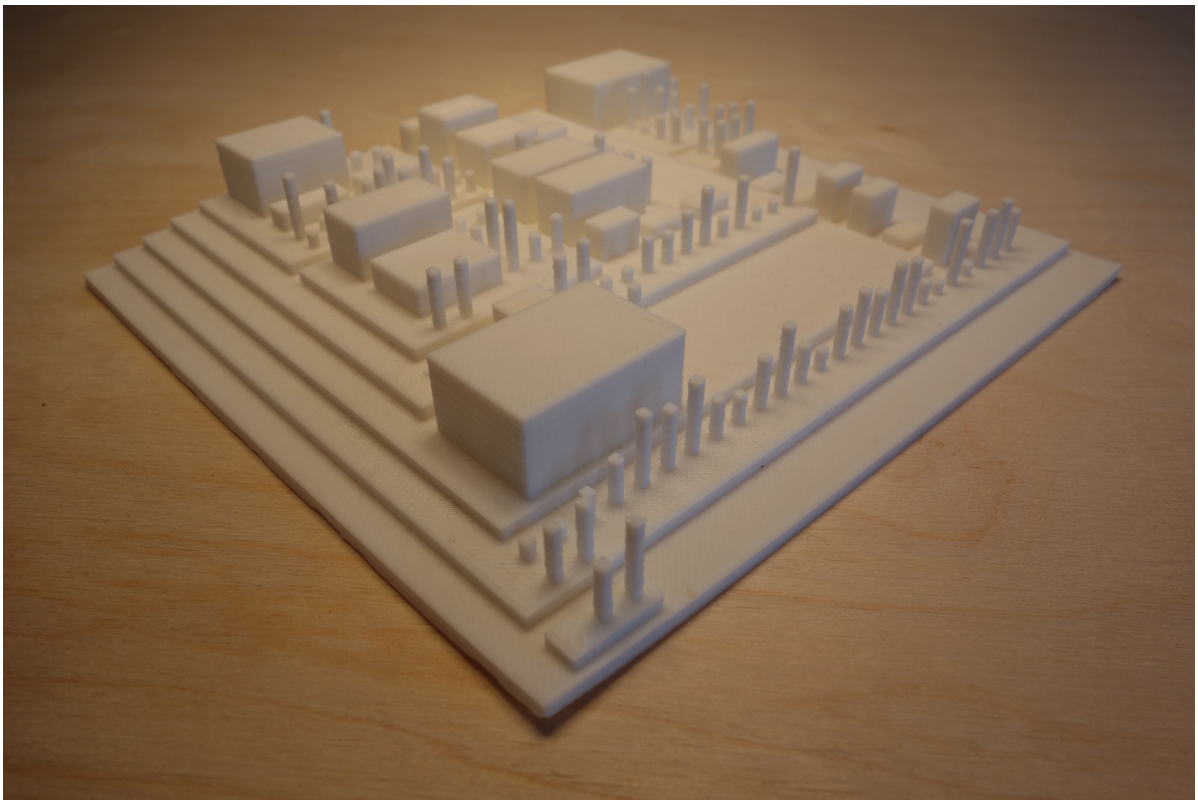

Figure 9. 3D printed physical 3D Model of EPrints (PLA)

Furthermore, the EPrints Team printed their own physical 3D Model, in combination with ace resin, on another 3D Printer, which is displayed at Figure 10. Additionally, we employed a VR approach to explore our generated software visualizations by using a head-mounted display and gesture-based interaction [Fittkau et al. 2015a].

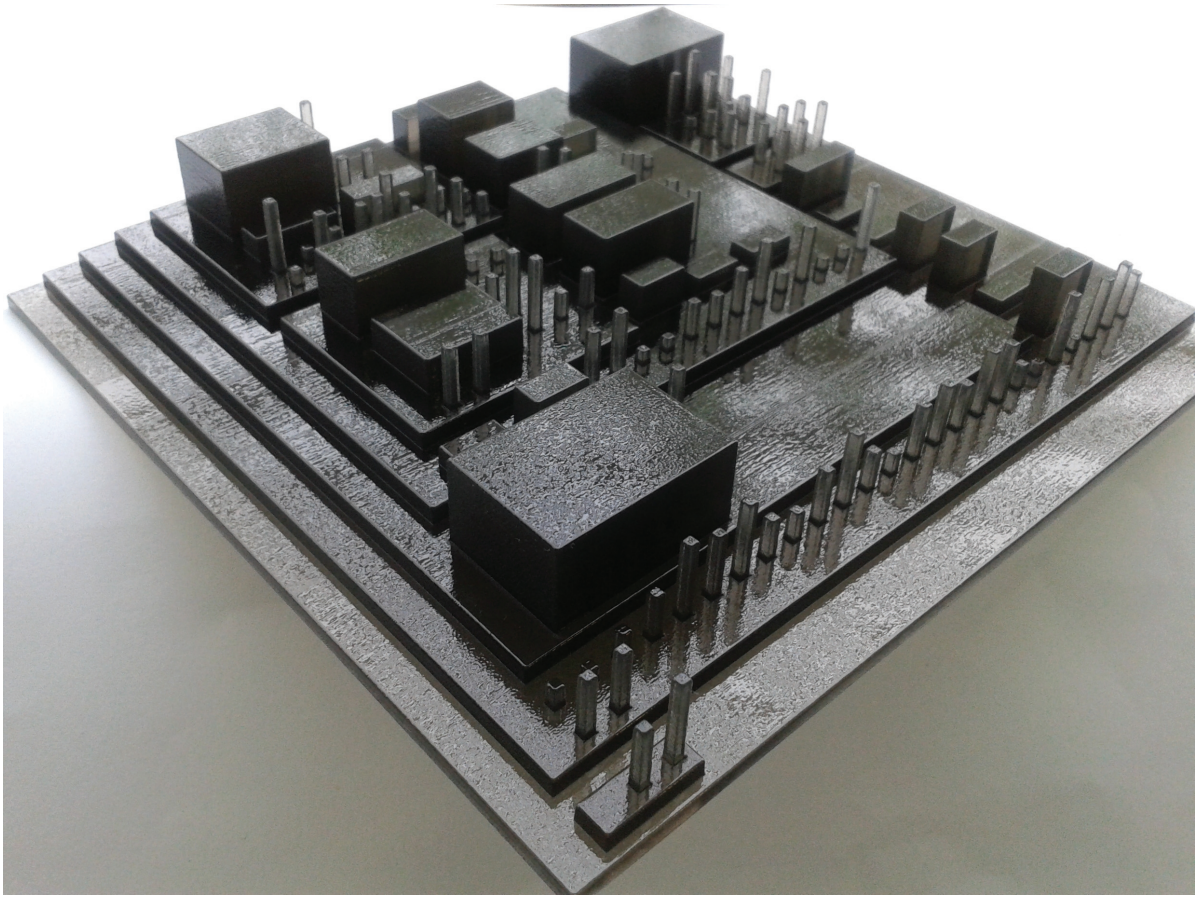

Figure 10. 3D printed physical 3D Model of EPrints (ace resin)

Acknowledgment

The authors would like to thank Sebastien Francois for his contributions to the EPrints instrumentation and analysis.

Bibliography

[Bastian et al. 2009] M. Bastian, S. Heymann, M. Jacomy, et al. Gephi: an open source software for exploring and manipulating networks. In: *Proceedings of the Third International AAAI Conference on Weblogs and Social Media*. 2009, pages 361–362. (Cited on pages 3 and 7)

[Beazley 2010] M. R. Beazley. EPrints institutional repository software: a review. *Partnership: the Canadian Journal of Library and Information Practice and Research* 5.2 (2010). (Cited on page 3)

[Cornelissen et al. 2007] 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 International Conference on*. 2007. (Cited on page 12)

[Cornelissen et al. 2009] 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 (2009), pages 684–702. (Cited on page 3)

[De Pauw et al. 2005] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. Morar. Web Services Navigator: Visualizing the execution of Web Services. *IBM Systems Journal* 44.4 (2005). (Cited on page 12)

[Duvall et al. 2007] P. M. Duvall, S. Matyas, and A. Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, July 2007. (Cited on page 13)

[Ehlers et al. 2011] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In: *Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011)*. Karlsruhe, Germany: ACM, June 2011, pages 197–200. (Cited on page 11)

[Fittkau et al. 2013] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. 2013. (Cited on page 12)

[Fittkau et al. 2015a] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In: *3rd IEEE Working Conference on Software Visualization*. IEEE, 2015. (Cited on page 15)

[Fittkau et al. 2015b] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. In: *Proceedings of the 23rd European Conference on Information Systems (ECIS 2015)*. 2015. (Cited on page 14)

[Fittkau et al. 2015c] F. Fittkau, E. Koppenhagen, and W. Hasselbring. Research Perspective on Supporting Software Engineering via Physical 3D Models. In: *3rd IEEE Working Conference on Software Visualization*. IEEE, 2015. (Cited on page 14)

[Florian Fittkau and Erik Koppenhagen and Wilhelm Hasselbring 2015] Florian Fittkau and Erik Koppenhagen and Wilhelm Hasselbring. Research Perspective on Supporting Software Engineering via Physical 3D Models. Research Report. Kiel University, 2015. (Cited on page 14)

[Gansner and North 2000] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience* 30.11 (2000), pages 1203–1233. (Cited on pages 3 and 6)

[Harnad et al. 2004] S. Harnad, T. Brody, F. Valliéres, L. Carr, S. Hitchcock, Y. Gingras, C. Oppenheim, H. Stamerjohanns, and E. R. Hilf. The access/impact problem and the green and gold roads to open access. *Serials Review* 30.4 (2004), pages 310–314. (Cited on page 3)

[Hasselbring 2011] W. Hasselbring. Reverse engineering of dependency graphs via dynamic analysis. In: *Proceedings of the 5th European Conference on Software Architecture*. ECSA '11. ACM, 2011. (Cited on page 7)

[Kiczales 1996] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.* 28.4 (Dec. 1996). (Cited on page 4)

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "'An overview of AspectJ'". In: *ECOOP 2001 Object-Oriented Programming*. Springer, 2001, pages 327–354. (Cited on page 5)

[Knoche et al. 2012] H. Knoche, A. van Hoorn, W. Goerigk, and W. Hasselbring. Automated Source-Level Instrumentation for Dynamic Dependency Analysis of COBOL systems. In: *Proceedings of the 14. Workshop Software-Reengineering (WSR '12)*. Bad Honnef, Germany, May 2012, pages 45–46. (Cited on pages 3 and 7)

[Lange and Nakamura 1995] D. B. Lange and Y. Nakamura. Program Explorer: A Program Visualizer for C++. In: *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'95)*. 1995. (Cited on page 12)

[Srinivasan 1997] S. Srinivasan. Advanced Perl Programming. O'Reilly Media, 1997. (Cited on page 3)

[Van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009, 27 pages. (Cited on pages 6, 7)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA: ACM, 2012, pages 247–248. (Cited on page 3)

[Wechselberg 2013] N. B. Wechselberg. Monitoring Perl-based Web Applications with Kieker. Bachelor Thesis. Kiel University, Apr. 2013. (Cited on page 5)