A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring

Jan Waller and Wilhelm Hasselbring
Department of Computer Science, Kiel University, Kiel, Germany
{jwa, wha}@informatik.uni-kiel.de

Abstract: Application-level monitoring frameworks, such as Kieker, provide insight into the inner workings and the dynamic behavior of software systems. However, depending on the number of monitoring probes used, these frameworks may introduce significant runtime overhead. Consequently, planning the instrumentation of continuously operating software systems requires detailed knowledge of the performance impact of each monitoring probe.

In this paper, we present our benchmark engineering approach to quantify the monitoring overhead caused by each probe under controlled and repeatable conditions. Our developed MooBench benchmark provides a basis for performance evaluations and comparisons of application-level monitoring frameworks. To evaluate its capabilities, we employ our benchmark to conduct a performance comparison of all available Kieker releases from version 0.91 to the current release 1.8.

1 Introduction

Understanding complex software systems requires insight into their internal behavior. Monitoring these systems with application-level monitoring frameworks, such as Kieker [vHWH12, vHRH$^+$09], can provide the required information at the cost of additional runtime overhead. Especially when planning the instrumentation of deployed and continuously operating software systems, detailed knowledge of the performance impact of each used monitoring probe is instrumental in reducing the actual performance overhead. To the best of our knowledge, no benchmark targeting the overhead of monitoring itself exists. Additionally, several authors [Hin88, Pri89, Sac11, FAS$^+$12, VMSK12] recognize the lack of an established benchmark engineering methodology.

We sketch such an methodology and employ it with our MooBench micro-benchmark. It can be used to measure the monitoring overhead of traces containing method executions under controlled and repeatable conditions. Furthermore, we provide a classification of the possible causes of monitoring overhead into three portions and additionally quantify each portion with our benchmark. Finally, we evaluate our benchmark with the help of a performance comparison of different releases of the Kieker framework, starting from version 0.91 (released Apr. 2009) to the current version 1.8 (released Oct. 2013).

The rest of this paper is structured as follows. In Sections 2 and 3, we introduce our notion of monitoring overhead and our benchmark to measure it. We evaluate our benchmark in Section 4. Finally, we discuss related work and draw the conclusions in Sections 5 and 6.

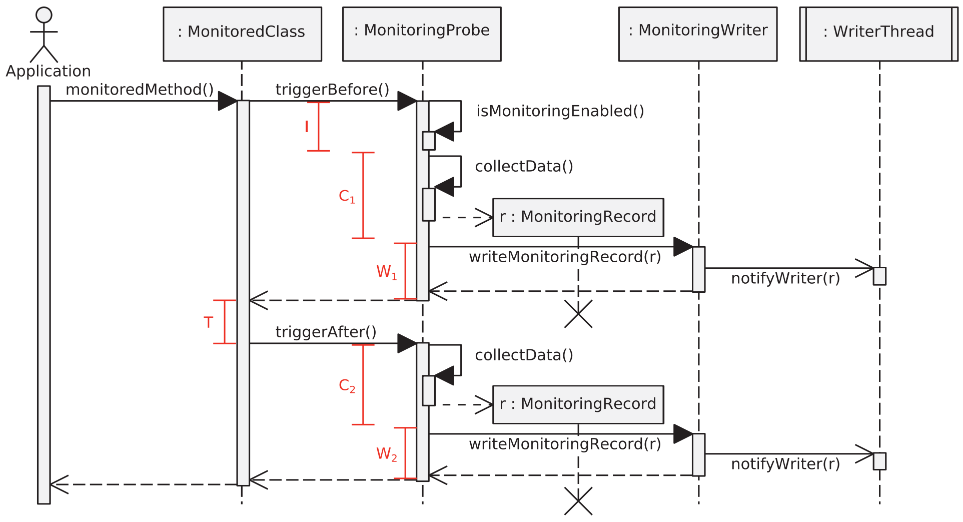

Figure 1: UML sequence diagram for method monitoring with the Kieker framework

2 Monitoring Overhead

The so-called *probe effect* is the influence a monitoring framework has on the behavior of a monitored software system [Jai91]. It is caused by the monitoring framework using resources (e. g., CPU-time) of the monitored system. In this paper, we are interested in parts of the probe effect causing increases in the response time of the system. This additional response time is the *monitoring overhead* of the monitoring framework.

A simplified UML sequence diagram for monitoring method executions with the Kieker monitoring framework using event based probes and an asynchronous writer is presented in Figure 1. A similar sequence diagram for Kieker using state based probes is presented in [WH12, vHRH$^+$09]. In addition to the control flow, the sequence diagram is annotated in red with timings of the uninstrumented original method (T) and the different portions of monitoring overhead (I, C_1, C_2, W_1, and W_2). These portion form the three common causes of monitoring overhead: (1) the instrumentation of the monitored system (I), (2) collecting data within the system ($C = C_1 + C_2$), and (3) either writing the data into a monitoring log or transferring the data to an online analysis system ($W = W_1 + W_2$). Refer to [WH12] for a detailed description of the identified portions of monitoring overhead.

In the case of asynchronous monitoring writers using buffers, as is usually the case with Kieker, we can determine an upper and lower bound on the actual monitoring overhead of writing (W). The lower bound is reached, if the *WriterThread* writes the records faster than they are produced. In the other case, the buffer reaches its maximum capacity and the asynchronous thread becomes effectively synchronized with the rest of the monitoring framework. Thus, its execution time is added to the caused runtime overhead of W.

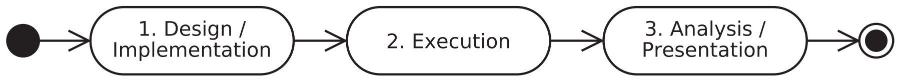

Figure 2: The three phases of our benchmark engineering methodology

3 Our Benchmark Engineering Methodology

Several authors [Hin88, Pri89, Sac11, FAS+12, VMSK12] recognize the lack of an established methodology for developing benchmarks. Furthermore, besides many authors describing specific benchmarks, only few publications, e. g., [Gra93, Hup09], are focused on such a methodology. In this section, we first sketch such a general methodology to develop, execute, and present a benchmark. Next, we present details of our concrete methodology for the MooBench benchmark for measuring the monitoring overhead of application-level monitoring frameworks.

According to [Sac11], a development methodology for benchmarks should include their development process as well as their execution and the analysis of their results. He has introduced the term *benchmark engineering* to encompass all related activities and concepts. Generally speaking, a benchmark engineering methodology can be split into three phases, as depicted in Figure 2, each with its own set of requirements:

1. The first phase is the actual *design and implementation of a benchmark*. Often this phase is specific for a class of SUTs, allowing the execution of multiple benchmark runs and subsequent comparison of results with different SUTs. A benchmark engineering methodology should provide general benchmark design requirements (representative, repeatable, fair, simple, scalable, comprehensive, portable, and robust) as well as requirements specific to the class of SUTs, e. g., possible workload characterizations and measures.

2. The second phase is the *execution of a benchmark*. Within this phase, one or more benchmark runs are performed for specific SUTs. The results of each run are usually recorded in a raw format and analyzed in the next and final phase. The methodology should provide solutions to common problems with the respective benchmark. The main requirement for this phase is the need for robust benchmark executions. This can be ensured by, e. g., repeated executions, sufficient warm-up, and an otherwise idle environment.

3. The third phase is the *analysis and presentation of benchmark results*. Here, the gathered raw performance data are statistically processed and interpreted. For the analysis, the methodology should provide guidelines for a statistically rigorous evaluation and validation of the collected data. Furthermore, it should provide guidelines for the presentation of the statistical results, e. g., present confidence intervals in addition to the mean. To ensure replicability, it should additionally provide guidelines for the description of the performed benchmark experiments.

The MooBench Benchmark for Monitoring Overhead

Our MooBench micro-benchmark has been developed to quantify the previously introduced three portions of monitoring overhead for application-level monitoring frameworks under controlled and repeatable conditions. In the following, we will detail some exemplary aspects of our benchmark engineering methodology and its three phases. The source code and further details on our benchmarks are available at the Kieker home page.[1]

Design & Implementation Phase The general architecture of our benchmark consists of two parts: First, an artificial *Monitored Application* instrumented by the monitoring framework under test. And second, there is the *Benchmark Driver* with one or more active *Benchmark Threads* accessing the *Monitored Application*.

The *Monitored Application* is a very basic implementation, consisting of a single *MonitoredClass* with a single *monitoredMethod()*. This method has a fixed, but configurable, execution time and can simulate traces with the help of recursion. It is designed to perform busy waiting, thus fully utilizing the executing processor core to simulate the work load and also to prevent elimination by JIT compiler optimizations.

The *Benchmark Driver* first initializes the benchmark system and finally collects and persists the recorded performance data. While executing the benchmark, each *Benchmark Thread* calls the *monitoredMethod()* a configurable number of times and records each measured response time.

The configuration of the total number of calls prepares our design for *robustness* by including a configurable warm-up period into runs. The method's execution time and recursion depth are used to control the number of method calls the monitoring framework will monitor per second (parts of the design requirement *representativeness*, *scalability*, *comprehensiveness*, and *portability*). For instance, we demonstrated a linear rise of monitoring overhead with additional calls per time frame by modifying the recursion depths with constant execution times in [WH12]. The remaining design requirements (*repeatability*, *fairness*, and *simpleness*) are inherent properties of the benchmark.

Each experiment consists of four independent runs, started on fresh JVM invocations, used to quantify the individual portions of the monitoring overhead:

1. Only the execution time of the calls to the *monitoredMethod()* is measured (T).
2. The *monitoredMethod()* is instrumented with a deactivated probe ($T + I$).
3. An activated probe adds data collection without writing any data ($T + I + C$).
4. The addition of an active writer represents full monitoring ($T + I + C + W$).

This way, we can incrementally measure the different portions of monitoring overhead as introduced in Section 2.

[1] `http://kieker-monitoring.net/overhead-evaluation/`

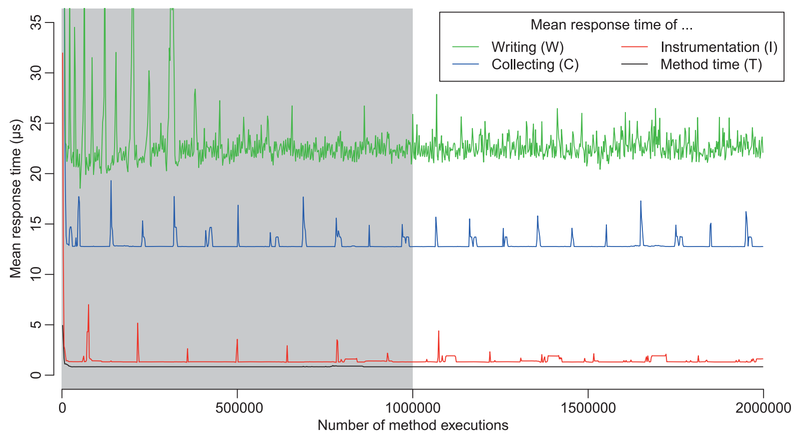

Figure 3: Exemplary time series diagram of measured timings with Kieker

Execution Phase Each benchmark experiment is designed to consist of four independent runs. Each run can be repeated multiple times on identically configured JVM instances to minimize the influence of different JIT compilation paths. The number of method executions in each run can be configured to ensure sufficient warm-up.

We analyze visualizations of time series diagrams of our experiments in combination with JIT compilation and garbage collection logs to determine sufficient warm-up periods. As a result for our evaluation, we recommend running 2 000 000 repeated method executions with Kieker and discarding the first half as warm-up. Similar studies are required for other monitoring frameworks, configurations, or hard- and software platforms. An exemplary time series diagram for Kieker 1.8 with event based probes and a binary writer is presented in Figure 3. The recommended warm-up period is shaded in gray in the figure.

Furthermore, the benchmark enforces initial garbage collection runs before the actual experiment starts, as this minimizes the impact of additional runs during the experiment. These additional garbage collections are also visible in Figure 3 by the regular spikes in the measured response times.

The final execution phase requirement is the need for an otherwise idle system. Thus, the benchmark and its components as well as the tested monitoring framework should be the only running tasks on the used hardware and software system.

Analysis & Presentation Phase For details on rigorous statistical evaluations of Java benchmarks, we refer to [GBE07]. For instance, our benchmark provides the mean and median values of the measured timings across all runs instead of reporting only best or worst runs. Additionally, the lower and upper quartile, as well as the 95% confidence interval of the mean value are included.

Finally, to ensure replicability, detailed descriptions of the experiment's configurations and environments have to be provided.

4 Performance Comparison of Released Kieker Versions

In this section, we evaluate the capabilities of our MooBench benchmark by conducting a performance comparison of the different released Kieker versions. The earliest version we investigate is version 0.91 from April 2009. It is the first version supporting different monitoring writers and thus the first version supporting all four measurement runs of our benchmark without any major modifications to the code. We compare all further released version up to the current version 1.8, that was released in October 2013. In all cases, we use the required libraries, i. e., AspectJ and Commons.Logging, in the provided versions for the respective Kieker releases. Additionally, we performed minor code modifications on the earlier versions of Kieker, such as adding a dummy writer for the third run and making the buffer of the writer thread in the fourth run blocking instead of terminating.

4.1 Experimental Setup

We conduct our performance comparison with the Oracle Java 64-bit Server VM in version 1.7.0_45 with up to 4 GiB of heap space provided to the JVM. Furthermore, we utilize an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM running Solaris 10. This hardware and software system is used exclusively for the experiments and otherwise held idle. The instrumentation of the monitored application is performed through load-time weaving using the respective AspectJ releases for the Kieker versions.

Our experiments are performed using probes producing *OperationExecutionRecords*. For measuring the overhead of writing (W), we use the asynchronous ASCII writer, producing human-readable csv files, that is available in all Kieker releases. Starting with Kieker version 1.5, we also repeat all experiments using the asynchronous binary writer, producing compact binary files, and probes producing *kieker.common.record.flow* event records. The event records are able to provide further details compared to the older records. In all cases, Kieker is configured with an asynchronous queue size of 100 000 entries and blocking in the case of insufficient capacity.

We configure the MooBench benchmark to use a single benchmark thread. Each experiment is repeated ten times on identically configured JVM instances. During each run, the benchmark thread executes the *monitoredMethod()* a total of 2 000 000 times with a configured execution time of 0 µs and a recursion depth of ten. As recommended, we use a warm-up period of 1 000 000 measurements.

In the case of state based probes, a total of ten *OperationExecutionRecords* is collected and written per measured execution of the *monitoredMethod()*. In the case of event based probes, a total of 21 *kieker.common.record.flow* records is produced and written.

To summarize our experimental setup according to the taxonomy provided by [GBE07], it can be classified as using multiple JVM invocations with multiple benchmark iterations, excluding JIT compilation time and trying to ensure that all methods are JIT-compiled before measurement, running on a single hardware platform with a single heap size, and on a single JVM implementation.

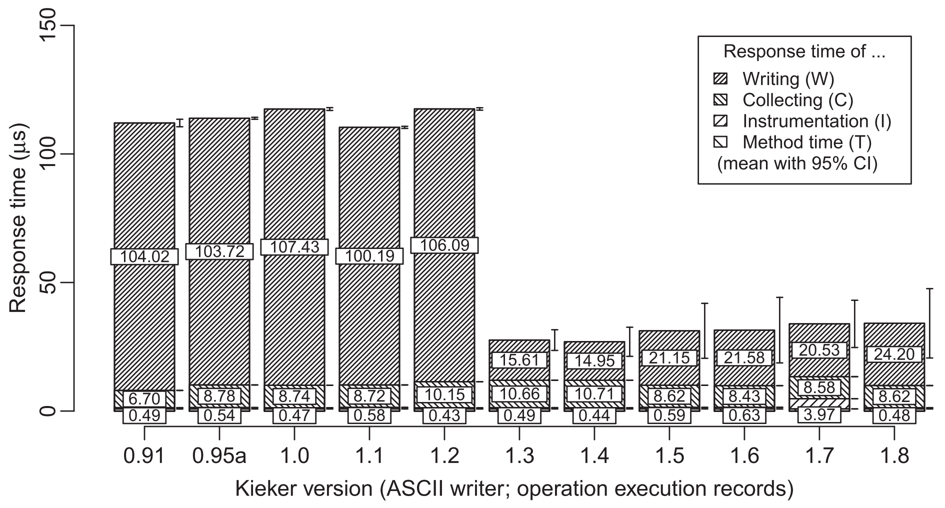

Figure 4: Performance comparison of eleven different Kieker versions

4.2 Performance Comparison: ASCII Writer & OperationExecutionRecords

Our first performance comparison restricts itself to the use of the asynchronous ASCII writer and state based probes producing *OperationExecutionRecords*. This restriction is necessary, as this is the only combination of writers and probes available in all versions. A diagram containing mean response times with 95%-confidence intervals for the three causes of monitoring overhead is presented in Figure 4. Quartiles are omitted in the diagram to reduce visual clutter.

With the exception of Kieker 1.7, the response time overhead of instrumentation (I) is constant with about 0.5 µs. Version 1.7 contains a bug related to the extended support of adaptive monitoring. This bug effectively causes Kieker to perform parts of the collecting step even if monitoring is deactivated.

The overhead of collecting monitoring data (C) stays within the same magnitude for all versions. For instance, the improvement between version 1.4 and 1.5 is probably related to the added support for immutable record types and other performance tunings.

The most interesting and most relevant part is the overhead for writing the collected monitoring data (W). The obvious change between versions 1.2 and 1.3 corresponds to a complete rewriting of the API used by the monitoring writers. This new API results in lots of executions with very low overhead, e. g., Kieker 1.8 has a median overhead of 1.3 µs. However, a small percentage of execution has extremely high response times of more than one second, as is also evident through the large span of the confidence intervals.

Additionally, we repeated this experiment with a configured method time of 200 µs. This way, we can determine a lower bound on monitoring overhead portion W. For instance, the first five versions of Kieker have a lower bound of below 4 µs of writing overhead W.

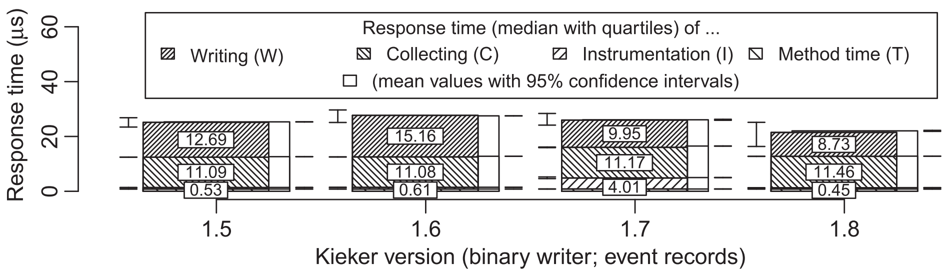

Figure 5: Performance comparison of Kieker versions employing a binary writer and event records

4.3 Performance Comparison: Binary Writer & Event-Records

For our second performance comparison of different Kieker versions, we employ the event based probes introduced with Kieker 1.5. Furthermore, we use the asynchronous binary writer, also introduced in version 1.5. The results are summarized in Figure 5.

Similar to our previous comparison, the overhead of instrumentation (I) stays constant with the exception of Kieker 1.7. Refer to the previous subsection for details.

The overhead of collecting monitoring data (C) with event based probes is higher, compared to the overhead when using state based probes (cf., Figure 4). However, this behavior is to be expected, as the event based probes produce twice the amount of monitoring records per execution. Comparing the event based probes among the different Kieker versions hints at constant overhead.

Finally, the overhead of writing using the binary writer (W) has been improved in the most recent Kieker versions. Furthermore, compared to the ASCII writer of the previous subsection, the average performance has much improved (especially considering twice the amount of records) and is much more stable, as is evident by the minimal confidence intervals and by the median being very close to the respective mean.

4.4 Further Performance Comparisons & Access to the Raw Experiment Results

For reasons of space, we provide further comparisons online for download.[2] Refer to the site `http://kieker-monitoring.net/overhead-evaluation/` for details.

In addition to our benchmark results, we also provide the pre-configured benchmark itself with all required libraries and Kieker versions.[2] Thus, repetitions and validations of our experiments are facilitated. Furthermore, we include all measurements for the described experiments and additional graphs plotting the results. Finally, the results of several additional experiments are included as well. For instance, we repeated all experiments on a different hardware platform, equipped with two AMD Opteron 2384 processors running at 2.7 GHz. Although the result graphs differ in the details, the overall trend of the results remains similar.

[2]Data sets (doi:10.5281/zenodo.7615) and the pre-configured benchmark (doi:10.5281/zenodo.7616)

5 Related Work

Although several authors [Hin88, Pri89, Sac11, FAS^{+}12, VMSK12] recognize the lack of an established benchmark engineering methodology, only few papers are concerned with establishing one. [Gra93] gives a list of four requirements for benchmarks (relevant, portable, scalable, and simple), that are also included in our list of requirements. However, the author focuses on the design phase, neglecting the execution and analysis phase. [Hup09] provides five benchmark characteristics (relevant, repeatable, fair, verifiable, and economical). The first four are also included in our set of requirements, the fifth is outside of our scope. Although mainly focusing on the design and implementation, the author also recognizes the need for a robust execution and for a validation of the results. [Sac11] provides similar requirements, but focuses on five workload requirements (representativeness, comprehensiveness, scalability, focus, and configurability). The first three are also part of our set, while the final two are included within other requirements. Additionally, we follow the author's advice on providing a benchmark engineering methodology.

Furthermore, to the best of our knowledge no other monitoring framework has been evaluated with a specialized benchmark targeting the overhead of monitoring itself. However, several publicized monitoring frameworks also provide brief performance evaluations. For instance, [App10] uses a commercial application to compare response times with and without monitoring. Similarly, [KRS99] and [ES12] use the SKaMPI or SPECjvm2008 macro-benchmarks to determine the overhead of monitoring.

6 Conclusions and Outlook

This paper presents our realization of a benchmark engineering methodology to measure the overhead of application-level monitoring. Particularly, we present our MooBench micro-benchmark to determine three identified causes of monitoring overhead. Our benchmark methodology is evaluated by creating a series of performance comparisons of eleven different released versions of the Kieker monitoring framework. In summary, our benchmark engineering methodology is capable of determining the causes of monitoring overhead within several different version of the Kieker monitoring framework. Furthermore, our evaluation demonstrates the importance Kieker places on high-throughput monitoring.

The presented division of monitoring overhead into three common causes and its measurement with the MooBench micro-benchmark has already been evaluated in the context of the Kieker framework, e. g., [WH12, vHWH12]. As future work, we will verify these portions with the help of additional lab experiments conducted on further scientific and commercial monitoring frameworks. We also plan to perform performance comparisons between Kieker and these additional monitoring frameworks. To further validate our results, we will compare them to results of macro-benchmarks and to results of using a meta-monitoring approach, i. e., monitoring a monitoring framework.

Finally, benchmarking is often considered to be a community effort [SEH03]. Consequently, we provide our benchmarks as open-source software and invite the community to use our tools to verify our results and findings.

References

[App10] AppDynamics. *AppDynamics Lite Performance Benchmark Report*, 2010.

[ES12] Holger Eichelberger and Klaus Schmid. Erhebung von Produkt-Laufzeit-Metriken: Ein Vergleich mit dem SPASS-Meter-Werkzeug. In *Proceedings of the DASMA Metrik Kongress*, pages 171–180. Shaker Verlag, 2012. In German.

[FAS$^+$12] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What it Should, Can, and Cannot Be. In *Proceedings of the 4th TPC Technology Conference on Performance Evaluation & Benchmarking*, pages 173–188. Springer, 2012.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Conference on Object-Oriented Programming Systems and Applications*, pages 57–76. ACM, 2007.

[Gra93] Jim Gray, editor. *The Benchmark Handbook: For Database and Transaction Systems*. Morgan Kaufmann, 2 edition, 1993.

[Hin88] David F. Hinnant. Accurate Unix Benchmarking: Art, Science, or Black Magic? *IEEE Micro*, 8(5):64–75, 1988.

[Hup09] Karl Huppler. The Art of Building a Good Benchmark. In *First TPC Technology Conf. on Performance Evaluation and Benchmarking*, pages 18–30. Springer, 2009.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[KRS99] Dieter Kranzlmüller, Ralf H. Reussner, and Christian Schaubschläger. Monitor overhead measurement with SKaMPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, pages 43–50. Springer, 1999.

[Pri89] Walter J. Price. A Benchmark Tutorial. *IEEE Micro*, 9(5):28–43, 1989.

[Sac11] Kai Sachs. *Performance Modeling and Benchmarking of Event-Based Systems*. PhD thesis, TU Darmstadt, Germany, 2011.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th International Conference on Software Engineering*, pages 74–83. IEEE Computer Society, 2003.

[vHRH$^+$09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical Report 0921, Department of Computer Science, Kiel University, Germany, 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proc. of the 3rd ACM/SPEC Int. Conf. on Performance Engineering*, pages 247–248. ACM, 2012.

[VMSK12] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience Benchmarking. In *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301. Springer, 2012.

[WH12] Jan Waller and Wilhelm Hasselbring. A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. In *Multicore Softw. Engineering, Performance, and Tools*, pages 42–53. Springer, 2012.