

Overhead Comparison of OpenTelemetry, inspectIT and Kieker

David Georg Reichelt¹, Stefan Kühne¹ and Wilhelm Hasselbring²

¹University Computing Centre, Research and Development, Universität Leipzig

²Software Engineering Group, Christian-Albrechts-Universität zu Kiel

Abstract

Having low performance overhead when monitoring the performance is crucial for exact measurements. Especially when trying to identify performance changes at code level, the performance overhead needs to be as low as possible. Due to changes in monitoring frameworks, performance benchmarks need regular updates. Due to changes in virtual machines, operating systems and hardware environments, performance benchmarking results also need regular updates.

Therefore, we describe an extension of the benchmark *MooBench* which includes the emerging monitoring framework OpenTelemetry in *MooBench*, and the results of its execution on a Raspberry Pi 4 in this paper. We find that Kieker is creating slightly less overhead than inspectIT and OpenTelemetry when processing traces.

Keywords

performance measurement, performance monitoring, performance benchmarking, software performance engineering

1. Introduction

To assure that performance requirements are met, the performance of parts of a system needs to be measured under real conditions. This measurement in live operation is called monitoring [1, p. 45]. Monitoring data can be used to identify performance issues, to extract performance models or for online capacity management. To measure the performance, monitoring tools add monitoring probes, i.e. pieces of code capable of measuring the resource usage, into the monitored system and serialize the monitoring records. The instrumentation, the measurement itself and the serialization cause monitoring overhead. Especially for the identification of performance changes at code level [2], the monitoring overhead needs to be as low as possible.

Benchmarking compares different methods, techniques and tools and is used widely to compare the performance of different implementation [3]. Therefore, the *MooBench* benchmark has been introduced to compare the performance overhead of different monitoring frameworks [4]. Originally, *MooBench* was able to measure the performance of the performance monitoring frameworks Kieker [5], inspectIT¹ and SPASSmeter [6].


Recently, the monitoring framework OpenTelemetry² emerged. It provides monitoring

SSP'21: Symposium on Software Performance, November 09–10, 2021, Leipzig, Germany

 <https://www.urz.uni-leipzig.de/fue/DavidGeorgReichelt/> (D. G. Reichelt)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

¹<https://www.inspectit.rocks/>

²<https://opentelemetry.io/>

support for a variety of languages and frameworks, and can therefore be used in different contexts. This paper presents an extension of MooBench that enables measuring the overhead of OpenTelemetry and Kieker and results of the execution of the extended MooBench.

In the remainder of this paper, we first describe the benchmark MooBench and our extension of MooBench for measurement of OpenTelemetry in Section 2. Afterwards, we describe the measurement results of our extended MooBench version in Section 3. In Section 4, we discuss related work. Finally, we summarize this paper and give an outlook to future work in Section 5.

2. Supporting OpenTelemetry in MooBench

This section first gives an overview of the benchmark MooBench and describes our extension of MooBench afterwards.

2.1. MooBench

MooBench is a benchmark for measuring the overhead of monitoring frameworks [4]. Performance measurement in the JVM is influenced by non-deterministic effects such as Just-In-Time-Compilation (JIT), Garbage Collections and memory fragmentation. Therefore, performance measurements need to be repeated inside of one started JVM, called VM in the remainder. Since warmup may end up in different steady states, multiple VMs need to be started and their results need to be analyzed by statistic tests such as T-Test [7].

MooBench provides a basic Java application that repeats busy waiting in the leaf node of a tree of nodes with only one child with a given recursion depth. For every monitoring framework, a Bash script automates the VM starts of the benchmark for the correspondent framework configuration. For the frameworks, the configurations contain (at least) the baseline (no instrumentation) and regular monitoring with serialization of the results. They may also contain deactivated monitoring (but enabled instrumentation) and different monitoring configurations (e.g. writing the results as generic text or as binary in Kieker). The measurement result of each VM run is saved as CSV into a result directory with a name denoting the VM configuration.

2.2. Extension of MooBench

To make OpenTelemetry runnable with MooBench, we implemented the instrumentation with OpenTelemetry. Additionally, the existing benchmarks for inspectIT were updated.

For instrumenting OpenTelemetry, we added a script. This script was built to support corresponding calls for each Kieker Call, i.e. (1) the baseline (no instrumentation), (2) monitoring with disabled sampling with `otel.traces.sampler=always_off`, i.e. no measurement will be done, but the instrumentation is still present, like Kieker with deactivated probe, (3) logging all method executions to standard output, like Kieker writing to hard disc, (4) logging all method executions (spans) to Zipkin and (5) logging metrics of executions to Prometheus, like Kieker writing to TCP. The measurement of Prometheus is disabled on 32-bit systems, since Prometheus only runs on 64-bit systems. Since results are saved in the MooBench CSV format, existing R scripts can be used for data analysis. Our forked version of MooBench is available on GitHub.³

³<https://github.com/DaGeRe/moobench-fork/>

3. Measurement Results

To facilitate reproducibility of our results, we decided to run our benchmarks on a Raspberry Pi like Knoche and Eichelberger [8]. We used the latest Raspberry Pi 4 running Raspberry OS (formerly known as Raspbian) 5.10.17-v7l+ in the 32 bit version and OpenJDK 11.0.11. Since Raspberry OS runs on 32-bit by default, the Jaeger serialization could not be used. To compare the Raspberry Pi results to a regular Desktop system, we also measured the same benchmarks on an i7-4770 CPU @ 3.40GHz with 16 GB RAM, running Ubuntu 20.04 and OpenJDK 11.0.11. We executed the benchmark with *call tree depth 10* (like Knoche and Eichelberger [8]) and with *growing call tree depth*.

3.1. Call Tree Depth 10

Our results of the execution of 2 000 000 calls with recursion depth 10 and 10 VM starts are depicted in Table 1 and Table 2. On Raspberry Pi and i7-4770, all relations stay equal, e.g. deactivated OpenTelemetry is slower than deactivated Kieker both on Raspberry Pi and i7-4770. Nevertheless, we see that the ratio of execution times changes, e.g. baseline execution is approximately 25 times faster on i7-4770 than on Raspberry Pi but execution with deactivated OpenTelemetry instrumentation is only approximately 5 times faster. Due to the limited instruction set of ARM processors, benchmark results on Raspberry Pi are probably not corresponding to benchmark results on desktop or server systems. Therefore, Raspberry Pi might not be a suitable hardware for benchmarking in every use case.

For the configurations, we see the following results: *Baseline*: The Raspberry Pi 4 with current software environment shows (as expected) slight improvements over the measurement values from Knoche

Variant	Raspberry Pi		i7-4770	
	95 % CI	σ	95 % CI	σ
Baseline	[1.5;1.5]	0.1	[0.057;0.058]	0.026
Kieker				
Deactivated Probe	[4.1;4.1]	7.5	[0.4;0.4]	7.1
DumpWriter	[51.9;52.0]	14.6	[8.5;8.5]	12.2
Logging (Text)	[743.3;799.4]	14315.8	[103.0;103.3]	56.4
Logging (Binary)	[59.8;87.8]	7149.4	[3.4;3.4]	15.8
TCP	[45.6;45.7]	14.6	[4.6;4.7]	10.4

and Eichelberger [8] on Raspberry Pi 3. *Deactivated*: The execution of Kieker with deactivated probe creates less overhead than the deactivated execution of OpenTelemetry and inspectIT. *Logging to hard disc*: With activated logging to the file system, Kieker with binary logging is faster than OpenTelemetry. Regular text logging is slower with Kieker. *External data processing*: With external data processing, by Zipkin for OpenTelemetry and TCP sending by Kieker, Kieker is slightly (but significantly) faster than OpenTelemetry and inspectIT. inspectIT is faster when metrics are processed by Prometheus.

3.2. Growing Call Tree Depth

Figure 1 shows the average of warmed up measured durations of all VMs with growing call tree depth. Even if call tree depth are discrete values, we chose to draw lines for better visibility.

Variant	OpenTelemetry				inspectIT			
	Deactivated Probe	StdOut	Zipkin	Prometheus	Deactivated Probe	Dump Writer	Zipkin	Prometheus
Pi 4								
CI	[26.8;26.9]	[483.0;508.1]	[53.4;53.6]	[44.4;44.5]	[9.9;9.9]	[87.2;87.5]	[97.2;97.8]	[32.3;32.4]
σ	20.4	6408.7	46.7	25.2	10.5	78.7	149.6	16.6
i7-4770								
CI	[4.9;5.0]	[56.9;57.7]	[6.8;6.9]	[6.9;6.9]	[1.3;1.4]	[10.3;10.4]	[10.9;11.2]	[4.0;4.0]
σ	4.1	222.5	8.5	4.9	8.2	17.4	57.4	4.1

Table 2
Measurement Results for OpenTelemetry and inspectIT

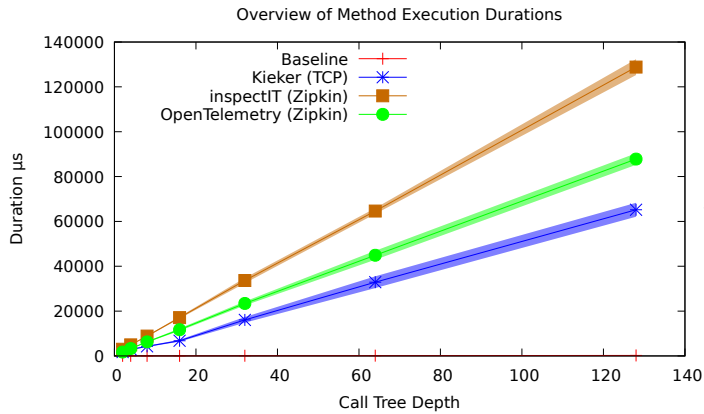


Figure 1: Growing Call Tree Depth with i7-4770

It show two things: (1) The overhead lineary increases with growing call tree depth, which is equal to growing count of instrumented methods. Therefore, instrumenting the whole application will always create big overhead. (2) The relations from call tree depth 10 persist: Sending the trace to Zipkin from OpenTelemetry or inspectIT creates more overhead then sending the trace using Kieker.

4. Related Work

Benchmarking is widely used for testing the performance of software in continuous integration [9]. To measure the performance, benchmarking harnesses like the Java Microbenchmarking Harness *JMH*⁴ provide an execution environment for workload specification and measurement. According to a study of Stefan et al. [9], only 3,4 % of all open source projects continously benchmark their software performance. Widespread frameworks like Hadoop [10] or Java itself⁵ contain benchmarks for continuous performance evaluation. Besides application monitoring overhead, other benchmarks cover other system classes as stream processing engines [11] or ML systems [12]. In contrast to these works, we examined the performance overhead of application performance monitoring.

Ahmed et al. [13] compare different APM tools by executing a load test on different systems and researching whether a performance regression could be identified. Afterwards, they check whether performance issues could be detected by thresholds in the commercial APM tools New Relic, AppDynamics and Dynatrace, and the open source tool Pinpoint⁶. They did not research

⁴<http://openjdk.java.net/projects/code-tools/jmh/>

⁵<https://www.spec.org/jvm2008/>

⁶<https://pinpoint-apm.github.io/pinpoint/>

the overhead of the tools, but their suitability for identification of performance changes.

In contrast to this work, MooBench [4] measures the overhead of performance monitoring tools. It is used continuously for measuring the performance overhead of Kieker. MooBench has been extended and used for testing the replicability of performance measurements on the Raspberry Pi by Knoche and Eichelberger [8] [14]. They used different benchmarks to assess the replicability of performance measurements on the Raspberry Pi. They find that the Raspberry Pi is capable of providing an infrastructure for replicable benchmark execution. In contrast to our work, they did not consider OpenTelemetry and use a Raspberry Pi 3. OpenTelemetry itself maintains continuous performance benchmarks for the performance of its python implementation.⁷ While the users of OpenTelemetry do occasional overhead measurement,⁸ no continuous benchmarking or benchmarking against other frameworks is done. Hence, a comparison of the monitoring overhead of OpenTelemetry in Java and Kieker has not been done so far.

Waller and Hasselbring [15] research the effects of activation of processor cores and multithreading to the monitoring overhead. They find that using one processor core with hyperthreading yields the lowest overhead in their configuration, since synchronization overhead between different processor cores increases the monitoring overhead. In contrast to their work, this work focusses on the comparison of different monitoring frameworks.

5. Summary and Outlook

We compared the monitoring overhead of OpenTelemetry and Kieker. Therefore, we extended the MooBench benchmark. By execution of the benchmarks on a Raspberry Pi 4 and a regular Desktop PC, we found that Kieker has better performance with serialization to hard disc and with processing the results with TCP. This relation also persists with growing call tree depth. We also see that the ratios between execution durations on Raspberry Pi 4 and the regular Desktop PC vary for different benchmark configurations. Therefore, the Raspberry Pi might not be a suitable hardware for benchmark execution in every use case.

In the future, benchmarks are required that cover real world usages of application monitoring frameworks better. Therefore, the following extensions are necessary from our point of view: (1) Real world programs are not built out of single-children trees with workload only in the one leaf node. The current tree structure leads to a regular execution order consisting of a constant number of monitored method executions and one busy wait. More complex trees containing a more complex distribution of the workload would make it possible to measure more realistic overhead. In a binary tree with busy wait in every leaf, the count of executions before the leaf node is called would vary. (2) Real world monitoring overhead is also caused by monitoring of specific frameworks, e.g. Jersey, CXF and Spring. To benchmark the overhead created by the probes for these frameworks, separate benchmarked application would need to be created (or adopted for this use case) and maintained. (3) Monitoring overhead is only one measurable property of monitoring. For practical purposes, like root cause analysis for performance problems or anomaly detection, accuracy is also a main property. Accuracy could be checked by how well certain root cause analysis algorithms perform with the examined

⁷<https://open-telemetry.github.io/opentelemetry-python/benchmarks/index.html>

⁸<https://github.com/open-telemetry/opentelemetry-java-instrumentation/discussions/2104>

monitoring framework like Ahmed et al. [13].

Acknowledgments This work is funded by the German Federal Ministry of Education and Research within the project “Performance Überwachung Effizient Integriert” (PermanEnt, BMBF 01IS20032D).

References

- [1] J. Waller, Performance Benchmarking of Application Monitoring Frameworks, BoD–Books on Demand, 2015.
- [2] D. G. Reichelt, S. Kühne, W. Hasselbring, PeASS: A Tool for Identifying Performance Changes at Code Level, in: Proceedings of the 33rd ACM/IEEE ASE, ACM, 2019. (in press).
- [3] W. Hasselbring, Benchmarking as Empirical Standard in Software Engineering Research, CoRR abs/2105.00272 (2021). URL: <https://arxiv.org/abs/2105.00272>. arXiv:2105.00272.
- [4] J. Waller, N. C. Ehmke, W. Hasselbring, Including Performance Benchmarks into Continuous Integration to Enable DevOps, ACM SIGSOFT Software Engineering Notes 40 (2015) 1–4. URL: <http://eprints.uni-kiel.de/28433/>. doi:doi:10.1145/2735399.2735416.
- [5] W. Hasselbring, A. van Hoorn, Kieker: A monitoring framework for software engineering research, Software Impacts 5 (2020) 100019. doi:<https://doi.org/10.1016/j.simpa.2020.100019>.
- [6] H. Eichelberger, K. Schmid, Flexible resource monitoring of Java programs, Journal of Systems and Software 93 (2014) 163–186.
- [7] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, ACM SIGPLAN Notices 42 (2007) 57–76.
- [8] H. Knoche, H. Eichelberger, The Raspberry Pi: A Platform for Replicable Performance Benchmarks?, Softwaretechnik-Trends 37 (2017) 14–16.
- [9] P. Stefan, V. Horky, L. Bulej, P. Tuma, Unit Testing Performance in Java Projects: Are We There Yet?, in: Proceedings of ACM/SPEC ICPE 2017, ACM, 2017, pp. 401–412.
- [10] S. Huang, J. Huang, Y. Liu, L. Yi, J. Dai, HiBench: A Representative and Comprehensive Hadoop Benchmark Suite, in: Proc. ICDE Workshops, 2010, pp. 41–51.
- [11] S. Henning, W. Hasselbring, Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures, Big Data Research 25 (2021) 100209.
- [12] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Damos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, et al., MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance, IEEE Micro 40 (2020) 8–16.
- [13] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, W. Shang, Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report, in: IEEE/ACM MSR, IEEE, 2016.
- [14] H. Knoche, H. Eichelberger, Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper, in: Proceedings of the 2018 ICPE, 2018, pp. 305–316.
- [15] J. Waller, W. Hasselbring, A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring, in: ICMSEPT, Springer, 2012.