EasyChair Preprint

№ 4371

Graph-Based Performance Analysis at System- and Application-Level

Richard Müller and Tom Strempel

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 12, 2020

Graph-Based Performance Analysis at System- and Application-Level

Richard Müller
rmueller@wifa.uni-leipzig.de
Leipzig University, Leipzig, Germany

Tom Strempel
ts66cyqo@studserv.uni-leipzig.de
Leipzig University, Leipzig, Germany

Abstract

The Kieker plugin for jQAssistant transforms monitored log data into graphs to support software engineers with performance analysis. In this paper, we describe how we have extended and improved this plugin to support performance analysis at system- and application-level and how we have evaluated its correctness and scalability using data from recent experiments. This is a first step to replicate complete experiments in the field of performance analysis using graphs.

1 Introduction

Graphs can be used to store and integrate data from different software artifacts [1]. The graph data support software engineers to make informed decisions. To create this graph-based data source we use the jQAssistant[1] framework and Neo4j[2].

We have developed a jQAssistant plugin that scans Kieker log data and stores them in a Neo4j graph database [3]. However, the initial version of the plugin had some issues. First, it was not possible to scan system-level information, such as CPU and system memory utilization. Second, the graph schema contained some redundant information leading to high storage space requirements. Third, there were some implementation flaws leading to long scan times. Finally, the plugin was only evaluated with regard to feasibility but not to scalability.

Hence, the contributions of this paper are as follows. We extend and improve the Kieker plugin to solve the aforementioned issues and evaluate its correctness and scalability using data from recent experiments. We also provide a reproduction package to replicate our results.

2 Related Work

The Kieker framework provides monitoring, analysis, and visualization support for application and system performance analysis as well as reverse engineering [4]. It has been used in several research projects and has a steadily growing open source community.

There are two recent experiments using Kieker log data for empirical analysis. Pitakrat et al. [2] combine component failure predictors with architectural knowledge to improve failure prediction. They use Kieker log data monitored at application- and system-level, such as method calls, CPU, and system memory utilization. Schnoor and Hasselbring [5] investigate the correlation between weighted dynamic and static coupling metrics. They want to find out how weighted dynamic coupling measurements can support software engineers to evaluate the architectural quality of software systems. Therefore, they use Kieker log data from monitoring production use of a commercial software system over a period of four weeks.

We have shown that graphs are suitable to store Kieker log data and form a sound basis for further analysis and visualization [3]. In this paper, we extend and improve this work and evaluate it using data from the aforementioned experiments.

3 Kieker Plugin

Next, we will show how the Kieker plugin[3] has been extended and improved to support performance analysis at system- and application-level. The changes are summarized in the Kieker graph schema shown in Figure 1.

3.1 Extensions

The Kieker plugin has been extended to support system-level measurements including `CPUUtilization`, `DiskUsage`, `LoadAverage`, `MemSwapUsage`, and `NetworkUtilization` with their corresponding properties. We have chosen the common node type `Measurement` instead of `Record` for these nodes to avoid confusion with the existing node `Record` which contains both, methods and measurements.

Furthermore, we have added two concepts to generate the weighted dynamic dependency graph at package level. The first concept creates the `CONTAINS` relationship between `Package` and `Type` nodes. The second concept creates the `DEPENDS_ON` relationship between `Package` nodes based on the dependency graph at class level, where the dependency originally refers to method calls.

Now, it is possible to scan and store system measurements and to automatically create the weighted dynamic dependency graph at package level.

[1] `https://jqassistant.org/`
[2] `https://neo4j.com/`

[3] `https://github.com/softvis-research/jqa-kieker-plugin`

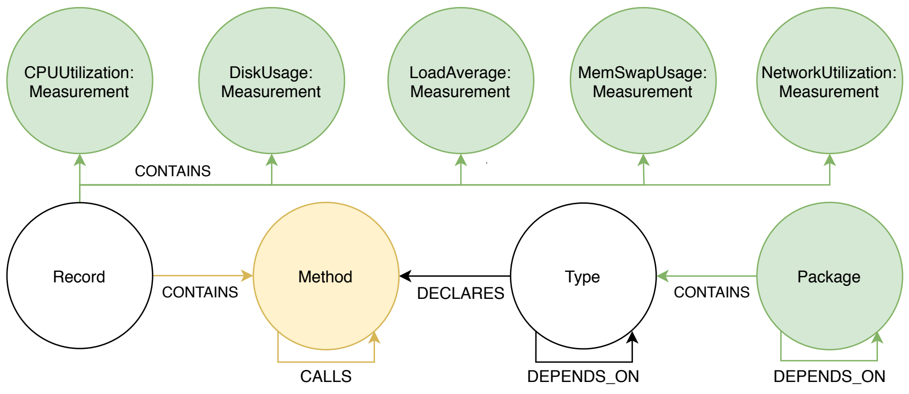


Figure 1: The extended (green) and improved (yellow) Kieker graph schema

3.2 Improvements

In the initial version of the Kieker graph schema each call between two methods was mapped to a unique `CALLS` relationship [3]. Now, there is one relationship between two calling methods and the number of calls is stored in the property `weight` of the `CALLS` relationship. Additionally, the properties `incomingCalls` and `outgoingCalls` of a `Method` node were removed as they can be derived from the call graph through counting incoming and outgoing `CALLS` relationships of a `Method` node.

The `Event` nodes of type `Execution` and `Call` acted as helper nodes to calculate the `Method` property `duration` and the `CALLS` relationship between methods. Both events could be removed as they are now calculated and created during the scan. Moreover, the `Trace` node was removed as it does not store any relevant information for the analyses. Hence, the `Record` node has a direct relationship `CONTAINS` to a `Method` node.

These changes lead to a significant reduction of the graph's storage space requirement, to time savings during the scan, and simultaneously to increased scalability. The graph schema can be adapted with manageable effort if further data from the monitoring log is required.

4 Evaluation

We evaluate the new functionality and the scalability of the Kieker plugin by scanning Kieker log data from two experiments to reproduce charts and values. Both analyses can be replicated online with our reproduction package[4].

4.1 Performance Analysis at System-Level

Pitakrat et al. [2] published their monitoring data in the online version of their article[5]. We use these data for our first evaluation to show that the Kieker plugin scans system-level measurements correctly. Therefore, we reproduce two line charts showing the system-level measures CPU and system memory utilization of the second business-tier instance (BT2) from [2].

After scanning the monitoring data, we get the CPU utilization measurements of BT2 by executing the Cypher query in Listing 1.

Listing 1: Cypher query for all CPU measurements of BT2

```
MATCH (r:Record)-[:CONTAINS]->(c:CpuUtilization)
WHERE r.fileName =~ '.*/1-MemoryLeak-5/kieker-logs/
kieker-20150820-064855519-UTC-middletier2-KIEKER'
RETURN c.timestamp AS timestamp, c.cpuID AS cpuID,
c.totalUtilization * 100 AS cpuUtilization
ORDER BY timestamp
```

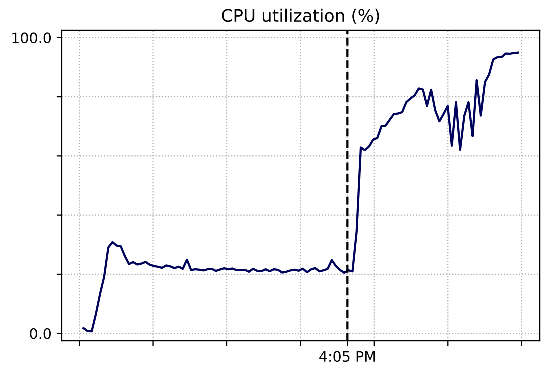
Next, the data are filtered, transformed, and cleaned. The complete analysis and the Cypher query for system memory utilization can be found in our reproduction package. Finally, we have reproduced the line charts for CPU and system memory utilization of BT2 shown in Figure 2. The charts slightly differ from the originals as we do not know all data transformations. Nevertheless, the courses of the line charts correspond to the originals.

4.2 Performance Analysis at Application-Level

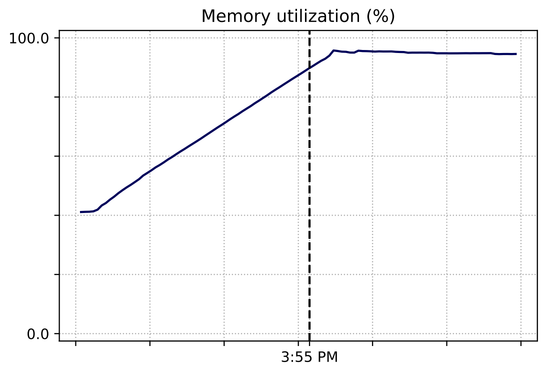
Schnoor and Hasselbring [5] published the monitoring data that they used for their analysis on Zenodo. We use the fourth dataset[6] from September 2018 for our second evaluation of the scalability as this is the biggest one with 58 users and 2,409,688,701 method calls.

We have measured the time for scanning the monitoring data on a HP EliteBook 850 G4 (Intel Core i7-7600U CPU @ 2.8 GHz (4 CPUs), 16 GB RAM) with Windows 10 as operating system and a Java development toolkit in version 8. It takes 1 h 38 min 29 s to scan all monitoring data including the creation of the dynamic dependency graphs at class and package level.

[4] https://github.com/softvis-research/SSP2020
[5] https://ars.els-cdn.com/content/image/1-s2.0-S0164121217300390-mmc1.zip
[6] https://doi.org/10.5281/zenodo.3648269



(a) CPU utilization



(b) System memory utilization

Figure 2: Reproduced line charts of system-level measurements of BT2 from [2]

The original dataset is in a packed binary format and occupies 8.89 GB disk space. The graph database needs 110 MB disk space. This reduction is mainly due to omitting the node types `Event` and `Trace` including their properties.

To show that the Kieker plugin can handle such a large amount of data, we have executed the Cypher query in Listing 2 and received the expected result of 2,409,688,701 method calls.

Listing 2: Cypher query to get the total number of method calls of dataset 4 from [5]

```
MATCH (:Method:Kieker)-[calls:CALLS]->(:Method:Kieker)
RETURN SUM(calls.weight) AS methodCalls
```

After the scan, the Kieker plugin automatically creates the weighted dynamic dependency graphs at class and package level using concepts. To examine the correctness of these graphs, we compare the average export coupling degrees of both graphs with the corresponding values from the experiment [5]. We have executed the Cypher query in Listing 3 and received the expected result of 370,821 as average export coupling degree at class level. The Cypher query for the average export coupling degree at package level also returned the correct value of 1,868,664. The complete analysis can be found in the reproduction package.

Listing 3: Cypher query to calculate the average export coupling degree at class level of dataset 4 from [5]

```
MATCH (t:Type:Kieker)
WHERE (t)-[:DEPENDS_ON]->() OR ()-[:DEPENDS_ON]->(t)
WITH t
OPTIONAL MATCH (t)-[out:DEPENDS_ON]->()
WITH t, SUM(out.weight) AS import
OPTIONAL MATCH ()-[in:DEPENDS_ON]->(t)
WITH t, import, SUM(in.weight) as export
RETURN ROUND(AVG(export)) AS averageExport
```

5 Conclusion and Future Work

We have presented extensions and improvements of the Kieker plugin to support performance analysis at system- and application-level using graphs. The modified plugin has been evaluated using Kieker log data from two experiments. First, we could reproduce charts from a comprehensive system performance analysis. Second, we could show that the plugin is able to process 2,409,688,701 method calls and to reproduce weighted dynamic dependency graphs at class and package level used in an application performance analysis. Both analyses can be reproduced with the provided reproduction package.

In the future, we plan to replicate the complete experiment comparing static and dynamic metrics [5]. We will use the Kieker plugin to generate dynamic dependency graphs and the Java bytecode scanner plugin[7] to generate static dependency graphs.

References

[1] R. Müller et al. "Towards an Open Source Stack to Create a Unified Data Source for Software Analysis and Visualization". In: *Proceedings of the 6th IEEE Working Conference on Software Visualization.* Madrid, Spain: IEEE, 2018.

[2] T. Pitakrat et al. "Hora: Architecture-aware online failure prediction". In: *Journal of Systems and Software* 137 (2018), pp. 669–685.

[3] R. Müller and M. Fischer. "Graph-Based Analysis and Visualization of Software Traces". In: *10th Symposium on Software Performance: Joint Developer and Community Meeting of Descartes/Kieker/Palladio.* Würzburg, Germany, 2019.

[4] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (Aug. 2020), pp. 1–5.

[5] H. Schnoor and W. Hasselbring. "Comparing Static and Dynamic Weighted Software Coupling Metrics". In: *Computers* 9.2 (Mar. 2020), p. 24.

[7] `https://github.com/jQAssistant/jqa-java-plugin`