

Performance Benchmarking of Application Monitoring Frameworks

Dissertation

Jan Waller

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2014

Kiel Computer Science Series (KCSS) 2014/5 v1.0

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via <https://www.informatik.uni-kiel.de/kcss>

The author can be contacted via <http://jwaller.de>

Published by the Department of Computer Science, Kiel University

Software Engineering Group

Please cite as:

- ▷ Jan Waller. *Performance Benchmarking of Application Monitoring Frameworks*. Number 2014/5 in Kiel Computer Science Series. Department of Computer Science, 2014. Dissertation, Faculty of Engineering, Kiel University.

```
@book{thesisWaller,  
  author   = {Jan Waller},  
  title    = {Performance Benchmarking of Application Monitoring Frameworks},  
  publisher = {Department of Computer Science, Kiel University},  
  year     = {2014},  
  month    = {dec},  
  number   = {2014/5},  
  isbn     = {978-3-7357-7853-6},  
  series   = {Kiel Computer Science Series},  
  note     = {Dissertation, Faculty of Engineering, Kiel University}  
}
```

© 2014 by Jan Waller

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

Herstellung und Verlag: BoD – Books on Demand, Norderstedt

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit <http://www.informatik.uni-kiel.de/kcss> for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Examiner: Prof. Dr. Wilhelm Hasselbring
Kiel University

2. Examiner: Prof. Dr. Klaus Schmid
University of Hildesheim

Date of the oral examination (*Disputation*): December 12, 2014

Abstract

Application-level monitoring of continuously operating software systems provides insights into their dynamic behavior, helping to maintain their performance and availability during runtime. Such monitoring may cause a significant runtime overhead to the monitored system, depending on the number and location of used instrumentation probes. In order to improve a system's instrumentation and to reduce the caused monitoring overhead, it is necessary to know the performance impact of each probe.

While many monitoring frameworks are claiming to have minimal impact on the performance, these claims are often not backed up with a detailed performance evaluation determining the actual cost of monitoring. Benchmarks can be used as an effective and affordable way for these evaluations. However, no benchmark specifically targeting the overhead of monitoring itself exists. Furthermore, no established benchmark engineering methodology exists that provides guidelines for the design, execution, and analysis of benchmarks.

This thesis introduces a benchmark approach to measure the performance overhead of application-level monitoring frameworks. The core contributions of this approach are 1) a definition of common causes of monitoring overhead, 2) a general benchmark engineering methodology, 3) the MooBench micro-benchmark to measure and quantify causes of monitoring overhead, and 4) detailed performance evaluations of three different application-level monitoring frameworks. Extensive experiments demonstrate the feasibility and practicality of the approach and validate the benchmark results. The developed benchmark is available as open source software and the results of all experiments are available for download to facilitate further validation and replication of the results.

Preface

by Prof. Dr. Wilhelm Hasselbring

Benchmarks are effective vehicles to boost comparative evaluations and progress in research. Via benchmarks, it is possible to identify and distinguish those approaches that are promising and those approaches that are futile. Repeatability of scientific experiments is essential to evaluate scientific results and to extend those experiments for further investigations.

Monitoring frameworks are a domain where benchmarks are not, as yet, established for evaluating these frameworks, despite the fact that, particularly from monitoring frameworks, high efficiency and reliability is expected. Monitoring for observing a software system should not impact the observed systems with respect to its performance and reliability. However, some performance overhead caused by monitoring is inevitable. This overhead should be small and it should scale linearly with increasing monitoring coverage.

In this thesis, Jan Waller designs and evaluates a new benchmark engineering method. Besides the conceptual work, this work contains a significant experimental part with an implementation and a multifaceted evaluation. Specific contributions are extensions to the Kieker monitoring framework and the MooBench benchmark for identifying the specific causes for overhead in such monitoring frameworks.

If you are interested in designing benchmarks and in monitoring software systems, this is a recommended reading for you.

*Wilhelm Hasselbring
Kiel, December 2014*

Contents

1	Introduction	1
1.1	Motivation and Problem Statement	2
1.2	Overview of our Approach and Contributions	4
1.3	Preliminary Work	6
1.4	Document Structure	11
I	Foundations	15
2	Software and Application Performance	17
2.1	Software and Application Performance	18
2.2	Performance Measurement Terminology	20
2.3	Software Performance Engineering (SPE)	24
3	Benchmarks	31
3.1	Definitions for Benchmarks	32
3.2	Classifications of Benchmarks	34
3.3	Benchmarks in Computer Science	37
3.4	Statistical Analysis of Benchmark Results	40
4	Monitoring	43
4.1	Profiling and Monitoring	44
4.2	Application Performance Monitoring (APM)	47
4.3	Instrumentation	48
4.4	States, Events, and Traces	67
4.5	The Kieker Framework	69

II	Benchmarking Monitoring Systems	77
5	Research Methods and Questions	79
5.1	Research Methods	81
5.2	Research Questions and Research Plan	82
6	Monitoring Overhead	87
6.1	Monitoring Overhead	89
6.2	Causes of Monitoring Overhead	90
6.3	Further Monitoring Scenarios and Overhead in Resource Usage	96
7	Benchmark Engineering Methodology	99
7.1	Benchmark Engineering	101
7.2	Phase 1: Design and Implementation	103
7.3	Phase 2: Execution	114
7.4	Phase 3: Analysis and Presentation	119
7.5	Challenges Benchmarking Java Applications	127
8	Micro-Benchmarks for Monitoring	129
8.1	The MooBench Micro-Benchmark	131
8.2	Phase 1: Design and Implementation	131
8.3	Phase 2: Execution	147
8.4	Phase 3: Analysis and Presentation	150
8.5	Threats to Validity	151
9	Macro-Benchmarks for Monitoring	153
9.1	Macro-Benchmarks for Monitoring	154
9.2	Benchmarks with the Pet Store	155
9.3	The SPECjvm2008 Benchmark	157
9.4	The SPECjbb2013 Benchmark	159
10	Meta-Monitoring: Monitoring the Monitoring Framework	163
10.1	Meta-Monitoring with Kieker	165
10.2	SynchroVis: Visualizing Concurrency in 3D	166
10.3	ExplorViz: Visualizing Software Landscapes	173

III Evaluation	177
11 Evaluation of Kieker with MooBench	179
11.1 Introduction to our Evaluation	181
11.2 Performance Comparison of Kieker Versions	182
11.3 The Scalability of Monitoring Overhead	191
11.4 Experiments with Multi-core Environments	194
11.5 Performance Tunings of Kieker	199
11.6 Benchmarking the Analysis Component	210
11.7 Using MooBench in Continuous Integration	215
12 Comparing Monitoring Frameworks with MooBench	221
12.1 The inspectIT Monitoring tool	222
12.2 The SPASS-meter Monitoring tool	236
13 Evaluation of Kieker with Macro-Benchmarks	249
13.1 Experiments with the Pet Store	251
13.2 Experiments with the SPECjvm2008	253
13.3 Experiments with the SPECjbb2013	257
13.4 Summary and Further Macro-Benchmarks	262
14 Meta-Monitoring of Kieker	267
14.1 Experimental Setup	268
14.2 Analyzing our Results with Kieker	269
14.3 Visualizing our Results with SynchroVis	275
14.4 Visualizing our Results with ExplorViz	276
15 Related Work	279
15.1 Benchmark Engineering Methodology	280
15.2 Measuring Monitoring Overhead	282

Contents

IV Conclusions and Future Work	293
16 Conclusions and Future Work	295
16.1 Conclusions	296
16.2 Future Work	300
List of Figures	303
List of Tables	307
List of Listings	309
Acronyms	311
Bibliography	315

Introduction

In this chapter, we provide an introduction to this thesis. First, we state the *motivation* for our research (Section 1.1). Next, we present our *approach* and the *contributions* of this thesis (Section 1.2). In Section 1.3, we briefly summarize our *preliminary work*. Finally, we present the *structure* of this document in Section 1.4.

How is this even science, without the possibility of death?

—GLaDOS, artificially intelligent computer system from the game Portal

1. Introduction

1.1 Motivation and Problem Statement

Performance is one of the most important quality attributes and nonfunctional requirements for software systems [Smith 1990; Balsamo et al. 2004]. The necessity of evaluating the performance of software application is a well known and studied challenge [Lucas 1971]. However, even in more recent industrial surveys [Snatzke 2008], performance of software systems and measurements of performance remain one of the most important challenges in software engineering.

Software Performance Engineering (SPE) is the collection of software engineering activities related to performance [Woodside et al. 2007]. There are three different approaches to SPE: performance modeling, testing, and monitoring. Performance monitoring is a measurement-based technique that is often employed in the later stages of the software engineering process, i. e., during the operation of a system.

In practice, monitoring solutions often focus on high abstraction levels, such as monitoring a database server or a web server. Similarly, monitoring is often restricted to infrastructure information, such as CPU usage. To actually understand the behavior of modern software systems, especially continuously operating systems, it is necessary to continuously monitor their internal behavior [Jones 2010].

Application-level monitoring frameworks, such as Kieker [van Hoorn et al. 2012], inspectIT [Siegl and Bouillet 2011], or SPASS-meter [Eichelberger and Schmid 2014], can provide these required insights at the cost of additional performance overhead. This overhead is caused by the monitoring probes that instrument the monitored system, effectively executing additional monitoring code within the targeted system. Depending on the implementation of the monitoring framework, the used probes, and the workload of the monitored system, each execution of a monitored part of the software system incurs an additional performance overhead compared to the uninstrumented execution.

The challenge of determining and handling this overhead is long known research question [Plattner and Nievergelt 1981]. However, high overhead remains a common problem with many monitoring tools [Jeffery 1996]. Even more recent approaches, such as cloud monitoring, recognize monitor-

1.1. Motivation and Problem Statement

ing overhead as a major challenge [Shao et al. 2010]. Similarly, practitioners are often searching for performance measurement tools with minimal overhead [Sieg] and Bouillet 2011]. Thus, detailed knowledge of the caused monitoring overhead aids in the selection of monitoring tools. Furthermore, the required trade-off between a detailed monitoring instrumentation and overhead [Reimer 2013] can be guided by this information.

As a consequence, monitoring overhead is considered one of the most important requirements when designing monitoring frameworks [Kanstrén et al. 2011]. Especially when developing such a framework, performance evaluation becomes important [Bloch 2009]. These evaluations can help in the development of the framework and aid in the early detection of performance regressions.

While many monitoring frameworks are claiming to have minimal impact on the performance, these claims are often not backed up with a detailed performance evaluation determining the actual cost of monitoring. Nowadays, scientific publications require empirical evaluations of these performance claims [Dumke et al. 2013]. These kinds of performance evaluations are traditionally performed with benchmarks [Vokolos and Weyuker 1998]. To the best of our knowledge, no benchmark specifically targeting the overhead of monitoring itself exists.

Benchmarks are used in computer science to compare, for instance, the performance of CPUs, database management systems, or information retrieval algorithms [Sim et al. 2003]. As discussed by Tichy [1998, 2014], benchmarks are an effective and affordable way of conducting experiments in computer science. As Georges et al. [2007] state, benchmarking is at the heart of experimental computer science and research. Furthermore, benchmarks often lead to improvements in the benchmarked area [Adamson et al. 2007].

However, several authors [e. g., Hinnant 1988; Price 1989; Sachs 2011; Folkerts et al. 2012; Vieira et al. 2012] identify the lack of an established benchmark engineering methodology. A benchmark engineering methodology provides guidelines for the design, execution, and analysis of benchmarks [Sachs 2011]. Thus, such a methodology could be employed for a benchmark to measure the performance overhead of application monitoring frameworks.

1. Introduction

1.2 Overview of our Approach & Contributions

This thesis describes a benchmarking-based approach to measure the performance overhead of application-level monitoring frameworks. Besides measuring the overhead, our approach also provides an analysis of common causes of this monitoring overhead. Thus, we provide an approach to measure and quantify the three common causes of performance overhead for monitoring frameworks.

To realize this approach, we provide a general benchmark engineering methodology. Then, this methodology is applied to develop, execute, and analyze/present our MooBench micro-benchmark for measuring the monitoring overhead of application-level monitoring frameworks.

Our contributions within this thesis can be grouped into five parts:

1.2.1 Common Causes of Monitoring Overhead

We provide a common definition of monitoring overhead with respect to the response times of monitored applications. Furthermore, we analyze the control flow of event-based and state-based monitoring of method executions with application-level monitoring frameworks on the example of Kieker. This analysis leads to three causes of overhead common to most application-level monitoring frameworks: (I) instrumentation of the system under monitoring, (C) collection of monitoring data, and (W) writing or transferring the collected data.

1.2.2 A Benchmark Engineering Methodology

We describe a benchmark engineering methodology encompassing three phases: (1) the design and implementation of a benchmark, (2) the execution of the benchmark, and (3) the analysis and presentation of the benchmark results. For each of these phases, we discuss several requirements that a benchmark should adhere to. In addition, we sketch a benchmark engineering process with hints on the selection of workloads and measures. Finally, we present recommendations on the publication of benchmark results.

1.2.3 Benchmarks to Quantify Monitoring Overhead

In order to measure and quantify the previously established portions of monitoring overhead in an application-level monitoring framework, we propose the MooBench micro-benchmark. It is created in accordance with our benchmark engineering methodology, splitting the benchmark engineering process into three phases.

In addition to the use of our micro-benchmark, we propose the use of established macro-benchmarks, e. g., the SPECjvm[®]2008 or the SPECjbb[®]2013 benchmarks. These benchmarks provide additional scenarios to our own micro-benchmark. We present an adaptation of three existing macro-benchmarks to also measure this overhead. Thus, they can be used to validate the results of our micro-benchmark.

1.2.4 Evaluation of our MooBench Micro-Benchmark

We evaluate our MooBench micro-benchmark in a series of benchmark experiments of the Kieker framework to demonstrate its capabilities in several different scenarios. Additionally, we demonstrate its feasibility to benchmark different application-level monitoring frameworks with several experiments on the inspectIT and SPASS-meter monitoring tools. Finally, we validate our micro-benchmark results with additional experiments using macro-benchmarks.

1.2.5 Performance Evaluations of Monitoring Frameworks

Due to our evaluation of our benchmarking approach, we also provide detailed performance evaluations of the Kieker, inspectIT, and SPASS-meter monitoring frameworks and tools. These evaluations lead to new insights into the inner workings of these frameworks and can steer their further development, e. g., to prevent performance regressions or to further minimize their impact on the targeted systems under monitoring.

1. Introduction

1.3 Preliminary Work

This thesis builds upon previous work that has already been published in several research papers. Parts of this thesis replicate or expand these previous publications. Their contents and classification within this thesis is summarized below.

Furthermore, several student theses, papers, and projects have been co-supervised in the context of this work. These works have been conducted at the Kiel University in collaboration with further researchers and industrial partners: Florian Fittkau, Sören Frey, Wilhelm Hasselbring, André van Hoorn, Reiner Jung, Stefan Kaes (XING), Jasminka Matevska (Airbus Defence & Space), and Stefan Thiele (Thales). We present a selection of co-supervised bachelor, master, and diploma theses and their respective contexts in this work below.

Previous Publications

[van Hoorn et al. 2009b] 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

This publication contains the first application of an early version of our MooBench micro-benchmark (Chapters 8 and 11) on Kieker. Similarly, a first evaluation of causes of monitoring overhead (Section 6.2) as well as of the linear scalability (Section 11.3) of monitoring overhead is performed.

[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)*. ACM, June 2011, pages 197–200

This paper contains an additional brief evaluation of the different causes of monitoring overhead for Kieker with MooBench (Chapter 11) and

1.3. Preliminary Work

argues that the deactivation of monitoring probes provides sufficient performance for adaptive monitoring solutions.

[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)*. ACM, Apr. 2012, pages 247–248

Within this paper, we have published a short quantification of the causes of Kieker overhead with MooBench (Chapter 11) as well as the results of a macro-benchmark with the SPECjEnterprise[®]2010 to verify our micro-benchmark (Section 13.4).

[Waller and Hasselbring 2012a] J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In: *Multicore Software Engineering, Performance, and Tools (MSEPT)*. Springer, June 2012, pages 42–53

In this publication, we have investigated the influence of several different multi-core platforms on the monitoring overhead of Kieker with the help of our MooBench micro-benchmark (Section 11.4). In addition, we demonstrate the linear scalability of the measured monitoring overhead with increasing workloads (Section 11.3). It also contains the first detailed descriptions of our benchmark experiments to facilitate replications and validations of our results (Section 7.4).

[Waller et al. 2013] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring. SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE Computer Society, Sept. 2013, pages 1–4

This publications builds upon the result of two student theses [Wulf 2010; Döhring 2012]. It describes an advanced analysis and visualization tool to study concurrency in monitored applications (Section 10.2).

[Fittkau et al. 2013a] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The

1. Introduction

ExplorViz approach. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE Computer Society, Sept. 2013, pages 1–4

Similarly to the previous publication, this paper also describes an advanced analysis and visualization tool for monitoring data. In this case, the focus is on understanding large software landscapes and the interaction of application components (Section 10.3).

[Waller and Hasselbring 2013a] J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68

This paper contains a description of our evaluation of causes of monitoring overhead (Section 6.2) and of our employed benchmark engineering methodology for MooBench (Chapters 7 and 8). Furthermore, a detailed evaluation of the monitoring overhead of different Kieker versions is presented (Section 11.2).

[Fittkau et al. 2013b] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 89–98

Within this publication, we have made a performance comparison of Kieker and ExplorViz with help of MooBench. The focus of the evaluation lies on the impact of an active live analysis on the monitoring overhead (Section 11.6).

[Ehmke et al. 2013] N. C. Ehmke, J. Waller, and W. Hasselbring. Development of a concurrent and distributed analysis framework for Kieker. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 79–88

1.3. Preliminary Work

This publication builds upon a master thesis by Ehmke [2013]. It is concerned with an extension of the Kieker analysis part, including performance evaluations of the analysis components. However, these evaluations are using a set of custom micro-benchmarks outside of the direct scope of this thesis.

[Waller 2013] *J. Waller*. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013

In this publication, we have summarized the approach described in this thesis (Part II), including our research questions (Chapter 5) and our evaluation (Part III).

[Waller et al. 2014a] *J. Waller*, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69

This paper describes our causes of monitoring overhead (Section 6.2) as well as the MooBench micro-benchmark (Chapter 8) in detail. Furthermore, the benchmark is employed to steer performance tunings of Kieker (Section 11.5).

[Waller et al. 2015b] *J. Waller*, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes* (2015). Submitted, pages 1–4

This paper discusses the application of MooBench in a continuous integration environment to facilitate an early detection of performance regressions within monitoring frameworks (Section 11.7).

[Fittkau et al. 2015] F. Fittkau, S. Finke, W. Hasselbring, and *J. Waller*. Comparing trace visualizations for program comprehension through controlled experiments. In: *International Conference on Program Comprehension (ICPC 2015)*. Submitted. 2015, pages 1–11

In this paper, we describe a series of controlled experiments to evaluate the trace visualization technique employed by ExplorViz (Section 10.3).

1. Introduction

Co-Supervised Bachelor, Master, and Diploma Theses

- ▷ Wulf [2010] provides the basis for the advanced visualization of monitoring data in Waller et al. [2013] (Section 10.2).
- ▷ Kroll [2011] compares several monitoring frameworks and presents guidelines for the continuous integration of monitoring into the software development processes.
- ▷ Konarski [2012] is concerned with visualizing core utilization in multi-core systems with a focus of the assignment of Java threads to cores. Building upon our work on the influence of different multi-core environments [Waller and Hasselbring 2012a] (Section 11.4), this work enables further insights into the inner workings of these platforms.
- ▷ Döhring [2012] describes the advanced analysis and visualization techniques for Java concurrency that have been published in Waller et al. [2013] (Section 10.2).
- ▷ Beye [2013] has evaluated different communication technologies between the monitoring and analysis components for Kieker and ExplorViz. This evaluation forms the basis for the implementation of the TCP writer that is employed in many experiments of this thesis (Chapters 11 and 13).
- ▷ Ehmke [2013] is concerned with an extension of the Kieker analysis part, including a performance evaluation of the analysis components. The results of this work have been published in Ehmke et al. [2013].
- ▷ Harms [2013] describes the reverse engineering of an existing software application with Kieker. Some of his employed analysis techniques are similar to the ones used in Section 14.2.
- ▷ Frotscher [2013] is concerned with the anomaly detection in software system with the help of gathered monitoring data. Some of his detection techniques can be adapted to enhance our inclusion of micro-benchmarks into continuous integration systems to detect performance anomalies (Section 11.7).

- ▷ Zloch [2014] develops a plugin to execute micro-benchmarks within continuous integration environments. We have expanded upon this first attempt in Section 11.7.

1.4 Document Structure

The remainder of this thesis is structured into four parts:

- ▷ **Part I** contains the foundations of this work.
 - ▷ **Chapter 2** introduces the field of software and application performance and establishes a common performance measurement terminology.
 - ▷ **Chapter 3** provides a general introduction to benchmarks and their role in software engineering. Additionally, hints for the statistical analysis of benchmark results are presented.
 - ▷ **Chapter 4** details the concept of monitoring with a focus on several different instrumentation techniques that can be employed. Furthermore, the Kieker monitoring framework is introduced.
- ▷ **Part II** describes our approach for performance benchmarks of application monitoring frameworks.
 - ▷ **Chapter 5** describes our research methods and questions. In addition, it outlines our approach and summarizes the results.
 - ▷ **Chapter 6** introduces the concept of monitoring overhead, possible causes of overhead and a methodology to quantify these causes.
 - ▷ **Chapter 7** presents a benchmark engineering methodology that consists of three phases: design, execution, and analysis/presentation of benchmarks.
 - ▷ **Chapter 8** describes the MooBench micro-benchmark to quantify the previously introduced portions of monitoring overhead.
 - ▷ **Chapter 9** describes several existing macro-benchmarks that can be adapted to validate the results of measurements from the MooBench micro-benchmark.

1. Introduction

- ▷ **Chapter 10** introduces the concept of meta-monitoring as well as advanced analysis and visualization techniques for the Kieker monitoring framework.
- ▷ **Part III** contains the evaluation of our approach.
 - ▷ **Chapter 11** presents several experiments and their respective results of benchmarking experiments with MooBench and Kieker.
 - ▷ **Chapter 12** presents additional experiments with MooBench and two further application-level monitoring frameworks: inspectIT and SPASS-meter.
 - ▷ **Chapter 13** contains our validation of our micro-benchmark results with the help of three different macro-benchmarks.
 - ▷ **Chapter 14** describes meta-monitoring experiments with Kieker, including more advanced visualizations of Kieker with SynchroVis and ExplorViz.
 - ▷ **Chapter 15** discusses related work to the contributions of this thesis.
- ▷ **Part IV** draws the conclusions.
 - ▷ **Chapter 16** summarizes the evaluation of our benchmarking approach, draws the conclusions and presents an outlook on future work.

Finally, the back matter of this thesis includes lists of figures, tables, and listings, as well as the used acronyms and the bibliography.

Part I

Foundations

Software and Application Performance

In this chapter, we introduce the fundamental terms and concepts used in the field of software and application performance. We start by presenting varied common definitions of *software and application performance* (Section 2.1) and by discussing a common *performance measurement* terminology (Section 2.2). Next, we introduce the field of *Software Performance Engineering* (SPE) (Section 2.3), providing a proper context for the rest of this thesis.

*If it is fast and ugly, they will use it and curse you;
if it is slow, they will not use it.*

— David Cheriton, computer science professor & billionaire entrepreneur

2. Software and Application Performance

2.1 Software and Application Performance

Several well known organizations in the areas of computer science or standardization, such as the Institute of Electrical and Electronics Engineers (IEEE), the International Electrotechnical Commission (IEC), or the International Organization for Standardization (ISO), provide definitions of the terminology used in this thesis. Furthermore, several authors have published influential scientific articles or books concerning software and the performance of software or applications. In the following, we present a selection of some of the common definitions concerning software and application performance.

Application performance is a special case of the more general term software performance. As the name implies, it concerns the performance of an application. In contrast to the general term software, *applications* are usually more complex and require other (complex) software to operate, such as operating systems and middleware solutions. Software, as the superclass, includes simple code snippets, procedures, frameworks, data, etc., as well as complex programs, such as operating systems or applications. As such, applications and application performance are a subset of the terms software and software performance. Thus, the definitions we provide for software performance are also applicable or easily adaptable for application performance.

The IEEE systems and software engineering vocabulary [ISO/IEC/IEEE 24765] gives a general definition of the term *performance* in the context of software systems.

Definition: (Software) Performance

The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

— ISO/IEC/IEEE 24765 [2010]

Specialized standards concerning software quality can provide a more in-depth definition. The ISO/IEC 25010 [2011] standard, the successor of the well-known ISO/IEC 9126 [2001] standard, defines *software quality* as the degree to which software satisfies specific characteristics. One of the

2.1. Software and Application Performance

major characteristics influencing software quality is *performance efficiency*, subdivided into the three subcharacteristics *time behavior*, *resource utilization*, and *capacity*. Performance efficiency is defined as performance relative to the amount of resource utilization. In contrast to this general concept, the definitions of the three subcharacteristics can serve as a definition for software performance:

Definition: Software Performance

Time behavior: *degree to which the response and processing times and throughput rates of a product [...] meet requirements.*

Resource utilization: *degree to which the amounts and types of resources used by a product [...] meet requirements.*

Capacity: *degree to which the maximum limits of a product [...] parameter meet requirements.*

—ISO/IEC 25010 [2011]

This definition of software performance is comprehensive in including resource utilization and capacity explicitly into the definition, but it neglects to correlate time behavior with resource utilization and capacity. A similar definition is given by Jain [1991], while Smith and Williams [2001] provide a definition focusing on the time behavior.

Definition: Software Performance

Performance is the degree to which a software system or component meets its objectives for timeliness.

—Smith and Williams [2001, p. 4]

According to the definition of Smith and Williams [2001], *timeliness* can be measured in terms of *response time* and *throughput* with the two dimensions responsiveness and scalability. *Responsiveness* is the degree to which response time and throughput meet requirements. It corresponds to the definition of time behavior in performance efficiency. *Scalability* is the ability to fulfill these requirements with increasing demand. This definition of scalability is a correlation of the definition of time behavior with the definitions of resource utilization and capacity. A detailed definition of the used terms is given in the next section.

2. Software and Application Performance

It is important to note that any performance metric used in these definitions requires *measurements* as a basis. These measurements can concern the time behavior or the resource utilization of the application. A short definition of measurement and related terms is given in the next section. A detailed view on the field of *monitoring*, the main measurement method used in this thesis, is presented in Chapter 4.

2.2 Performance Measurement Terminology

In this section, we provide definitions of the terms used in the thesis in the context of software performance and software performance measurement. There are several contradictory definitions of terms in the context of measurement. In our definitions, we follow the terminology proposed by García et al. [2006], which is consistent to the one used by, e. g., ISO/IEC 25010 [2011]. Becker [2008] provides similar definitions based upon the older ISO/IEC 9126 [2001] standard. Further definitions are given, for example, by the OMG Architecture-Driven Modernization (ADM) Task Force [OMG 2006; OMG 2012], by the Joint Committee for Guides in Metrology (JCGM) [JCGM 200:2008], by the Standard Performance Evaluation Corporation (SPEC) [SPEC 2013a], or by Fenton and Pfleeger [1998].

A *measure*, often also called *metric* [e. g., Böhme and Freiling 2008], is a variable to which a value is assigned according to a measurement method as the result of a measurement. For instance, the response time is a measure that uses the measurement method of recording time to assign specific timings. *Measurement* is the action of measuring a system, thus applying the measures. The result of measuring is called *measurement result*, that is a set of values assigned to measures.

Refer to Fenton and Pfleeger [1998] for a comprehensive list of common software measures. An example of a typical measure when determining the performance of a software system is the *response time*. It is generally defined as the time between a user's request and a system's response [Jain 1991; Menascé et al. 2004]. In Figure 2.1, we adapted a graphical representation of a common partitioning of the response time from Ehlers [2012] that is originally based upon illustrations by Jain [1991] and Menascé et al. [2004].

2.2. Performance Measurement Terminology

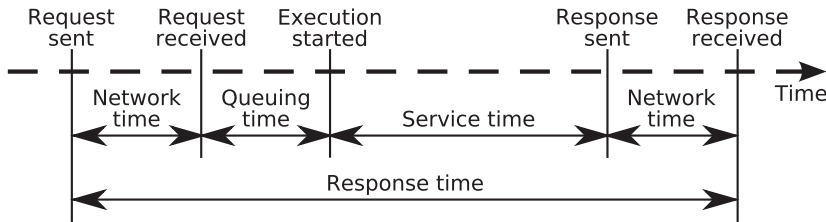


Figure 2.1. A common partitioning of the response time (adapted from Ehlers [2012], Jain [1991], and Menascé et al. [2004])

This conventional definition is best suited for distributed systems, such as web-based services. For instance, a user requests a web page in his browser from an enterprise application, such as a web shop. This corresponds to sending the request in the figure. The network time is the delay between sending the request and its reception by the server, where it is typically enqueued for later processing by the enterprise middleware system. The actual processing time of the enterprise application (service time) starts after the queuing time has passed and the enqueued request is handled. After the processing is finished, a response is sent, adding a second network time to the total response time.

This kind of end-to-end measurement is usually a good starting point in analyzing the performance of software systems. Other measurement methods, such as monitoring (see Chapter 4), provide a more in-depth view into the systems and require different definitions of the measure *response time*. When monitoring the execution of an application's method, the response time of the method is the time between starting the execution of the method and returning from it, i. e., the service time in Figure 2.1.

Another important measure for the performance of a software system is the *throughput* of the system. It is defined as the number of requests that can be processed within a fixed time interval [Jain 1991; Menascé et al. 2004; Molyneaux 2009]. For instance, in the case of a web application, it is typically defined as web page requests per second. Response time and throughput form a relationship, that is a low response time usually leads to a high throughput, while a high response time causes a low throughput.

2. Software and Application Performance

Besides the response time, the throughput of a system is depending on the *workload*. The workload of a system is the number of requests performed in a fixed time period, for example, the number of concurrent users per second that request a web page. On the one hand, a low workload, i. e., few requests to the system, results in low throughput, on the other hand, a too high workload leads to a high resource utilization and results in higher response times causing lower throughput as well.

A further important measure is the *resource utilization*. Resources in a software system can be hardware resources, such as CPU time or available memory, but can also be software resources, such as available threads in a thread pool or free database connections. With increasing workload, the resources become more and more used, until one or more resources are fully utilized and can not service further requests, thus creating a bottleneck.

The *capacity* of a system is the amount of available resources in relation to the workload, i. e., the possible workload without any adverse effect on the response time or throughput of the system.

The mentioned relations between response time, throughput, and resource utilization on the one axis and increasing workload on the other axis are illustrated in Figure 2.2. As long as the depicted workload is below the capacity of the system, the measures scale approximately linearly with the increasing workload. When the resource utilization reaches its maximum, the response time starts to increase exponentially and the throughput decreases.

As defined by Smith and Williams [2001], a system has a good *responsiveness* if it meets its performance requirements in terms of throughput and response time for a given workload, thus if its workload is below its capacity. Furthermore, a system has a good *scalability* if increasing workloads have little effect on the measures, thus if the actual workload is as far as possible lower than the capacity.

2.2. Performance Measurement Terminology

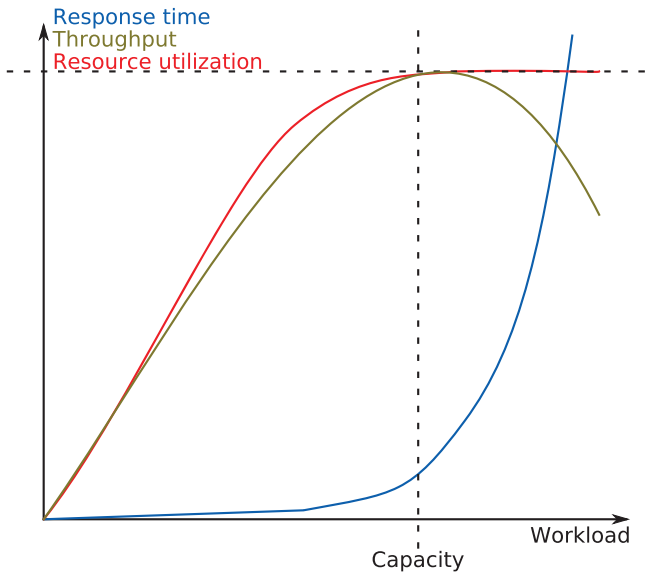


Figure 2.2. Typical behavior of response time, throughput, and resource utilization with increasing workloads (based upon Jain [1991], Smith and Williams [2001], and Menascé et al. [2004])

2. Software and Application Performance

2.3 Software Performance Engineering (SPE)

The term *Software Performance Engineering (SPE)* was coined by Smith [1990, 1994] for a model-based approach to construct software systems that meet performance requirements. Smith and Williams [2001] applied this approach to object-oriented systems. In their approach, a *performance model* is constructed during the early development of the software system to predict the expected performance of the system. A history of the field of SPE is presented by Smith [1994].

Woodside et al. [2007], on the other hand, provide a broader definition of software performance engineering, also including measurement-based approaches. *Performance measurement* complements and validates performance prediction models, but requires prototypical implementations or earlier versions of the software system.




Definition: Software Performance Engineering (SPE)

Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to meeting performance requirements.

—Woodside et al. [2007]

2.3.1 A Converged SPE Process

In accordance to the definition of SPE, Woodside et al. [2007] provide a simplified domain model for a resulting *converged SPE process* which is presented in Figure 2.3. The notation of the domain model is based upon the Object Management Group (OMG) Software & Systems Process Engineering Metamodel (SPEM) 2.0 [OMG 2008]. A description of the converged SPE process is provided after the short overview on SPEM models.

SPEM models describe activities  interacting with artifacts and outcomes. Artifacts  are tangible work products, for example documents describing performance requirements, used as inputs for activities. Outcomes  are usually non-reusable work products that are the result of activities, for example, a performance model for the current analysis is the result of the

2.3. Software Performance Engineering (SPE)

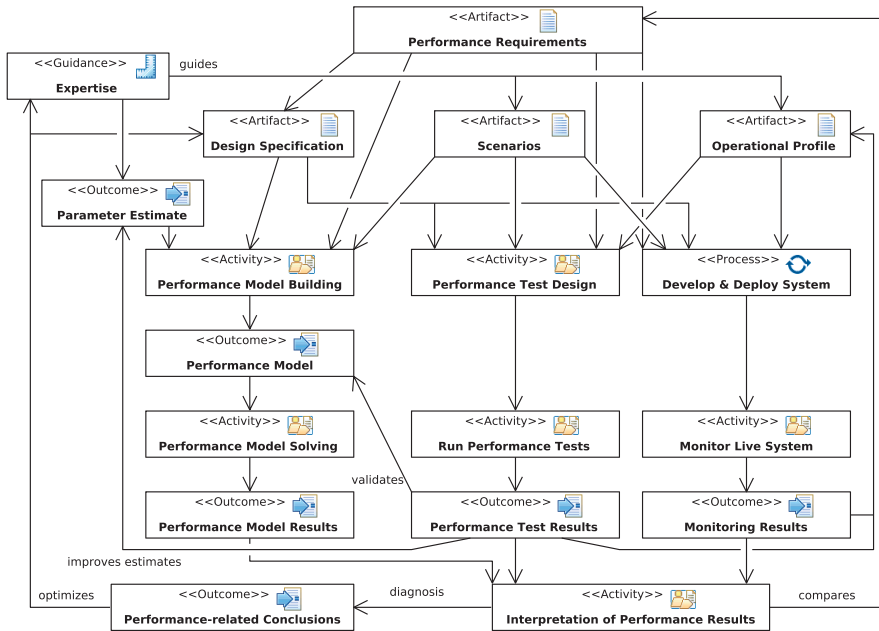







Figure 2.3. A simplified domain model for a converged SPE process (SPEM 2.0 based diagram, adapted from Woodside et al. [2007])

activity performance model building. These outcomes are further used to adapt existing artifacts. Finally, process elements are complex adaptable activities and guidance elements provide further information to work products, such as artifacts or outcomes. This short overview on the SPEM 2.0 meta-model is sufficient for the domain model given in Figure 2.3. However, a complete description is provided in the Software & Systems Process Engineering Metamodel 2.0 [OMG 2008].






Description of the Converged SPE Process The converged SPE process (Figure 2.3) of Woodside et al. [2007] combines model-based performance prediction approaches with measurement-based approaches. In the following, we provide a description of this process.







2. Software and Application Performance

In the top of the diagram, the required artifacts of each software system analysis with the SPE process are shown. The list of performance requirements  acts as the intended objective of the performance analysis and optimization. These requirements influence the design specification of the software system. The requirements in combination with this design specification  determine the performance model building, the performance test design, and the development and deployment of the system.

The final two artifacts, scenarios and operational profiles, are usually provided through the expertise  of a domain or performance specialist. Scenarios  of the software system correspond to typical use cases of the system. Thus, they influence all three paths of performance engineering. Operational profiles  include knowledge of the actual or expected usage of the system, thus prototypical implementations are usually required and they are only applicable to the measurement-based approaches.

In the middle part of the diagram, the two separate approaches to software performance engineering are depicted. On the left side of this part, a model-based performance prediction approach is presented, while on the right side, two different measurement-based approaches are presented.

In the case of the model-based performance prediction approach, the performance requirements, the design specification, and the scenarios act as the input for the activity of performance model building . The parameter estimate  is provided by the expertise of a specialist and fulfills a similar role as the operational profiles for measurement-based approaches. The outcome of this activity is the performance model  of the software system. This model is evaluated in the step performance model solving , resulting in the performance model results .

The measurements-based approaches distinguish between performance tests and monitoring. Performance tests, such as profiling the system, are usually performed during the development time of a software project, while monitoring is usually performed on the live system under realistic conditions. As such, the performance test design  can work with prototypical or partial implementations, while monitoring requires the development and deployment of the whole system (develop & deploy system ). The next activities are to run the performance tests  or to monitor the live system  resulting in performance test results  or monitoring results .

2.3. Software Performance Engineering (SPE)

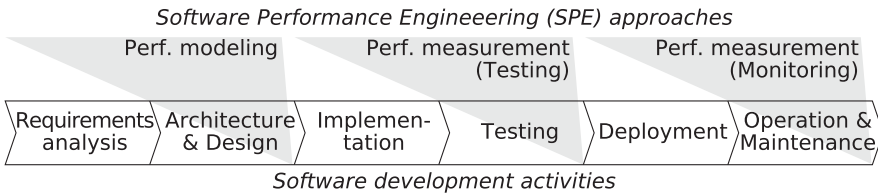




Figure 2.4. Integration of SPE in the software development process [Ehlers 2012]

Finally, the performance results of the three paths can help to improve parameter estimates, to fine-tune operational profiles, or to validate the constructed performance model. The interpretation of performance results  leads to performance-related conclusions  which can be used to optimize the design specification and to improve the expertise. Furthermore, the interpretation can be compared with the performance requirements, thus determining if further tuning is necessary or if the results are already satisfactory. Thus, besides the mentioned combination of model-based performance prediction approaches with measurement-based approaches, the converged SPE process also includes a performance tuning cycle.

2.3.2 Performance Modeling, Testing, and Monitoring

In the previous section, we introduced the modeling-based Software Performance Engineering approach, *performance modeling*, as well as the two different measurement-based SPE approaches, *performance testing* and *performance monitoring*. In this section, we provide further details on these three approaches.

The converged SPE process represents the viewpoint of a software performance engineer. In contrast, Figure 2.4 presents the three SPE approaches from the viewpoint of a more conventional software engineer. Thus, they are included into the phases of a typical sequential software development process, such as the waterfall model. As mentioned in the converged SPE process, performance modeling is usually performed during the early phases of software development. Performance measurement approaches, such as testing and monitoring, usually require at least a prototype. Perfor-

2. Software and Application Performance

mance tests, such as profiling of the system, are often started in the early implementation steps, while performance monitoring is usually performed on the live system after its deployment.

A similar classification of performance evaluation techniques into performance modeling and performance measurement is done by John [2005b]. Performance modeling, consisting of simulation and analytic modeling, is conducted at the beginning of the development process. Performance measurements are made in the later stages of the development process to validate the existing models.

Similar to its integration into the converged SPE process, *performance tuning* can be integrated into software development processes. It is usually done by adding one or more checks for acceptable performance into the model, combined with back branches into earlier stages of the development process if the performance criteria are not met [e. g., Wilson and Kesselman 2000; Hunt and John 2011].

The model-based performance prediction approach of the converged SPE process is a simplification and generalization of the Software Performance Engineering process introduced by Smith [1990]. The focus of this thesis is on measurement-based approaches, so we refer to Smith and Williams [2001] for further details on the activities and artifacts of the original SPE process for object-oriented systems.

A survey of further model-based performance prediction approaches can be found in Balsamo et al. [2004]. A more detailed description is provided by Balsamo et al. [2003]. Surveys and an overview of model-based and measurement-based approaches for component-based software systems are presented by Becker et al. [2004, 2009] and Koziolok [2010]. Woodside et al. [2007] provide a general overview on different SPE approaches. A comparison of the general advantages and disadvantages between different model-based and measurement-based performance prediction approaches is given by Jain [1991] and Becker et al. [2009]. An example of a combination of model-based techniques with measurement-based ones is presented by Merriam et al. [2013].

In contrast to model-based performance prediction, *performance testing* is conducted during the development of the system. In order to conduct any tests, at least an initial, not necessarily complete, implementation of the

2.3. Software Performance Engineering (SPE)

system has to be present. Thus, performance testing is part of the dynamic analysis of software systems [Ball 1999; Cornelissen et al. 2009; Marek 2014]. An introduction to the field of performance testing of applications is provided by Barber [2004]. More general descriptions of approaches to performance testing are given by Vokolos and Weyuker [1998], Weyuker and Vokolos [2000], Denaro et al. [2004], Molyneaux [2009], or Westermann et al. [2010].

Performance testing tools can be separated into two categories: *workload generation tools* and *profiling tools* [Parsons 2007; Denaro et al. 2004]. The scenarios and operational profiles mentioned in the converged SPE process act as the input for the workload generation tools and define the desired output of these tools. As such, these tools, also named *load drivers* [Sabetta and Koziolok 2008] or *load generators* [SPEC 2013a], provide a synthetic workload for the System Under Test (SUT). In the case of a web-based system, user requests are automatically created and executed according to the scenarios and operational profiles [Menascé et al. 1999; Menascé 2002]. A commonly used open source workload generation tool is JMeter [Apache JMeter]. Further details on workload generation with JMeter and its extension Markov4JMeter are given by van Hoorn et al. [2008].

Profiling tools are used to collect events and runtime information of the system under test. Typically collected information include heap usage, object lifetime, wasted time, and time-spent [Pearce et al. 2007]. Generally speaking, there are two different approaches to profiling a software system: exact profiling and sampling-based profiling [Parsons 2007; Pearce et al. 2007]. Exact profiling is usually very precise and collects all events during the execution of the system, thus causing a high overhead (see Chapter 6). On the other hand, sampling-based profiling usually only records a subset of the available information and uses statistical methods to infer details on the whole system. Further details and collections of profiling tools are provided by Pearce et al. [2007], Parsons [2007], or Hunt and John [2011].

Software performance tests are often classified into three types: load tests, stress tests, and endurance tests [e.g., Subraya and Subrahmanya 2000]. In *load tests*, the normal or expected workload of a system under test is used. Thus it provides hints on the normal or day to day behavior of the system. In *stress tests*, a greater than normal workload is employed.

2. Software and Application Performance

These tests aim at finding the upper capacity limits of the system under test. In *endurance tests*, either a load or stress workload is employed for a long duration. Thus, it is possible to detect potential leaks and to evaluate the performance for long running systems.

Benchmarks are a special case of performance tests. Similar to performance test, they usually include a specified set of workloads. This workload is used to measure the performance of specific aspects of the system under test. Contrary to performance tests, the focus of benchmarks is not on finding and tuning performance problems, but on comparing different systems under standardized conditions. We discuss benchmarks in greater detail in Chapter 3.

As the name implies, *monitoring of live systems* is performed during the runtime of the system. Instead of synthetic workload generation, the real workload of the system is used to gather performance relevant data. And contrary to profiling tools, which can cause high runtime and memory usage overhead, monitoring tools have to minimize their respective overhead. Chapter 4 provides further details on the topic of monitoring.

Benchmarks

In this chapter, we provide a general introduction to the field of benchmarks in software engineering. In the first section (Section 3.1), we give common *definitions of benchmarking terminology*. Next, we present a *classification of the two common benchmark types*, micro- and macro-benchmarks (Section 3.2). In Section 3.3, we give an overview on the importance and role of *benchmarks in software engineering*. The final section of this chapter (Section 3.4) is concerned with the *statistical analysis of benchmark results*.

The reputation of current “benchmarking” claims regarding system performance is on par with the promises made by politicians during elections.

—Kaivalya Dixit, SPEC president, The Benchmark Handbook, 1993

3. Benchmarks

3.1 Definitions for Benchmarks

Benchmarks are used to compare different platforms, tools, or techniques in experiments. They define standardized measurements to provide repeatable, objective, and comparable results. In computer science, benchmarks are used to compare, for instance, the performance of CPUs, database management systems, or information retrieval algorithms [Sim et al. 2003]. Aside from performance evaluations, benchmarks in computer science can also employ other measures, for example the number of false positives or negatives in a detection algorithm.

Accordingly, several organizations and standards provide different definitions of the general term benchmark. In the following, we present multiple common definitions and conclude with our own definition:

The ISO/IEC 25010 [2011] standard for software quality provides a very general definition of the term *benchmark*.

Definition: Benchmark

A standard against which results can be measured or assessed.

— ISO/IEC 25010 [2011]

Similar definitions are provided by the IEEE systems and software engineering vocabulary [ISO/IEC/IEEE 24765]:

Definition: Benchmark

1. *A standard against which measurements or comparisons can be made.*
2. *A procedure, problem, or test that can be used to compare systems or components to each other or to a standard.*

— ISO/IEC/IEEE 24765 [2010]

3.1. Definitions for Benchmarks

A definition more focused on the performance of computer systems is provided by the glossary of the Standard Performance Evaluation Corporation (SPEC) [SPEC 2013a]:

Definition: Benchmark

A benchmark is a test, or set of tests, designed to compare the performance of one computer system against the performance of others.

—SPEC [2013a]

We employ a similar, narrow definition, focusing on software performance measurements. Further similar definitions are provided by Joslin [1965], Hinnant [1988], Price [1989], Jain [1991], Vokolos and Weyuker [1998], Kähkipuro et al. [1999], Smith and Williams [2001], Menascé [2002], and Vieira et al. [2012]. So, *benchmarking* is the process of measuring the performance of a system with the help of a benchmark.

Definition: Benchmark

A benchmark is a standardized software system (benchmark system) used to measure the performance of a platform or another system (platform/system under test (SUT)) interacting with it.

The *benchmark system* can be of any size or complexity, from a single addition instruction to a whole enterprise system. An important part of the definition of this benchmark system is the definition of the measures and measurements used to determine the performance. Furthermore, especially for complex benchmark systems, a *defined usage profile* (operational profile and scenario in Figure 2.3) of the software system is required to produce repeatable results. The *platform/system under test (SUT)* interacting with the benchmark system could be a hardware or software platform, a software component, or even a single operation. Similar to the benchmark system, its complexity can be anything from a single addition instruction to a whole enterprise system. Additional details on developing benchmarks and the required design decisions are provided in Chapter 7.

As mentioned in Section 2.3.2, benchmarks are a special case of performance tests. As such, we can adopt a classification for different kinds of testing to benchmarks. Utting and Legéard [2007] employ three dimensions

3. Benchmarks

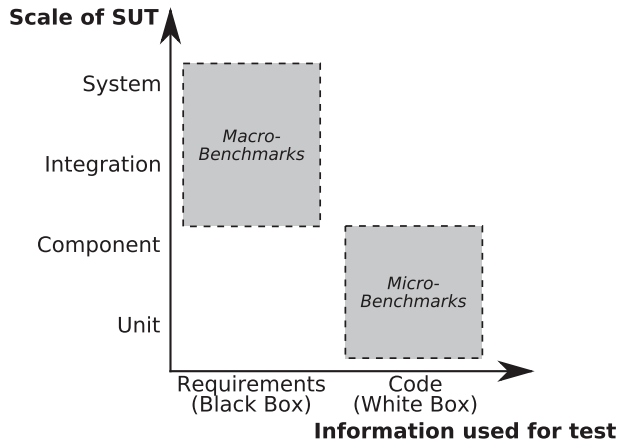


Figure 3.1. Different kinds of performance tests and benchmarks (based upon Utting and Legeard [2007])

to classify testing: the characteristic being tested, the scale of the SUT, and the information used to design the test. In the case of benchmarking or performance tests, the characteristic is fixed at performance. So, we can focus on the remaining two axes in Figure 3.1. The scale of the SUT in Figure 3.1 corresponds to the size or complexity of the benchmark system. Similarly, the information used to design the tests corresponds to the information used to design the benchmarks. Thus, we can categorize benchmarks by their complexity, as well as by their design intend.

3.2 Classifications of Benchmarks

Most benchmarks can be classified into two categories: micro- and macro-benchmarks [Seltzer et al. 1999; Wilson and Kesselman 2000]. *Micro-benchmarks* are designed to evaluate the performance of a very specific, usually small part of a software system. *Macro-benchmarks* are large and often complex benchmark system, designed to simulate a real system or a part thereof.

3.2.1 Micro-Benchmarks

Most *micro-benchmarks*, also called *synthetic* [e. g., Lucas 1971; Weicker 1990; John 2005a], *narrow spectrum* [e. g., Saavedra-Barrera et al. 1989], *kernel* [e. g., Cybenko et al. 1990], or *simple benchmarks* [e. g., Bulej et al. 2005; Kalibera 2006], are written to compare basic concepts, such as a single operation, or narrow aspects of a larger system. Typical examples of these benchmarks are the comparison of different sorting algorithms or of the performance of an operation on different hardware platforms. Depending on the size of the benchmarks, further subcategories of micro-benchmarks are used [e. g., Dongarra et al. 1987; Lilja 2000; Bull et al. 2000].

Micro-Benchmarks usually correspond to the lower right corner of the scale and information axes in Figure 3.1. They are focussing on a small and specific part of a system, i. e., a single unit such as an operation or a class. Additionally, they often use a white-box approach in their design, that is they are designed with the actual system under test in mind.

In theory, micro-benchmarks excel at their given task of comparing well-defined, small properties. But it is often hard to find these small, relevant task-samples. Thus, lots of micro-benchmarks have only a very limited real-life applicability.

On the other hand, their basic concepts make most micro-benchmarks easy to automate. They can be included in continuous integration setups to automatically record performance improvements and regressions [Bulej et al. 2005; Kalibera 2006; Weiss et al. 2013; Reichelt and Braubach 2014; Zloch 2014] (see also Section 11.7). In continuous integration setups, code changes are tested for incompatibilities with other code changes and for new bugs introduced (almost) immediately. Usually, these tests are performed automatically on an integration system which notifies the developers of any problems. Refer to Fowler [2006], Duvall et al. [2007], and Meyer [2014] for further information on continuous integration.

The major drawback and danger of micro-benchmarks is a result of their simpleness. They often neglect other influencing factors and focus only on a single aspect of the complex interactions of a system. This can lead to false conclusions and harmful performance tunings, improving the performance for the micro-benchmark, but decreasing it in other scenarios [Mogul 1992].

3. Benchmarks

Details on typical challenges of micro-benchmarks are given in Chapter 7. An in-depth example of a micro-benchmark for monitoring systems is presented in Chapter 8.

3.2.2 Macro-Benchmarks

Macro-benchmarks, also called *natural* [e. g., Hinnant 1988], *application* [e. g., Cybenko et al. 1990; Lilja 2000; Bull et al. 2000], or *complex benchmarks* [e. g., Bulej et al. 2005; Kalibera 2006], are supposed to represent a relevant task-sample including other influencing factors. Thus, they often encompass a large portion of the actually possible tasks. They are used to overcome the shortcomings of micro-benchmarks [Cybenko et al. 1990].

Macro-benchmarks usually correspond to the upper left corner of the scale and information axes in Figure 3.1. They are representing large parts of systems or even complete systems. Furthermore, they usually take a black-box approach, that is they are not designed with a specific system under test in mind, but rather with a more general requirements specification.

In the best case, the macro-benchmark is the actual system under test with a realistic task-sample [Weicker 1990], for example the macro-benchmark of an online shop could be a new instance of the shop system, deployed on comparable hard- and software, and used with realistic task-samples. In the case of a performance benchmark, the workload produced by the task-samples could be higher than the expected workload to benchmark for performance bottlenecks.

In most cases, macro-benchmarks are representations of real systems. Besides varying the hard- and software running the benchmark, the macro-benchmark itself can be an abstraction or reduction of the real system. For example, instead of using the real application, the benchmark consists of a generalized, abstract online shop, simulating a real one. An example of such a macro-benchmark is the SPECjbb[®]2013 application benchmark, described in greater detail in Section 9.4.

Especially when the system under test is independent from the benchmark system, an abstract benchmark, simulating typical task-samples, is common. In the case of the SPECjbb[®]2013 application benchmark, the system under test is usually a combination of a hardware platform with a

3.3. Benchmarks in Computer Science

specific application server, while the benchmark system is an online shop, simulating typical task-samples for application servers.

A similar, often used benchmark system is the Pet Store in one of its many implementations. It was originally developed by Sun Microsystems as part of the J2EE Blueprints program [Java Pet Store] and later adapted to different platforms and technologies. Although it was originally not intended to be a performance benchmark, several versions were used to compare products or technologies [Almaer 2002]. A recent implementation of the Pet Store that is used for example by Kieker [2014], is provided by the MyBatis Project [MyBatis JPStore]. Further details on the Pet Store are given in Section 9.2.

Besides finding a good tradeoff between a realistic benchmark system and the added complexity of such a system, the determination of the task samples is usually hard. The domain knowledge of experts is an invaluable asset in these tasks. Furthermore, monitoring of real systems during the operation is a good method to get valid usage samples, as seen in Figure 2.3 for operational profiles and scenarios.

Apart from the higher complexity, and thus usually higher cost of macro-benchmarks, it is often harder to pinpoint the actual cause of (performance) problems detected with these benchmarks, compared to specialized micro-benchmarks [Saavedra et al. 1993]. Due to the common inclusion of lots of influencing factors, the relevant ones are harder to determine. Additionally, depending on the quality of the task-samples, detected problems or the lack thereof may or may not be indicative. But, contrary to micro-benchmarks, this risk is less pronounced.

3.3 Benchmarks in Computer Science

Several authors stress the importance of benchmarks for the field of computer science:

The importance of benchmarks has been recognized for several years [Hinnant 1988]. They are an effective and affordable way of conducting experiments in computer science [Tichy 1998; 2014]. Using well-known benchmarks that are accepted as representatives of important applications, in

3. Benchmarks

experimental designs suggests a general applicability of the results [Berube and Amaral 2007]. The successful evaluation of ideas with these kinds of benchmarks often plays an important role in the acceptance of said ideas [Adamson et al. 2007].

Thus, benchmarks are a central part of scientific investigations [Blackburn et al. 2012]. They are able to shape the field of computer science [Patterson 2002] and steer research and product development into new directions [Blackburn et al. 2006; Adamson et al. 2007]. Hence, the use of benchmarks is frequently accompanied by rapid technological progress [Sim et al. 2003]. Especially the employment of performance benchmarks has contributed to improve generations of systems [Vieira et al. 2012]. In summary, benchmarking is at the heart of experimental computer science and research [Georges et al. 2007]. But benchmarking is also an important activity at the business level [Menascé 2002].

According to Sachs [2011], the development of benchmarks has turned into a complex team effort with different goals and challenges compared to traditional software. Tichy [1998, 2014] states that constructing benchmarks is hard work, best shared within a community. Furthermore, benchmarks need to evolve from narrowly targeted tests to broader, generalizable tests in order to prevent overfitting for a specific goal. Carzaniga and Wolf [2002] also stress the importance of benchmark design as a community activity, resulting in wider acceptance and adoption of the final benchmark. Sim et al. [2003] further pursue the community idea and state that benchmarks must always be developed and used in the community, rather than by a single researcher. Good benchmarks originate from a combination of scientific discovery and consensus in the community, both equally important. The SPEC Research Group (SPEC RG) is an example of such a community that is regularly involved in the development of benchmarks.

We argue that it is sufficient to start the development process of a new benchmark with a small group of researches as an offer to a larger scientific community. This first (or proto-) benchmark can act as a template to further the discussion of the topic and to initialize the consensus process. Further details on developing benchmarks and the required design decisions are provided in Chapter 7. A resulting proto-benchmark for monitoring systems is presented in Chapter 8.

3.3. Benchmarks in Computer Science

However, most popular benchmarks are provided by larger consortiums or research communities. Some of the best-known groups are the already mentioned Standard Performance Evaluation Corporation (SPEC),¹ the Transaction Processing Performance Council (TPC),² and the Defense Advanced Research Projects Agency (DARPA).³ The SPEC is a consortium with several committees creating a variety of benchmarks. Their main focus is on benchmarks comparing different hardware systems or software environments. The TPC defines transaction processing and database benchmarks while the DARPA provides a multitude of different benchmarks, for example image processing or speech recognition benchmarks. Several of these benchmarks are described in *The Benchmark Handbook* by Gray [1993]. Short overviews on important past benchmarks are given by Price [1989], Weicker [1990], Lilja [2000], John [2005a], and Sabetta and Koziolok [2008]. Additionally, two popular benchmarks (SPECjvm[®]2008 and SPECjbb[®]2013) are detailed in Chapter 9.

As mentioned before, benchmarks are part of the measurement-based approaches in the field of Software Performance Engineering (see Figure 2.3). Besides this categorization in the field of SPE, benchmarks are also an important part of the field of empirical research in software engineering.

Two common empirical research and evaluation methods in software engineering are *formal experiments* and *case studies* [Pfleeger 1995]. Experiments require a high level of control over all variables affecting the outcome but also provide reproducibility and easy comparability. Case studies, on the other hand, require less control but are also seldom replicable and hard to generalize. Benchmarks are in-between formal experiments and case studies, containing elements of both empirical methods [Sim et al. 2003]. Similar to experiments, benchmarks aim for a high control of the influencing variables and for reproducibility. On the other hand, the actual platform, tool, or technique evaluated by the benchmark can be highly variable, thus each benchmark run is similar to a case study.

A general classification of the research methods used in this thesis, including benchmarks, is given in Section 5.1.

¹<http://www.spec.org/>

²<http://www.tpc.org/>

³<http://www.darpa.mil/>

3. Benchmarks

3.4 Statistical Analysis of Benchmark Results

In this section, we present some general statistical definitions needed to analyze the results of benchmarks. A more detailed explanation of this topic can be found in any textbook on probability and statistics [e. g., Montgomery and Runger 2010].

The presented methods and techniques are common within the field of Software Performance Engineering (SPE). Hence, several books concerned with measurements of software systems include chapters on the required statistical methods [e. g., Jain 1991; Fenton and Pfleeger 1998; Lilja 2000].

Similarly, several scientific papers concerned with software measurement or benchmarking either provide sections detailing these methods [e. g., Kalibera et al. 2005; Kalibera and Jones 2013] or even focus on the statistical analysis of data [e. g., Fleming and Wallace 1986; Lilja and Yi 2005; Maxwell 2006; Georges et al. 2007; Iqbal and John 2010; de Oliveira et al. 2013]. Here, we follow the guidelines of the very influential and good paper by Georges et al. [2007].

Most benchmarks produce a series of individual measurement results, at least as an implicit intermediary result. To present this series in a compact fashion, simple statistical methods, such as best, worst, or average performance, are employed [Georges et al. 2007]. These simple methods can lead to false conclusions and wrong indications. In accordance with the authors, we advocate a more statistically rigorous approach to data analysis, by including confidence intervals for mean values and additionally providing median values with quartiles.

The whole range of possible measurement results of a benchmark system is usually not limited and thus impossible to observe. Accordingly, the results of the benchmark have to be based upon a *sample* of values. This sample is taken from all possible results of measurements that are modeled by a probability distribution, such as the normal distribution or the Student's t-distribution. The sample consisting of the actual measurement results is called (x_1, x_2, \dots, x_n) with $n \in \mathbb{N}$ as the number of performed measurements.

3.4. Statistical Analysis of Benchmark Results

3.4.1 Mean and Confidence Intervals

One of the most common statistical methods applied to such a sample is the determination of the mean \bar{x} . The *mean* \bar{x} is defined as the average value of the results of all measurements. Thus, it is calculated as follows:

$$\bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n} = \frac{\sum_{i=1}^n x_i}{n} \quad (\text{mean } \bar{x})$$

The *Confidence Interval* (CI) for \bar{x} is the interval $[l, u]$ around \bar{x} that contains the true value of the mean for the whole set of possible measurements with a certain probability, typically 90%, 95% or 99%. So, the 95% CI of \bar{x} has a confidence of 95% of containing the actual mean.

We can determine l and u with the help of \bar{x} , the standard deviation s , and constant c . The standard deviation s is calculated as follows:

$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1} \quad (\text{standard deviation } s)$$

According to the *central limit theorem*, for large samples with $n \geq 30$ we can use $c = z_{1-\alpha/2}$ which is defined for normal distributions. Its value is best obtained from precalculated tables. In the case of the 95% CI: $\alpha = 0.05$, $z_{1-\alpha/2} = z_{0.975} = 1.96$. For smaller samples with $n < 30$, the Student's t-distribution provides a better approximation with $c = t_{1-\alpha/2, n-1}$. It is best obtained from precalculated tables, too. Thus, the confidence interval is calculated as:

$$l = \bar{x} - c \frac{s}{\sqrt{n}} \quad u = \bar{x} + c \frac{s}{\sqrt{n}} \quad (\text{CI } [l, u])$$

The confidence interval allows for a simple comparison of measurement results. If the intervals of two means overlap, these two means cannot be assumed to differ but might be the result of random effects. If the intervals do not overlap, we can assume with a certain confidence that the two means differ significantly. Advanced techniques, such as ANOVA, can be employed to compare more than two alternatives [Georges et al. 2007].

3. Benchmarks

3.4.2 Median and Quantiles

The second, common statistical technique that is used to compare measurement results is the median and the quantiles. Assuming an ascending order of the measurement results (x_1, x_2, \dots, x_n) , the *median* \tilde{x} is defined as the middle value. If such a middle value does not exist (n is even), the median is defined as the average of the two middle values. Thus, it is defined as (depending on n being odd or even):

$$\tilde{x} = x_{\frac{n+1}{2}} \quad \text{or} \quad \tilde{x} = \frac{(x_{\frac{n}{2}} + x_{\frac{n}{2}+1})}{2} \quad (\text{median } \tilde{x})$$

The median is usually accompanied by the quantiles (often also referred to as quartiles). The median corresponds to the *50%-quantile*, i. e., 50% of the measurement results are below \tilde{x} . The other two common quantiles are the *25%-* and the *75%-quantile*. Similarly, 25% or 75% of the results are below this values. Both are calculated by determining the median of the lower or upper half of measurement results, respectively. Thus, these three values split the results into four sections (hence the name quartiles).

The final two important quantiles are the *0%-quantile*, commonly known as *minimum*, and the *100% quantile*, commonly known as *maximum*.

3.4.3 Combination of Methods

The combination of the average (including its CI) with the median and its quantiles allows for further analyses of the measurement results. For instance, in the case of the mean few high values in combination with many small ones can have a huge impact on the calculated mean. This impact of only a few outliers is usually visible with the help of the median. In this case, the median would be small, compared to the mean. Similarly, a high distance between the 25%- and 75%-quantile hints at unsteady results that can be hidden by simply presenting the mean. On the other hand, the mean with its confidence interval can enable simpler comparisons of results.

In summary, we recommend a statistically rigorous evaluation of benchmark results, as sketched above and detailed by Georges et al. [2007]. Furthermore, we propose to include the mean with its CI as well as the median and the quantiles into presentations of measurement results.

Monitoring

In this chapter, we provide an overview on monitoring software systems. First, we introduce the concept of *monitoring software systems* and differentiate between *profiling tools* and *monitoring tools* (Section 4.1). Next, we present the related marketing term *Application Performance Monitoring* (APM) that is commonly used in the industry (Section 4.2). The third section (Section 4.3) contains an overview of different *instrumentation techniques* to gather information on a monitored system. Section 4.4 introduces the monitoring concepts of *states, events, and traces*. Finally, we provide an introduction to the *Kieker monitoring framework* (Section 4.5).

Anything that is measured and watched, improves.

— Bob Parsons, founder of GoDaddy.com

4. Monitoring

4.1 Profiling and Monitoring

The IEEE software engineering vocabulary [ISO/IEC/IEEE 24765] uses monitoring in its literal meaning, but provides a definition of the term monitor, i. e., the tool or device used when monitoring.

Definition: Monitor

A software tool or hardware device that operates concurrently with a system or component and supervises, records, analyzes, or verifies the operation of the system or component.

— ISO/IEC/IEEE 24765 [2010]

In this definition, a distinction is made between hardware and software monitor on two different levels. First, the monitor itself can either be a software tool, a hardware device, or a hybrid combination of both. Second, the monitored system or component, also named System Under Monitoring (SUM) or System Under Test (SUT), can either be a software system or a hardware device. The focus of this thesis lies on monitoring software systems with software monitors, so we include the specialized definition of a software monitor as well.

Definition: Software Monitor

A software tool that executes concurrently with another program and provides detailed information about the execution of the other program.

— ISO/IEC/IEEE 24765 [2010]

An important part of this definition is the usage of the word concurrently. That is, the monitor is always used while the monitored system is running itself. This corresponds to the definition of *dynamic analysis*, i. e., the analysis of data gathered from running programs [Plattner and Nievergelt 1981; Ball 1999; Cornelissen et al. 2009]. The opposite approach is *static analysis*, i. e., all observations are made without executing the software. In the case of Software Performance Engineering, static analysis is used to predict performance with models, while dynamic analysis is used to measure the performance of systems [Smith and Williams 2001]. Often, a combination of both approaches can provide better results.

4.1. Profiling and Monitoring

Table 4.1. Typical differences between profiling tools and monitoring tools

	Profiling	Monitoring
Employed during	development	operation
Workload	generated	real
Data gathered	detailed	specific
Acceptable overhead	high	low

In Section 2.3 and Figure 2.3, Woodside et al. [2007] have differentiated between performance profiling (testing) and performance monitoring. The given definition of a software monitor is applicable to both cases. In the literature, both terms are sometimes used interchangeably [e. g., Viswanathan and Liang 2000; Sabetta and Koziolok 2008]. However, a distinction between both terms is common [e. g., John 2005b; Bulej 2007; Hunt and John 2011; Isaacs et al. 2014]. A further distinction is sometimes made on the employed data gathering technique (see Section 4.3). For instance, Mohror and Karavanic [2007, 2012] or Isaacs et al. [2014] refer to profiling when a sampling technique is employed and to monitoring or tracing when detailed information is collected. According to Lucas [1971], monitoring is focussed on gathering the actual performance of an existing system.

In this thesis, we distinguish the terms profiling and monitoring. We expand the definition of a software monitor accordingly:

Definition: Software Monitoring Tool

A software tool used to collect information about the execution of a software system during its live operation under real conditions.

Table 4.1 summarizes the differences between profiling tools and monitoring tools according to our definition. The main difference is the typical time of usage and the resulting execution of the software system under test (cf., Figure 2.4).

Profiling tools are usually employed during the development time of a software system. As a consequence, the system is executed with an artificial workload generator on dedicated hardware. Any measurement result obtained via profiling is only as good as the profile used to generate

4. Monitoring

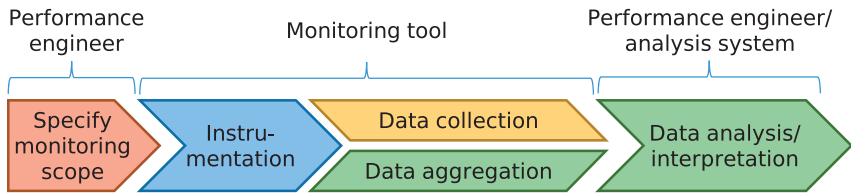


Figure 4.1. The monitoring process (based upon Eichelberger and Schmid [2014])

the workload. On the other hand, any overhead caused by the profiling does not affect any legitimate users of the system. So, profiling tools often collect very detailed data, thus causing relatively high overhead (see Chapter 6).

In the case of *monitoring tools*, the measurement is usually performed during the operation of the live system under real (in contrast to realistic) conditions. Thus, the monitoring results provide a good description of the actual events in the system. But the overhead caused by the monitoring tool has to be minimal, limiting the amount of data retrieved. Most monitoring tools can be adapted to be used as profiling tools, but the opposite does not necessarily apply.

Although the rest of the chapter concerns itself with monitoring of software system, most statements are applicable to profiling as well.

Eichelberger and Schmid [2014] propose a monitoring process common to most monitoring tools (see Figure 4.1). First, a performance engineer determines the intended monitoring scope for the SUM. This step is highly dependant on the system and on the intended final analysis. Refer to Focke et al. [2007], van Hoorn et al. [2009c], or Bertolino et al. [2013] for details on determining an appropriate scope. Next, the monitoring tool performs an instrumentation of the SUM and gathers monitoring data (see Section 4.3). Depending on the monitoring tool, this data can be further aggregated or simply collected for a concurrent or subsequent analysis. This analysis is usually performed by the performance engineer with the help of an analysis system that is often part of the monitoring tool. A brief example of such an analysis with the help of the Kieker monitoring framework is presented in Chapter 14.

4.2 Application Performance Monitoring (APM)

In the industry, Application Performance Monitoring (APM) is established as a marketing term with over 2 billion US\$ spent worldwide each year on monitoring licenses and maintenance contracts [Kowall and Cappelli 2013]. Application Performance Monitoring is also often synonymously used with Application Performance Management (APM), although the scope of the former is more limited and more tool-oriented [Sydor 2010]. The latter is characterised by Menascé [2002] as a collection of management processes to ensure that the performance of applications meets the business goals. From the business perspective it can be seen as a superset of earlier monitoring techniques [Sydor 2010].

Gartner Research provides an own definition of Application Performance Monitoring (APM) that is composed of five dimensions [Cappelli and Kowall 2011; Kowall and Cappelli 2012a; b; 2013]:

1. *End-user experience monitoring*: Monitoring the experience of the users of an application. That is, the tracking of availability or response times from the user perspective, either by monitoring real users or by introducing artificial users executing simulated queries.
2. *Runtime application architecture discovery, modeling and display*: Discovering the software and hardware components of the application and their relationships with each other during the execution of the application.
3. *User-defined transaction profiling*: The tracing of events caused by the user request across the application components.
4. *Component deep-dive monitoring in application context*: The fine-grained monitoring of the supporting infrastructure of an application in context of the detected traces and components, for example middleware, database, or hardware systems.
5. *Analytics*: The analysis of the data gathered by the other dimensions.

These five dimensions are complementary to each other and a successful APM tool should fulfill each one of them. Research tools, such as Kieker (see Section 4.5) or SPASS-meter (see Section 12.2), can usually not compete with commercial products in all of these dimensions. However, they enable the

4. Monitoring

study of new technologies and permit research which is often not possible with commercial tools. For instance, most commercial APM tools explicitly forbid performance comparisons with other tools within their licensing terms. Refer to Kowall and Cappelli [2013] for a recent overview on the commercial market of APM tools.

In the context of this thesis, we are mostly concerned with the first four dimensions of APM and their influence on the runtime behavior of the application or system under monitoring. However, we also briefly touch the subject of analyzing monitoring data and the resulting impact on the runtime behavior of performing an online analysis.

4.3 Instrumentation

Instrumentation is a technique used by monitoring tools for gathering data about a system under test by inserting probes into that system. It is often used in combination with accessing already existing data sources, such as hardware performance counters. The IEEE vocabulary [ISO/IEC/IEEE 24765] defines instrumentation as:

Definition: Instrumentation

Devices or instructions installed or inserted into hardware or software to monitor the operation of a system or component.

—ISO/IEC/IEEE 24765 [2010]

Similar definitions can be found in the literature [e. g., Jain 1991; Smith and Williams 2001; Woodside et al. 2007; Pearce et al. 2007; Bulej 2007; Marek 2014]. The inserted devices or instructions are usually called *probes*.

A related approach to instrumentation is *sampling* [Mytkowicz et al. 2010], sometimes also referred to as profiling [Mohror and Karavanic 2007]. Instead of inserting a probe into the system to continuously collect information, data is only collected during short intervals. Usually, these intervals are of random length and have randomized gaps [McCanne and Torek 1993]. This leads to greatly reduced overhead (cf., Chapter 6), although at the cost of incomplete and missing data. Sampling is a technique mostly employed by profiling tools.

4.3. Instrumentation

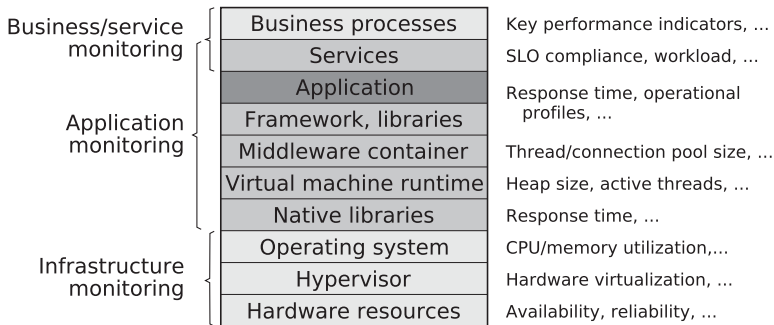


Figure 4.2. Monitoring the different abstraction layers of a software system (adapted from Ehlers [2012] and Menascé et al. [2004])

4.3.1 Abstraction Layers of a System

The instrumentation of a software system can be performed on different abstraction levels of a system [Menascé et al. 2004; Bulej 2007]. A typical set of abstraction layers for a modern enterprise software system running in a managed runtime system, for example a Java platform Enterprise Edition (EE) application, is presented in Figure 4.2.

The top two layers provide the most abstract views on the system: business processes and services provided by the application. Monitoring these layers is called *business* or *service monitoring* and provides information on key performance indicators or Service Level Objective (SLO) compliance and workload, respectively.

The shaded area of the figure represents the layers associated with *application monitoring*. The boundary to service monitoring is blurred, as the instrumentation of service interfaces is often also a part of application monitoring. The most common point of instrumentation is the application itself in combination with the used frameworks and libraries. Common measures are the response time, operational profiles, or traces.

Most middleware containers and virtual machines already include options for specific instrumentation points and thus provide a multitude of data, such as thread or connection pool sizes, heap size, or number of active

4. Monitoring

threads. Finally, on the lowest level of application monitoring resides monitoring the native libraries of applications, i. e., libraries running outside of the virtual machine and providing access to the operating system and lower layers. The common measures are similar to the ones of regular libraries.

The bottom three layers consist of the operating system, an optional virtualization hypervisor, and the actual hardware of the computer system. Monitoring of these layers is called *infrastructure monitoring* and provides, for example, information on the CPU or memory utilization, the used hardware virtualization, or the availability and reliability of hardware components.

The instrumentation of the hardware resource layer is a particular case. All other layers use software instrumentation, i. e., special inserted instructions, while the bottom layer uses hardware instrumentation, i. e., special devices inserted to capture the state of the hardware.

Applications running in a managed runtime system, such as the JVM, are different from conventional software systems written in languages like C, C++, or Fortran. Instead of a direct compilation into executable machine-dependent code, Java programs are compiled into an intermediate bytecode. This bytecode is in turn executed by the managed runtime system, providing features like automatic runtime optimization, Just-In-Time (JIT) compilation, memory management, garbage collection, reflection, security policies, and runtime exception checking. Furthermore, the bytecode is portable across different hardware configurations and operating systems. As a downside, the added virtualization layers complicate monitoring or benchmarking of Java applications (see also Section 7.5).

In the following, we describe different approaches to instrument the most common layers of a software system. Although the descriptions are coined for the Java platform and the Java Virtual Machine (JVM), similar approaches exist for other platforms, such as the Microsoft .NET framework with the Common Language Runtime (CLR) virtual machine.

4.3.2 Hardware Instrumentation

Historically, hardware instrumentation with external devices was a common method to measure the performance of computer systems [Smith and Williams 2001]. It provided low overhead access to a multitude of detailed

information, although the mapping to higher abstraction levels is not always easy [Jain 1991; Menascé et al. 2004]. However, modern computer systems are usually not accessible by external measurement devices. Thus today, these methods are often restricted to more accessible parts, such as network connections [Sabetta and Koziolok 2008].

An alternative common hardware instrumentation technique for modern computer systems is the use of *hardware performance counters* [e. g., Anderson et al. 1997; Ammons et al. 1997; Sweeney et al. 2004; Schneider et al. 2007; Bulej 2007; Mytkowicz et al. 2008a; Treibig et al. 2012; Isaacs et al. 2014]. Hardware performance counters are a special set of registers included in modern processors providing information on certain CPU instructions, cache misses, or memory accesses.

As they are already included in the processing systems, the usage of hardware performance counters produces only minimal overhead and needs no insertion of special instructions into the monitored system. On the other hand, the method to access the counters and the information stored in them are usually not portable between different hardware architectures and the information provided by them is very low-level.

First approaches to link these low-level information with executing high-level Java applications have been provided by Sweeney et al. [2004]. The authors collect low-level events, such as pipeline stalls or cache misses, and link them to the native threads they are occurring in. Then they create a mapping between the Java threads and the respective native threads, thus linking the low-level events to events within the Java application. Hauswirth et al. [2004] improve upon this approach with the concept of vertical profiling by including information from several abstraction layers into their monitoring system. Finally, Georges et al. [2004] provide a linking of these low-level information to the individual Java methods.

As a downside, all of these approaches to link low-level information with higher abstraction layers require the use of a specifically prepared Java Virtual Machine (JVM), in this case the Jikes Research Virtual Machine (RVM) [Jikes RVM] by Alpern et al. [2005]. Thus, they are usually not applicable to most Java software projects. Furthermore, the performance results gathered with the help of these techniques might not be representable of the performance on the actual JVM used in production environments.

4. Monitoring

4.3.3 Instrumentation of the Operating System

Most of the commonly used operating systems for Java EE systems, such as Windows, Linux, or Solaris, already provide tools for their own instrumentation. These tools are usually sampling data, that is they are collecting information in specified time-intervals instead of continuously, by inserting themselves into the kernel of the operating system [Sabetta and Koziolk 2008]. Typically sampled data are information on the CPU scheduler run queue or the utilization of the CPU, memory, network, or disk. An overview on different tools to gather these information is provided by, e. g., Smith and Williams [2001], Bulej [2007], or Hunt and John [2011].

The CPU utilization reported by operating system tools is often grouped by running applications and divided into *user time* and *kernel time*, also called *system* or *privileged time*. The user time is the part of the CPU utilization spent executing the actual application, while the kernel time is the part spent executing calls of the application into the kernel of the operating system.

In the case of the Windows operating system, the commonly used tools to gather the CPU utilization are the Task Manager (taskmgr.exe), the Process Explorer (procxp.exe), the Performance Monitor (perfmon.exe), and the TypePerf tool (typeperf.exe). Similar graphical CPU utilization tools exist for Linux and Solaris, for example xosview or cpubar. But command-line tools are more commonly used, for example vmstat, mpstat, prstat, or top. In addition to the CPU utilization, the tools perfmon.exe, typeperf.exe, cpubar, and vmstat can determine the CPU scheduler run queue depth.

Similar to the CPU related performance indicators, the Windows tools perfmon.exe and typeperf.exe provide information on the memory utilization, context switches, network accesses, and disk utilization of a system. For Linux and Solaris a number of common tools can be used to collect the specific data, for example vmstat for memory utilization, mpstat or pidtstat for context switches, netstat or nicstat for network utilization, and iostat for disk utilization. A more universally useable tool for Linux and Solaris is the tool sar.

Refer to Hunt and John [2011] or the respective tool documentation for a detailed description of the mentioned tools. A screenshot of the Windows

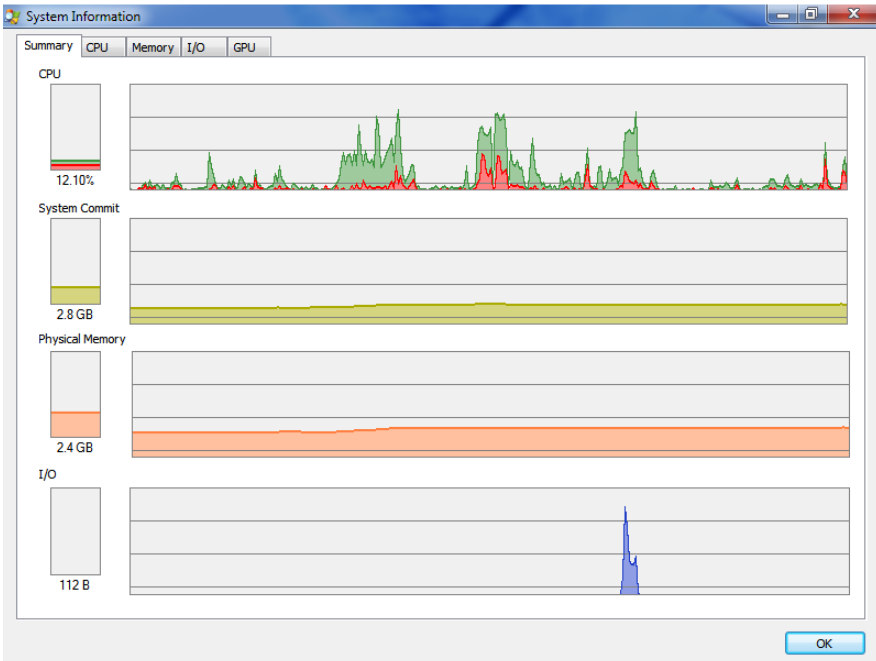


Figure 4.3. Screenshot of the Windows Process Explorer tool

Process Explorer tool displaying typical operating system level information is presented in Figure 4.3.

Similar to the instrumentation of the underlying hardware, it can be hard to match measurements in the operating system to events in an application. In addition to the combination of the mentioned tools with the use of specialized tools, such as Java stack printer (jstack), the vertical profiling approach of Hauswirth et al. [2004] provides a possible solution.

In summary, the instrumentation of the operating system provides easy access to general performance data. However, mapping this data to higher abstraction levels is not always possible and thus drawing conclusions from the collected data is not always easy.

4. Monitoring

4.3.4 Instrumentation of the Virtual Machine

A commonly used abstraction level for instrumentation is the virtual machine runtime. For instance, in the case of Java applications, the Java Virtual Machine (JVM) gets instrumented or already provides information via a standard interface [Parsons 2007]. Typically gathered information in the JVM are memory consumption, garbage collection, JIT compiler activity, and class loading [Hunt and John 2011].

Direct instrumentation of the JVM usually requires source code access and is often restricted to special Research Virtual Machines (RVMs), for example the Jikes RVM [Jikes RVM]. This reduces the portability and applicability of these instrumentation approaches, but provides access to a wide range of possible information. Sweeney et al. [2004] performed such an instrumentation of the Jikes RVM to access a mapping between Java threads and underlying operating system threads. Similar instrumentations of JVMs were performed by, e. g., Arnold and Grove [2005], Bond and McKinley [2005], or Xu et al. [2011].

A common interface available in JVMs since Java 1.5 is the Java Virtual Machine Tool Interface (JVM TI) [JVM TI]. It replaces the earlier Java Virtual Machine Profiler Interface (JVMPi) [Viswanathan and Liang 2000]. The interface is a two-way interface. A so-called agent registers with the interface and can either query the JVM directly or can be notified by the interface in case of special events. Typical events notifying the agent are, for example, entering or exiting a method in the running application. Furthermore, the agent can query running threads, get the CPU time, access stack traces and memory information, or even dynamically modify the bytecode of the running application. Nevertheless, its abilities are still limited by the provided interface. The agent itself has to be written in a native programming language, such as C or C++, and integrates with the JVM at load time. Thus, although the agent is portable across different JVMs, it is bound to a specific hardware and operating system. An example of such an instrumentation using the JVMPi is provided by Bellavista et al. [2002]. The JVM TI is employed by, for instance, Eichelberger and Schmid [2014] or Marek [2014].

Another possibility to instrument the JVM is the use of the Java Management Extensions (JMX) technology [JMX]. JMX is a standard Java technology

Listing 4.1. Querying the CPU time of the current thread in Java with JMX

```
1 ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
2 threadMXBean.getCurrentThreadUserTime();
```

that enables the management of Java resources with the help of so-called managed beans (MBean or MXBean). A managed bean is a special Java object registered with a JMX agent, that represents and manages Java resources, such as the application, objects in the applications, or specific devices in the JVM. The JMX agent manages the registered managed beans and makes them available to local or remote clients. The default JMX agent provides several default managed beans, providing interfaces to the JVM. Listing 4.1 demonstrates accessing the JVM to get the CPU time of the current thread with the help of an instance of the default managed bean `ThreadMXBean`. Similar managed beans are provided in the package `java.lang.management`.

One of the main advantages, besides the usage of portable Java code, is the easy availability of remote access to managed beans. A JMX agent can be configured to accept connections from within the JVM, from within the same machine, or even over a network. Furthermore, according to the Java Specification Request 77 [JSR 77], Java EE application servers (middleware systems) are required to provide access to their information via the JMX interface. Thus, it is usually also possible to instrument the middleware layer of Java EE applications with the help of the JMX technology to access information on thread pool sizes or database connectivity.

A similar brief overview on instrumentation techniques for the JVM is given by Marek [2014]. Hunt and John [2011] provide a detailed discussion on several further instrumentation and monitoring tools and technologies for the JVM abstraction level. For instance, the use of special command line parameters to gather information on garbage collection (e. g., `-XX:+PrintGCDetails`) or JIT compilation (`-XX:+PrintCompilation`), the use of the `JConsole` tool to access the JMX interface, and the use of the `VisualVM` tool to access a combination of these information. A screenshot of the Java `VisualVM` tool [VisualVM] displaying typical JVM level information is presented in Figure 4.4.

4. Monitoring

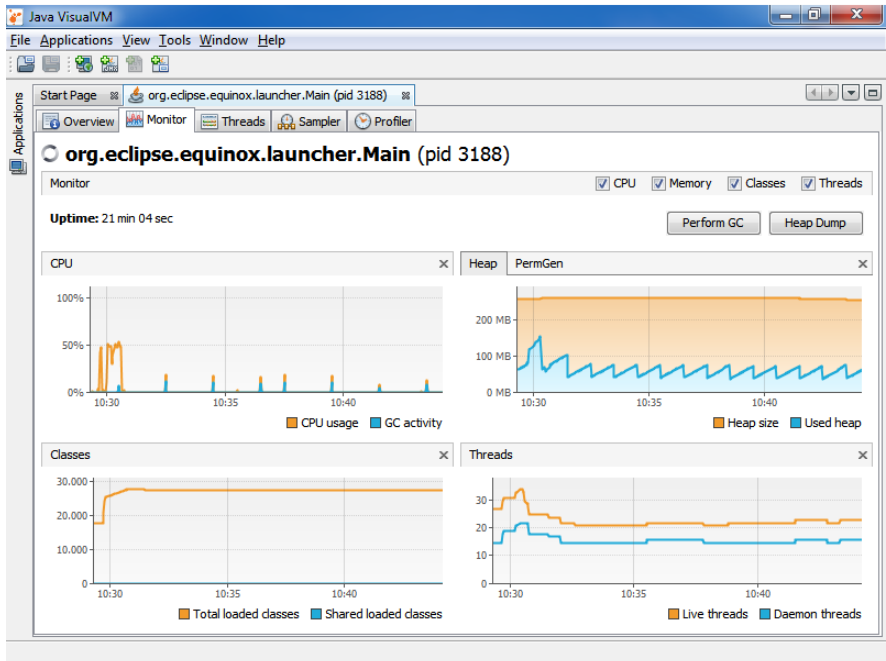


Figure 4.4. Screenshot of the Java VisualVM tool [VisualVM]

4.3.5 Application Instrumentation

In this section, we present different approaches to the instrumentation of Java applications. These approaches are applicable on the framework and library level as well as on the application-level of the monitored software system (see Figure 4.2). Furthermore, most of these approaches are also applicable to the middleware container, which is typically implemented as a complex Java application. Additionally, we include middleware interception as an approach to cover parts of service level monitoring from within the application monitoring.

Listing 4.2. Manual instrumentation of Java source code

```

1  import java.util.logging.Logger;
2  public class MonitoredClass {
3      private static final Logger LOG = Logger.getLogger("MonitoredClass");
4      public boolean monitoredMethod() {
5          LOG.entering("MonitoredClass", "monitoredMethod()");
6          // normal method code
7          // ...
8          LOG.exiting("MonitoredClass", "monitoredMethod()");
9          return true;
10     }
11 }

```

Manual Instrumentation

If the source code of the application is available, the easiest form of instrumentation for application-level monitoring is *manual source code instrumentation* [Bulej 2007]. The instrumentation code is simply added directly into the appropriate parts of the source code.

According to Smith and Williams [2001], there are three similar alternatives to implement this kind of instrumentation: (1) a direct integration of the monitoring code into the application logic, (2) an additional specialized class containing the monitoring logic, and (3) the use of existing system-event-recording tools.

Instead of developing an own monitoring solution, existing frameworks and libraries can be integrated into the monitored application. An example of such a specialized framework often used in manual instrumentation of C, C++, or Java applications is the Application Response Measurement (ARM) API [Johnson 2004]. Besides specialized monitoring frameworks, most logging frameworks can provide the required infrastructure to manually instrument the application.

For instance, the standard `java.util.logging` package can be used to perform simple instrumentations of Java applications. It is an example of using existing system-event-recording tools or logging frameworks for monitoring. A concrete example for basic monitoring of method entries and exits is

4. Monitoring

presented in Listing 4.2. Lines 5 and 8 have been added to the application in order to enable monitoring, while lines 1 and 3 have been added as necessary boilerplate code. Refer to Oliner et al. [2012] for further details on employing logging mechanisms.

Manual instrumentation is capable of acquiring almost any information from the inside of the application. Furthermore, it is easy to implement on small applications. But with larger applications, the entanglement of application logic with monitoring logic usually becomes problematic. The readability and maintainability of the resulting combined source code decrease. Furthermore, due to this entanglement, it is usually not easily possible to completely disable the monitoring logic, removing any side effects of the monitoring code from the application, e. g., the caused overhead.

In summary, manual instrumentation is a powerful tool, but it requires a major amount of work during the development and it can not be used if the source code is not available [Parsons 2007].

Bytecode Instrumentation

As mentioned before, managed runtime systems, such as the Java platform with the Java Virtual Machine, execute an intermediate bytecode instead of the machine code of the underlying hardware platform. In the case of the Java language, the source code of the Java classes (.java files) is compiled into a hardware- and operating system-independent binary format, stored in .class files [Lindholm et al. 2014]. Bytecode instrumentation is a technique for inserting probes into the bytecode, without any necessary knowledge of or access to the source code.

Bytecode instrumentation can be performed at three different stages of the program life cycle [JVM TI, (JVM TI Reference)]. First, the .class files itself can be modified (*compile-time* or *static instrumentation*). Static instrumentation can cause problems with signed .class files or multiple modifications and is not commonly used.

Second, when loading the .class files in the JVM, the bytecode can be modified (*load-time instrumentation*). This is the most common method for bytecode instrumentation, providing the greatest predictability and portability.

Listing 4.3. A Java agent for bytecode instrumentation

```

1  public class BytecodeInstrumentation {
2      public static void premain(String agentArgs, Instrumentation inst) {
3          inst.addTransformer(new ClassFileTransformer() {
4              @Override
5                  public byte[] transform(..., byte[] classfileBuffer)
6                      throws IllegalClassFormatException {
7                      // Instrument the bytecode stored in classfileBuffer
8                      return instrumentedClassfileBuffer;
9                  }
10         });
11     }
12 }

```

Third, the in-memory representation of the bytecode in the JVM can be modified (*runtime* or *dynamic instrumentation*). Although dynamic instrumentation provides greater flexibility, it is usually harder to implement and less common than load-time instrumentation.

The `java.lang.instrument` package provides built-in load-time bytecode instrumentation capabilities for the Java platform. An example of a Java agent using the `java.lang.instrument` package is presented in Listing 4.3. A Java agent is a class containing a static `premain` method, which is executed by the JVM before the actual `main` method of the application is started. In order to get recognized by the JVM, the Java agent has to be packaged into a jar-archive with a special `Premain-Class` annotation in its manifest file. Furthermore, a `-javaagent:path_to_jar` parameter has to be added to JVM command line.

Similar results can be achieved with bytecode instrumentation capabilities of the Java Virtual Machine Tool Interface [JVM TI]. As mentioned before, a JVM TI agent has to be written in a native language, such as C or C++, and integrates with the JVM at load time, but provides the additional capability of dynamic instrumentation, that is the loaded and perhaps already modified class can be remodified during the runtime of the Java application. Examples of tools using the instrumentation capabilities of JVM TI for pro-

4. Monitoring

filing are the Oracle Solaris Studio Performance Analyzer [Itzkowitz 2010] and the NetBeans Profiler included in the Java VisualVM tool [VisualVM].

Using either the `java.lang.instrument` package or the JVM TI technology, the implementation of the actual transformation requires detailed knowledge of the inner structure of `.class` files, as file contents are simply forwarded to the transformation method. Furthermore, references to values or bytecode addresses are stored as absolute values inside the `.class` files. So, minor changes in one part of the file can have major impact on other parts of the file. Details on the structure of `.class` files and Java bytecode instructions are presented by Lindholm et al. [2014].

Several special bytecode instrumentation libraries aim to assist with the details of bytecode instrumentation. The Byte Code Engineering Library (BCEL) [BCEL; Dahm 2001] provides a very low-level API to manipulate the bytecode of Java classes. Most of the problems with direct manipulation of the `.class` files are circumvented, but a detailed knowledge of the different bytecode instructions is needed nevertheless. The ASM framework [ASM; Bruneton et al. 2002] has a similar approach but its focus is on efficiency and a smaller memory footprint. The Javassist framework [Javassist; Chiba 2000] focuses on usability and further abstracts from the `.class`-file structure. It includes a simple Java compiler to allow the addition of on-the-fly compiled Java source code into existing classes. Thus, it enables bytecode instrumentation without knowledge of the structure of `.class` files or bytecode instructions.

Similar techniques exist for the instrumentation of other languages [Bulej 2007; Liu et al. 2013]. For instance, the `gprof` profiler [Graham et al. 1982] enables compile time instrumentation for C or C++ applications with the help of special support from the `gcc` compiler. Other common techniques to automate the instrumentation are program transformation systems [Visser 2005] or binary rewriters [e. g., Srivastava and Eustace 1994; Pearce et al. 2002].

In summary, bytecode instrumentation has the same capabilities as manual instrumentation, but avoids its issues. Source code access is no longer required and entangling of application logic with monitoring logic is avoided. However, bytecode instrumentation usually requires intimate knowledge of the inner working of Java bytecode and `.class` files.

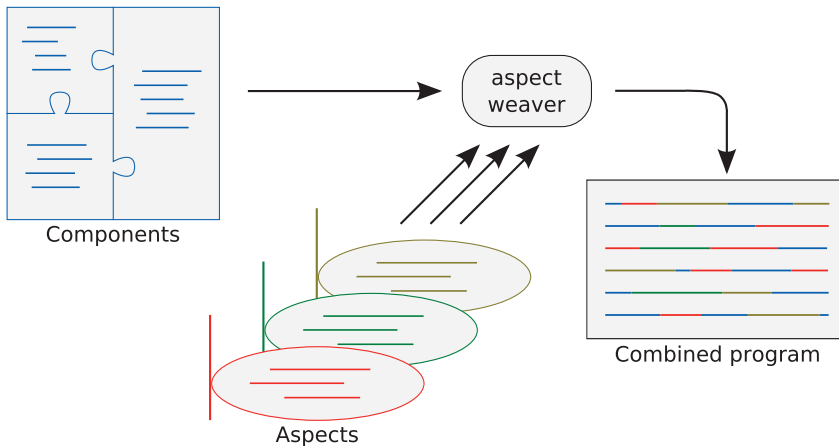


Figure 4.5. Aspect-Oriented Programming (AOP): The components of a system are woven together with the aspects of a system, resulting in the combined program (based upon Kiczales et al. [1996])

Aspect-Oriented Programming (AOP)

Kiczales et al. [1997] introduced the Aspect-Oriented Programming (AOP) paradigm. AOP distinguishes between components and aspects. A *component* is a self-contained part of a system, for example an object or a procedure. An *aspect* is a property of a system affecting more than one component, a so-called cross-cutting concern, for example performance optimization, error handling, or monitoring. The goal of AOP is to separate components from each other, aspects from each other, and components from aspects [Kiczales et al. 1997].

Components are implemented in a component language while aspects are implemented in an aspect language. Depending on the AOP implementation, both languages can either be distinct or share (at least) a common basis. An *aspect-weaver* is used to combine the components with the aspects according to specified set of rules into the final system. This combination, or weaving, can either occur at *compile-time* or during the *run-time* (or *load-time*) of the system.

4. Monitoring

The elements of an aspect-oriented program are presented in Figure 4.5. The figure is based upon earlier work of Kiczales et al. [1996] and depicts a system with three components written in a component language and three aspects written in three (perhaps different) aspect languages. The components and aspects are woven together into the combined program by the aspect weaver. The language of the combined program is typically either the component language or source code of an own language.

The most common AOP implementation for Java is AspectJ [AspectJ; Kiczales et al. 2001]. It was originally developed by Kiczales et al. at the Xerox Palo Alto Research Center, but later contributed to the Eclipse Foundation. In its current implementation the AspectJ aspect weaver uses the BCEL bytecode instrumentation framework [BCEL]. Thus, technically, the component language and the aspect language is Java bytecode. But for all practical purposes, the component language is Java while the aspect language is either a Java-based dialect or Java with the addition of special annotations.

In addition to the general AOP terms, AspectJ introduced several new terms for Aspect-Oriented Programming instrumentation [Kiczales et al. 2001]. These terms also got adopted by other AOP implementations.

Join point: A join point is the basis of program instrumentation. It corresponds to a well-defined point in the execution of a program, for example the execution of a specific method, access to a specific field, the construction of an object from a specific class, or the handling of an exception.

Pointcut: A pointcut is a collection of several join points sharing a specified property, for example all methods with names starting with `get`, or all methods within one class or package. Pointcuts are used to specify join points for instrumentation.

Advice: Advices are methods or sets of operations that are associated with a pointcut and that get executed at each join point specified by the pointcut. Usually, advices can be defined to get executed before, after, or around the specified join points.

Aspect: Finally, an aspect is a collection of all pointcuts and advices belonging to one cross-cutting concern. Aspects are similar to ordinary

Listing 4.4. AspectJ aspect for the instrumentation of Java source code

```

1  @Aspect
2  public class MonitoringAspect {
3      @Pointcut("execution(* *(..))")
4      public void monitoredOperation() { /* must be empty */ }
5
6      @Before("monitoredOperation()")
7      public void beforeOperation(final JoinPoint.StaticPart jp) {
8          System.err.println("Enter : " + jp);
9      }
10     @After("monitoredOperation()")
11     public void afterOperation(final JoinPoint.StaticPart jp) {
12         System.err.println("Leave : " + jp);
13     }
14 }

```

Java classes. They can use inheritance, implement interface, and have internal methods and data structures.

As mentioned before, an AspectJ aspect can be defined in different ways. Originally, AspectJ proposed an own Java dialect (or DSL) for the definition of aspects. With AspectJ 5, it became possible to define aspects as normal Java classes with the help of special annotations. An example of such an aspect is presented in Listing 4.4. Furthermore, existing abstract aspects can be made concrete by means of an XML configuration file.

The sample aspect in Listing 4.4 is similar to the manual instrumentation presented in Listing 4.2. The aspect consists of one pointcut definition, matching each join point where any method is executed, and two advices executed before or after each join point. The `JoinPoint.StaticPart` allows for access to static information of the currently executing join point, such as the names of the executing class and method. Thus, before and after each method execution, the corresponding messages are logged.

Currently, AspectJ supports compile-time weaving as well as load-time weaving. For *compile-time weaving*, a special aspect compiler (ajc) is used in addition or instead of the regular Java compiler (textcodejavac).

4. Monitoring

Listing 4.5. AspectJ: simple aop.xml file for Java instrumentation

```
1 <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "...">
2 <aspectj>
3   <weaver options="">
4     <include within="monitored.package..*" />
5     <exclude within="monitored.package.tests..*" />
6   </weaver>
7   <aspects>
8     <aspect name="MonitoringAspect" />
9   </aspects>
10 </aspectj>
```

For *load-time weaving*, a XML configuration file (META-INF/aop.xml) is required to specify the used aspects and further parameters, such as a list of packages to include into or exclude from weaving. In addition, the `-javaagent:aspectweaver.jar` parameter has to be added to the JVM command line. An example aop.xml file for the aspect in Listing 4.4 is presented in Listing 4.5. Here, weaving is enabled for all classes in the package `monitored.package` and all its subpackages, except for classes in `monitored.package.tests` and subpackages.

Examples of using AspectJ for the instrumentation of software systems for monitoring are provided, e. g., by Khaled et al. [2003], Richters and Gogolla [2003], Debusmann and Geihs [2003], Colyer et al. [2003], Chen and Roşu [2005], Avgustinov et al. [2006], Bodden and Havelund [2008], or Nusayr and Cook [2009]. An extensive tutorial of monitoring Java EE applications with AspectJ is provided by Bodkin [2005]. Pearce et al. [2007] use AspectJ to perform typical profiling activities. In the context of the Kieker framework, AspectJ is used as the default instrumentation provider (see Section 4.5).

In addition to its stand-alone implementation, AspectJ is also included as a possible realization of Spring AOP in the Spring framework [Spring]. A research focused reimplementaion of the AspectJ compiler (ajc) has been performed by Avgustinov et al. [2005a] with the AspectBench Compiler (abc). Experience reports from using AspectJ in industrial projects are provided,

e. g., by Colyer et al. [2003] or Hohenstein and Jäger [2009]. Refer to the AspectJ documentation [AspectJ] for a complete overview on the capabilities of the AspectJ AOP implementation.

Besides AspectJ, there are further AOP implementations for the Java language, for example GluonJ and DiSL. *GluonJ* [GluonJ; Chiba et al. 2010] is an AOP implementation based upon the Javassist bytecode instrumentation framework [Javassist; Chiba 2000]. It is focused on modularity and formality. The *DiSL framework* [DiSL; Marek et al. 2012; 2015; Sarimbekov et al. 2014; Marek 2014] is a mix between AOP and bytecode instrumentation. It is based upon the ASM bytecode instrumentation framework [ASM; Bruneton et al. 2002] and JVM TI [JVM TI] and allows for the instrumentation of arbitrary bytecode regions but provides instrumentation concepts typical of AOP implementations, for example join points, pointcuts, and advices (called snippets).

Aspect-Oriented Programming is not limited to the Java language. Examples for other languages include modern languages such as C [Coady et al. 2001], C++ [Spinczyk et al. 2002] or C# [Kim 2002], as well as legacy languages such as Visual Basic 6 [van Hoorn et al. 2011] or COBOL [Knoche et al. 2012].

In summary, Aspect-Oriented Programming has similar capabilities to manual or bytecode instrumentation, but is sometimes limited in its applicability, depending on the used AOP implementation. For instance, AspectJ is currently not able to target loops within method bodies, while DiSL has no such limitation. The advantages of AOP instrumentation are similar to bytecode instrumentation with the added benefit of abstraction from the inner workings of Java bytecode. As a downside, the added abstraction can restrict the possible instrumentation points and can introduce new sources of overhead.

Middleware Interception

The middleware interception approach to application-level monitoring aims to instrument the service interfaces of the services layer. Most Java EE applications are component-based software systems, i. e., the services of the software system are independent components, communicating with speci-

4. Monitoring

Listing 4.6. AspectJ pointcut for the instrumentation of Java servlets

```
1 @Pointcut("execution(* *.do*(..)) && args(request,response)")
2 public void monitoredServlet(HttpServletRequest request,
3                             HttpServletResponse response) {
4     // Aspect Declaration (MUST be empty)
5 }
```

fied interfaces [Szyperski et al. 2002]. The structure of these components is often determined by the used middleware systems. These interfaces can either be directly instrumented, using one of the previously introduced methods, or additional wrappers can be inserted to perform the instrumentation [Gao et al. 2000; 2001].

For instance, most Java EE applications are implemented as multitier systems, usually including at least a web tier [Jendrock et al. 2012]. The web tier provides a container for servlets or JavaServer Pages (JSPs). Servlets are special Java classes used to respond to server queries, for example a web browser requesting a web page from the server. JSPs are text-based documents describing dynamic web pages that get automatically converted into servlets by the Java EE server. Java EE servers use standardized interfaces to access the existing servlets. Furthermore, servlets can be extended by pre- and post-processing filters, executed before or after the actual execution of the servlet. Consequently, an instrumentation of the standardized interface, the use of filters, or a combination of both approaches enables an instrumentation of all communication between the web client and the application, thus providing an example of service-level instrumentation. Similar possibilities exist for all common tiers of Java EE applications.

An example of an AspectJ pointcut to access the standardized servlet interface, in this case the `do` methods (e.g., `doGet()` or `doPost()`) provided by the `HttpServletRequest` class, is given in Listing 4.6. A similar approach is also detailed by Debusmann and Geihs [2003].

Another example of middleware interception is provided by the COMPAS framework [Parsons et al. 2006; Parsons 2007]. The framework analyzes the metadata provided by Java EE applications to determine existing component interfaces and subsequently instruments the detected interfaces.

4.3.6 Business Process and Service Instrumentation

Business process or service instrumentation is concerned with the highest two abstraction layers of Figure 4.2. Rather than directly instrumenting the application that implements a service or business process, the instrumentation is performed on this higher level. The business process or services are usually modelled in a special modeling language, for instance, the Business Process Model and Notation (BPMN) [OMG 2011a]. These languages either already contain facilities to include monitoring probes or can be extended in such a fashion.

An example of such an extension for scientific workflows that is based upon Kieker is given by Brauer and Hasselbring [2012] or Brauer et al. [2014]. An example of instrumenting generated web services is provided by Momm et al. [2008]. A further approach to generate monitoring instrumentation based upon model-driven software engineering techniques has been developed by Bošković and Hasselbring [2009]. A more general approach is described by Jung et al. [2013].

On the highest level of abstraction, monitoring is usually employed to gather critical business information. These activities are usually called business activity monitoring [McCoy 2002].

4.4 States, Events, and Traces

Generally speaking, there are two possible kinds of measurements when monitoring a (software) system: (1) measuring the state of the system and (2) measuring events within the system [Lilja 2000; Smith and Williams 2001].

The *state* of a system is a period of time during which a system is doing something specific, for example a CPU having high utilization or the system executing an operation. Usually, the state of a system is measured periodically, for example getting the current CPU utilization every second. But, depending on the used instrumentation, a new measurement can be triggered by changes of state, for example, whenever a new operation is executed, a new execution state is measured.

4. Monitoring

An *event* in a monitored system is the change between two states of the system. Events are measured when they happen. For instance, at the start or the end of an operation execution, the start or end of the execution is measured.

This leads to two different monitoring approaches: *state-based monitoring* and *event-based monitoring*. Furthermore, a combination of both approaches is possible and common in many monitoring tools.

When monitoring the execution of the method `monitoredMethod()` in a state-based monitoring approach, we have a single measurement: *monitoredMethod() executing from time x to time y*. In an event-based monitoring approach, we have two measurements: *monitoredMethod() started executing at time x* and *monitoredMethod() stopped executing at time y*.

In this simple example, both approaches have the same expressiveness, but usually event-based monitoring provides for more detailed measurements, especially when compared to a periodic measurement approach (sampling) that is common for state-based monitoring. Event-based monitoring is a more fitting approach for method monitoring, while state-based monitoring is more fitting for, e.g., CPU or memory monitoring.

Additional details on the two monitoring approaches in the context of the Kieker framework are given in Section 4.5.2.

A series of associated measurements (either events or states) is called a *trace* [e.g., Jain 1991; Lilja 2000; Gao et al. 2000; 2001; Mohror and Karavanic 2007]. An example of a trace is an *execution trace* of a software system. It is the collection of events or states corresponding to the executions of the methods of the software system [Rohr et al. 2008]. A simple example of an event-based execution trace for a method `A()` that calls `B()` is as follows: *methodA() started executing at time 1; methodB() started executing at time 2; methodB() stopped executing at time 3; methodA() stopped executing at time 4*.

The events or states of a trace are distinctly ordered, for instance, according to timestamps or IDs. Furthermore, they have to include all necessary information, for example a timestamp, the type of event, or other parameters. This allows for a reconstruction of (or approximation upon) the original events in the monitored system. The act of finding such an execution path of a system is called *tracing* [Jain 1991; Lilja 2000].

4.5 The Kieker Framework

The Kieker framework [Kieker; Kieker 2014] by Rohr et al. [2008] and van Hoorn et al. [2009b, 2012] is an extensible framework for application performance monitoring and dynamic software analysis. The framework includes measurement probes for the instrumentation of software systems and monitoring writers to facilitate the storage or further transport of gathered data. Included analysis plugins operate on the gathered data and extract architectural models that get visualized and augmented by quantitative observations.

Large parts of the Kieker framework have been co-developed and evaluated in the context of this thesis. In the following, we provide a brief introduction to the parts relevant for our performance evaluation of Kieker. A recent, more detailed description of the framework and its development is provided in the dissertation of van Hoorn [2014].

The development of the Kieker framework has started in 2006 [Focke 2006]. It originally has been designed as a small tool for continuously monitoring response times of Java software operations. Since then, Kieker has evolved into a powerful and extensible dynamic analysis framework with a focus on Java-based systems. In 2011, the Kieker framework was reviewed, accepted, and published as a recommended tool for quantitative system evaluation and analysis by the SPEC Research Group (SPEC RG). Since then, the tool is also distributed as part of SPEC RG's tool repository.¹ As of this writing, the most current version of Kieker is 1.9, released in April, 2014.

Although originally developed as a research tool, Kieker has been evaluated in several industrial production systems [e. g., van Hoorn et al. 2009b; Rohr et al. 2010; van Hoorn 2014]. In order to extend its focus beyond Java-based systems, monitoring adapters for additional platforms have been added. Examples include legacy systems such as Visual Basic 6 [van Hoorn et al. 2011] or COBOL [Knoche et al. 2012] as well as modern platforms, such as the Microsoft .NET platform [Magedanz 2011] or cloud systems [Frey 2013].

¹<http://research.spec.org/projects/tools.html>

4. Monitoring

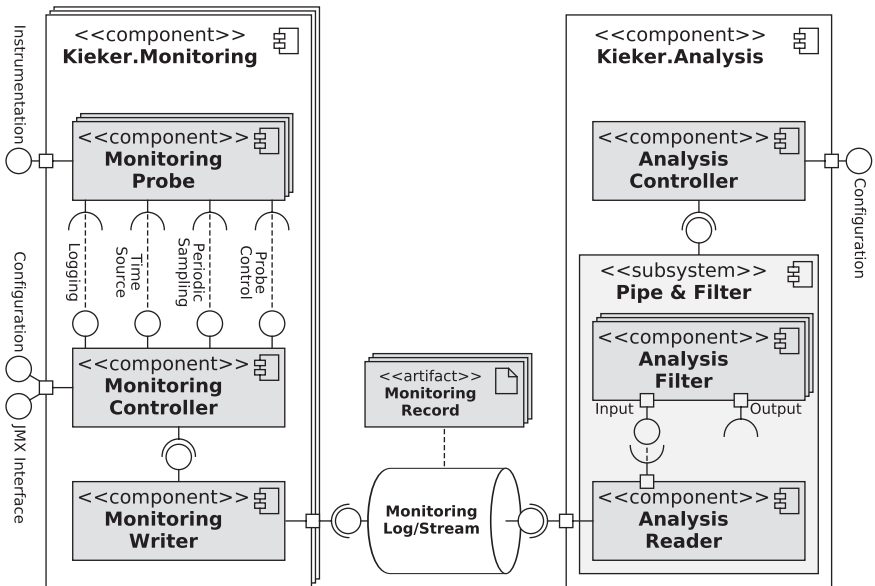


Figure 4.6. UML component diagram of a top-level view on the Kieker framework architecture

4.5.1 Kieker Architecture

The Kieker framework provides components for software instrumentation, collection of information, logging/transport of collected data, and analysis/visualization of this monitoring data. Each Kieker component is extensible and replaceable to support specific project contexts. A UML component diagram [OMG 2011b] representation of the top-level view on the Kieker framework architecture and its components is presented in Figure 4.6.

The general Kieker architecture is divided into two parts. First, the Kieker.Monitoring component that is used to actually monitor software systems. Second, the Kieker.Analysis component that is used to analyze and visualize the gathered data. These two top-level components are connected by a Monitoring Log or Stream.

The Monitoring Log or Stream

A Monitoring Log is used to decouple the analysis component from the monitoring component. It enables offline analysis of the gathered data, that is the analysis is performed independently from the collection. Offline analysis is usually performed to generate reports or it is triggered manually when problems occurred. The analysis is independent of the running system, so the monitoring logs can either be stored locally, to be retrieved later, or remotely in files or a database.

A Monitoring Stream, on the other hand, is used for online or live analysis of monitoring data. In this case, the data is analyzed as soon as it is collected, concurrent to the monitoring process. The live analysis is performed at runtime, usually on a separate physical system, and allows for immediate reactions to monitoring events, such as increased response times of methods or violations of Service Level Objectives (SLOs). The stream is typically realized by either directly connecting the monitoring component to the analysis component in Java (thus effectively removing the Monitoring Stream component) or by using a transportation technology, for example Java Message Service (JMS), Java Management Extensions (JMX), or Transmission Control Protocol (TCP).

The artifacts used to store and transport the collected monitoring data are called Monitoring Records. Each record is a data structure that contains the data associated with a single monitored event or state.

Particularly, in the case of monitoring distributed software systems, there may exist more than one instance of the `Kieker.Monitoring` component, usually one instance per separate software system. The Monitoring Records of all instances are pooled in the Monitoring Log/Stream and retrieved with the `Kieker.Analysis` component.

The Kieker.Monitoring Component

The `Kieker.Monitoring` component is the focus of our work in this thesis. In combination with the Monitoring Log or Stream, it is the most important part of the application monitoring framework. Thus, it is the focus of our attention in our performance benchmarks. One instance of the monitoring component is used to instrument each software system under monitoring.

4. Monitoring

The actual instrumentation of the applications is performed with the help of bytecode instrumentation or AOP libraries, such as AspectJ (see Section 4.3.5). This way, the provided Monitoring Probes are inserted into the targeted methods of the monitored system.

With any execution of these monitored or instrumented methods, the inserted probes collect data, e.g., the name of the monitored method, execution timestamps, or trace information. This collected data is stored in Monitoring Records.

The Monitoring Controller has a central role within the Kieker.Monitoring component. It configures and initializes the whole monitoring system. Furthermore, it coordinates the activation and deactivation of Monitoring Probes for adaptive monitoring [Ehlers 2012]. Additionally, it provides a configurable Time Source to the probes and it can trigger periodically running sampling probes, e.g., to collect CPU or memory information. Finally, it connects all active probes to the configured Monitoring Writer.

The Monitoring Writer receives the Monitoring Records and forwards them to the Monitoring Log/Stream. Depending on the actually employed writer, the records are either serialized and stored or directly forwarded to a connected analysis component. Most writers are asynchronous, i.e., all incoming records are inserted into a buffer while an additional thread handles the actual serialization and the writing. Usually, this decoupling of gathering data and writing data provides a performance advantage. A detailed analysis of this performance is given in Section 11.4.

The three most commonly used writer technologies within Kieker are: (1) the default ASCII writer, (2) the binary writer, and (3) the TCP writer. The *ASCII writer* produces human-readable CSV files. Thus, it is mostly suited for small projects or debugging of the performed instrumentation. The *binary writer* produces smaller, more efficient files. Thus, it is best suited for productive use with the intent of a subsequent offline analysis. Finally, the *TCP writer* sends binary encoded records to a connected Analysis Reader. Thus, it usually is the technology of choice for live analysis of monitoring data. The performance of these writers is evaluated and put into comparison in Chapter 11.

A more detailed description of the actual process of monitoring a method is presented in Section 6.2.

The Kieker.Analysis Component

The central part of the Kieker.Analysis component is the Analysis Controller. It configures and controls the underlying Pipes & Filters network [e. g., Hohpe and Woolf 2004] that realizes the actual analysis.

The Pipes & Filters network consists of an Analysis Reader and several interconnected Analysis Filters. The Analysis Reader retrieves the Monitoring Records from the Monitoring Log/Stream and delivers them to its connected filters. Each Analysis Filter performs certain analysis or visualization operations on the received data and sends the results to its successors.

The Analysis Filters available with the Kieker framework support the reconstruction of traces, the automated generation of UML sequence diagrams [OMG 2011b], dependency graphs, and call graphs. Refer to van Hoorn et al. [2009b] and van Hoorn [2014] for more information on analysis and visualization with Kieker. Furthermore, two 3D visualizations built upon the Kieker framework are presented in Chapter 10.

4.5.2 State-based and Event-based Monitoring

Kieker has always provided support for state-based monitoring. In its more recent version (since version 1.5 in 2009), support for event-based monitoring has been added.

The original *state-based monitoring* approach is centered around the `OperationExecutionRecord` and its associated Monitoring Probes. The probes are called during the execution of each monitored method and create a respective record. Each `OperationExecutionRecord` contains information about the execution of the method, such as its signature and timestamps for the start and end of its execution. Furthermore, in the case of distributed or web-based systems, the session identifier and the hostname of the execution are included as well.

Tracing support is added by providing a unique `traceId` to each record belonging to the same trace. Furthermore, an order index as well as the current stack depth are logged within each record to enable the reconstruction of the monitored execution traces. Refer to van Hoorn et al. [2009b] or van Hoorn [2014] for further details.

4. Monitoring

Additional state-based probes and records are available for CPU and memory monitoring. Here, all probes are called periodically and provide sampling information of the observed system properties.

The *event-based monitoring* approach of Kieker is based upon the probes and records provided in the `.flow` packages. These event records are more flexible than `OperationExecutionRecords`. Several different events can be associated with each execution of an operation and each such event can be recorded in an own record. All trace event records share the common superclass `AbstractTraceEvent`, defining the minimal set of information associated with each event in a trace. So, each trace event occurs at a specific timestamp, has a unique trace identifier shared by all events in the same trace, and an unique order index within the trace.

The two main trace event records used to replicate the scope of `OperationExecutionRecords` are `BeforeOperationEvent` and `AfterOperationEvent`. Both kinds of records add information on the class and operation signature to the event trace. They always occur in pairs allowing for the reconstruction of the stack depth of the executed operation.

In addition to before and after events, further specialized events can be associated with operation executions, e. g., `CallOperationEvents`. Each execution of an operation is started by a call of this operation within the context of a caller operation, creating caller/callee associations between operations. Besides generating additional timing information, the collection of call events allows for the detection of gaps in the monitoring coverage. Thus, it is possible to make assumptions on partially reconstruct traces involving unmonitored components, e. g., third-party libraries without any available source-code [Knoche et al. 2012].

Further examples of specialized events are `AfterOperationFailedEvent` (including information on exceptions), `...ConstructorEvents` (including information on the created object), `...ObjectEvents` (including information on the object instance the operation is called upon), or concurrency events such as `SplitEvents` (start of a new thread) or `Monitor...Events` (monitoring synchronization).

Part II

**Benchmarking Monitoring
Systems**

Research Methods and Questions

In this chapter, we present our employed *research methods* (Section 5.1). In addition, we pose our *research questions*, develop a *research plan*, and present an GQM goal to present our evaluation (Section 5.2).

*It is not enough to do your best;
you must know what to do, and then do your best.*

—W. Edwards Deming, statistician & professor

5. Research Methods and Questions

Previous Publications

Parts of this chapter are already published in the following works:

1. *J. Waller*. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013

5.1 Research Methods

In this section, we present the research methods that we employ within this thesis. Classifications of typical empirical research methods in software engineering are performed by Wohlin et al. [2006], Höfer and Tichy [2006], or Tichy and Padberg [2007]. These classifications are similar to established categories in other fields of empirical sciences, e. g., by Christensen [2006]. Further classifications in software engineering are given by Sjøberg et al. [2007] or Easterbrook et al. [2008].

According to Tichy [2014], benchmarks are one of the most important empirical methods in computer science. Refer to Section 3.3 for an overview on the role of benchmarks. For instance, benchmarks are typically employed to compare and evaluate different techniques, tools, or environments. An example of such a methodology employing benchmarks is given by Stantchev [2009]. Thus, by performing several case studies with benchmark experiments, we can evaluate the capabilities of the used benchmark systems.

Hints on performing these kinds of experiments are provided, for instance, by Fenton and Pfleeger [1998] or Blackburn et al. [2012]. Further detailed guidelines for the execution of empirical software engineering research are given by Kitchenham et al. [2002].

Of special mention is the importance of replication of experiments [e. g., Shull et al. 2002; Juristo and Gómez 2012; Crick et al. 2014]. With regard to benchmarks, we discuss the need for replicability as a central requirement of benchmark design in Chapter 7.

We employ the following research methods within this thesis:

- *Literature reviews* are a systematic method to review and compare previous approaches by other researchers in the literature. We employ this method, for instance, in our definition of a benchmark engineering methodology in Chapter 7.
- *Proof-of-concept implementations* demonstrate the technical feasibility of an approach. Subsequently, these implementations can be evaluated with further research methods. We employ this method to demonstrate the capabilities of our benchmark engineering methodology to create the MooBench micro-benchmark (Chapter 8).

5. Research Methods and Questions

- *Lab experiments* are our main research method. These experiments produce evaluation results in controlled environments. Detailed guidelines for the execution of benchmark experiments are given in the context of our benchmark engineering methodology in Chapter 7. Applications of benchmark experiments are described in Chapters 11–13.

In addition to these research methods, we employ the Goal, Question, Metric (GQM) approach [Basili et al. 1994; van Solingen and Berghout 1999] to summarize our evaluations of our benchmarking approach. It is considered a state of the art approach to evaluate software projects [Dumke et al. 2013]. GQM provides a structured paradigm to further define a set of goals with specific questions and to select and refine the required metrics to measure the fulfillment of these goals.

Based on broad conceptual goals, specific questions are developed to further characterize all different aspects of the respective goals. In turn, each question gets associated with a set of software metrics (measures) that provide the information to answer the questions. The previously mentioned research methods can serve to deliver the metrics in our application of GQM. In summary, the GQM approach defines the goals, questions, and metrics in a top-down approach and interprets the measurement results in a bottom-up approach.

5.2 Research Questions and Research Plan

The goal of this thesis is to measure the performance overhead of application-level monitoring frameworks. This section provides an overview on our research questions and our research plan.

This goal directly leads to our main research question: *RQ: What is the performance influence an application-level monitoring framework has on the monitored system?* Similar to the GQM approach, we can break this goal or question down into further sub-questions according to the contributions of our approach in this thesis (see Section 1.2).

Monitoring Overhead

The performance influence of monitoring (also called monitoring overhead) can be measured by the impact on the response times of monitored methods (cf., Chapter 6). Thus, our first research question is: *RQ1: What are the causes for observed changes in the response time of a monitored method?*

We investigate this research question in Chapter 6. There, we provide a common definition of monitoring overhead with respect to the response times of monitored applications. Our analysis leads to three common causes of monitoring overhead: (I) *instrumentation* of the system under monitoring, (C) *collection* of monitoring data, and (W) *writing* the collected data into a monitoring log or stream.

We evaluate these findings with Kieker in Chapter 11 using a series of *lab experiments* with benchmarks. In these experiments, we demonstrate the presence and measurability of these overhead causes within Kieker. Furthermore, we validate the findings on Kieker with additional lab experiments using two additional application-level monitoring frameworks in Chapter 12. Finally, in Section 11.6, we expand upon the approach of three common causes of monitoring overhead and introduce additional causes of monitoring overhead for live analysis concurrent to monitoring.

Benchmark Engineering Methodology

As motivated in Sections 1.1 and 3.3, benchmarks are a common method to measure performance. Thus, our next research question is: *RQ2: How to develop a benchmark to measure the causes of monitoring overhead?*

Within this subsection, we focus on the first part of the research question: *RQ2.1: How to develop a benchmark?* As mentioned in Section 1.1, no established benchmark engineering methodology exists. In Chapter 7, we describe such a methodology that is split into three phases: (1) The first phase is the *design and the implementation of the benchmark*. (2) The second phase is the *execution of the benchmark*. (3) The third phase is the *analysis and presentation of benchmark results*. For each of these phases, we present a set of requirements that a benchmark should adhere to.

Our main evaluation of this benchmark engineering methodology is performed within Chapter 7 with an extensive *literature review* of 50 different

5. Research Methods and Questions

publications concerned with benchmarking. We extract the benchmark requirements presented in each of these publications and integrate them into our approach. Thus, we obtain the resulting combination of fifteen primary and three secondary benchmark requirements.

In addition, we provide a *proof-of-concept implementation* of our benchmark engineering methodology for the MooBench micro-benchmark in Chapter 8. This implementation demonstrates the feasibility of our approach. The evaluation of this implementation is sketched in the next subsection.

Benchmarks to Quantify Monitoring Overhead

Within this subsection, we are concerned with the second part of our second research question (RQ2). Thus, our sub-question is: *RQ2.2: How to measure and quantify monitoring overhead using benchmarks?* We evaluate this research question using *lab experiments* with the benchmark and summarize the results using the GQM approach.

The GQM goal G of this research question is to *measure and quantify the performance overhead of a software monitoring framework with the MooBench micro-benchmark*. Our complete GQM model for this goal is presented in Table 5.1.

The first GQM question Q1 is targeted at the object and viewpoint of our goal: *Which monitoring tools or frameworks can be benchmarked with MooBench?* Its aim is to confirm the applicability of the benchmark for different tools or frameworks. As such, its metrics are (M1) the different tools or frameworks itself and (M2/3) the required changes or adaptations to the frameworks that are necessary for the benchmark to measure the total monitoring overhead (M2) or to quantify the causes of monitoring overhead (M3). We evaluate M1 in Chapter 11 with Kieker and ExplorViz. In Chapter 12, we furthermore add evaluations of inspectIT and SPASS-meter. Similarly, we evaluate the metrics M2 and M3 in these chapters for the respective monitoring frameworks.

The second question Q2 targets the effort of a performance evaluation with MooBench. Accordingly, we reuse the previous metrics M2 and M3 for the monitoring frameworks under test. Furthermore, we add M4 that

5.2. Research Questions and Research Plan

is concerned with the required run-time of the benchmark. We evaluate the minimal run-time in our discussion of the benchmark execution in Section 8.3 and add further run-times with our lab experiments in Chapters 11 and 12.

In our third question **Q3**, we evaluate the purpose of our goal: *Can the monitoring overhead be quantified with MooBench?* Thus, our main metric **M5** contains the scenarios within which we are able to determine the overhead. Our next metric **M6** is related: It is concerned with the configurability and adaptability of our benchmark to these scenarios. Finally, metric **M7** measures the reproducibility of our benchmark results. Our different evaluation scenarios for our benchmark are presented in Chapters 11 and 12 (**M5**). Similarly, we demonstrate the configurability and adaptability (**M6**) in the same chapters. The reproducibility of our results (**M7**) is also evaluated in these chapters by repeating each experiment multiple times and by comparing similar experiments with slightly different environments of our scenarios.

Our final question **Q4** is: *Are the benchmark results representative and relevant?* Without reproducibility, the results cannot be relevant. Thus, we again utilize metric **M7**. Furthermore, we compare our results to further benchmarks and performance evaluations (**M8**). As mentioned before, **M7** is evaluated in Chapters 11 and 12. We evaluate metric **M8** with the help of three different macro-benchmarks in Chapter 13. Additionally, we further validate these results with a comparison to the results of a meta-monitoring experiment with Kieker in Chapter 14.

Besides our evaluation in Part III (Chapters 11–14), we further discuss the answers to our research questions in the conclusions of this thesis (see Section 16.1).

5. Research Methods and Questions

Table 5.1. Definition of the GQM goal: “Measure and quantify the performance overhead of a monitoring framework with the MooBench micro-benchmark.”

Goal	G	
	Purpose	Measure and quantify
	Issue	the performance overhead of
	Object	a monitoring framework
	Viewpoint	with the MooBench micro-benchmark.
Question	Q1	Which monitoring tools can be benchmarked?
Metrics	M1	tools or frameworks
	M2	required changes for simple benchmarks
	M3	required changes for cause analysis
Question	Q2	What effort is required to benchmark?
Metrics	M2	required changes for simple benchmarks
	M3	required changes for cause analysis
	M4	required run-time of the benchmark
Question	Q3	Can the monitoring overhead be quantified?
Metrics	M5	different scenarios
	M6	configurability of the benchmark
	M7	reproducibility of benchmark results
Question	Q4	Are the benchmark results representative?
Metrics	M7	reproducibility of benchmark results
	M8	differences to other benchmarks

Monitoring Overhead

In this chapter, we provide an introduction to the concept of *monitoring overhead*. First, we introduce the general concept and the related terminology (Section 6.1). In Section 6.2, we investigate possible *causes of overhead* and sketch a methodology to quantify these causes. Finally, we discuss *further monitoring scenarios* and their associated overhead as well as overhead in resources other than execution time in Section 6.3.

If we knew what it was we were doing, it would not be called research, would it?

— Albert Einstein

6. Monitoring Overhead

Previous Publications

Parts of this chapter are already published in the following works:

1. 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
2. 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)*. ACM, June 2011, pages 197–200
3. J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In: *Multicore Software Engineering, Performance, and Tools (MSEPT)*. Springer, June 2012, pages 42–53
4. J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68
5. F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 89–98
6. J. Waller. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013
7. J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69

6.1 Monitoring Overhead

A monitored software system has to share some of its resources (e. g., CPU-time or memory) with the monitoring framework, resulting in the *probe effect*. The probe effect, often also referred to as *observer effect* [e. g., Cornelissen et al. 2009], *degradation* [e. g., Lucas 1971], *delays* [e. g., Plattner and Nievergelt 1981], *perturbation* [e. g., Hauswirth et al. 2004], *artifact* [e. g., Jain 1991], *overhead* [e. g., Eichelberger and Schmid 2014], *instrumentation uncertainty principle* [e. g., Malony et al. 1992], or (incorrectly) as *Heisenberg uncertainty principle* [e. g., Smith and Williams 2001], is the influence that measuring a system has on the behavior of the system [e. g., Gait 1986; Jain 1991; Fidge 1996; Bulej 2007; Mytkowicz et al. 2010; Marek 2014]. This influence includes changes to the probabilities of non-deterministic choices, changes in scheduling, changes in memory consumption, or changes in the timing of measured sections. Here, we take a look at the overhead causing parts of the probe effect.

Monitoring overhead is the amount of additional usage of resources by a monitored execution of a program compared to a normal (unmonitored) execution of the program. In this case, resource usage encompasses utilization of CPU, memory, I/O systems, and so on, as well as the time spent executing. Monitoring overhead in relation to execution time is the most commonly used definition of overhead. Thus, in the following, any reference to monitoring overhead concerns overhead in time, except when explicitly noted otherwise.

Following Bellavista et al. [2002] or Huang et al. [2012], we provide a simple mathematical definition of monitoring overhead. Let t_p be the normal execution time of program, i. e., without any monitoring present. Let t_m be the additional time used by added monitoring, so $t_p + t_m$ is the total execution time of the monitored program. Thus, the *percentage overhead* o is defined as $o = \frac{t_m}{t_p}$. Similarly, we can define the *monitoring percentage* m , i. e., the percentage of execution time spent monitoring, as $m = \frac{t_m}{t_p + t_m} = \frac{o}{1+o}$.

Although this definition allows for the representation of monitoring overhead by a single number, one has to consider the context of this derived value. Considering a program p with execution time $t_p = 500\mu\text{s}$ and a

6. Monitoring Overhead

monitoring system m requiring an additional overhead time $t_m = 4\mu s$, we derive an overhead of $o = 0.8\%$ and a monitoring percentage of $m = 0.8\%$. If this overhead time t_m is largely independent of the execution time t_p , for example t_p is the execution time of single method and m measures only the entry and exit times of the method, a change of t_p will have no or minimal influence on t_m , but it will have a huge impact on o and m . Thus, changing the program execution time $t_p = 1000\mu s$ without influencing t_m will result in $o = 0.4\%$ and $m = 0.4\%$. We are able to reduce the percentaged overhead as far as intended via increasing the execution time of a monitored method.

In consequence, we do not present a percentage as the summarizing result of our micro-benchmark experiments (Chapters 11 and 12). However, for specified scenarios, such as macro-benchmarks (Chapter 13), it may be useful to present the percentaged monitoring overhead, provided that the monitored application, the instrumentation and the usage profile are somehow representative for similar systems. Yet, any given percentaged monitoring overhead without any narrowly defined context has to be judged cautiously.

6.2 Causes of Monitoring Overhead

In this section, we investigate the exact causes of monitoring overhead. Furthermore, we explore techniques to quantify these causes. Specifically, we propose a separation into three different portions of monitoring overhead [Waller et al. 2014a; Waller and Hasselbring 2013a; 2012, a; van Hoorn et al. 2009b].

Our description of possible causes of monitoring overhead is based upon the Kieker framework (Section 4.5) and application-level instrumentation (Section 4.3.5). Although the actual details of the implementation of other application-level monitoring framework might differ, a similar division of overhead causes can be found.

For the sake of simplicity, we assume a `MonitoredClass` with a `monitoredMethod()`. Each execution of this method should be monitored by our monitoring framework. For this purpose, executions of a `MonitoringProbe` are interleaved with the execution of the `monitoredMethod()`. In other words,

6.2. Causes of Monitoring Overhead

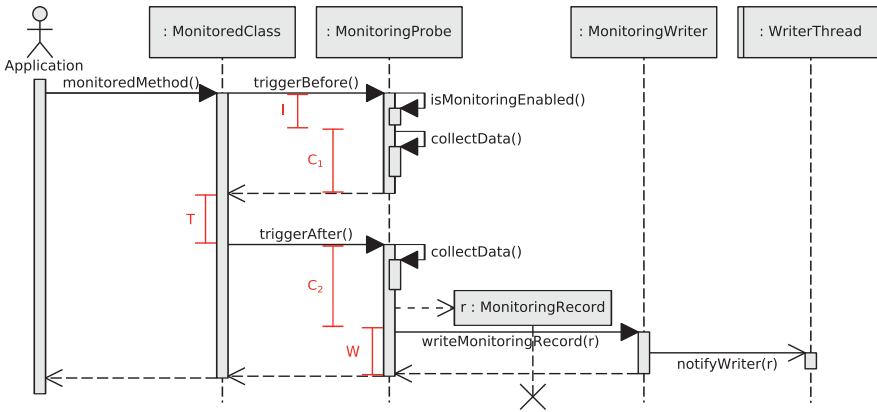


Figure 6.1. UML sequence diagram for state-based method monitoring with the Kieker framework using an asynchronous writer (based upon Waller and Hasselbring [2012a])

parts of the `MonitoringProbe` have to be executed before the actual execution of the `monitoredMethod()`, while other parts have to be executed afterwards. In the following, we name these parts `triggerBefore()` and `triggerAfter()`. The actual means of interleaving the `MonitoringProbe` with the `monitoredMethod()` are not important in the context of this section. Common possibilities are the use of manual insertion or employing aspect-oriented techniques (refer to Section 4.3.5 for details). Within the `MonitoringProbe`, the required data is collected and written into a monitoring log or stream.

In Figures 6.1 and 6.2, we present simplified UML sequence diagram [OMG 2011b] representations of the control flow for monitoring an execution of a `monitoredMethod()` with the Kieker monitoring framework. The first figure depicts a state-based method monitoring approach while the second figure depicts an event-based method monitoring approach (cf., Section 4.5.2). In both cases, an asynchronous writer is employed to write the gathered data.

In addition to these diagrams, we present the simplified Java source code for such a `MonitoringProbe` for state-based method monitoring in Listing 6.1. This code corresponds to the control flow depicted in Figure 6.1. The `triggerBefore()` and `triggerAfter()` parts of the `MonitoringProbe` are implicitly

6. Monitoring Overhead

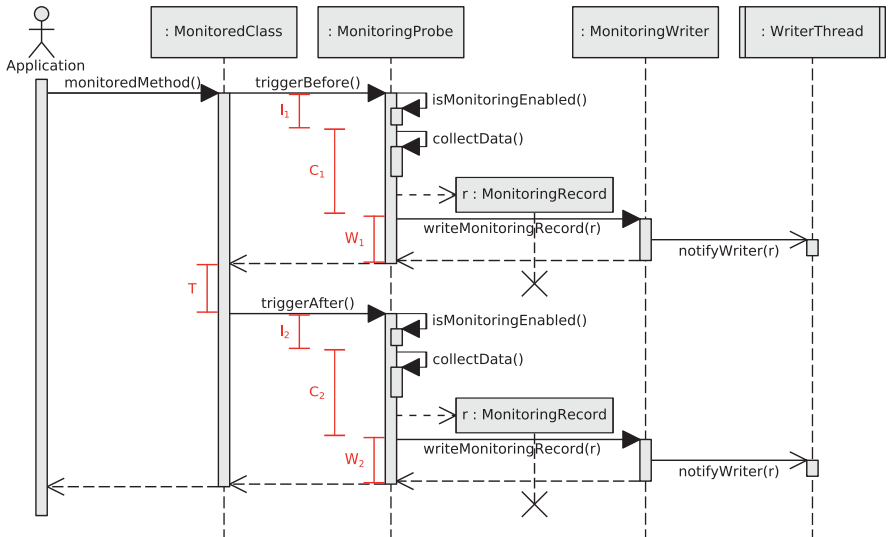


Figure 6.2. UML sequence diagram for event-based method monitoring with the Kieker framework using an asynchronous writer (based upon Waller and Hasselbring [2013a])

included within the code of the `monitoredOperationProbe()`. In addition, the `collectData()` operation is complemented by a `getTime()` operation to showcase a typical reason for two collection periods within the `MonitoringProbe`.

Regardless of the actual details of the monitoring framework and the depicted control flow in Figures 6.1 and 6.2 or in Listing 6.1, we can identify three common possible causes of overhead while monitoring an application: (I) the *instrumentation* of the `monitoredMethod()` including checks for possible deactivation of probes or monitoring, (C) the *collection* of monitoring data, e. g., operation names and timestamps, and (W) the actual *writing* of the collected data into a monitoring log or stream. These three causes and the *execution* time (T) of the application itself are illustrated in red color in the diagrams and the listing. In the following, we will further detail the execution time and the three causes of overhead, as well as the commonly executed operations of the monitoring framework.

6.2. Causes of Monitoring Overhead

Listing 6.1. Simplified Java source code for a Kieker monitoring probe

```
1 public Object monitoredOperationProbe(...) {
2     // instrumentation
3     if (!isMonitoringEnabled("monitoredOperation"))
4         return monitoredOperation(...);
5     // collect data
6     MonitoringData d = collectData();
7     long t1 = getTime();
8     // actually execute the called method
9     // and remember its return value
10    Object retval = monitoredOperation(...);
11    // collect additional data
12    long t2 = getTime();
13    Record r = new MonitoringRecord(t1, t2, d, ...);
14    // write collected data
15    // using a synchronous or asynchronous writer
16    MonitoringWriter.writeMonitoringRecord(r);
17    return retval;
18 }
```

T The actual execution time of the `monitoredMethod()`, i. e., the time spent executing the actual code of the method if no monitoring is performed, is denoted as *T*.

In Figure 6.1, Figure 6.2, and Listing 6.1, this time is annotated with a red *T*. Although sequence diagrams provide a general ordering of before and after, the depicted length of an execution carries no meaning. Thus, for reasons of space and clarity, the illustration of the actual execution time *T* in the figures is small compared to the sum of the three overhead timings. However, note that in actual systems the execution time *T* is often large compared to the sum of overhead.

In summary, *T* is the execution time of the `monitoredMethod()` without any influence of the monitoring framework.

I Before the code of the `monitoredMethod()` in the `MonitoredClass` is executed, the `triggerBefore()` part of the `MonitoringProbe` is executed. Within this part of the probe, `isMonitoringEnabled()` determines whether monitoring

6. Monitoring Overhead

is activated or deactivated for the `monitoredMethod()`. This check is used on the one hand to realize adaptive monitoring solutions [Ehlers 2012], and on the other hand to activate and deactivate the functionality of the monitoring framework as a whole. If monitoring is currently deactivated for the method, no further probe code will be executed and the control flow immediately returns to the `monitoredMethod()`. Otherwise, the execution continues with the remaining parts of the `MonitoringProbe`.

Besides these operations of the monitoring framework, I also includes any overhead caused by the instrumentation itself. For instance, when employing aspect-oriented instrumentation techniques with AspectJ (see Section 4.3.5), similar calls to our `triggerBefore()` are internally performed.

In Figure 6.1, Figure 6.2, and Listing 6.1, I indicates the execution time for the instrumentation of the method including the time required to determine whether monitoring of this method is activated or deactivated.

Depending on the implementation of the `MonitoringProbe`, for instance in the case of our diagram for event-based method monitoring (Figure 6.2), I can be partitioned into two parts with $I = I_1 + I_2$. This is typically the case if separate instrumentation points are employed, e. g., before and after advices with the DiSL framework instead of a single around advice in the case of the AspectJ framework (see Section 4.3.5). The second instrumentation part I_2 contains the execution of the `triggerAfter()` part of the `MonitoringProbe`, the potential additional call to `isMonitoringEnabled()`, and the associated overhead of the employed instrumentation technique.

In summary, I is the amount of overhead caused by the used instrumentation technique in addition to the overhead of checking whether monitoring is actually enabled for the `monitoredMethod()`.

- C If monitoring of the `monitoredMethod()` is active (as determined in the previous step with the `isMonitoringEnabled()` operation), the `MonitoringProbe` collects some initial data with its `collectData()` method, such as the current time and the operation signature. When the execution of the actual code of the `monitoredMethod()` finishes with activated monitoring, the `triggerAfter()` part of the `MonitoringProbe` is executed. Then, some additional data, such as the response time or the return values of the method, is collected.

6.2. Causes of Monitoring Overhead

The collected data is stored in so-called `MonitoringRecords`. These records are simple data structures used to store and transfer collected monitoring data. In the case of Kieker, the most commonly used records are the `OperationExecutionRecord` (e.g., for state-based method monitoring as depicted in Figure 6.1) or the combination of `BeforeOperationEvent` and `AfterOperationEvent` records (e.g., for event-based method monitoring as depicted in Figure 6.2). Depending on the implementation of the `MonitoringProbe` and the employed monitoring approach, one or more `MonitoringRecords` are created in memory and filled with the previously collected data.

In Figure 6.1, Figure 6.2, and Listing 6.1, the time needed to collect data of the `monitoredMethod()` and to create the required `MonitoringRecords` in main memory is $C = C_1 + C_2$. Similar to the division of I into two parts, C is split into C_1 for the overhead of the `triggerBefore()` part of the `MonitoringProbe` before the execution of the `monitoredMethod()` and into C_2 for the overhead of the `triggerAfter()` part of the `MonitoringProbe` after the execution of the `monitoredMethod()`.

In summary, C is the amount of overhead caused by collecting the monitoring data and by storing it in a memory record.

- W Each created and filled `MonitoringRecord` r is forwarded to a `MonitoringWriter` with the framework operation `writeMonitoringData(r)`. The `MonitoringWriter` either stores the collected data in an internal buffer, that is processed asynchronously by the `WriterThread` into the `Monitoring Log/Stream`, or it synchronously writes the collected data directly into the `Monitoring Log/Stream`.

Thus, W is the amount of overhead for writing the `MonitoringRecord` r . That is, in the case of a synchronous writer, the time spent writing the collected data to the `Monitoring Log/Stream`. In the case of an asynchronous writer, it is the time spent placing the data in an internal buffer via `notifyWriter(r)` and using an asynchronous `WriterThread` to write the collected data into the `Monitoring Log/Stream`.

Depending on the underlying hardware and software infrastructure and the available resources, the actual writing within this additional thread might have more or less influence on the results. For instance,

6. Monitoring Overhead

in cases where records are collected faster than they are written, the internal 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 . An alternative would be discarding additional records. Thus, the resulting monitoring traces would be incomplete, hindering a subsequent analysis of the observed traces. In other cases, with sufficient resources available, the additional overhead of the writer might be barely noticeable (see Section 11.4).

In Figure 6.1 and Figure 6.2, only asynchronous writing is illustrated. Due to the limited scope of the listing, Listing 6.1 can represent either synchronous or asynchronous writing. In Figure 6.1 and Listing 6.1, only a single source of writing overhead W is present, while a division of W into $W_1 + W_2$ similar to the division of I for the `triggerBefore()` and `triggerAfter()` parts of the `MonitoringProbe` is presented in Figure 6.2.

In summary, W is the amount of overhead caused by writing the collected monitoring data into a monitoring log or into a monitoring stream.

To summarize: in addition to the normal execution time of the monitored `Method()` T , there are three possible portions of overhead: (1) the instrumentation of the method and the check for activation of the probe (I), (2) the collection of data (C), and (3) the writing of collected data (W).

6.3 Further Monitoring Scenarios and Overhead in Resource Usage

Our previous discussion of causes of monitoring overhead is focused on overhead in relation to the execution time and on the scenario of monitoring method executions. However, our approach and concepts can easily be transferred and used for overhead in other resources as well as for other monitoring scenarios.

With our presented techniques, we can investigate monitoring scenarios besides monitoring method executions. For instance, in the case of monitoring method calls instead of executions, we usually collect a lot of additional

6.3. Further Monitoring Scenarios and Overhead in Resource Usage

monitoring data, i. e., calls from monitored methods to unmonitored ones, e. g., API calls, are gathered as well. Although this additional data can provide important details for a subsequent analysis [Knoche et al. 2012], the monitoring overhead is usually very severe. Further typical scenarios include monitoring of concurrency or resource utilization. In all cases, the resulting control flows (sequence diagrams) are similar to the ones already presented in this chapter (Figures 6.1 and 6.2). Consequently, we can again determine the same three causes of monitoring overhead.

In the context of this thesis, our focus lies on monitoring method executions. However, our findings can usually be transferred to these additional scenarios. For instance, monitoring method calls is similar to monitoring method executions with a higher load.

Furthermore, the additional utilization of resources due to monitoring, e. g., the utilization of CPU, memory, I/O systems, and so on, can be measured instead of or in addition to the measurements of the execution time. More indirect cost can be measured as well, e. g., lock contention, garbage collection calls, page faults, cache or branch prediction faults, or file system cache misses. Similar to overhead in relation to execution time, we can determine the same three causes of overhead: instrumentation (*I*), collection (*C*), and writing (*W*).

Although overhead in resources besides execution time can play an important role, our focus is on the measured response times of the monitored application. In most cases, e. g., memory, additional resource usage can be solved by providing the additional resources. In the case of execution time, this is usually not as easy. However, in the detailed raw data packages of our experiments, we usually also provide data on the resource utilization during our benchmarks. Furthermore, we mention selected cases of overhead in resource usage with our experiments, e. g., usage of the I/O system to write monitoring logs.

Instead of the flat monitoring cost imposed per `MonitoringRecord`, i. e., the actual change in a monitored method's *response time*, we can also investigate the monitoring *throughput*, i. e., the number of `MonitoringRecords` sent and received per second. The response time and the monitoring throughput are related: improving one measure usually also improves the other one.

6. Monitoring Overhead

However, with asynchronous monitoring writers (as in the case of most of our experiments) the relationship between throughput and response time can become less obvious.

In order to measure the maximal monitoring throughput, it is sufficient to minimize T while repeatedly calling the `monitoredMethod()`. Thus `MonitoringRecords` are produced and written as fast as possible, resulting in maximal throughput. As long as the actual `WriterThread` is capable of receiving and writing the records as fast as they are produced (see description of W above), it has no additional influence on the monitored method's response time. When our experiments reach the `WriterThread`'s capacity, the buffer used to exchange `MonitoringRecords` between the `MonitoringWriter` and the `WriterThread` blocks, resulting in an increase of the monitored method's response time without further affecting the throughput.

Benchmark Engineering Methodology

In this chapter, we introduce the *field of benchmark engineering* (Section 7.1) and provide our *benchmark engineering methodology*. This methodology consists of three phases: Phase 1 is concerned with the *design of benchmarks* (Section 7.2), phase 2 is concerned with the *execution of benchmarks* (Section 7.3), and phase 3 is concerned with the *analysis and presentation of benchmark results* (Section 7.4). Finally, we discuss *special challenges when benchmarking Java applications* (Section 7.5).

The trouble with using a benchmark program is that so many things can go wrong along the way.

— Bob Colwell, At Random column of IEEE computer

7. Benchmark Engineering Methodology

Previous Publications

Parts of this chapter are already published in the following works:

1. *J. Waller* and *W. Hasselbring*. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68
2. *J. Waller*. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013
3. *J. Waller*, *F. Fittkau*, and *W. Hasselbring*. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69

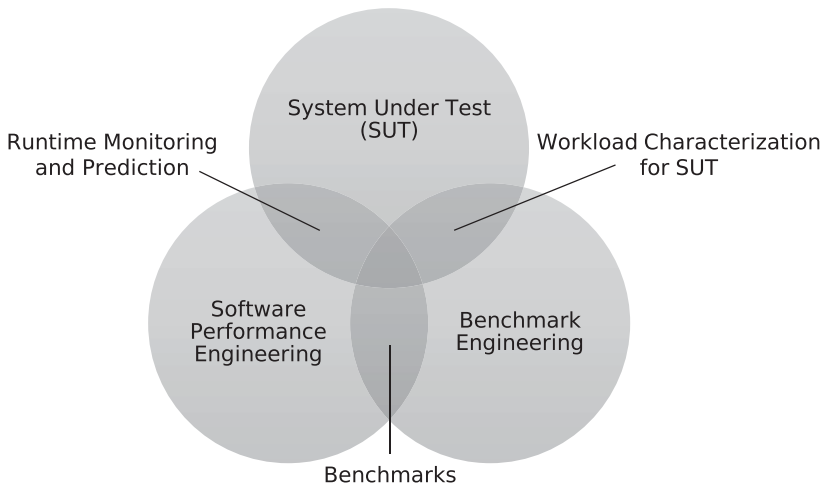


Figure 7.1. Classification of benchmark engineering (adapted from Sachs [2011])

7.1 Benchmark Engineering

Several authors [e.g., Hinnant 1988; Price 1989; Sachs 2011; Folkerts et al. 2012; Vieira et al. 2012] have recognized the lack of an established methodology for developing benchmarks. Furthermore, besides many authors describing specific benchmarks, only a few publications [e.g., Gray 1993; Huppler 2009] are focused on such a methodology.

According to Sachs [2011], 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.

A classification of benchmark engineering in the field of software engineering is presented in Figure 7.1. Benchmark engineering is dependant on two related fields: 1) the field of Software Performance Engineering (SPE) and 2) the field of the actual System Under Test (SUT) or the class of SUTs a benchmark is designed for.

7. Benchmark Engineering Methodology



Figure 7.2. The three phases of a benchmark engineering methodology [Waller and Hasselbring 2013a]

As mentioned in Section 2.3, the actual benchmarks are a part of SPE and rely on several established runtime measurement and prediction techniques. These techniques are usually influenced by the respective SUT. Furthermore, the SUT influences the used workload characterization for the benchmark.

Besides these overlapping areas, a benchmark engineering methodology consists of several individual components. Generally speaking, it can be split into three phases, as depicted in Figure 7.2, each with its own set of requirements:

1. The first phase is the actual *design and the 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 guidelines 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 additional requirements as solutions to common problems with the respective benchmarks, e. g., the need for robust benchmark executions.
3. The third phase is the *analysis and presentation of benchmark results*. Here, the gathered raw performance data is 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. To ensure replicability, it should additionally provide guidelines for the description of the performed benchmark experiments.

7.2. Phase 1: Design and Implementation

It is of note that the first phase is usually only executed once, while the second and third phase are repeated for each experiment with the benchmark developed in the first phase.

In the following sections, we will provide further details on the concepts and activities of each phase independent from any actual SUT. In particular, we will provide sets of requirements for each phase that are common to most benchmark engineering methodologies. A specific benchmark engineering methodology targeting benchmarks for monitoring systems will be presented in Chapter 8.

7.2 Phase 1: Design and Implementation

As mentioned before, the first phase of a benchmark engineering methodology (see Figure 7.2) is concerned with the *design and implementation* of benchmarks.

In the following subsection, we present eight primary design requirements and considerations common for most benchmarks. A good benchmark should adhere to all of these requirements. In addition to these primary requirements, we also include two secondary requirements which can be beneficial to benchmark design, but that are not necessarily required for good benchmarks.

In the second subsection (Section 7.2.2), we highlight additional design decisions that distinguish benchmark engineering projects from more general software engineering projects.

7.2.1 Benchmark Design Requirements

Most published descriptions of benchmarks include a set of common and more or less general design requirements the specific benchmark adheres to. Similar sets are presented by authors focusing on benchmark engineering. In this section, we present our own set of eight primary benchmark design requirements. The first three requirements (*representativeness*, *repeatability*, and *robustness*) are the most important ones and should be the main focus during benchmark engineering. The final five requirements can be relaxed,

7. Benchmark Engineering Methodology

Table 7.1. Benchmark design requirements in the literature

Publication	R1	R2	R3	R4	R5	R6	R7	R8	S1	S2
Joslin [1965]	✓						✓			
Lucas [1971]	✓						✓		✓	
Kuck and Sameh [1987]	✓						✓			
Dongarra et al. [1987]	✓	✓		✓			✓			
Hinnant [1988]	✓	✓	✓		✓					
Price [1989]	✓		✓	✓					✓	
Saavedra-Barrera et al. [1989]	✓		✓						✓	
Cybenko et al. [1990]	✓	✓	✓			✓	✓	✓		
Weicker [1990]	✓		✓		✓		✓			✓
Jain [1991]	✓		✓	✓	✓		✓			
Gray [1993]	✓				✓	✓		✓		✓
Reussner et al. [1998]	✓	✓			✓		✓	✓		
Vokolos and Weyuker [1998]	✓		✓		✓					
Seltzer et al. [1999]	✓						✓	✓		✓
Kähkipuro et al. [1999]	✓	✓	✓		✓	✓				
Bull et al. [2000]	✓		✓		✓			✓		
Lilja [2000]	✓				✓			✓		
Smith and Williams [2001, 2003]	✓	✓								
Carzaniga and Wolf [2002]	✓									✓
Menascé [2002]	✓	✓					✓	✓		
Buble et al. [2003]			✓	✓						
Sim et al. [2003]	✓	✓		✓	✓	✓		✓		✓
$\sum 22$	21	9	10	4	10	4	10	8	6	2

Req.	Description	sec. Req.	Description
R1	Representative / Relevant	S1	Specific
R2	Repeatable	S2	Accessible / Affordable
R3	Robust		
R4	Fair		
R5	Simple		
R6	Scalable		
R7	Comprehensive		
R8	Portable / Configurable		

7.2. Phase 1: Design and Implementation

Table 7.2. Benchmark design requirements in the literature (cont.)

Publication	R1	R2	R3	R4	R5	R6	R7	R8	S1	S2
Denaro et al. [2004]	✓		✓							
Kalibera et al. [2004]		✓	✓		✓	✓		✓		
Bulej et al. [2005]	✓	✓	✓							
Brebner et al. [2005]	✓	✓	✓		✓					
John [2005a]	✓				✓					
Araiza et al. [2005]	✓				✓			✓		
Kounev [2005]	✓	✓	✓	✓		✓	✓			
Kalibera [2006]	✓		✓		✓					
Blackburn et al. [2006]	✓		✓		✓	✓	✓			✓
Adamson et al. [2007]	✓		✓		✓					
Georges et al. [2007]			✓	✓						
Mytkowicz et al. [2008a,b]	✓	✓	✓							
Lange [2009]	✓				✓					
Binnig et al. [2009]	✓						✓			✓
Huppler [2009]	✓	✓	✓	✓	✓			✓		✓
Kuperberg et al. [2010]					✓		✓	✓		
Mytkowicz et al. [2010]	✓		✓	✓						
Sachs [2011]	✓	✓	✓	✓		✓	✓	✓		
Gil et al. [2011]			✓	✓			✓	✓		
Blackburn et al. [2012]	✓	✓	✓							
Folkerts et al. [2012]	✓	✓		✓	✓	✓	✓	✓		✓
Vieira et al. [2012]	✓	✓		✓	✓	✓	✓	✓		✓
SPEC [2013a]			✓	✓						
Weiss et al. [2013]	✓		✓							
Kalibera and Jones [2013]			✓	✓						
Saller et al. [2013]	✓	✓				✓		✓		
Wang et al. [2014]	✓	✓		✓	✓	✓	✓			✓
$\sum 27$	21	16	18	7	13	8	9	9	2	4

Publications	R1	R2	R3	R4	R5	R6	R7	R8	S1	S2
22 Publications before 2004	21	9	10	4	10	4	10	8	6	2
27 Publications after 2004	21	16	18	7	13	8	9	9	2	4
$\sum 49$	42	25	28	11	23	12	19	17	8	6

7. Benchmark Engineering Methodology

but should also always be considered and documented. It is of note that some of the requirements can overlap or influence other requirements. These cases are mentioned in our detailed description below.

- R1: Representative / Relevant
- R2: Repeatable
- R3: Robust
- R4: Fair
- R5: Simple
- R6: Scalable
- R7: Comprehensive
- R8: Portable / Configurable

This set is designed to incorporate the most common published benchmark requirements. Refer to Tables 7.1 and 7.2 for an overview on published design requirements. Although our requirements set is quite comprehensive, two secondary requirements are not included. These additional requirements are briefly discussed at the end of this section.

- S1: Specific
- S2: Accessible / Affordable

R1: Representative / Relevant

Representativeness or relevance is the most common and the most important requirement for benchmarks. A benchmark has to be representative of the measured (benchmarked) usage of the SUT. Otherwise, its results would be of little value. There are two parts to representativeness during benchmark design: (1) selection of a representative workload and (2) relevant usage of software and hardware features.

1) *Representative workload*: The designed workload has to be representative of the intended system. For instance, for a macro-benchmark modeling buyers interacting with a supermarket, the workload should model the actual behavior of buyers. The better this workload is designed, the closer

7.2. Phase 1: Design and Implementation

the benchmark will simulate the real system. On the other hand, for a micro-benchmark measuring the duration of a single operation call, a workload repeatedly executing this call as fast as possible might be sufficient.

2) *Relevant usage of software and hardware features*: The usage of software and hardware features has to be relevant for the intended benchmark. For example, a supermarket macro-benchmark to test a message-oriented middleware should use the same kinds of messages a real supermarket software uses. Similarly, access to other software or hardware features should mimic the intended behavior. To continue our micro-benchmark example, the benchmark of a single operation call should execute this call instead of other software features.

The purpose of this requirement is to allow for the user of the benchmark to draw conclusions regarding actual systems.

R2: Repeatable

A benchmark should be repeatable. That is, it should be possible to run the benchmark again and achieve the same (or very similar) results. Again, this has two implications on the benchmark design: (1) design to produce deterministic results and (2) design for repeatability.

1) *Deterministic results*: A benchmark should be designed to provide deterministic runs and results. The fulfillment of this requirement is not always entirely possible and could usually be somewhat relaxed. For example, workloads are often defined probabilistically, e. g., using probability distributions or using Markov chains [e. g., van Hoorn et al. 2008]. Furthermore, several operations can have lasting side-effects that influence future runs, e. g., writing to a database. However, resets to an initial consistent state, sufficient run repetitions, and long run times can be used to achieve adequately deterministic results. The necessary steps have to be considered and documented during the benchmark design process.

2) *Design for repeatability*: Besides providing deterministic results, a benchmark should be designed to publish enough details to repeat the benchmarking experiment. Although this requirement is mostly concerned with the third phase of our benchmark engineering methodology, the benchmark design should already consider this requirement. For instance,

7. Benchmark Engineering Methodology

the benchmark has to adapt to different environments while providing documentation of the influence of the different environmental parameters (see also R8: Portable / Configurable).

The purpose of this requirement is to achieve comparability and credibility for benchmark results.

R3: Robust

A robust benchmark is not influenced by factors outside of the benchmark's control. This requirement is related to the demand for deterministic results (R2). With robust benchmarks, a typical source of non-determinism is eliminated. An example for such a robustness requirement present in almost all benchmarks is the need for the SUT to be held idle besides executing the benchmark. Further common threats to robustness are, for instance, the influence of Just-In-Time compilers or caching effects of the operating system or hardware.

An additional aspect of a robust benchmark is its robustness against perturbation by its own measurements. This is related to the requirement for simple measures (R5).

In summary, this robustness requirement is on the one hand very encompassing and on the other hand mostly concerned with the execution of the benchmark. Thus, we will break it down further in our description of execution time requirements (Section 7.3). But, as most threats to robustness also have to be considered during design time, we include this requirement as a benchmark design requirement as well.

R4: Fair

A benchmark should be fair rather than being biased for a specific system or environment. That is, although a benchmark might be written for a specific product, it usually measures the performance of said product within a specific environment. In this case, different environments should still provide comparable and fair results. For example, a general benchmark for database systems should not utilize functions specific for a single database. On the other hand, a benchmark to determine the performance of a specific

7.2. Phase 1: Design and Implementation

database on different operating systems can utilize such functions. However, such a benchmark must not be biased towards a specific operating system.

R5: Simple

A benchmark should be designed to be simple and clear. First, it should be simple to configure and to execute the benchmark. Second, it should be simple to understand what exactly is measured by the benchmark and how to interpret the benchmark results. Clarity in understanding the measures is often enhanced by using simple and clearly defined measures. Additionally, the use of simple measures often leads to less perturbation within the benchmark due to these measurements.

R6: Scalable

Scalability is often partitioned into (1) horizontal and (2) vertical scalability.

1) *Horizontal scalability*: Horizontal scaling is concerned with scaling the system itself. For example, considering the already mentioned supermarket benchmark, the benchmark could be split across several systems, e.g., by using a cloud infrastructure, to increase the horizontal scale. The workload is usually similarly scaled to keep the effective workload per component constant.

2) *Vertical scalability*: In contrast, vertical scaling increases or decreases the workload assigned to the benchmark with a fixed system configuration. This allows for the performance evaluations of wide ranges of target systems and environments.

R7: Comprehensive

Comprehensiveness is a benchmark design requirement similar to relevance. Besides being representative and relevant, the benchmark also needs to cover all important parts of the SUT. A benchmark with maximal comprehensiveness would cover all possible parts of the system and the environment, thus it would probably have low relevance. Therefore, a good benchmark has to find a balance between relevance and comprehensiveness.

7. Benchmark Engineering Methodology

R8: Portable / Configurable

A portable or configurable benchmark can be executed on different environments or for different SUTs. For instance, a Java benchmark is inherently more portable than a benchmark compiled into native machine code. On the other hand, portable code might not always be representative (R1). A typical case for configurability is the workload, e. g., to support scaling the benchmark (R6). Furthermore, a design for repeatability (R2) might require certain configuration possibilities.

Secondary Requirements

Besides our eight primary requirements (R1–R8), several authors propose two additional secondary requirements (S1 and S2). Refer to Tables 7.1 and 7.2 for the use of these secondary requirements in the literature.

S1: Specific This requirement is in parts contrary to the portability requirement (R8). A (domain-)specific benchmark is designed for special SUT or class of SUTs. Although mentioned by several authors, we do not include this requirement into our set of primary requirements, as it is too specific. Additionally, this requirement is only included by two of the more recent papers. In general, benchmarks are not required to be specific, but it is rather desirable to have portable ones (R8).

S2: Accessible / Affordable A benchmark should be accessible and affordable. On the one hand, the benchmark should be publicly available, for example as a downloadable open-source system. On the other hand, it should be cheap and easy to employ and to understand by its users. The last parts of this additional requirement are already included in our simpleness requirement (R5).

Although high accessibility and affordability can improve a benchmark's acceptance within its community (see Section 3.3), it is no actual requirement in designing benchmarks. For instance, most benchmarks published by the SPEC are rather expensive and require a certain amount of familiarization from the user. Nevertheless, these benchmarks are often employed within

7.2. Phase 1: Design and Implementation

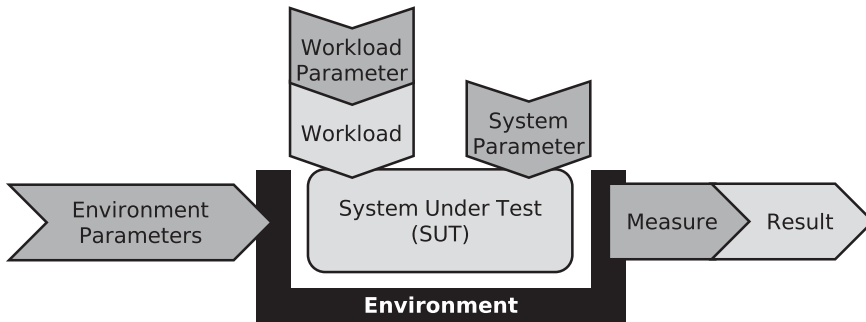


Figure 7.3. A generic model of a benchmark (based upon Saller et al. [2013])

their respective communities. Thus, we do not include this requirement into our primary list but we encourage the production of open and accessible benchmarks.

7.2.2 The Benchmark Engineering Process

The software development process used within a benchmark engineering methodology can employ any common software development model, as long as the benchmark design requirements are respected. Therefore, we only highlight additional design decisions specific for benchmark design, i. e., choosing the workload and the measure.

A benchmark is designed to test a SUT within a specific environment. For instance, consider benchmarking a specific JVM running within an OS on a hardware platform. The OS and the hardware platform are parts of the environment, while the JVM is the SUT.

A generic benchmark model including the environment and its influencing parameters has been developed by Saller et al. [2013]. A schematic representation is visualized in Figure 7.3. The main components of the benchmark are the *workload* that drives the SUT, the *SUT* itself, and the *environment* the SUT is embedded within. All three components are influenced by their respective *parameters*. In addition, we add a fourth component to this generic model: the *measure* used to compute the *benchmark results*.

7. Benchmark Engineering Methodology

Usually, the SUT and the environment are given when developing a new benchmark. Thus, the main focus of the design and implementation phase is on the intended workload and on the used measure [e. g., Menascé 2002]. In the following, we provide further details on benchmark workloads and measures.

Workload

The workload is often the most important component of a benchmark. Hence, most of our design requirements directly influence its selection. The most important one is the requirement for a representative and relevant workload (R1). Additionally, a chosen workload should be deterministic (R2), scalable (R6), and comprehensive (R7). Finally, the workload should not bias certain SUTs (R4), be easy to comprehend (R5), and portable across environments and SUTs (R8).

Similar to the division of benchmarks into micro- and macro-benchmarks (see Section 3.2), workloads are often categorized as synthetic or real-world workloads [e. g., Saller et al. 2013].

Synthetic workloads are artificial and target a specific subset of the possible workloads of the intended SUTs. Thus, their fulfilment of the representative and relevance requirement (R1) is often a tradeoff with the portability requirement (R8): They can be very relevant for a narrowly defined SUT, as commonly found in micro-benchmarks, but porting the benchmark to other systems often removes this relevance. On the other hand, they are often easily scalable (R6), simple (R5), and can be very comprehensive (R7) within their intended SUT. Hence, they are usually used in connection with micro-benchmarks and they are the main reason for these benchmarks' limited real-world applicability.

Real-world workloads are usually based on real-world data which is, for instance, gathered by monitoring the actual user behavior of a target system. This information can be transformed into workloads with the help of domain-experts (see Figure 2.3 and Section 2.3.1). Real-world workloads are usually employed within macro-benchmarks and lead to these benchmarks' higher representativeness and applicability (R1). The downsides of these more realistic workloads are higher complexity (R5) and problems with

7.2. Phase 1: Design and Implementation

the comprehensiveness (R7) of the workload: While synthetic workloads can explicitly target corner cases not common in the typical system usage, real-world workloads usually focus on the main usage patterns. Due to the nature of real-world workloads, probability distributions are often used to describe the simulated user's behavior when interacting with the system. This can lead to lower repeatability and less deterministic behavior (R2).

Both kinds of workload can either be minimal, average, or stress workloads. For additional details on designing workloads refer to the existing literature [e. g., Jain 1991; Vokolos and Weyuker 1998; Menascé et al. 1999; Smith and Williams 2001; Molyneaux 2009; Saller et al. 2013].

Measure

The second important decision, separating benchmark engineering projects from other software engineering projects, is the choice of the used measure. As mentioned in Section 2.2, there are, generally speaking, three different kinds of measures: response time, throughput, and resource utilization. All three are common within benchmarks and in an optimal case, a benchmark should use all applicable types.

The main concern with adding all possible measures is the probe effect (see Section 6.1). Each measurement has an inherent cost and influence on the measured system. An additional source of imprecision is the measurement accuracy of the system. For instance, depending on the hardware infrastructure and the used operating system, timer requests might be resolved with low accuracy by the Time Stamp Counter (TSC) or with high accuracy by the High Precision Event Timer (HPET). In order to produce acceptable and meaningful results, the introduced measurement error by probe effect and accuracy must be small in comparison to the measured values. This corresponds to our robustness benchmark design requirement (R3).

The second requirement concerning the choice of measure is the need for simplicity (R5). The chosen measure should be common, well understood, and easy to gather. Typical examples are average response times, calls per second, or average utilization. Rather than combining these measurement results within the benchmark into a single number, it is often better to gather

7. Benchmark Engineering Methodology

and report all collected data, enabling more detailed analyses in the third phase (see Section 7.4). Especially the combination of multiple measures, e. g., response times with CPU utilization, can improve the understanding of results.

For further details on choosing measures refer to the existing literature [e. g., Saavedra-Barrera et al. 1989; Jain 1991; Fenton and Pfleeger 1998; Smith and Williams 2001].

7.3 Phase 2: Execution

The second phase of a benchmark engineering methodology (see Figure 7.2) is concerned with the actual *execution of the benchmark* that was designed and developed in the first phase. In addition to the eight primary benchmark design requirements, we present four primary benchmark execution requirements:

- R9: Robust Execution
- R10: Repeated Executions
- R11: Warm-up / Steady State
- R12: Idle Environment

This set of requirements incorporates common published hints for the execution of benchmarks. Refer to Tables 7.3 and 7.4 for an overview on the use of these requirements in the literature. It is worth noting that most authors focus on the design of benchmarks rather than their execution. Even most papers listed in the two tables provide only brief hints or remarks concerning the execution. Only within more recent papers authors start to pay attention to the execution of benchmarks and the associated challenges present during runtime.

All four benchmark execution requirements are usually also considered during the design time in the context of the robustness requirement (R3). Furthermore, the goal of all four execution requirements is an increased repeatability (R2) of the benchmark.

R9: Robust Execution

The main concern during the execution of the benchmark is *robustness* against any kind of unintended perturbation influencing the benchmark results. Most possible perturbation sources should have been already considered during the design phase (R3). However, their actual avoidance, as well as further influences of the environment, must be controlled during the execution phase.

The three further requirements (R10–R12) are strictly speaking common ways of dealing with three typical kinds of perturbations present during benchmarking. However, due to their common nature, they are important enough to warrant their role as own benchmark execution requirements. Thus, we do not include R10–R12 into this requirement (R9). So, this robustness requirement is concerned with more general and additional techniques to avoid perturbation. Refer to the literature for examples of these techniques.

R10: Repeated Executions

A common requirement of benchmark executions is the need for multiple repetitions. There are two different approaches to repeated executions that are usually used in combination. First, the measurement within the benchmarking experiment should be repeated multiple times. Second, the experiment itself should be repeated. These repetitions of the experiment are often performed within the same environment, but different environments can be used to even their influence on the results. However, parameters for the environment and the workload should remain constant.

Ideally, every measurement of both kinds of repetitions should be recorded and later aggregated and analyzed with the help of appropriate statistical methods during the third benchmark engineering phase (see Section 7.4). An early aggregation and analysis during the benchmark execution can lead to loss of information and, depending on the used statistical methods, to additional perturbation. Similarly, the recording of the additional measurement data should not induce further perturbation.

7. Benchmark Engineering Methodology

Table 7.3. Benchmark execution requirements in the literature

Publication	R9	R10	R11	R12
Hinnant [1988]	✓			✓
Price [1989]	✓			
Saavedra-Barrera et al. [1989]	✓	✓		
Cybenko et al. [1990]		✓		
Weicker [1990]	✓			
Jain [1991]	✓			
Reussner et al. [1998]		✓	✓	
Vokolos and Weyuker [1998]				✓
Kähkipuro et al. [1999]	✓			
Bull et al. [2000]	✓			
Buble et al. [2003]	✓		✓	
Σ 11	8	3	2	2

Requirement	Description
R9	Robust Execution
R10	Repeated Executions
R11	Warm-up / Steady State
R12	Idle Environment

7.3. Phase 2: Execution

Table 7.4. Benchmark execution requirements in the literature (cont.)

Publication	R9	R10	R11	R12
Denaro et al. [2004]		✓		
Kalibera et al. [2004]	✓			
Bulej et al. [2005]	✓		✓	
Brebner et al. [2005]	✓		✓	
Kounev [2005]	✓	✓	✓	✓
Kalibera [2006]	✓	✓	✓	
Blackburn et al. [2006]		✓	✓	
Adamson et al. [2007]	✓		✓	
Georges et al. [2007]	✓	✓	✓	
Mytkowicz et al. [2008a,b]	✓	✓		✓
Huppler [2009]	✓			
Kuperberg et al. [2010]		✓	✓	
Mytkowicz et al. [2010]		✓	✓	✓
Sachs [2011]			✓	
Gil et al. [2011]	✓	✓	✓	✓
Blackburn et al. [2012]	✓			
SPEC [2013a]			✓	
Weiss et al. [2013]		✓		
Kalibera and Jones [2013]	✓	✓	✓	
Ortin et al. [2014]		✓	✓	
$\sum 20$	12	12	14	4

Publications	R9	R10	R11	R12
11 Publications before 2004	8	3	2	2
20 Publications after 2004	12	12	14	4
$\sum 31$	20	15	16	6

7. Benchmark Engineering Methodology

R11: Warm-up / Steady State

The next common requirement of benchmark executions is the division of the actual benchmark execution into a warm-up period and a steady state period. Kalibera and Jones [2013] propose a further break-down of the steady state into an initialized state and into an independent state.

Initially, each benchmark execution is in its *warm-up period*. That is, the measurements of the benchmark are strongly influenced by other tasks, such as class loading, initialization of data, or just-in-time compilation. Thus, the resulting measurements are typically highly erratic.

As soon as these tasks are finished, they no longer influence the measurement. Thus, the benchmark execution reaches its *initialized state*. This is a necessary condition for being considered a steady state. However, there might still be an ongoing trend in the measurements, e. g., caused by a database slowly filling and thus degrading the performance.

An *independent state* is reached when each measurement is independent from the previous ones. That is, remaining factors are eliminated and each measurement produces stable results. Obviously, an independent state is the ideal condition for steady state measurements. However, not every benchmark actually reaches such an independent state in a reasonable time. In these cases, it is usually sufficient to achieve an initialized state.

Ideally, the warm-up period and the steady state periods execute identical code, i. e., the same measurements are performed in each period. Thus, the accidental introduction of a new warm-up period for the added measurements can be avoided. The actual split between warm-up and steady state can be determined during the data analysis in the third phase (see Section 7.4). Then, the gathered measurements outside of the steady state get discarded before further analyses are applied. Nevertheless, it is imperative to have a sufficiently long steady state for several repeated measurements (see R10).

R12: Idle Environment

Our final benchmark execution requirement is the need for an idle environment. That is, the environment the SUT is embedded in should be minimal and be used exclusively by the benchmark. For instance, no software tasks

7.4. Phase 3: Analysis and Presentation

should be running that are not directly related to the benchmark or the SUT. The aim of this requirement is to minimize external and unintended perturbation of the benchmark results.

A contrary approach to using an idle environment is the use of a realistic background workload. This approach can improve the relevance of the benchmark, as it relates more closely to an intended real-world system. However, it is usually hard to determine such a realistic background workload. Furthermore, the influence of this additional perturbation is dependent on the used environment. Thus, the repeatability and comparability of the benchmark can be reduced.

In our opinion, the benefits of an idle environment (less perturbation, better repeatability) outweigh the benefits of a realistic background workload (higher real-world applicability) for most benchmarks. This is especially true in the case of micro-benchmarks.

7.4 Phase 3: Analysis and Presentation

The third phase of a benchmark engineering methodology (see Figure 7.2) is the actual *analysis and presentation of the benchmark* that was designed, developed, and executed in the previous phases. The respective benchmark analysis and presentation requirements for this phase are presented in the following subsection (Section 7.4.1). The second subsection (Section 7.4.2) further details the good scientific practice for the actual publication of benchmark results, i. e., the application of the respective requirements.

7.4.1 Benchmark Analysis and Presentation Requirements

In addition to the eight benchmark design requirements and the four benchmark execution requirements, we present three primary benchmark analysis requirements:

- R13: Statistical Analysis
- R14: Reporting
- R15: Validation

7. Benchmark Engineering Methodology

Table 7.5. Benchmark analysis requirements in the literature

Publication	R13	R14	R15	S3
Kuck and Sameh [1987]				✓
Dongarra et al. [1987]	✓			
Hinnant [1988]	✓	✓	✓	
Price [1989]			✓	
Saavedra-Barrera et al. [1989]	✓			
Cybenko et al. [1990]	✓			✓
Jain [1991]	✓	✓		
Reussner et al. [1998]		✓		✓
Kähkipuro et al. [1999]		✓		
Bull et al. [2000]		✓		
Lilja [2000]	✓			
Buble et al. [2003]	✓	✓		
Σ 12	7	6	2	3

Req.	Description	add. Req.	Description
R13	Statistical Analysis	S3	Public Results Database
R14	Reporting		
R15	Validation		

7.4. Phase 3: Analysis and Presentation

Table 7.6. Benchmark analysis requirements in the literature (cont.)

Publication	R13	R14	R15	S3
Denaro et al. [2004]	✓	✓		
Kalibera et al. [2004]				✓
Bulej et al. [2005]	✓			
Brebner et al. [2005]		✓		
Kounev [2005]		✓		✓
Blackburn et al. [2006]		✓		
Georges et al. [2007]	✓	✓		
Mytkowicz et al. [2008a,b]	✓	✓	✓	
Huppler [2009]			✓	
Kuperberg et al. [2010]	✓	✓		
Mytkowicz et al. [2010]	✓	✓		
Sachs [2011]	✓	✓	✓	
Blackburn et al. [2012]	✓	✓		
Folkerts et al. [2012]		✓		
Vieira et al. [2012]		✓	✓	
SPEC [2013a]	✓	✓		✓
Kalibera and Jones [2013]	✓	✓		
Saller et al. [2013]	✓	✓		
Ortin et al. [2014]	✓	✓	✓	
\sum 19	12	16	5	3

Publications	R13	R14	R15	S3
12 Publications before 2004	7	6	2	3
19 Publications after 2004	12	16	5	3
\sum 31	19	22	7	6

7. Benchmark Engineering Methodology

This set is designed to incorporate the most common published benchmark analysis and presentation requirements. Refer to Tables 7.5 and 7.6 for an overview on these requirements in the literature. Although our requirements set is quite comprehensive, an additional secondary requirement, concerned with the presentation of results, is not included:

- S3: Public Results Database

This secondary presentation requirement will be briefly discussed at the end of this section.

R13: Statistical Analysis

The most important primary requirement for the analysis of the benchmark results is the need for a sound and rigorous statistical analysis. Refer to Section 3.4 and the referenced literature for further details on this requirement. In the following, we present two hints on minimal requirements for a good benchmark analysis.

First, the benchmark should utilize *multiple statistical aggregation methods*, such as both the arithmetic mean and the median, instead of simply providing a mean value. Furthermore, each of these reported values should be accompanied by its confidence interval or the other quartiles, respectively.

Second, the statistical analysis should be based upon the *raw experimental data*, also called primary data, collected during the benchmark execution phase. This set of raw data should always be made available in addition to all other published results (see R14 and R15).

The goal of this requirement is to provide the instruments for meaningful comparisons between different executions of the benchmark. Often these instruments are reduced to a single number, although additional details are usually necessary to perform these comparisons with confidence.

R14: Reporting

The second primary requirement for the analysis and presentation phase is the need of detailed reporting. Similar to the phases of our benchmark engineering methodology, there are three parts to good reporting: the

7.4. Phase 3: Analysis and Presentation

benchmark itself, the *execution* of the benchmark, and the *results* of the benchmark.

First, the used *benchmark* itself should be described. In the case of popular or previously published benchmarks, a reference to previous publications and documentations is sufficient. In the case of open source or publicly available benchmarks, in addition to the description, an access to the benchmark should be provided. In other cases, the benchmark should be described comprehensively with reference to the eight benchmark design requirements and its adherence to them.

Next, the *execution* of the benchmark should be described. Here, an exact description of the SUT, the environment, the workload, and the used measures, including their respective parameters, is required (see Figure 7.3). The goal of the description should be to enable replications of the performed benchmark experiments to provide additional verifiability and replicability.

Finally, the benchmark *results* should be described. That is, the results of the statistical analysis, as well as details on the actual analysis, should be published. In addition, the raw or primary data of all benchmark runs should be made available to provide further credibility to the results.

R15: Validation

The third primary benchmark requirement for the analysis and presentation phase is the need for validation of the benchmark results. This requirement is twofold: (1) the validation by the performer of the benchmark and (2) the possibility for others to validate the results. The first part is concerned with the validity of results, e. g., taking measurement bias of the benchmark into account or performing causality analyses. The second part is concerned with providing enough details to enable replication and validation by others, i. e., providing a thorough reporting and making all required data and code available (see R14).

S3: Public Results Database

Our fourth analysis and presentation phase requirement is a secondary one. That is, this requirement is not necessary for good benchmarks, but parts of

7. Benchmark Engineering Methodology

its principle can help. Several authors (see Tables 7.5 and 7.6) promote the idea of public benchmark databases for benchmarks and their results.

Examples of existing public benchmark result repositories for specific benchmarks are the results collections of the SPEC benchmarks.¹ However, these collections only accumulate and compare the results of single benchmarks, as opposed to a real public benchmark results database, collecting and comparing results of multiple benchmarks.

Such a database could provide additional benefits, e. g., finding similar benchmarks by comparing the results for similar environments and SUTs. Additionally, it could be possible to validate micro-benchmark results with the help of macro-benchmark results. In order to achieve these comparisons and validations, the benchmarks, the environments, the SUTs, and the raw result data must be comparable. For instance, in the case of the raw result data, a common data format is required, capable of satisfying the needs of several very different benchmarks to store their respective data.

Given these limitations and requirements, more specialized forms of publication currently promise better results, e. g., the aforementioned SPEC repositories. Further possibilities to publish raw benchmark results will be discussed in the next section.

7.4.2 Publishing Benchmark Results

Several of our requirements from the three phases of our benchmark engineering methodology aim at improving the replicability and verifiability of benchmarks, e. g., R2, R3, R9–R12, and R13–R15. Successful replication and verification highly depends on the amount and detail of published information (R14).

According to Peng [2011], scientific publications should ideally be judged by full replication (the gold standard). However, full replication is often prohibitively expensive and time consuming. Thus, a verification of the scientific results based on the original data is often considered sufficient (called reproducibility by Peng). As a consequence, Peng introduced a reproducibility spectrum for scientific publications, that we adapt for the publication of benchmark results in Figure 7.4.

¹E. g., for the SPECjbb®2013 benchmark [SPEC 2013b], <http://www.spec.org/jbb2013/results/>

7.4. Phase 3: Analysis and Presentation

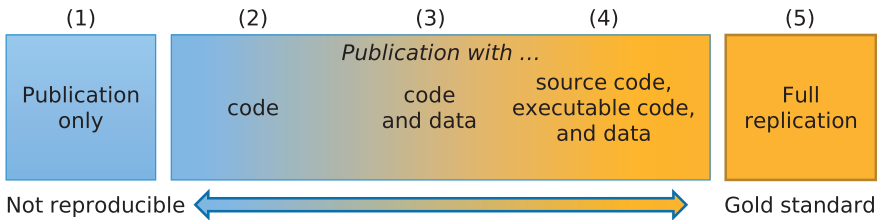


Figure 7.4. Reproducibility spectrum for scientific publications (adapted for benchmark results from [Peng 2011])

In this figure, publication only (1) refers to publishing the benchmark results without any means to verify them. The addition of code (2) is usually satisfied by using well-known or open source benchmarks and providing detailed descriptions of the benchmark environment, the SUT, and the configuration of the benchmark. The next quality level is achieved by also providing the raw results (3), while the final level is reached by providing the source code of the benchmark as well as a pre-configured and ready to run version of the benchmark in addition to the raw data (4). Full replication, on the other hand, can only be achieved by other, independent benchmarkers, repeating the whole benchmark experiments (5).

Examples of our own publication in the categories 2–4 are as follows:

- (2) For our monitoring overhead evaluation in van Hoorn et al. [2009b], we have employed an open-source benchmark and provide detailed descriptions of our experiments.
- (3) For our evaluation of trace processing overhead [Fittkau et al. 2013b; c], we have employed an open-source benchmark, provide detailed descriptions of our experiments, and make all raw data available.
- (4) For our comparison of different Kieker versions [Waller and Hasselbring 2013a; b; c], we additionally provide a pre-configured and directly executable version of the benchmark, including all used SUTs.

Replicability is often considered as good scientific practice. For instance, the Deutsche Forschungsgemeinschaft (DFG) states in its seventh recommendation for good scientific practice, that the primary data should be stored

7. Benchmark Engineering Methodology

and preserved for ten years [DFG 2013]. However, privately storing data is not sufficient for replicability. Crick et al. [2014] similarly demand the publication of all code in addition to produced data.

In accordance with Gray [2009], Bell et al. [2009] name data and exploration of data the fourth paradigm of science. Thus, data itself should be published in addition to simply storing it. Kunze et al. [2011] further refine this idea and propose so-called data papers. A data paper is similar to a normal paper (e. g., including a title, authors, and an abstract), but contains data instead of the usual paper contents. As such, it is usually not printed but digitally published and distributed. Examples of data papers are the already mentioned publications of our raw benchmark results, for instance, Waller and Hasselbring [2013b].

The publication of scientific data is more common in other scientific communities besides computer science, especially within earth and life sciences. For instance, the PubFlow project aims at automating the data publication process in ocean science [Brauer and Hasselbring 2013].

There are already a number of existing specialized data centers, so called World Data Centers (WDCs), for several scientific disciplines. They are capable of handling the amounts of scientific data produced with each experiment. In the case of ocean sciences, an example WDC is the World Data Center for Marine Environmental Sciences [WDC-MARE]. Refer to Leadbetter et al. [2013] for further details and recommendations on publishing data.

To the best of our knowledge, there are no WDCs focussing on computer science. However, a few scientific data centers are open for all fields of science and also provide the required licensing capabilities often required for hosting software, e. g., pre-configured open-source benchmarks. One example is ZENODO, a data center provided by CERN [ZENODO].

As a final remark on the publication of scientific data, there are further problems similar to the ones mentioned for a public benchmark repository. Thus, any publication of raw benchmark results should employ clearly defined and documented data formats. Refer to, for instance, Kalibera et al. [2004], Gray [2009], or Hinsen [2012] for hints on data modeling and examples of possible data formats. For the publication in public data centers, human-readable and self-documenting formats have to be preferred, e. g., CSV, XML, or JSON.

7.5 Challenges Benchmarking Java Applications

When benchmarking applications running in managed runtime systems, such as Java applications running in the JVM, additional challenges are present. The JVM acts as a black-box, influencing the runtime performance of the benchmark in non-deterministic ways. The main causes of this non-determinism are Just-In-Time (JIT) compilation, garbage collection, or thread scheduling. Each of these components is run in one or more threads that are started in addition to all application threads.

These challenges are mentioned in several scientific publications concerned with performance measurements of benchmarks in Java [e. g., Bull et al. 2000; Georges et al. 2004; Blackburn et al. 2006; Adamson et al. 2007; Georges et al. 2007; 2008; Mytkowicz et al. 2009; 2010; Kuperberg et al. 2010; Gil et al. 2011]. Furthermore, several professionals within the Java community have provided insights into this topic [e. g., Goetz 2004; Boyer 2008; Bloch 2009; Click 2009; Hunt and John 2011].

In the following, we briefly introduce the most commonly mentioned challenges posed by the JIT compiler and its associated optimizations. In addition, we sketch common solutions to its challenges. For a more detailed description and further challenges, refer to the given literature.

As mentioned in Section 4.3, Java source is compiled into an intermediary bytecode instead of directly executable machine code. At the start of the execution of a Java application, this bytecode is executed in an interpreted mode. Each execution is analyzed and after sufficient analysis it gets compiled into machine code that gets directly executed. In addition to this initial compilation, further optimizations are performed during the runtime. For instance, elimination of dead code branches, inlining of short methods, or unrolling of simple loops. All of these optimizations are usually non-deterministic and can differ between several executions running in identical environments with identical parameters.

The most important solution to the challenge of code compilations of the JIT compiler is a sufficiently long *warm-up period*. Sufficient in this context means including all performed compilations and optimizations. The performed compilations within the JVM can be logged with the help of the `-XX:+PrintCompilation` parameter.

7. Benchmark Engineering Methodology

To combat the inherent *non-determinism of JIT compilation*, usually three solutions are proposed: First, the compilation could be disabled with the `-Djava.compiler=NONE` parameter. However, the resulting performance measurements usually have nothing in common with realistic scenarios. Thus, other solutions should be preferred. Second, replay compilation could be employed. In this case, an order of compilations and optimizations is recorded and reused for all subsequent executions. However, depending on the recorded replay, the resulting performance can be more or less representative of the system's actual performance. Furthermore, special JVMs are required to support this replay [Georges et al. 2008], additionally limiting the relevance of the results. Finally, the execution can be repeated multiple times on fresh instances of the JVM. With the help of appropriate statistical methods, the non-deterministic influence on the results can be minimized [e. g., Mytkowicz et al. 2010; Gil et al. 2011]

Finally, the performed optimizations of the JIT compiler pose *challenges to the implementation of the benchmark* itself. Especially in the case of micro-benchmarks, no unintended unrollings or optimizations of measurement loops must be performed. A common strategy is the introduction of side effects to prevent these kinds of optimizations. Detailed descriptions of common mistakes in constructing Java micro-benchmarks are given by Bloch [2009] or Click [2009]. Additional insights into these challenges and further details on solutions are presented by Hunt and John [2011].

Micro-Benchmarks for Monitoring

In this chapter, we describe the *MooBench micro-benchmark* that we have developed to measure and quantify the overhead of monitoring frameworks (Section 8.1). In Sections 8.2–8.4, we describe its *benchmark engineering process* and our adherence to the requirements of each of the three phases of benchmark engineering. Finally, we present common *threats to the validity* of benchmark experiments with MooBench that are inherent to micro-benchmarks in general and to MooBench in particular (Section 8.5).

Micro-benchmarks are like a microscope.

Magnification is high, but what the heck are you looking at?

—Dr. Cliff Click, CTO and Co-Founder of 0xdata

8. Micro-Benchmarks for Monitoring

Previous Publications

Parts of this chapter are already published in the following works:

1. 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
2. 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)*. ACM, June 2011, pages 197–200
3. J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In: *Multicore Software Engineering, Performance, and Tools (MSEPT)*. Springer, June 2012, pages 42–53
4. J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68
5. F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 89–98
6. J. Waller. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013
7. J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69

8.1 The MooBench Micro-Benchmark

The MooBench (MONITORING Overhead BENCHMARK) micro-benchmark has been developed to measure and quantify the previously introduced three portions of monitoring overhead (see Section 6.2) for application-level monitoring frameworks under controlled and repeatable conditions. It is provided as open source software [Apache License, Version 2.0] with each release of the Kieker framework and on the Kieker home page.¹ Furthermore, it is archived at ZENODO [Waller 2014a].

In order to achieve representative and repeatable performance statistics for a software system, benchmarks have to adhere to a benchmark engineering methodology. This methodology describes a benchmark engineering process with guidelines for all three phases of benchmark engineering:

1. the design and implementation of a benchmark,
2. the execution of the benchmark, and
3. the analysis and presentation of the benchmark results.

Such an encompassing benchmark engineering methodology is presented in Chapter 7. In the following, we employ this methodology for all three benchmark engineering phases of the MooBench micro-benchmark. For each phase, we first give a general overview of our benchmark within the phase and detail our fulfillment of and adherence to the respective phase's set of requirements.

8.2 Phase 1: Design and Implementation

The first phase of our benchmark engineering methodology is concerned with the design and implementation of the benchmark. In this section, we first introduce our chosen architecture and the resulting implementation for MooBench. Next, we present our general approach to measure the three causes of monitoring overhead using our benchmark. Afterwards, we discuss our adherence to the benchmark design requirements and additional design decisions concerning the benchmark engineering process.

¹<http://kieker-monitoring.net/MooBench/>

8. Micro-Benchmarks for Monitoring

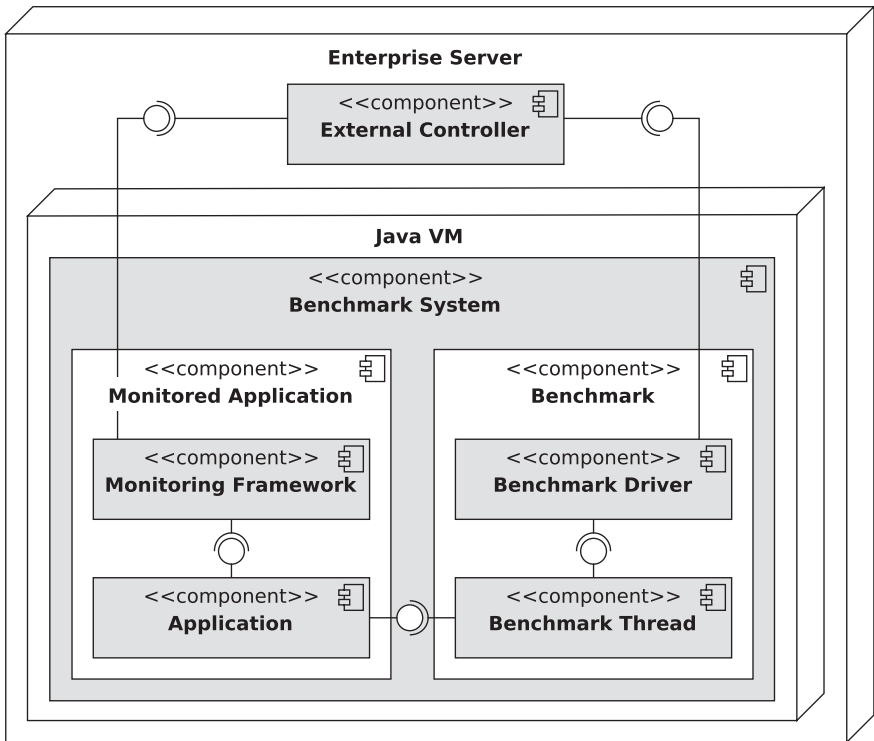


Figure 8.1. The general architecture of the benchmark setup

8.2.1 Architecture and Implementation of MooBench

An overview on the designed and implemented component structure of the MooBench micro-benchmark is presented in Figure 8.1. The figure combines a UML component diagram with a UML deployment diagram [OMG 2011b]. Thus, in addition to the used components, it depicts a typical deployment of the benchmark onto a server machine with a single JVM.

The MooBench micro-benchmark consists of two main components: the Benchmark System running in a JVM and an External Controller initializing and operating the system. The Benchmark System itself is again divided

8.2. Phase 1: Design and Implementation

Listing 8.1. Excerpt of an External Controller script

```
1  #!/bin/bash
2  # path
3  JAVABIN="./jdk/bin/"
4  # configuration parameters
5  SLEEPTIME=30
6  NUM_LOOPS=10
7  THREADS=1
8  RECURSIONDEPTH=10
9  TOTALCALLS=2000000
10 METHODTIME=500000
```

into two parts: First, the Monitored Application, consisting of the Application instrumented by the Monitoring Framework component. Second, the Benchmark, consisting of the Benchmark Driver with one or more active Benchmark Threads accessing the Monitored Application.

The External Controller

The External Controller provides the user of the benchmark with configuration possibilities and acts as a facade for the rest of the benchmark. That is, the controller initializes and configures the Benchmark component with the desired parameters. Furthermore, it ensures that the Monitoring Framework component is correctly integrated into the Monitored Application. Finally, it also controls the execution of the benchmark in the second benchmark engineering phase and initializes the analysis for the third benchmark engineering phase.

For its implementation, the External Controller employs a series of pre-configured and easily adaptable shell scripts that are contained within MooBench's `bin/` folder and its subfolders. Typically, only the top few lines of a chosen External Controller script need to be adapted for the intended benchmarking experiment. An excerpt of these lines is presented in Listing 8.1.

For instance, the path to the required Java executable has to be adapted for the respective SUT. Furthermore, several parameters controlling the

8. Micro-Benchmarks for Monitoring

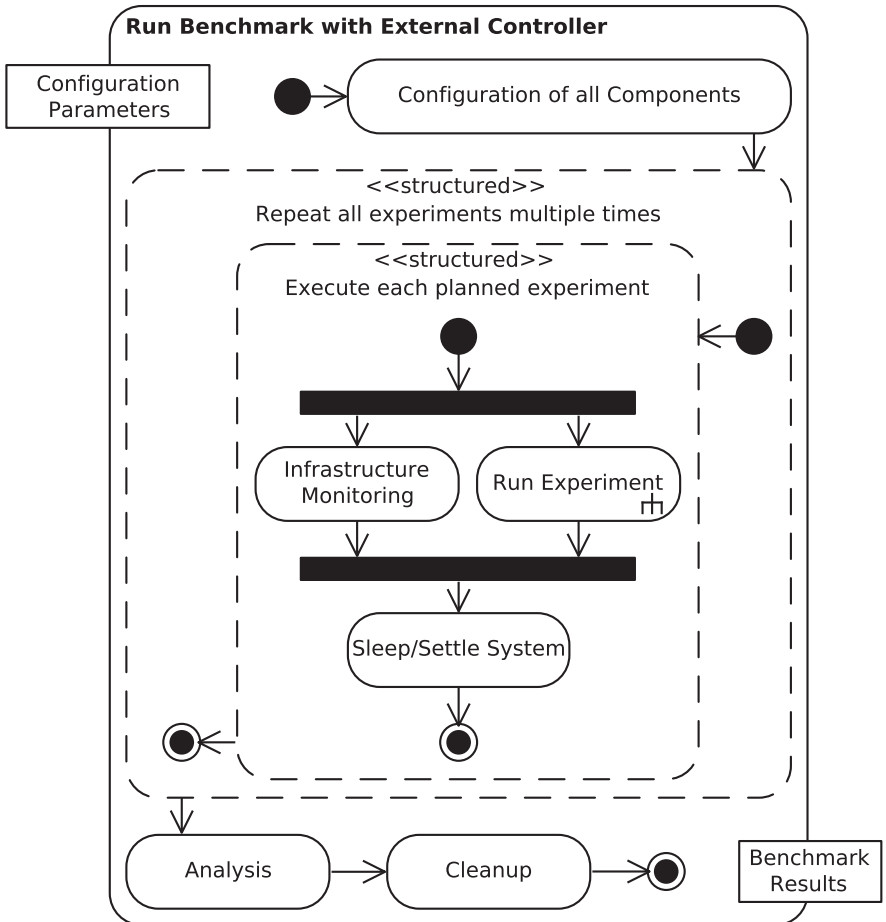


Figure 8.2. UML activity diagram for a typical External Controller

8.2. Phase 1: Design and Implementation

execution of the benchmark in the second phase can be managed. For instance, `NUM_LOOPS` controls the repetitions of the experiments in fresh JVMs, i. e., a value of 10 results in each benchmark experiment being repeated ten times. `SLEEPTIME`, on the other hand, controls the idle time between these repetitions. Refer to the descriptions of the other components of the MooBench micro-benchmark for further details on the other parameters.

It is of note that it is easily possible to implement an own version of an External Controller using other means. For instance, a Java-based implementation is used by Zloch [2014] to execute our benchmark within continuous integration systems (see also Section 11.7).

A UML activity diagram [OMG 2011b] that details the inner workings of such a typical External Controller is presented in Figure 8.2. The modeled activity is running a benchmark with the External Controller. Hence, the input parameter is the set of desired configuration parameters, while the output is the results of the performed benchmark experiments. First, the configurations of the JVM and the Benchmark System are prepared. Next, two nested loops are executed.

The outer loop controls the repetitions of the experiments, while the inner loops execute each planned experiment, e. g., different experiments to determine the three portions of monitoring overhead. Concurrently to each experiment, infrastructure monitoring is performed. For instance, the CPU, memory, disk, and network utilization are logged during each experiment. After each experiment, a short sleep time is introduced to let the system settle down and conclude pending operations. After all repetitions are finished, an analysis of the benchmark results is triggered. Finally, some cleanup tasks are executed, e. g., deleting temporary files or zipping results.

The Monitored Application

The Monitored Application consists of two components: the Application itself and the Monitoring Framework (also called System Under Test (SUT)) instrumenting the Application. For benchmarking the Kieker framework, the Monitoring is realized by the `Kieker.Monitoring` component (see Section 4.5). For benchmarking other monitoring frameworks, this component can be replaced accordingly.

8. Micro-Benchmarks for Monitoring

Listing 8.2. Required Java interface for the Monitored Application

```
1 public interface MonitoredClass {
2     public long monitoredMethod(long methodTime, int recDepth);
3 }
```

Listing 8.3. Basic implementation of the MonitoredClass interface

```
1 public final class MonitoredClassSimple implements MonitoredClass {
2     public final long monitoredMethod(long methodTime, int recDepth) {
3         if (recDepth > 1) {
4             return this.monitoredMethod(methodTime, recDepth - 1);
5         } else {
6             final long exitTime = System.nanoTime() + methodTime;
7             long currentTime;
8             do {
9                 currentTime = System.nanoTime();
10            } while (currentTime < exitTime);
11            return currentTime;
12        }
13    }
14 }
```

It is important to note that the Monitoring Framework component is not actually a part of the MooBench micro-benchmark, but that it rather is the SUT that is benchmarked. Prepared configurations, including required changes to the External Controller for different monitoring frameworks besides Kieker, are provided with releases of MooBench, e. g., SPASS-meter [Eichelberger and Schmid 2014] and inspectIT [Siegl and Bouillet 2011].

The Application component is usually realized as a very basic implementation, consisting of a single `MonitoredClass` with a single `monitoredMethod()`. The required Java interface for all implementations of this component is provided in Listing 8.2. This `monitoredMethod()` has a fixed execution time, specified by a parameter `methodTime`, and can simulate `recDepth` nested method calls (forming one *trace*) within this allocated execution time.

A basic implementation of the interface (`MonitoredClassSimple`) that is provided with MooBench is presented in Listing 8.3. During the execution

8.2. Phase 1: Design and Implementation

of the `monitoredMethod()`, busy waiting is performed, thus fully utilizing the executing processor core. Due to its side effects and calculated return value, the loop of the method cannot be easily eliminated by JIT compiler optimizations, thus avoiding common pitfalls in Java micro-benchmark systems.

Depending on the used hardware architecture and software stack, e.g., older machines, it might be advisable to replace the calls to `System.nanoTime()` with `System.currentTimeMillis()`. Furthermore, the implementation of busy waiting is only correct in the case of single threaded benchmarks on an otherwise unoccupied system. In the case of concurrency, other threads could consume all available time on the executing processor core. However, due to the implementation of the `monitoredMethod()`, this time would still be counted as the busy waiting time.

In order to correctly simulate a method using the CPU for a period of `methodTime` despite the activities of other threads in the system, we use `getCurrentThreadUserTime()` of JMX's `ThreadMXBean`. Such an advanced implementation of the `MonitoredClass` interface is provided with `MooBench` in the class `MonitoredClassThreaded`. However, note that the use `ThreadMXBean` is usually expensive compared to querying the system's timer. Thus, benchmark configurations with small `methodTime` parameters might become less accurate. Especially with older or virtualized systems, e.g., cloud infrastructures, these calls might be prohibitively expensive.

Finally, it is easily possible to add own implementations of the `MonitoredClass` interface for special cases or scenarios. For instance, the custom implementation of the `monitoredMethod()` can act as a facade for calls to a more complex or more realistic application.

The Benchmark

The Benchmark component is at the core of the `MooBench` micro-benchmark. It consists of the Benchmark Driver and one or more active Benchmark Threads.

A UML activity diagram [OMG 2011b] for the process of running a single experiment with the Benchmark Driver is presented in Figure 8.3. The Benchmark Driver gets started by the External Controller and first checks its committed configuration. Then, it initializes the rest of the Benchmark ac-

8. Micro-Benchmarks for Monitoring

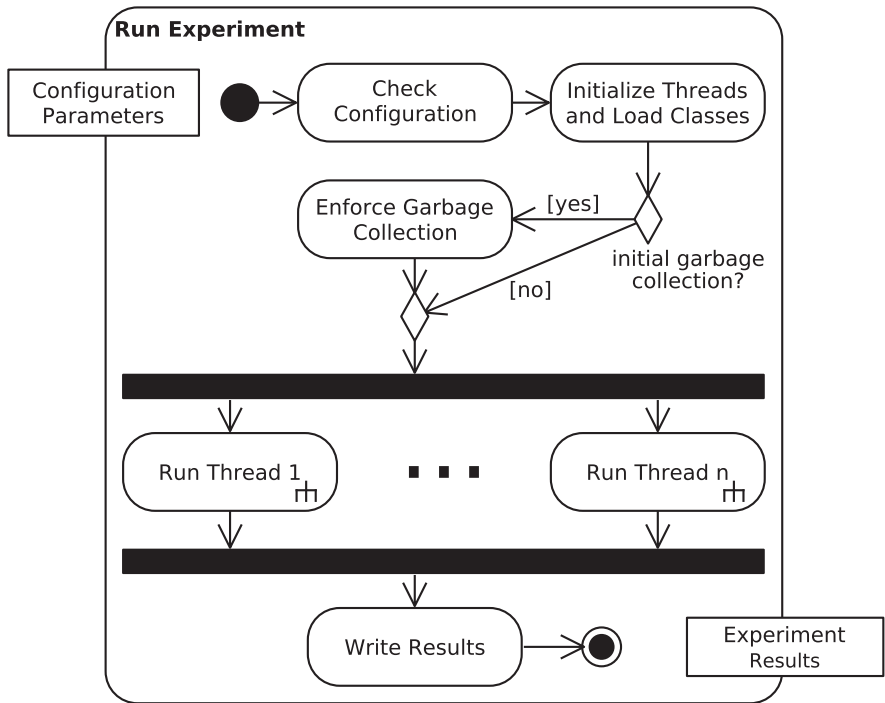


Figure 8.3. UML activity diagram for the Benchmark Driver

According to this configuration and starts the Monitored Application and the required number of Benchmark Threads.

Depending on the configuration (`--quickstart`), an initial garbage collection can be enforced by consuming and releasing all available memory. According to our experience, such an initial garbage collection shortens the required warm-up period by letting the system reach a steady state earlier.

Afterwards, the configured number of Benchmark Threads is started. Each thread performs the actual calling of the `monitoringMethod()` and handles the required measurements. Finally, after all Benchmark Threads are finished, the Benchmark Driver collects and persists the performance data.

8.2. Phase 1: Design and Implementation

Listing 8.4. Excerpt of the Benchmark Thread's run method

```
1 public final void run() {
2     long start_ns;
3     long stop_ns;
4     for (int i = 0; i < totalCalls; i++) {
5         start_ns = System.nanoTime();
6         monitoredClass.monitoredMethod(methodTime, recursionDepth);
7         stop_ns = System.nanoTime();
8         timings[i] = stop_ns - start_ns;
9     }
10 }
```

An excerpt of the Java code for the `run()` method of the Benchmark Threads is presented in Listing 8.4. One or more concurrently executing instances of these Benchmark Thread call the `monitoredMethod()` of the Monitored Application while recording the response times of each call using the `System.nanoTime()` method. Each thread is parameterized with a *total number of calls*, as well as the *method time* and the *recursion depth* of each call.

Contrary to the External Controller and to the Monitored Application, the Benchmark component and its parts are not specifically designed to be exchangeable. However, MooBench is available as open-source software, so each aspect of the benchmark can be tailored to the specific requirements. For instance, it is easy to create another class implementing the Benchmark-Thread interface in order to employ a different measuring method from `System.nanoTime()`. Due to this employed abstraction, only a single line in the Benchmark Driver would be modified for this adaptation.

Finally, the set of all available configuration parameters for the Benchmark component is listed in Listing 8.5. Most of these parameters correspond to similar parameters of the External Controller. However, there are additional parameters available. For instance, the `--application` parameter allows for the selection of the `MonitoredClass` implementation, with `MonitoredClassThreaded` being the default. Similarly, the `--runnable` parameter enables custom initializations for specific monitoring frameworks, e. g., preconfiguring settings for a used logger.

8. Micro-Benchmarks for Monitoring

Listing 8.5. Configuration parameters for the Benchmark

```
usage: mooBench.benchmark.Benchmark
-a,--application <classname>    Class implementing the MonitoredClass
                                  interface.
-d,--recursiondepth <depth>      Depth of recursion performed.
-h,--totalthreads <threads>      Number of threads started.
-m,--methodtime <time>           Time a method call takes.
-o,--output-filename <filename>  Filename of results file. Output is
                                  appended if file exists.
-q,--quickstart                   Skips initial garbage collection.
-r,--runnable <classname>        Class implementing the Runnable
                                  interface. run() method is executed
                                  before the benchmark starts.
-t,--totalcalls <calls>          Number of total method calls performed.
```

8.2.2 Design of an Experiment to Measure the Three Causes of Overhead

Each experiment to measure the three causes of monitoring overhead (see Section 6.2) is designed to consist of four independent runs, that are started by the external controller (refer to Figure 8.2). With each run, the benchmark is configured to measure a different execution time associated with one portion of overhead. This way, we can incrementally measure all three portions of monitoring overhead, as outlined below:

1. In the first run, only the execution time of each call of the monitored-Method() is determined (T), including potential additional recursive calls. That is, the Monitoring Framework component is not present and the Monitored Application is executed without monitoring. Thus, we can determine T , i. e., the time spent executing the actual code of the monitored-Method() if no monitoring is performed.
2. In the second run, the Monitoring Framework component is added. That is, the monitoredMethod() is instrumented with a MonitoringProbe which is deactivated for the monitoredMethod(). Thus, with each execution of the method the duration $T + I$ is measured.

8.2. Phase 1: Design and Implementation

3. In the third run, we activate monitoring for the `monitoredMethod()`. Thus, the collection of data is added to our measurements. However, each thus created `MonitoringRecord` is immediately discarded. This results in measuring $T + I + C$ with each call of the `monitoredMethod()`.
4. The fourth run finally represents the measurement of full monitoring with the addition of an active `Monitoring Writer` and potentially an active `Writer Thread`. Thus, with each call of the `monitoredMethod()`, we measure the execution time of $T + I + C + W$.

In summary, this set of four independent runs allows for the estimation of the three portions of monitoring overhead by calculating the differences between each run with the help of appropriate statistical methods.

It is of note that we can easily add additional runs. For instance, in the case of live analysis, we could add further runs to determine the additional costs that performing the analysis has on the monitoring of the Application. Refer to Chapters 11 and 12 for further details and scenarios.

Similarly, it is possible to remove one or more runs. For instance, if the monitoring framework under test does not support deactivation of writing, we are unable to perform the third run and cannot determine the individual portions of C or W , but rather only the combined overhead of $C + W$. An extreme example would be the reduction to two runs: with and without monitoring. This way, it is possible to determine the total overhead of any monitoring framework.

8.2.3 Benchmark Design Requirements for MooBench

In this section, we discuss our adherence to our set of the eight benchmark design requirements (R1 – R8) introduced in Section 7.2. Furthermore, we briefly discuss the two secondary requirements S1 and S2.

R1: Representative / Relevant Although representativeness is the most important requirement, it is also the weakness of our MooBench micro-benchmark. However, this weakness is inherent with all kinds of micro-benchmarks due to their limited focus and applicability. Regardless of this general weakness, MooBench is still representative of and relevant for the

8. Micro-Benchmarks for Monitoring

important use case of simple application traces. Additionally, we discuss the comparison of the results of our micro-benchmark with results of more representative macro-benchmarks in Chapter 9.

Concerning the subdivision of representativeness into two parts, the strength of our benchmark lies with the relevant usage of software and hardware features. We directly include the monitoring framework under test into our benchmark and employ it as intended. Thus, we automatically use the same software and hardware features as normal (not benchmarking related) monitoring scenarios.

With regards to a representative workload, our benchmark targets the monitoring of simple method execution traces in common Java applications. The actual workload is configurable using several parameters, for instance, the method's execution time and recursion depth. However, more complex scenarios, e. g., monitoring of concurrency or measuring the memory consumption of a monitoring framework in relation to the number of instrumented methods, require additional work.

In summary, MooBench provides a representative and relevant micro-benchmark for measuring the overhead of monitoring execution traces in applications.

R2: Repeatable MooBench is designed to provide repeatable and deterministic results. First, the generated workload is deterministic. Second, usually there are no lasting side-effects of our benchmark experiments. In the case of special monitoring frameworks, e. g., employing a database, necessary steps to remove these side effects must be taken, e. g., clearing the database after every run. However, this largely depends on the monitoring framework under test and is outside of the direct area of control for the benchmark.

In addition, MooBench is designed to provide and record all parameters and results. The parameters of the benchmark as well as those of the system running the benchmark are collected within log files by the External Controller and the Benchmark Driver, while the results form the output of each benchmark experiment. While the External Controller starts an analysis of the gathered data, the raw results are never discarded. Thus, our results can be published with sufficient details to repeat all experiments.

8.2. Phase 1: Design and Implementation

R3: Robust Robustness is mostly concerned with the execution of the benchmark. However, within the External Controller we already provide the means for robustness in the execution phase. For instance, both the number of repetitions within each benchmark run and the number of repetitions of the whole experiment are configurable. Each repetition of the whole experiment is performed on a different JVM, thus alleviating the influence of different JIT compilation paths (see Section 7.5). Similarly, the configuration of the total number of calls prepares our design for robustness by including a configurable warm-up period into the runs.

In order to protect our results against perturbation from our own measurements, the `System.nanoTime()` method is employed. This method internally accesses the system's performance counters. Furthermore, it is often employed by the monitoring framework under test, too. However, as mentioned before, a change of the used measurement technique to reduce its influence on different monitoring frameworks under test only requires a small adjustment within our open source benchmark.

R4: Fair Although MooBench is designed with the Kieker monitoring framework in mind, its features are not specifically tailored for it. This is demonstrated in Chapter 12, where we compare results of our benchmark for different monitoring frameworks under test. It is of note that the employed measuring methodology and the used workload are neither tailored nor biased for the Kieker framework. Additionally, MooBench is not limited to a specific environment. Thus, different software and hardware environments can be compared with the help of the benchmark.

R5: Simple The simplicity of our benchmark is inherent with most micro-benchmarks. That is, the benchmark is easy to configure, to tailor for specific scenarios, and to execute using the External Controller. The used workload and the employed measures are easily comprehensible.

R6: Scalable Due to its nature as a micro-benchmark, MooBench is mostly concerned with vertical scalability. Thus, it is easy to scale the used workload, e. g., by configuring the number of Benchmark Threads or the moni-

8. Micro-Benchmarks for Monitoring

toredMethod()'s execution time and recursion depth. On the other hand, the Monitored Application is too simple for any meaningful horizontal scalability. However, due to the configurability, it is easy to exchange the Application for a more complex one, leading in the direction of macro-benchmarks (see Chapter 9).

R7: Comprehensive The comprehensiveness of the MooBench micro-benchmark is mostly limited to monitoring application traces. Similarly to our discussion of representativeness, this is inherent to the small scale and simplicity of a micro-benchmark. However, this comprehensiveness is sufficient for the usual use cases of application-level monitoring tools.

Furthermore, due to the configurable workload and number of Benchmark Threads, several different, typical monitoring scenarios can be benchmarked and compared. Finally, due to the extensibility of the benchmark, additional parts of specific monitoring frameworks under test can be easily covered.

R8: Portable / Configurable Concerning the portability, MooBench is not restricted to any specific environment or system, as long as Java and the monitoring framework under test are supported. The provided External Controller is implemented in the form of a bash script, so Linux-based systems are preferred. However, other controller implementations are easily done [Zloch 2014]. The configurability and extensibility of the benchmark has already been discussed in the previous requirements (also refer to Listings 8.1 and 8.5).

Additional Requirements Due to its restricted nature as a micro-benchmark, MooBench is mostly specific (S1) for the subclass of application-level monitoring frameworks, rather than, for instance, hardware monitoring frameworks. However, we designed no additional or artificial restrictions for specific platforms or classes of SUTs.

Furthermore, our MooBench micro-benchmark is both accessible and affordable (S2), thus adhering to the second additional requirement. Both parts are fulfilled by making the benchmark available as an open source

tool (refer to the website: <http://kieker-monitoring.net/MooBench>) as well as bundled with releases of the Kieker monitoring framework and archived at [Waller 2014a]. Finally, we already discussed useability with the simplicity requirement (S5).

8.2.4 Decisions of the Benchmark Engineering Process

The environment of our benchmark consists of the hardware and software platform, including the operating system and the JVM. For optimal repeatability, this environment and all necessary environmental parameters need to be documented. Design decisions concerning the workload and the employed measures are further detailed below.

Workload

The chosen workload for our benchmark is a very simple, but configurable, synthetic workload: Each Benchmark Thread performs a specified amount of calls (recursiondepth) to the `monitoredMethod()` per configurable time unit (methodtime). Each of these calls is monitored by the Monitoring Framework under test. Thus, by tuning these parameters, we can regulate the workload for the monitoring framework, i. e., the number of method calls the monitoring framework will monitor per second.

For instance, using a minimal methodtime of, e. g., 0 ns, we are able to determine an upper bound for the monitoring overhead per method call. Thus, we can measure the maximal monitoring throughput for a system. With regards to the UML diagram presented in Figure 6.1 on page 91, we minimize T . This way, `MonitoringRecords` are produced and written as fast as possible, resulting in maximal throughput.

As long as the actual `WriterThread` is capable of receiving and writing the records as fast as they are produced, it has no additional influence on the monitored method's response time. When the `WriterThread`'s capacity is reached, the buffer used to exchange `MonitoringRecords` between the `MonitoringWriter` and the `WriterThread` blocks, resulting in an additional increase of the `monitoredMethod()`'s response time. In other words: with this *stress workload*, the method calls are performed as fast as the monitoring

8. Micro-Benchmarks for Monitoring

framework is able to process the gathered information, including the time required for actually writing the `MonitoringRecords`.

On the other hand, with a large `methodtime`, i. e., more time per call to the `monitoredMethod()` than is required by the monitoring framework to handle each call, we are able to determine a lower bound on the monitoring overhead per call. Regardless of the actual load on a system, the overhead of monitoring a single method call cannot drop below this value. This results, depending on the actual configuration, in a *minimal* or *average workload*.

Similarly, a linear rise of overhead with each additional call can be demonstrated by increasing the `recursiondepth` over multiple benchmark experiments with a large, but constant, `methodtime`. Several experiments using these varied workloads are presented in Chapters 11 and 12. A similar result could be obtained with an increasing number of total calls and a decreasing execution time of each method call. However, a constant method time simplifies the analysis of the results.

Measure

Our MooBench micro-benchmark focusses on the measure of *response time*. It is the primary measure that is collected with each call the `Benchmark Threads` perform to the `monitoredMethod()`. For the implementation of the measure, the `System.nanoTime()` method is employed, as discussed in the description of the `Benchmark` component. On contemporary systems, this implementation ensures a high accuracy by utilizing the High Precision Event Timer (HPET). The robustness concerns of this implementation are already discussed in the description of our adherence to the robustness requirement (R3).

Similar to the measure of response time, we can also calculate a derived measure for the monitoring *throughput*. That is the number of executions of the `monitoredMethod()` per second, instead of the flat monitoring cost imposed per execution, i. e., the actual change in the `monitoredMethod()`'s *response time*. This throughput measure can be calculated in the third phase (analysis and presentation) of our benchmark engineering methodology. As mentioned before, our MooBench collects the response times of the `monitoredMethod()`. These response time measurements are accumulated in a number of bins,

each containing one second worth of method executions. The number of response times per bin corresponds to the reported throughput of method executions per second.

The third measure which our benchmark employs is a measurement of resource utilization that is initialized by the External Controller concurrently to each experiment run. For instance, the CPU, memory, disk, and network utilization are logged. To realize this infrastructure monitoring, common system tools are employed, for instance, the tools `mpstat`, `vmstat`, or `iostat`.

Finally, the used JVM instance is instructed to gather details on inner operations related to performance, for instance, garbage collection activity or JIT compilations. Depending on the used configuration of the External Controller, typical command line switches are employed. For instance, `-XX:+PrintCompilation`, `-XX:+PrintInlining`, or `-XX:+UnlockDiagnosticVMOptions` `-XX:+LogCompilation` to gather data on JIT activity, and either `-verbose:gc` or `-XX:+PrintGCDetails` to gather garbage collection details.

8.3 Phase 2: Execution

The second phase of our benchmark engineering methodology is concerned with the actual execution of the benchmark for specific SUTs. In the case of MooBench, each SUT is a monitoring framework. Most of the groundwork for the execution phase has already been laid in the first phase. For instance, several configuration parameters are included in the benchmark components to fine tune its behavior during the execution.

The actual benchmark execution of each benchmark experiment is controlled by the External Controller. As mentioned in Section 8.2.2, each benchmark experiment to determine the four causes of monitoring overhead is designed to consist of four independent runs. Furthermore, each of these runs can be repeated multiple times (cf., Figure 8.2).

Prior to its first execution, the benchmark has to be built using the `ant` tool. Several build targets exist to directly pre-configure MooBench for its supported monitoring frameworks, e. g., `build-kieker` for Kieker, `build-inspectit` for inspectIT, and `build-spasmeter` for SPASS-meter. Otherwise, the default build target can be employed. In all cases, the prepared benchmark with

8. Micro-Benchmarks for Monitoring

all required libraries is available in the `dist/` folder. The provided External Controller scripts can be used to start a fresh benchmark experiment.

In the following, we will elaborate on the execution time requirements presented in Section 7.3. In addition, we discuss default values for the benchmark's configuration parameters (see Listings 8.1 and 8.5).

R9: Robust Execution The requirement for *robustness* against any kind of unintended perturbation influencing the benchmark results is mostly handled by the next three subrequirements (R10–R12).

In addition, we recommend to employ the correct implementation of the Monitored Application (see Section 8.2.1). For instance, in our experiments with virtualized cloud environments, the use of the `MonitoredClassSimple` implementation caused less perturbation and resulted in more stable results. However, in other scenarios the `MonitoredClassThreaded` provided better performance and more realistic results.

Similarly, the method used to gather the employed measure of response time, i. e., `System.nanoTime()`, can be exchanged to employ the same methodology as the monitoring framework under test.

R10: Repeated Executions The requirement for multiple executions consists of two parts. First, each measurement has to be repeated multiple times. This is ensured with the External Controller's configuration parameter `TOTALCALLS`. With each performed call of the `monitoredMethod()`, a measurement is performed (see Listing 8.4). Each of these measurements is recorded and available for analysis in the third phase.

The second part of this requirement is the need for multiple repetitions of the whole experiment. The number of repetitions can be controlled by the `NUM_LOOPS` parameter. Assigning values greater than one causes each independent experiment run to be repeated multiple times on identically configured JVM instances. All results are recorded and statistical methods can be applied in the third phase to minimize the influence of, for instance, different JIT compilation paths or other random factors.

The actual number of `TOTALCALLS` is more important for a sufficient warm-up period. Thus, its value will be discussed with the next requirement.

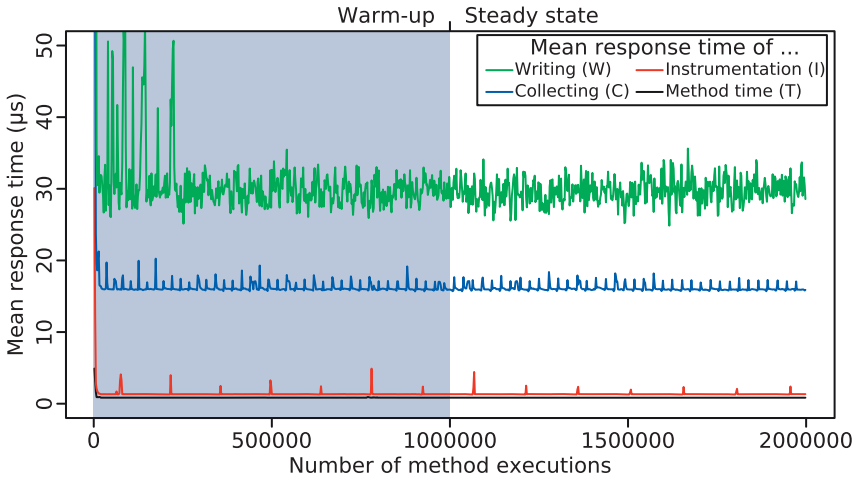


Figure 8.4. Exemplary time series diagram of measured timings with Kieker

Regarding the `NUM_LOOPS` parameter, we suggest at least ten repetitions of each experiment, i. e., setting `NUM_LOOPS = 10`. Refer to Section 7.5 and the accompanying literature for details.

R11: Warm-up / Steady State The total number of calls to the monitored `Method()` (`TOTALCALLS`) in each experiment run has to be sufficiently large to include the warm-up period and a sufficient portion of the steady state. In order to determine the steady state of experiment runs, the benchmark user can analyze the result stream as a time series. An exemplary time series diagram for an early Kieker 1.8 nightly build with event-based probes and a binary writer containing four experiment runs is presented in Figure 8.4.

Our experiments as well as our analyses of JIT compilation and garbage collection logs of benchmark runs with the Kieker framework on our test platform suggest discarding the first half of the executions (for 2,000,000 repeated method executions) to ensure a steady state in all cases. We propose similar analyses with other monitoring frameworks, configurations, or hard- and software platforms to determine their respective steady states. However,

8. Micro-Benchmarks for Monitoring

a value of `TOTALCALLS = 2000000` should be sufficient in most cases. This recommended warm-up period is shaded in gray in Figure 8.4.

Furthermore, the Benchmark Driver enforces the garbage collection to run at least once at the beginning of the warm-up phase. Our experiments suggest that this initial garbage collection reduces the time until a steady state is reached. Additional garbage collections are also visible in Figure 8.4 by the regular spikes in the measured response times.

R12: Idle Environment 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.

8.4 Phase 3: Analysis and Presentation

The third and final phase of our benchmark engineering process is concerned with the analysis of the benchmark results and their subsequent presentation. The External Controller performs a statistical analysis of the benchmark results with the help of several provided R scripts [R; Ihaka and Gentleman 1996]. These scripts can also provide initial visualizations of the results. However, all created raw data remains available for an additional and more detailed analysis.

R13: Statistical Analysis In accordance with Georges et al. [2007] and Section 3.4, our default analysis of the benchmark results provides the mean and median values of the measured timings across all runs instead of reporting only best or worst runs. In addition, it includes the quartiles as well as the 95% confidence interval of the mean value.

R14: Reporting The first part of the reporting requirement for MooBench is fulfilled by the description of the benchmark within this chapter. The second part is concerned with the actual execution of the benchmark. To facilitate repetitions and verifications of experiments, the benchmark user

has to provide a detailed description of the used configurations and environments. Refer to the detailed descriptions of our benchmark experiments in Chapters 11 and 12 for examples. In all cases, the benchmark environment and workload with all required parameters are described in detail. In addition, the results of each benchmark are provided in sufficient detail to draw meaningful conclusions. Additional details of the results are always made available for download.

R15: Validation The validation of our benchmark results is done by comparing our results to the results of several macro-benchmarks in Chapter 13. Additionally, we provide the benchmark, a detailed description of our experiments, and all raw result data. Thus, any interested benchmark user can replicate our results and validate our findings.

S3: Public Results Database Although the results of our benchmark are currently not collected in a public results database, the benchmark could serve as a basis for starting a comprehensive database of monitoring performance. Additionally, the public availability of results would further simplify additional validations of the benchmark results. However, such a platform does not yet exist as of writing this thesis.

8.5 Threats to Validity

Common threats to the validity of micro-benchmarks are their relevance and systematic errors. Although our benchmark is highly configurable and adaptable to different platforms, it bases on repeatedly calling a single method. However, by performing recursive calls, the benchmark is able to simulate larger call stacks. To further validate our approach, additional comparisons of our results with larger and more complex benchmarks or applications are required. Refer to R15 and Chapter 13 for details.

Furthermore, if the configured method execution time is negligible compared to the measured overhead, these comparisons may be inappropriate. However, our experiments with different method execution times suggest that the results are still valid.

8. Micro-Benchmarks for Monitoring

Further threats to validity are inherent to Java benchmarks (see Section 7.5). For instance, different memory layouts of programs or JIT compilation paths for each execution might influence the results. This is countered by multiple executions of our benchmark. However, the different compilation paths of our four measurement runs to determine the individual portions of monitoring overhead might threaten the validity of our results. All benchmark runs include multiple garbage collections which might influence the results. However, this is also true for long running systems that are typically monitored.

Macro-Benchmarks for Monitoring

In this chapter, we present several existing macro-benchmarks that can be used to validate the results of our micro-benchmarks. In Section 9.1, we give a general *introduction* to the topic. In the following three sections, we briefly introduce *three different macro-benchmarks* with increasing complexity: the Pet Store (Section 9.2), the SPECjvm[®]2008 (Section 9.3), and the SPECjbb[®]2013 (Section 9.4).

What can you do? Measure, measure, and measure again!

—Joshua Bloch, software engineer and author of *Effective Java*

9.1 Macro-Benchmarks for Monitoring

To the best of our knowledge, there are currently no macro-benchmarks that are specifically designed to measure or benchmark software monitoring tools or frameworks. However, several existing macro-benchmarks can be employed to act as the system under monitoring and as the load driver when benchmarking monitoring frameworks. In the simplest case, we can execute an existing benchmark two times: once without monitoring and once with monitoring. Afterwards, we can compare the benchmark results to determine the monitoring overhead.

Similar to our micro-benchmark experiments to determine the causes of monitoring overhead (see Section 6.2), we can perform more complex series of experiments to compare the influences of different stages of monitoring instrumentation. However, even in macro-benchmarks, only specific scenarios are benchmarked. That is, the measured performance might differ from any real application. This is especially true for macro-benchmarks that have been developed to benchmark a specific aspect of a system and not the monitoring framework interacting with the benchmark itself.

For instance, a benchmark simulating a shopping system might include hard-disk accesses for its simulated database, that would be located on an external system in a real application. These hard-disk accesses have a huge impact on a monitoring writer that is also using this hard-disk. Of course, this challenge can be avoided by also externalizing such a database. However, due to its greater complexity, it is harder to anticipate these challenges compared to a simpler micro-benchmark. Thus, although we usually get an improved representativeness by employing macro-benchmarks, the simplicity is reduced and we might even risk losing some representativeness. Furthermore, it is often harder to pinpoint the actual cause of (performance) problems detected with these benchmarks, compared to specialized micro-benchmarks [Saavedra et al. 1993].

Our goal with our macro-benchmark experiments is to validate our micro-benchmark results. To avoid the challenges discussed above, we utilize a series of different macro-benchmarks that are presented below. The actual execution of our benchmark experiments and the respective results are detailed in Chapter 13.

9.2. Benchmarks with the Pet Store

In the following three sections, we present a selection of three different macro-benchmarks. Starting with the simple Pet Store, each presented benchmark is more complex than the previous one. Thus, we can validate our micro-benchmarking findings on a wide range of different kinds of macro-benchmarks.

Note that the presented selection of macro-benchmarks only contains a small subset of all available benchmarks. Other common examples of Java-based macro-benchmarks include CoCoME [Herold et al. 2008], the Qualitas Corpus [Tempero et al. 2010], the SPECjEnterprise[®]2010 [SPEC 2012] and its predecessor [Kounev 2005], or the DaCapo benchmark suite [Blackburn et al. 2006].

9.2 Benchmarks with the Pet Store

As already mentioned in our general introduction to macro-benchmarks (see Section 3.2.2), there are several different versions of the Pet Store. In our evaluation, we employ the most recent Pet Store version of the MyBatis project (JPetStore 6) [MyBatis JPetStore] that is also used as an example application by Kieker [2014].

MyBatis JPetStore 6 is a small Java EE web application with an implementation based upon Spring and Stripes. It consists of 24 Java classes as well as several JSPs and XML configuration files. The required database is realized with an embedded in-memory HSQLDB database. The execution of the Pet Store requires a typical servlet container, such as Tomcat or Jetty. In our evaluation, we employ the more lightweight Jetty server that is also used by the Kieker live-demo. For a more detailed description of the Pet Store, we refer to its documentation [MyBatis JPetStore].

The Pet Store is no benchmark, but rather the System Under Test (SUT) within our envisioned macro-benchmark. The remaining two required components (according to Figure 7.3 on page 111) are the workload and the measure. In our case, we employ Apache JMeter [Apache JMeter] with the Markov4JMeter extension [van Hoorn et al. 2008] to provide both.

For the measure of our macro-benchmark, we employ the gathered response times provided by jMeter. This measure includes the service time

9. Macro-Benchmarks for Monitoring

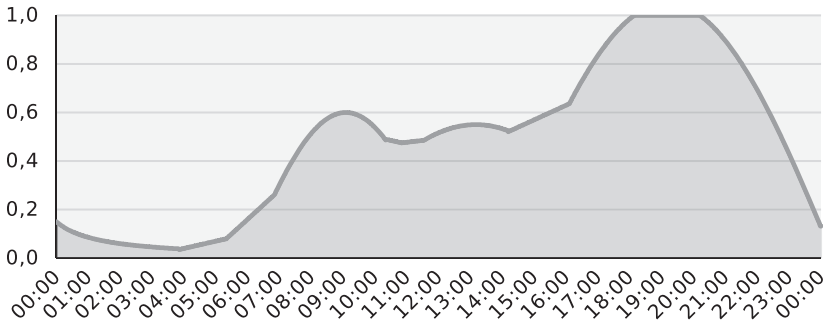


Figure 9.1. Workload simulating a 24 hour period on a typical enterprise web site

of the measured request, as well as the network and queuing time (see Figure 2.1 on page 21). The workload consists of two components: first the actual load, i. e., the number concurrent users accessing the Pet Store, and second the order of accessed pages.

Considering the first part, we employ up to 2,500 concurrent users that are simulated by JMeter. To generate a more realistic workload, we employ a workload curve as presented in Figure 9.1. This workload simulates a 24 hour period on a typical enterprise website [van Hoorn et al. 2009a; van Hoorn 2014]. In our experiments, we reduce this time period to a 24 minute run. Thus, only during the peak times between minute 18 and 20, all 2,500 simulated users are accessing the Pet Store. Otherwise, only a fraction of users is active.

Considering the second part of the workload, we use an updated version of the Pet Store example provided by van Hoorn et al. [2008] and van Hoorn [2014]. The Markov model simulates two different user types: either browsers or buyers. The former focusses on accessing the web site without actually buying any pets, while the latter has a 50% probability to actually order pets. In sum, a total of twelve different actions are performed, e. g., signon, viewProduct, or viewCart. A more detailed description of this workload is given by van Hoorn et al. [2008] and van Hoorn [2014].

9.3. The SPECjvm2008 Benchmark

A possible problem in the evaluation of this macro-benchmark could be the large number of failed orders in the default configuration of the Pet Store. About 7% of the final order requests fail due to a primary key violation of the database. This is caused by two or more buyers accessing the database at the same time. However, this problem is equally present in all benchmark runs. Thus, the results should still be comparable.

A prepared version of the JPetStore 6, including all required configurations, as well as a prepared jMeter application are provided for download [Waller 2014b]. This package also includes the results from our evaluation, as presented in Section 13.1.

9.3 The SPECjvm2008 Benchmark

The SPECjvm[®]2008 benchmark [SPEC 2008a; Shiv et al. 2009] has been developed as a replacement for the SPECjvm[®]98 benchmark. Its focus lies on evaluating the performance of the Java Virtual Machine (JVM) as well as of the system running the JVM. Contrary to its earlier release, the 2008 benchmark specifically targets multi-threaded behavior in addition to single-threaded Java applications. Furthermore, typical server-sided Java tasks are benchmarked in addition to typical client tasks. Contrary to most SPEC benchmarks, the SPECjvm[®]2008 is available free of charge.

The benchmark consists of a set of eleven Java applications, each with one or more associated workloads. Thus, in a normal run, 39 different sub-benchmarks are executed to calculate the final score. Each of them provides an *operations per minute* measure that is used to calculate the final score with the help of a nested geometric mean [Shiv et al. 2009]. In the following, we will briefly discuss the set of included applications and their applicability as benchmarks for monitoring. Refer to Shiv et al. [2009] or to the benchmark user's guide [SPEC 2008b] for a more detailed description.

Startup: With a total of 17 different workloads, this benchmark measures the startup performance of a JVM. One of the main contributing factors to this performance is class loading. However, with most monitoring frameworks, class loading is especially costly. First, several additional classes associated with the monitoring framework are loaded. Second,

9. Macro-Benchmarks for Monitoring

common instrumentation mechanisms transform the classes during their loading, causing additional overhead (see Section 4.3). Although the startup performance of Java applications and VMs is important, it is outside of the scope of our evaluation of monitoring frameworks. Thus, this set of benchmarks is omitted.

Compiler: The two workloads of the benchmark measure the compilation performance of the included Java compiler. As most monitoring frameworks are concerned with the execution of Java programs rather than with their compilation, this benchmark is omitted, too.

Compress: In the compress benchmark, a data set is compressed and decompressed. The focus of this benchmark is on JIT optimizations performed within the JVM. Thus, it consists of many very small, but optimizable methods. Any instrumentation of these methods usually prevents these kinds of optimizations and has a huge impact on the measured performance. As a consequence, an instrumentation of all these methods is not feasible and a more elaborate approach has to be chosen to instrument this benchmark. Depending on the chosen instrumentation points and on their amount, the results will be different. Thus, we omit this benchmark from our evaluation.

Crypto: The three workloads of the crypto benchmark measure three different cryptographic approaches. Depending on the workload, either several methods of the benchmark are executed (usually instrumented by the monitoring framework) or an already provided implementation of the JVM vendor is executed. As the code provided by the JVM is usually not instrumented, the performance impact of the monitoring tool differs accordingly.

Derby: The derby benchmark simulates accesses to database written in Java as well as calculations with large numbers. Thus, this benchmark represents a typical representative of small server applications.

Mpegaudio: In this benchmark, a Java-based library is called to decode audio files. This benchmark represents a typical representative of a small client application.

9.4. The SPECjbb2013 Benchmark

Scimark: The set of Scimark benchmarks with its nine workloads is based upon the popular SciMark 2.0 benchmark suite [NIST 2004]. Here, it is further subdivided into a small and a large benchmark, each containing four workloads, and an additional workload, based upon a Monte Carlo approximation. Only the last workload of this set poses challenges to our intention of benchmarking monitoring frameworks. Similar to the previously mentioned *compress* benchmark, the Monte Carlo workload is designed to test the capabilities of the JIT, specifically inlining. Thus, its monitored performance is very low and we omit its execution from our selection of benchmarks.

Serial: In the serial benchmark, a set of Java objects is serialized and deserialized in a typical producer/consumer scenario.

Sunflow: The sunflow benchmark simulates multi-threaded visualizations using ray-tracing. Its focus lies on floating-point operations as well object-creation and garbage collection.

Xml: The final benchmark contains two workloads. Both are concerned with typical tasks of handling XML files.

The results of a benchmark run with Kieker are presented in Section 13.2. All utilized scripts and configuration parameters, as well as the full set of results, are available for download [Waller 2014b].

9.4 The SPECjbb2013 Benchmark

The SPECjbb[®]2013 benchmark [SPEC 2013b; Pogue et al. 2014] has been developed by the Standard Performance Evaluation Corporation (SPEC). It is the successor to the SPECjbb[®]2005 benchmark [Adamson et al. 2007]. Its goal is to measure the performance and scalability of Java platforms for business applications. As part of this goal, the benchmark can be used to evaluate the whole system performance: from the underlying hardware and OS layers to the actual Java Runtime Environment (JRE) executing the benchmark application. It mainly consumes CPU, memory, and (depending

9. Macro-Benchmarks for Monitoring

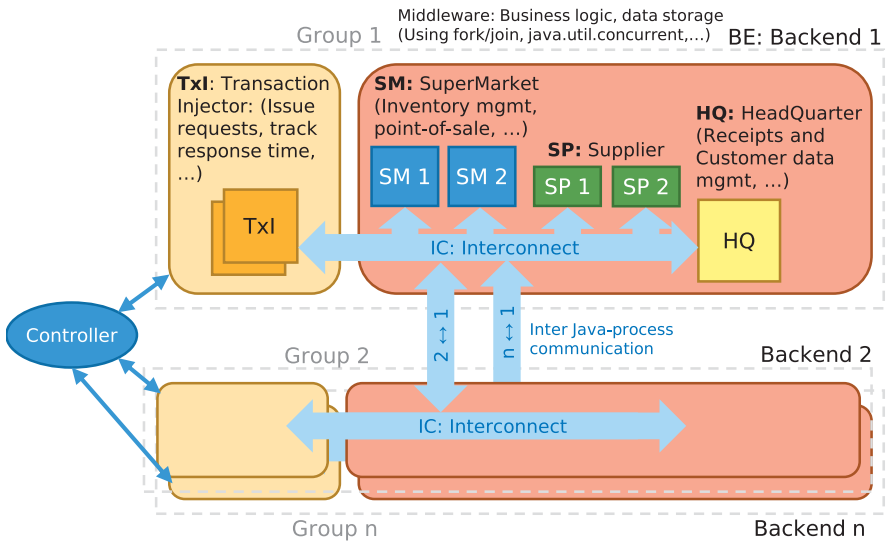


Figure 9.2. Architecture of the SPECjbb2013 benchmark (based upon SPEC [2013b])

on the used configuration) network resources while disk access is no part of the tested use cases.

The benchmark models a large supermarket company, from local supermarkets to its supply chain management, including data mining operations of the company headquarters. It is built to support the features of Java 7 as well as modern multi-core or cloud infrastructures. The SPECjbb[®]2013 consists of three components: a controller, a transaction injector (load driver), and a backend (SUT). A combination of transaction injector and backend is called group. Each group can be replicated and distributed to simulate different scenarios. Refer to the benchmark's documentation for additional details [SPEC 2013c; d].

The benchmark provides several default configurations of its components. The simplest version is deployed in a single JVM that contains a copy of all three components. For additional complexity, the three components can be distributed on different JVMs, either on the same hardware platform

9.4. The SPECjbb2013 Benchmark

or in a distributed network. Finally, the backend and the transaction injector can be replicated multiple times to simulate even larger applications.

An overview of the architecture of the SPECjbb[®]2013 is presented in Figure 9.2. Here, several groups, each consisting of one transaction injector and one backend, are depicted with their respective communication structure. In addition, the inner structure of the backend components is displayed. Each consists of several subcomponents that represent the supermarket company, i. e., the supermarkets, the supplier, and the headquarters.

Besides the provided scalability in the components of the benchmark, the employed workload is highly configurable. In its default configuration, the workload is increased until the measured response-times indicate reaching the capacity for the SUT. This maximal workload (max-jOPS) acts as one of its two measures. The other one is the maximal workload that still satisfies certain performance SLOs (critical-jOPS). In addition, we can configure a fixed workload that is utilized for a certain time. With such a steady workload, we can perform direct comparisons between the measured response times of our benchmark experiments.

Finally, the monitoring instrumentation of the SPECjbb[®]2013 is not trivial. To simulate a monitoring scenario for the supermarket company, we restrict our instrumentation to the respective Java packages that implement the components of the backend. As with all of our benchmarks, all utilized scripts and configuration parameters, as well as the full set of results, are available for download [Waller 2014b].

Meta-Monitoring: Monitoring the Monitoring Framework

In this chapter, we introduce the concept of *meta-monitoring* and its application to Kieker (Section 10.1). Additionally, we present two different 3D visualization approaches that can be employed to analyze and visualize our experimental results of meta-monitoring. The first approach is *SynchroVis*, a 3D city metaphor approach focusing on concurrency (Section 10.2). The second approach is *ExplorViz*, a combination of a 2D view of large software landscapes with a detailed 3D view of single systems (Section 10.3).

*Quis custodiet ipsos custodes?
Who watches the watchmen?*

—Juvenal (Roman poet), Satire VI

10. Meta-Monitoring: Monitoring the Monitoring Framework

Previous Publications

Parts of this chapter are already published in the following works:

1. *J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring. SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: 1st IEEE International Working Conference on Software Visualization (VISOFT 2013). IEEE Computer Society, Sept. 2013, pages 1–4*
2. *F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: 1st IEEE International Working Conference on Software Visualization (VISOFT 2013). IEEE Computer Society, Sept. 2013, pages 1–4*
3. *J. Waller. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013*

10.1 Meta-Monitoring with Kieker

Meta-monitoring usually either refers to monitoring or controlling of low-level monitoring tools from a higher abstraction level [e. g., Petcu 2013], or to the observation and collection of data from one or more other monitoring sources [e. g., Büge et al. 2010]. In our case, *meta-monitoring* refers to the monitoring of a monitoring framework while it monitors a system.

Generally speaking, all our benchmark experiments with monitoring systems could be classified as meta-monitoring. However, our goal with meta-monitoring is to observe the runtime behavior of an active monitoring system with the help of the monitoring system itself. That is, for instance, we instrument the Kieker monitoring framework with Kieker and observe its behavior.

Usually, this kind of self-monitoring is prevented in monitoring frameworks, as it would easily lead to infinite loops. For instance, assume a method call is monitored and a corresponding record is created. This act of monitoring is again monitored and again a record is created. In turn, this monitoring is monitored, and so on. This challenge can be avoided by cloning and renaming the monitoring framework. In the case of Kieker, we clone the project and change all references of Kieker to Kicker, e. g., package names or references to classes. This way, we retain the original monitoring framework, that is prevented from monitoring itself. However, we gain an identical framework, that can monitor Kieker, but also cannot monitor itself, thus avoiding any aforementioned problems.

Thus, with the help of the Kicker monitoring framework, we are able to gather program traces and performance data of the Kieker monitoring framework. Kieker provides a set of useable analyses within its analysis component that can be performed on the resulting monitoring data (see Section 4.5). These analyses can also be applied to our meta-monitoring logs to create visualizations of the monitoring process within Kieker.

The employed analyses and visualizations can provide insight in the inner performance of the monitoring framework, e. g., by quantifying the observed execution times of monitoring components. However, these measurements are very imprecise due to the combined monitoring overhead of both monitoring frameworks.

10. Meta-Monitoring: Monitoring the Monitoring Framework

Our modified Kicker monitoring framework is available with MooBench. The provided jar-file contains a ready to use experiment to instrument Kieker. The necessary configuration files are contained in the META-INF folder of the file.

In the following, we present two 3D visualization that we employ to analyze and visualize the results of our meta-monitoring experiments. These results are provided with our evaluation of MooBench in Chapter 14. An overview on further typical visualization of monitoring data is given by Isaacs et al. [2014].

10.2 SynchroVis: Visualizing Concurrency in 3D

The SynchroVis visualization approach has already been published in Waller et al. [2013]. It is originally based upon the bachelor thesis of Wulf [2010] and upon the master thesis of Döhring [2012]. Its goal is visualizing the static structure of a software system, e. g., classes and packages, in combination with dynamic information, e. g., traces and concurrent behavior.

The city metaphor is a 3D visualization approach displaying a software system as a large city. The viewers day-to-day familiarity in navigating a city (e. g., reading a street map, orienting with the help of large buildings, etc.) supports the understanding of the visualized application [Wettel and Lanza 2007].

In our SynchroVis approach, we employ the city metaphor to improve program comprehension for software systems. Our visualization of the system's static structure is based on a source code analysis, while the dynamic behavior is gathered from information collected in monitoring traces. The focus of our approach is on providing a detailed visualization of the system's concurrent behavior. Typical tasks include understanding the existence and interaction (e. g., the mutual calling behavior) of classes or components in a software system under study, as well as understanding concurrency (e. g., locking and starvation).

The SynchroVis tool, including some examples, is available as open source software.¹

¹<http://kieker-monitoring.net/download/synchrovis/>

10.2.1 Our City Metaphor

A schematic view of our city metaphor used as the basis for our visualization approach is depicted in Figure 10.1a. We employ the three general concepts of (1) *districts* to break our city into parts, of (2) *buildings* to represent static parts of the software system, and of (3) *streets* to connect our static parts according to dynamic interactions.

An example of visualizing the static information of a large software system is depicted in Figure 10.2. In this case, the packages, classes, and relations of the Java-based Vuze Bittorrent Client are displayed.

Districts Packages (e. g., in Java) or components (as a more general concept) form the districts of our city metaphor. Each package is visualized as a rectangular layer with a fixed height. Several packages are stacked upon one another to display their subpackage hierarchies with increasing lightness.

Buildings Each class is represented by a building which is placed in its corresponding district (determined by its package). The ground floor of the building represents the actual class object, while the upper floors represent dynamically created class instances.

Streets Operation calls contained in the program trace are represented by streets, i. e., colored arrows between floors of the same or different buildings. Each color corresponds to a single thread to simplify the comprehension of different traces. Arrows entering the ground floor represent either calls of static operations or of constructors. A constructor call also adds a new floor to the building. Arrows entering upper floors represent operation calls on the corresponding object instances.

In addition, SynchroVis allows to show or to hide static relations of either a selected class or of all classes in our visualization. All of these relations are displayed as arrows connecting building roofs. Black arrows symbolize inheritance relationships, gray arrows symbolize interface implementations, while white arrows symbolize general associations.

10. Meta-Monitoring: Monitoring the Monitoring Framework

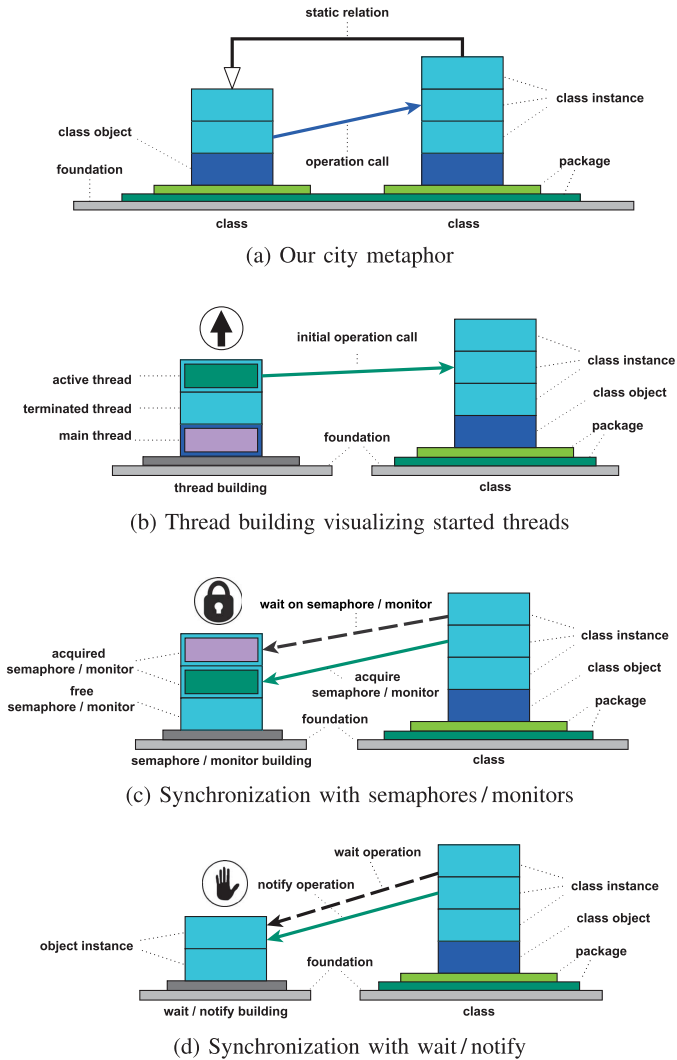


Figure 10.1. Schematic view of our SynchroVis approach (based upon [Döhning 2012])

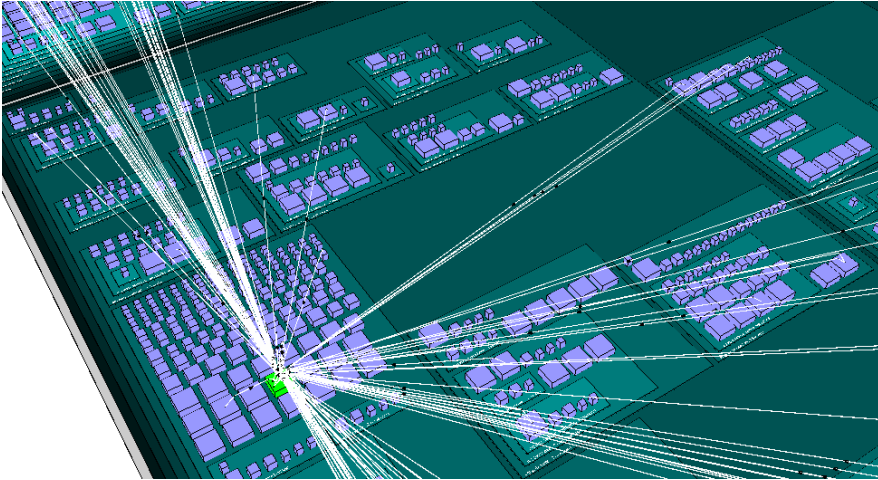


Figure 10.2. Visualizing the static structure of the large software system Vuze

10.2.2 Mechanisms for Visualizing Concurrency

Besides visualizing the interleaving of concurrent threads as described in the previous section, we also provide mechanisms to visualize four specific synchronization concepts.

Threads All program traces start at a special thread building in an external district. The ground floor of the thread building represents the starting thread of the program execution. Each time a new thread starts, a new floor is added, colored in the respective thread's color. The initial arrow of the new trace starts within the respective floor of this thread building. In case of thread termination, the associated floor loses its color. Refer to Figure 10.1b for a schematic representation.

Monitor/Semaphore A classic synchronization concept is the monitor, e.g., realized by the synchronized keyword in Java. It is similar to the concept of binary semaphores. Each semaphore or monitor is visualized

10. Meta-Monitoring: Monitoring the Monitoring Framework

by a separate floor on a specific semaphore building next to the thread building. If the monitor is acquired, the floor gets colored by the respective thread's color, otherwise it is uncolored. Furthermore, any successful acquire or entry operation is depicted with a solid arrow directed between source class instance floor and the semaphore floor. Blocking operations are depicted with similar dashed arrows. Thus, a waiting thread is visible by its directed arrow entering a differently colored semaphore. Similarly, it is possible to spot deadlocks by comparing the different semaphore floors and their respective waiting threads. This semaphore / monitor mechanism is depicted in Figure 10.1c.

Wait/Notify The next synchronization concept supported by our SynchroVis approach is the wait and notify mechanism. Again, we add a special building to the extra district and assign a floor to each object a thread is waiting on. Each wait operation is depicted with a dashed arrow between the source floor and the respective floor of the special building, while each notify or notifyAll operation is depicted with a solid arrow. This allows for a visualization very similar to the visualization of locking and deadlock behavior of semaphores. This mechanism is illustrated in Figure 10.1d.

Thread Join The final synchronization concept realized within our approach is the joining concept of threads, where threads wait upon the completion of another thread. This is visualized by dashed arrows into the respective floor of the thread building that the other threads are waiting upon.

10.2.3 Displaying and Navigating Collected Program Traces

The SynchroVis tool provides the usual means for interactively navigating the city (e. g., moving, rotating, and zooming), as well as for searching and locating specific entities within. It is also possible to select an arbitrary scene element to get further information on it.

Furthermore, SynchroVis provides specific support for navigating program traces. The user is able to use both a chronological and a hierarchical display of program traces. The *chronological display* allows to iterate over all

10.2. SynchroVis: Visualizing Concurrency in 3D

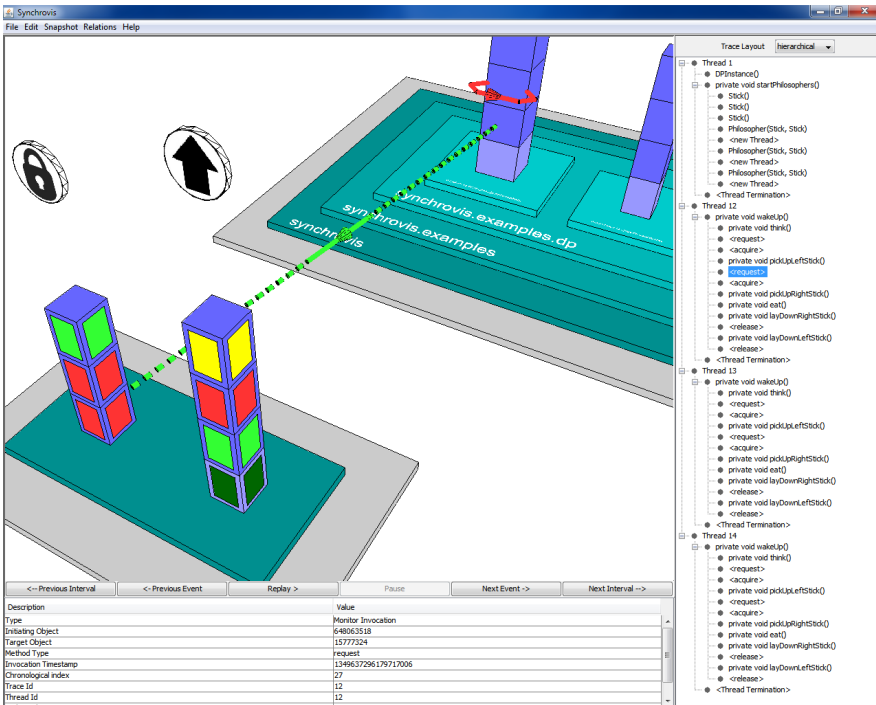


Figure 10.3. Visualizing a normal run of the dining philosophers problem

events of all threads sorted by time in ascending order, while the *hierarchical display* allows to iterate over the events of a single thread. Additionally, it is possible to use a *time-based stepping* with a configurable interval.

For enhanced usability, SynchroVis provides the option to directly jump to a specific point in time or to a specific event. Moreover, it offers the possibility to automatically step through the trace by means of a *movie mode*. In this mode, the user is able to watch and to study the recorded behavior of the application.

10. Meta-Monitoring: Monitoring the Monitoring Framework

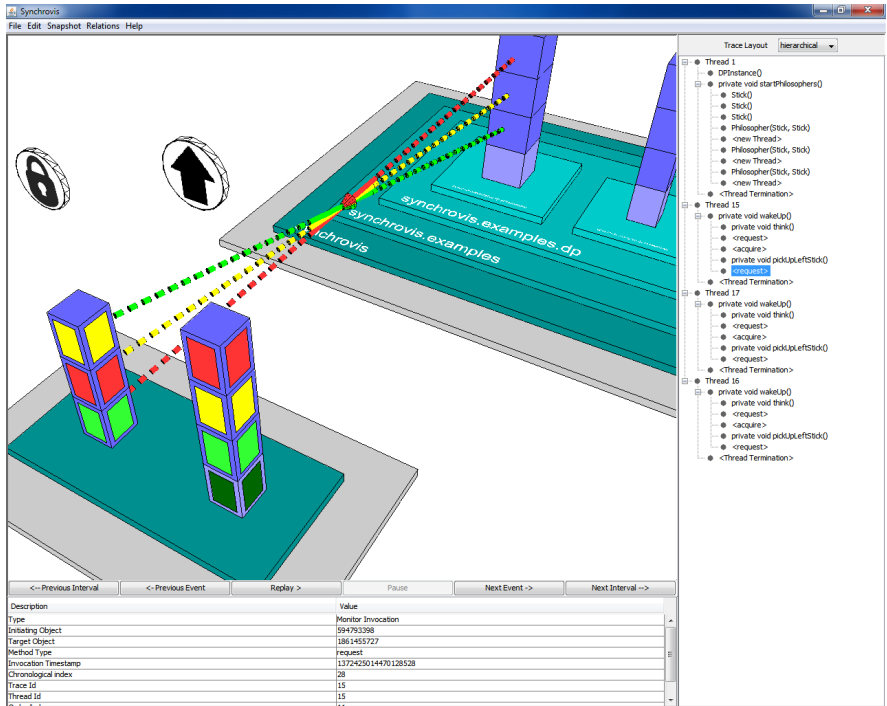


Figure 10.4. Visualizing a deadlock in the dining philosophers problem

10.2.4 Exemplary Concurrency Analysis with Synchronis

As an example scenario, we describe a typical concurrency analysis of the well known dining philosophers problem with Synchronis. This scenario is often used to study locking behavior, especially deadlocks. For the sake of simplicity, we assume three present philosophers, realized by separate threads, and three shared forks, realized by separate monitors.

A normal run of the dining philosophers problem is presented in Figure 10.3. Here, the philosopher represented by the red thread has acquired his two forks (monitors) and is executing his eating time. The green philosopher has acquired only a single fork and is waiting for the release of his

second fork, currently held by the red thread. This is visualized by the green arrow pointing at the red floor of the semaphore / monitor building.

A run resulting in a deadlock is presented in Figure 10.4. In this case, all three philosophers have managed to acquire a single fork. Thus, all three threads are waiting upon each other to release the respective monitors. This is visualized by the colored arrows pointing at the differently colored floors of the semaphore / monitor building. The navigation within the program traces allows for an analysis of the cause of this deadlock.

10.3 ExplorViz: Visualizing Software Landscapes

The visualization approach of ExplorViz has already been published in Fittkau et al. [2013a]. It is part of the ExplorViz project,² that also provides a high-throughput optimized version of Kieker (see our evaluation in Section 11.5).

The focus of ExplorViz is visualizing large software landscapes of typical enterprise systems. A corresponding perspective of our visualization provides knowledge about the interaction of different applications and nodes in the software landscape to enhance system comprehension. It uses a 2D visualization with a mix of UML deployment and activity diagram elements. This landscape perspective is combined with a 3D city metaphor, visualizing the communication of single software systems or components within the landscape.

For large software landscapes, a lot of information must be processed and presented within a limited time span. Thus, our presentation of information must be easy to understand and limited to the actually required data. As a solution, our ExplorViz approach reveals additional details on demand, e. g., communications on deeper system levels.

Our goal of meta-monitoring is not within the actual focus of ExplorViz. On the landscape level, most monitoring tools only provide one or a maximum of two systems, including the SUM. However, the system level perspective can provide insights into the inner workings and communications of the monitoring system under test.

²<http://www.explorviz.net/>

10. Meta-Monitoring: Monitoring the Monitoring Framework

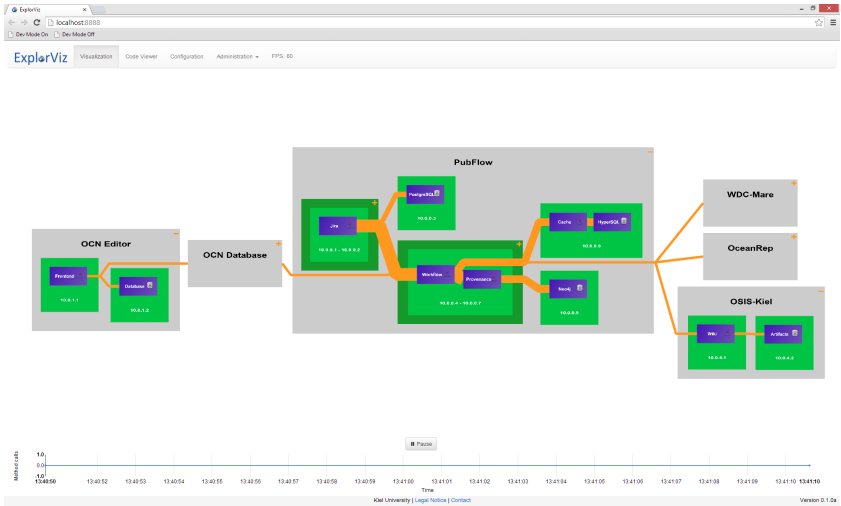


Figure 10.5. Example of the ExplorViz landscape level [Fittkau et al. 2013a]

10.3.1 Landscape Level Perspective

An example of the *landscape level perspective* is provided in Figure 10.5. Several applications, their respective deployment, and their communication with each other are visualized. Edges depict the communication, the respective thickness of the edges represents the number of requests in the currently active data set.

Refer to Fittkau et al. [2013a] for further details on the landscape level perspective. Due to its limited applicability for our intended meta-monitoring scenario, we omit further details.

10.3.2 System Level Perspective

The system level perspective is based upon a 3D city metaphor (see also Section 10.2). Similar to our visualization with SynchroVis, our city consists of several components: districts, buildings, and streets. An example of a system visualized on the system level is provided in Figure 10.6.

10.3. ExplorViz: Visualizing Software Landscapes

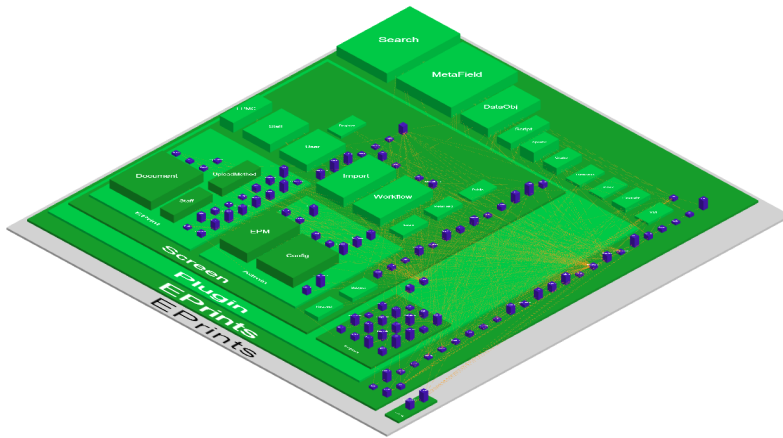


Figure 10.6. Example of the ExplorViz system level [Fittkau et al. 2013a]

Districts Components or subcomponents (e. g., packages in Java) form the districts in our city metaphor. Each component is visualized as a rectangular layer with a fixed height. Several components are stacked upon one another to display their subcomponent hierarchies.

Buildings Buildings represent entities, i.e., components, subcomponents, or classes. In our city metaphor, buildings become districts when they are opened to provide additional details on their inner workings. The maximal count of current instances in each entity maps to the height of the corresponding building. If the entity is a class, the current instance count of the class forms the height of the building. Furthermore, the width of a building is determined by the number of classes inside the represented entity. If the entity is a class, the width is a constant minimal value.

Streets Streets visualize the communication between districts and buildings. They are represented by pipes between the respective elements. The thickness of the pipes represents the call count between the entities within the data set.

Part III

Evaluation

Evaluation of Kieker with MooBench

In this chapter, we evaluate several aspects of the Kieker monitoring framework with the help of MooBench. First, we provide a general *introduction* to our experiments with Kieker (Section 11.1). In Section 11.2, we compare the *performance of several release versions of Kieker*. Next, we demonstrate the *linear scalability* of monitoring overhead (Section 11.3). Several experiments to compare the *influence of different multi-core environments* are presented in Section 11.4. In Section 11.5, we employ MooBench to *steer performance tunings* of Kieker. The performance influence of *live analysis with Kieker* is evaluated in Section 11.6. Finally, in Section 11.7, we employ MooBench within a continuous integration setting to *continuously observe and control the monitoring overhead*.

*The fundamental principle of science, the definition almost, is this:
the sole test of the validity of any idea is experiment.*

—Richard P. Feynman, recipient of the Nobel price

11. Evaluation of Kieker with MooBench

Previous Publications

Parts of this chapter are already published in the following works:

1. J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In: *Multicore Software Engineering, Performance, and Tools (MSEPT)*. Springer, June 2012, pages 42–53
2. J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68
3. F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 89–98
4. J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69
5. J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes* (2015). Submitted, pages 1–4

11.1 Introduction to our Evaluation

Our goal within this chapter is to evaluate the capabilities of MooBench. This evaluation is performed with the help of a series of experiments with the Kieker monitoring framework (see Section 4.5). In each experiment, we evaluate a specific aspect of the performance and of the monitoring overhead that is caused by Kieker. In doing so, we can demonstrate the feasibility of MooBench as a benchmark for Kieker.

We further reinforce the general feasibility of MooBench as a micro-benchmark to determine the overhead of monitoring frameworks in the next chapters. In Chapter 12, we perform further experiments to demonstrate the capabilities of MooBench when benchmarking additional monitoring tools. Furthermore, we confirm our findings by comparing our results to the results of macro-benchmarks (Chapter 13). Finally, we compare our findings to the results of a meta-monitoring experiment performed with Kieker (Chapter 14).

11.1.1 Validation of our Experiments

All experiments with MooBench are executed, analyzed, and presented according to our benchmark engineering process (see Chapter 8). The general presentation of all our benchmark experiments is similar: First, we describe our scenario. Next, we describe the parameters that are relevant for our benchmark and its environment. Finally, we present our measurements and discuss the results.

In addition to the description of our experiments, we provide pre-configured downloads of MooBench to enable repetitions and validation of our findings. Depending on the experiment and on our findings, our reports of the measurement results are more or less detailed. However, in all cases, sufficient details are presented to draw meaningful conclusions. Furthermore, the complete list of findings, including all raw data, is available for download with each experiment. Thus, all of our findings can be validated.

11. Evaluation of Kieker with MooBench

11.1.2 Note on the Presentation of Results

In our presentation of results, we intend to highlight the division of overhead into its three causes (see Section 6.2). Thus, we do not employ box plots [Montgomery and Runger 2010] but rather a variation thereof.

We employ composite bar charts with stacked columns. One of the more complex examples is presented in Figure 11.4 on page 189. Each shaded area represents one of the causes of monitoring overhead (refer to the legend for details). In this case, the shaded areas represent the median values, while the remaining quartiles are displayed to the left of the column. In addition, we include a second stacked column behind the first one. It represents the mean values with the respective 95%-confidence intervals annotated to the right of the column.

11.2 Performance Comparison of Kieker Versions

In this section, we evaluate the capabilities of our MooBench benchmark by conducting a performance comparison of the different released Kieker versions using our experimental design to measure the three causes of monitoring overhead that have been introduced in Section 8.2.2 on page 140. We have originally published parts of these findings in Waller and Hasselbring [2013a]. All results, as well as a prepared and pre-configured version of our benchmark setup, are published online [Waller and Hasselbring 2013b; c; Waller 2014b].

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 versions up to the current version 1.9, that has been released in April, 2014.

In all cases, we use the required libraries, i. e., AspectJ and Apache Commons Logging, in the provided versions for the respective Kieker releases. Additionally, we perform 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 in the case of a full buffer.

11.2.1 General Benchmark Parameters

All experiments that we perform on different version of the Kieker monitoring framework utilize the same set of environment, workload, and system parameters. We conduct our performance comparison running on Solaris 10 with the Oracle Java 64-bit Server JVM in version 1.7.0_45 with up to 4 GiB of heap space provided to the JVM. The instrumentation of the monitored application is performed through load-time weaving using the AspectJ releases corresponding to the Kieker versions.

We perform all experiments on two different sets of hardware: First, we utilize an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM. Next, we employ an X6240 Blade Server with two AMD Opteron 2384 2.7 GHz Quadcore processors and 16 GiB RAM. The respective hardware and software system is used exclusively for the experiments and is otherwise held idle. This combination of different hardware architectures provides additional information on the behavior of Kieker on different target platforms.

The majority of our experiments are performed using probes producing `OperationExecutionRecords`. For measuring the overhead of writing (W), we focus on the asynchronous ASCII writer, producing human-readable CSV files, that is available in all tested 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 `OperationExecutionRecords`. 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 recursion depth of ten. As recommended, we use a warm-up period of 1,000,000 measurements. An exemplary time series diagram for Kieker 1.8 with event based probes and a binary writer is presented in Figure 11.1. The recommended warm-up period is shaded in gray in the figure.

11. Evaluation of Kieker with MooBench

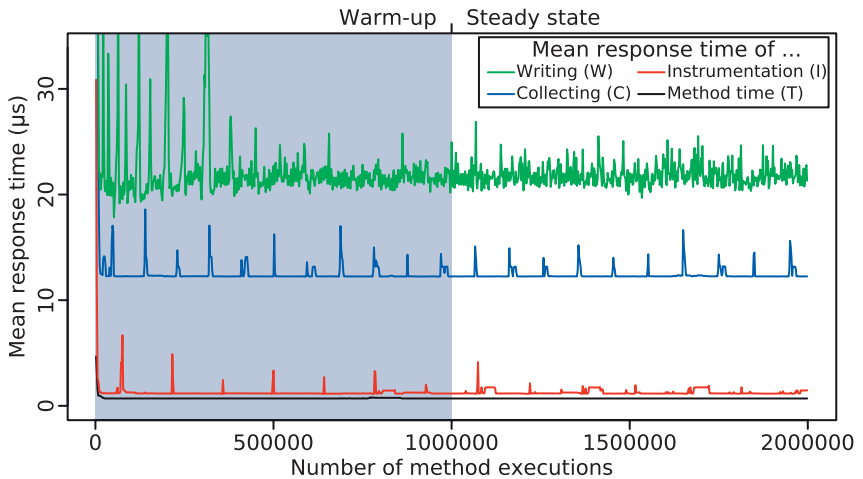


Figure 11.1. Exemplary time series diagram of measured timings with Kieker [Waller and Hasselbring 2013a]

Furthermore, we either use a configured execution time of $0\ \mu\text{s}$ or a configured execution time of $200\ \mu\text{s}$. This way, we can estimate an upper bound as well as a lower bound on the monitoring overhead of the employed monitoring writers. Refer to the description of the required decision concerning the workload in Section 8.2.4 on page 145 for further details.

In the case of state-based probes, a total of ten `OperationExecutionRecords` is collected and written per measured execution of the monitored `Method()`. 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 Georges et al. 2007, 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.

11.2. Performance Comparison of Kieker Versions

Table 11.1. Response times for Kieker release 1.9 (in μs)
(using an asynchronous ASCII writer and `OperationExecutionRecords`)

	No instr.	Deactiv.	Collecting	Writing
Mean	0.83	1.35	9.90	34.62
95% CI	± 0.00	± 0.01	± 0.01	± 3.60
Q ₁	0.83	1.27	9.81	10.80
Median	0.83	1.30	9.83	12.25
Q ₃	0.83	1.36	9.86	15.52
Min	0.80	1.20	9.73	10.11
Max	128.25	4612.80	4406.34	3520088.05

11.2.2 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 tested versions of Kieker.

A diagram containing mean response times with 95%-confidence intervals for the three causes of monitoring overhead is presented in Figure 11.2. Although our benchmark also measured and calculated the quartiles, minima, and maxima, we omit these in the diagram to reduce visual clutter and to improve the clarity of our results. However, as mentioned before, all benchmark results are available for download. Furthermore, we present the full set of results for Kieker release 1.9 in Table 11.1 as an example.

With the exception of Kieker 1.7, the mean 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 ver-

11. Evaluation of Kieker with MooBench

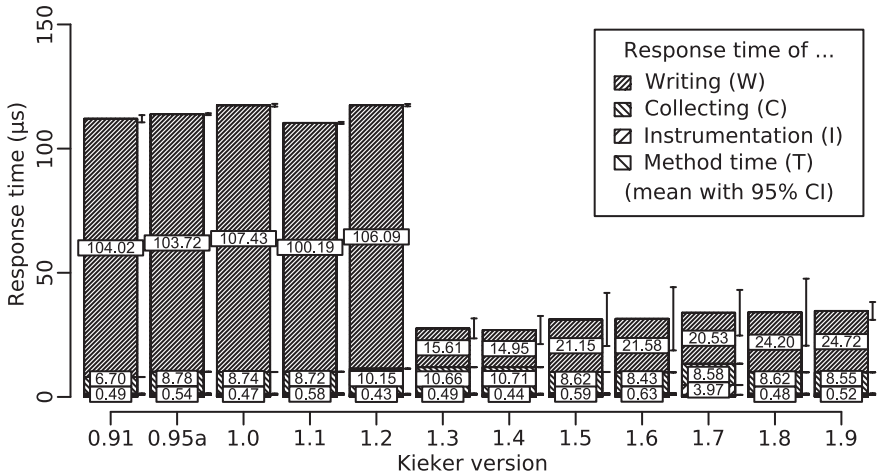


Figure 11.2. Performance comparison of twelve different Kieker versions using an asynchronous ASCII writer and `OperationExecutionRecords` with a configured method execution time of $0\mu\text{s}$ (based upon Waller and Hasselbring [2013a])

sion 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 1.9 has a median writing overhead of $2.42\mu\text{s}$ ($12.25\mu\text{s} - 9.83\mu\text{s} = 2.42\mu\text{s}$). However, a small percentage of executions has extremely high response times of more than one second, as is also evident through the large span of the confidence intervals and the measured maximal response time of over 3.5 s.

As a next step, we repeat our performance benchmarks with a configured method execution time of $200\mu\text{s}$. This way, we can determine a lower bound on monitoring overhead portion *W*. The results of these runs are presented in Figure 11.3.

11.2. Performance Comparison of Kieker Versions

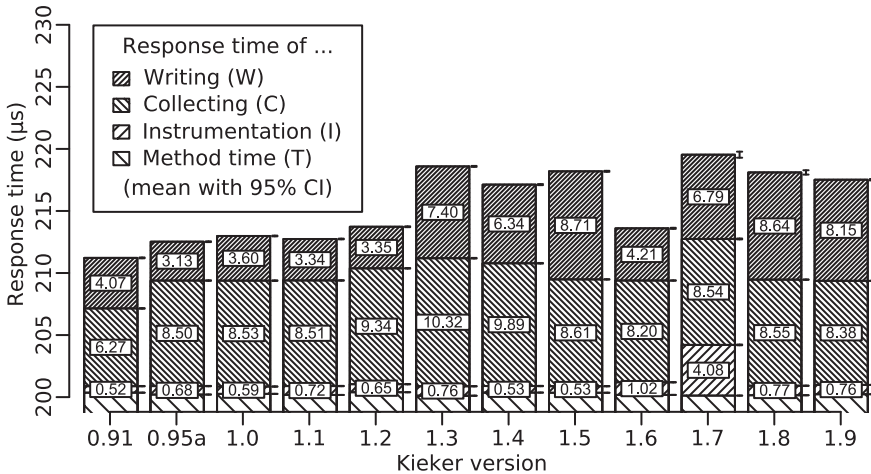


Figure 11.3. Performance comparison of twelve different Kieker versions using an asynchronous ASCII writer and `OperationExecutionRecords` with a configured method execution time of $200\ \mu\text{s}$ (based upon Waller and Hasselbring [2013a])

The overhead of instrumentation (*I*) is similar to or slightly above our benchmark results with a method execution time of $0\ \mu\text{s}$. This small increase can probably be explained by the changes in timings and the resulting differences in JIT compilations. A similar result can be seen with regards to the overhead of collecting (*C*). Here, the measured response times are similar to or slightly below our previous experiments. Again, these changes are probably due to the overall changes in timing, affecting the JIT as well as the performed garbage collections.

However, a major difference can be spotted regarding the overhead of writing (*W*). With all versions of Kieker, the average writing overhead is considerably lower compared to our previous experiments. For instance, the first five versions of Kieker have a lower bound of about $4\ \mu\text{s}$ average writing overhead *W*. This suggests that the earlier versions of Kieker provided a better performance with low workloads compared to newer versions. However, the more recent versions also contain additional features.

11. Evaluation of Kieker with MooBench

Of special note is Kieker version 1.6. Here, the overhead of writing with a low workload (4.21 μs) is comparable to the earlier versions of Kieker. This improvement is achieved by performance tunings in the handling of Strings, especially important for the ASCII writer. The performance degradation with version 1.7 (6.79 μs) is caused by additional error handling that has been missing in previous versions.

With this minimal workload, the 95% confidence intervals are very small, hinting at constant overhead with very few outliers. This is additionally confirmed by an analysis of the median overhead values. In summary, this demonstrates the asynchronous decoupling of the writer thread.

The combination of the two series of benchmark experiments, one with a stress workload and one with a minimal workload, allows for an estimate of the actual performance overhead per set of ten method executions on similar systems. In most cases, the actually observed overhead will be between the two measured values of the minimal and maximal workload.

Finally, we repeat the series of benchmark experiments on a different hardware infrastructure, e. g., employing AMD processors. Although the actual measured numbers differ, the overall results remain the same and confirm our previous findings. We can observe similar changes in overhead with the respective Kieker versions as in the Intel-based experiments. The full set of results and analyses of these additional experiments are available online [Waller and Hasselbring 2013b].

11.2.3 Performance Comparison: Binary Writer & Event Records

For our second performance comparison of different Kieker versions, we employ event-based probes producing `kier.common.record.flow` event records introduced with Kieker 1.5. Furthermore, we use the asynchronous binary writer, also introduced in version 1.5. The experiments are executed once with a configured method execution time of 0 μs and once with a configured method execution time of 200 μs . The results are summarized in Figure 11.4.

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.

11.2. Performance Comparison of Kieker Versions

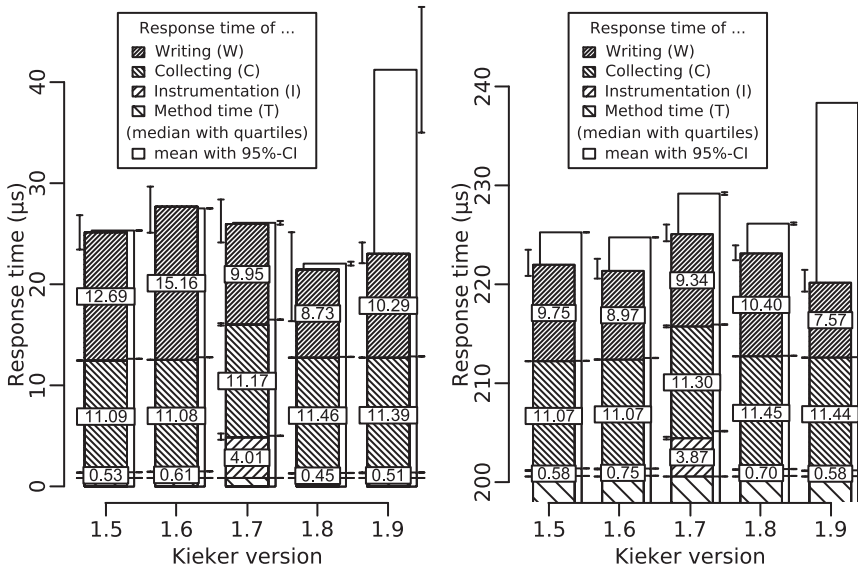


Figure 11.4. Performance comparison of five different Kieker versions using an asynchronous binary writer and event-based probes with a configured method execution time of either 0 μs or 200 μs (based upon Waller and Hasselbring [2013a])

The overhead of collecting monitoring data (C) with event-based probes is higher, compared to the overhead when using state-based probes (cf., Figure 11.2 and Figure 11.3). 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 Kieker versions 1.7 and 1.8. Furthermore, compared to the ASCII writer of the previous subsection, the average performance has improved considerably (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.

11. Evaluation of Kieker with MooBench

In our evaluation of Kieker version 1.9, we can detect a new performance degradation when employing the binary writer. This is especially visible in the mean overhead of writing. This degradation is caused by changes in the employed serialization mechanism and will probably be handled with the next Kieker release.

As with the previous set of experiments, our findings with the AMD hardware infrastructure confirm our findings on the Intel platform. These additional results are available online [Waller and Hasselbring 2013b].

11.2.4 Further Performance Comparisons

In addition to the two presented series of benchmark experiments, we have also tested other combinations. That is, event-based records with the ASCII writer and state-based probes with the binary writer. However, these additional experiments and analyses result in no further findings, but rather confirm our already presented ones. For instance, the performance degradation of the binary writer in Kieker version 1.9 is also evident when using `OperationExecutionRecords` instead of event-based records.

These additional results as well as the already presented ones, including the experiments utilizing the AMD platform, are available online [Waller and Hasselbring 2013b; Waller 2014b]. In addition to our benchmark results, we also provide the pre-configured benchmark itself with all required libraries and Kieker versions [Waller and Hasselbring 2013c]. Thus, repetitions and validations of our experiments are facilitated.

11.2.5 Conclusions

Our performance comparison of up to twelve different Kieker versions with the help of the MooBench micro-benchmark demonstrates the benchmark's capabilities. Three main conclusions can be drawn from our results:

1. MooBench is capable of benchmarking all released versions of Kieker. Only minor adjustments are necessary with the earlier versions to determine all individual portions of monitoring overhead. The total overhead can be determined without any changes to the released versions of the monitoring framework.

11.3. The Scalability of Monitoring Overhead

2. With the help of our benchmark results, we are able to compare different releases of Kieker with each other. Furthermore, we are also able to compare the influence of different configurations or environments. Due to these comparisons, we are able to verify intended performance changes, e. g., the API rewrite between versions 1.2 and 1.3.
3. Finally, we are able to detect unintended performance regressions in our benchmark results. Two already mentioned major examples have been found in version 1.7 and in the binary writer with release 1.9. In both cases, this unintended behavior has only been detected due to our benchmark.

Especially the third conclusion highlights an important contribution of our benchmark. After their detection, these performance regressions have been transformed into tickets in the Kieker issue tracking system: ticket #996¹ and ticket #1247.² This way, these regressions can be fixed for future releases. Ideally, our benchmark would automatically catch these problems before any public release. This leads to our work on including the benchmark into our continuous integration system (see Section 11.7).

11.3 The Scalability of Monitoring Overhead

In our next series of benchmark experiments, we demonstrate the expressiveness of the recursion depth in our MooBench micro-benchmark. These findings have originally been published in Waller and Hasselbring [2012a]. The results and a pre-configured benchmark are available online [Waller and Hasselbring 2012b; c]. Similar results for an earlier release of Kieker are published in van Hoorn et al. [2009b].

For monitoring frameworks, only a linear increase of monitoring overhead with increasing workload is acceptable for good scalability. In order to determine whether the increase of the amount of monitoring overhead with each additional monitored method call is linear, we perform a series of benchmark experiments with increasing recursion depths.

¹Ticket #996: <http://trac.kieker-monitoring.net/ticket/996>

²Ticket #1247: <http://trac.kieker-monitoring.net/ticket/1247>

11. Evaluation of Kieker with MooBench

11.3.1 Configuration of the Benchmark Parameters

We utilize a similar benchmark environment as in our previous experiments. Our hardware platform is again a X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM running Solaris 10 and an Oracle Java 64-bit Server JVM in version 1.6.0_26 with 1 GiB of heap space. We use a nightly build of Kieker release 1.5 (dated 09.01.2012) as the monitoring framework under test. AspectJ release 1.6.12 with load-time weaving is used to insert the particular `MonitoringProbes` into the Java bytecode of the `MonitoredClass`. Aside from the experiment, the server machine is held idle and is not utilized.

The rest of our benchmark parameters is similar to our previous experiments, too. We repeat the experiments on ten identically configured JVM instances, utilizing a single benchmark thread, and calling the `monitoredMethod()` 2,000,000 times on each run with a configured method time of 500 μ s per method call. Thus, the configured workload is a minimal one. We discard the first 1,000,000 measured executions as the warm-up period and use the second 1,000,000 steady state executions to determine our results. The configured recursion depth varies within our series of experiment runs. An exemplary time series diagram illustrating the warm-up period in this series of experiments is presented in Figure 11.5

Kieker is configured to create `OperationExecutionRecords` and to utilize the asynchronous ASCII writer with a configured queue size of 100,000 entries.

11.3.2 Linear Increase of Monitoring Overhead

In order to demonstrate the linear increase in monitoring overhead, we increase the workload of our benchmark. Thus, in each experiment run, each call of the `monitoredMethod()` results in additional recursive (monitored) calls of this method, enabling us to measure the overhead of monitoring multiple successive method calls.

The results of this experiment are presented in Figure 11.6 and described below. The measured overhead of instrumentation I increases with a constant value of approximately 0.1 μ s per call. The overhead of collecting data C increases with a constant value of approximately 0.9 μ s per call.

11.3. The Scalability of Monitoring Overhead

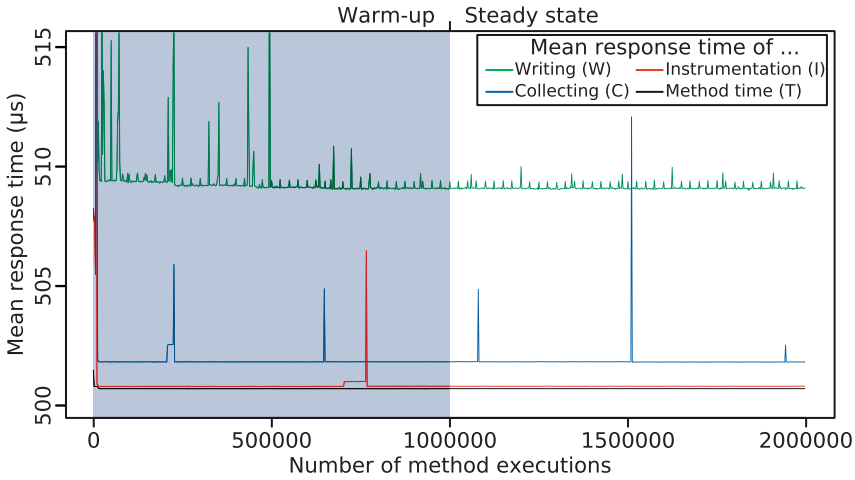


Figure 11.5. Exemplary time series diagram of measured timings with Kieker [Waller and Hasselbring 2012a]

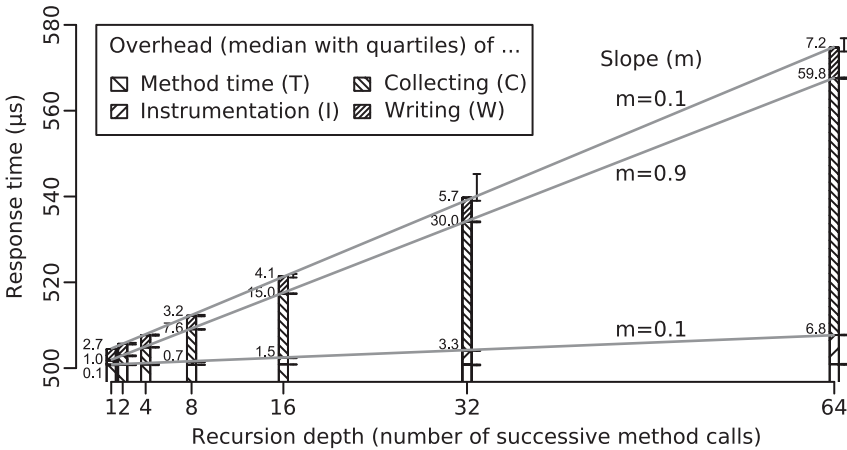


Figure 11.6. Linear Increase of Monitoring Overhead [Waller and Hasselbring 2012a]

11. Evaluation of Kieker with MooBench

The overhead of writing W consists of two parts: a constant overhead of approximately $2.5\ \mu\text{s}$ during the period of $500\ \mu\text{s}$ and an increasing value of approximately $0.1\ \mu\text{s}$ per additional call.

Our experiments include recursion depths up to 64 method calls per configured method time of $500\ \mu\text{s}$. With higher values of the recursion depth, the monitoring system records method calls faster than it is able to store monitoring records in the file system, resulting in a stress workload.

In each experiment run, the Monitoring Writer has to process 362 MiB of monitoring log data per step of recursion depth. In the case of a recursion depth of 64, 23 GiB Kieker monitoring log data were processed and written to disk within the 20 minutes execution time (at $19.3\ \text{MiB/s}$).

11.3.3 Conclusions

With the described series of benchmark experiments, we are able to demonstrate the linear scalability of monitoring overhead in the Kieker framework with increasing workload. Furthermore, we can demonstrate the capability of MooBench to provide scaling workloads by configuring the recursion depth parameter besides the method time parameter.

11.4 Experiments with Multi-core Environments

Our next series of experiments has also already been published in Waller and Hasselbring [2012a]. Hence, the results and a pre-configured benchmark are available online [Waller and Hasselbring 2012b; c], too.

The original goal of the experiments is testing the influence of different multi-core architectures and of available cores on the monitoring overhead. In doing so, we can demonstrate the feasibility of performing these comparisons with MooBench.

11.4.1 General Benchmark Parameters

The configuration of our environment and our benchmark parameters is similar to the previous series of experiments (see Section 11.3.1). However,

11.4. Experiments with Multi-core Environments

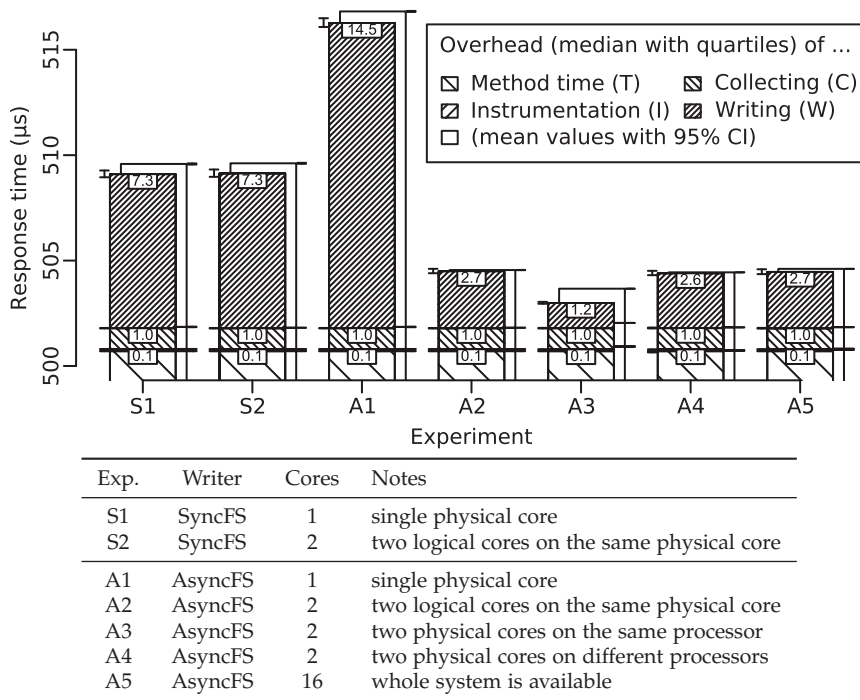


Figure 11.7. Single-Threaded Monitoring Overhead [Waller and Hasselbring 2012a]

we employ a constant recursion depth of one, i. e., no additional recursive calls are performed.

In this configuration, each experiment consists of four independent runs and each run takes a total time of 20 minutes. Each run with an active Monitoring Writer produces at least 362 MiB of Kieker monitoring log files.

We configure the Kieker monitoring framework to either use the asynchronous ASCII writer (AsyncFS) or a synchronous version of the same writer (SyncFS). Otherwise, the configuration of Kieker is not adjusted.

Our employed X6270 Blade Server contains two processors, each processor consists of four cores, and each core is split into two logical cores via hyper-threading. We are using operating system commands to assign

11. Evaluation of Kieker with MooBench

only a subset of the available cores to the benchmark system containing the monitored application and the monitoring framework. On our Solaris 10 server we use the `psrset` command. Similar commands are available on other operating systems.

11.4.2 The Influence of Available Cores

The focus of this series of experiments is to quantify the three portions of monitoring overhead and to measure the influence of different assignments of multiple cores or processors to the application (and to the Monitoring component) on the monitoring overhead.

The results of the experiments are presented in Figure 11.7 and described below. Additionally, the assignment of cores and the employed writer are also documented in the table accompanying the figure.

- S1 We start our series of experiments with a synchronous ASCII writer, thus disabling the internal buffer and the asynchronous `WriterThread`, yielding a single-threaded benchmark system. First, we assign a single physical core to the application and disable its second logical core, thus simulating a single core system. The main portion of overhead in S1 is generated by the writer W ($7.3 \mu\text{s}$) that has to share its execution time with the monitored application. The overhead of the instrumentation I is negligible ($0.1 \mu\text{s}$), the overhead of the data collection C is low ($1.0 \mu\text{s}$).
- S2 In Experiment S2 we activate two logical cores (hyper-threading) in a single physical core and repeat the experiment with the synchronous writer. There is no significant difference between one or two assigned cores. For this reason we omit further synchronous experiments. Only with asynchronous writing, multiple processing units may reduce the monitoring overhead.
- A1 We continue the rest of our series of experiments with the asynchronous ASCII writer. Similar to experiment S1, we assign a single physical core to the application and disable its second logical core. The portion W of the overhead caused by the writer ($14.5 \mu\text{s}$) is almost doubled compared to the synchronous writer. This can be explained by the writer thread

11.4. Experiments with Multi-core Environments

sharing its execution time with the monitored application. Compared to the experiment S1, context switches and synchronization between the two active threads degrade the performance of the system.

- A2 Next, we activate two logical cores in a single physical core. The additional core has no measurable influence on the overhead of instrumentation I ($0.1 \mu\text{s}$) and collecting data C ($1.0 \mu\text{s}$). Due to the additional available core, which is exclusively used by the writer thread, the overhead of writing the data W ($2.7 \mu\text{s}$) is significantly reduced. Even though both logical cores have to share the resources of a single physical core, the second logical core has been an enormous improvement. The overhead has been reduced by 55% compared to the overhead of the synchronous writer (S1) and by 76% compared to the overhead of the single core system with the asynchronous writer (A1).
- A3 In this experiment we assign two different physical cores on the same processor to the benchmark system. This setup provides the best results of the series of experiments with a greatly improved writer performance W ($1.2 \mu\text{s}$). The improvement can be explained by no longer sharing the processing resources of a single physical core by two logical cores (via hyper-threading). Thus, the overhead of monitoring has been reduced by 73% compared to the overhead of the synchronous writer (S1) and by 85% compared to the overhead of the single core system with the asynchronous writer (A1).
- A4 Next, we assign two physical cores of two different processors on the motherboard. The increased synchronization overhead between two different processors causes results similar to A2.
- A5 Finally, we activate all physical and logical cores in the system. Since the monitored software system uses a single thread and the monitoring framework uses an additional writer thread, no additional benefit of more than two available cores is measurable: the two threads (one for the application and one for monitoring) cannot exploit more than two cores.

Overall, the experiments provided very stable results with the mean response time being only slightly above the measured median. In summary,

11. Evaluation of Kieker with MooBench

the Kieker monitoring framework can benefit greatly from a single available core that can be used exclusively for writing the monitoring data. However, in high load environments with sparse processing resources, a synchronous monitoring writer can provide better performance compared to an asynchronous writer.

11.4.3 The Influence of Different Multi-core Architectures

In this experiment, we compare the results of our benchmarks on several different multi-core architectures with each other. Similar results with an AMD platform have already been sketched in Section 11.2.

Besides the X6270 Blade server with two Intel Xeon E5540 processors (Intel), we use the already mentioned X6240 Blade with two AMD Opteron 2384 2.7 GHz processors (AMD), a T6330 Blade with two Sun UltraSparc 1.4 GHz T2 processors (T2), and a T6340 Blade with two Sun UltraSparc 1.4 GHz T2+ processors (T2P).

On each server, we compare two different benchmark experiments. The first run is performed with a synchronous writer (S) and is similar to experiment S2. The second run is performed with an asynchronous writer (A) and corresponds to experiment A5. The results of these experiments are presented in Figure 11.8 and are described below.

Compared to our Intel experiments, the AMD architecture provides slightly improved performance in the collecting portion C with a similar performance of the synchronous writer, while the performance gain of the asynchronous writer is slightly worse. The Sun UltraSparc architectures provide lots of slower logical cores (64 on the T2, 128 on the T2+) compared to the Intel or AMD architectures. The result is a significant increase of the monitoring overhead. Yet, an asynchronous writer provides an even greater benefit compared to a synchronous writer. In the case of the T2 processor, the overhead of writing W is reduced from 69.4 μs to 9.4 μs . In the case of the T2+ processor, the overhead is reduced from 64.9 μs to 13.7 μs .

In all experiments, the writing portion W of the overhead can be greatly reduced with the usage of an asynchronous monitoring writer and available cores.

11.5. Performance Tunings of Kieker

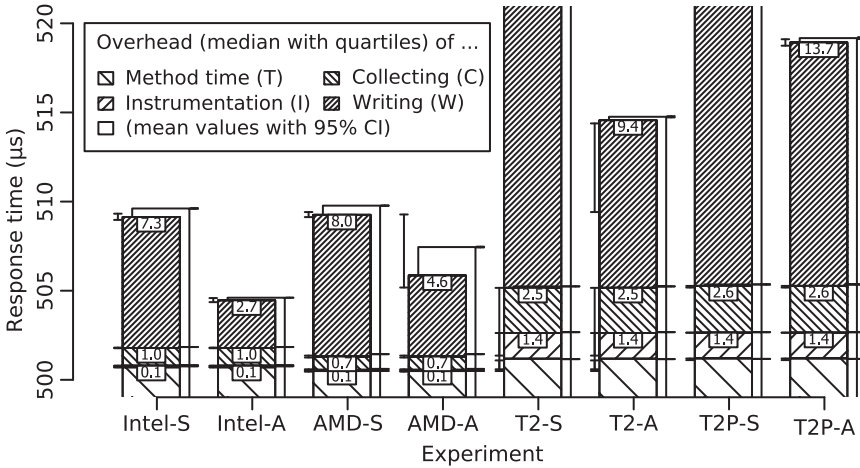


Figure 11.8. A comparison of different multi-core architectures [Waller and Hasselbring 2012a]

11.4.4 Conclusions

Again, these experiments demonstrate the capabilities of MooBench to compare different environments with each other. Furthermore, the flexibility of the benchmark with respect to possible applications is established.

11.5 Performance Tunings of Kieker

In this section, we employ our MooBench micro-benchmark in a structured performance engineering approach (see also Section 2.3.1) to reduce the monitoring overhead of the Kieker framework. We report on our exploration of different potential optimization options and our assessment of their impact on the performance as well as on the maintainability and usability of the framework.

While high-throughput is very important to observe distributed systems, the maintainability trade-off should be minimal. Otherwise, the framework may become unusable for a broader audience, effectively rendering the

11. Evaluation of Kieker with MooBench

optimization useless. Parts of these results have already been presented in Waller et al. [2014a,b].

Within our structured performance engineering approach, MooBench is used to measure the three individual portions of monitoring overhead. The results of the benchmark are then used to guide our performance tunings of Kieker. The tuned version is again evaluated and compared to the previous one with the help of our benchmark. Thus, we provide an example of how micro-benchmarks can be used to steer a structured performance engineering approach.

11.5.1 Experimental Setup

Our benchmarks are executed on the Java reference implementation by Oracle, specifically an Oracle Java 64-bit Server VM in version 1.7.0_25 running on an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM with Solaris 10 and up to 4 GiB of available heap space for the JVM.

In our experiments, we use modified versions of Kieker release 1.8 as the monitoring framework under test. All modifications are available in the public Kieker Git repository with tags starting with 1.8-pt-. Furthermore, access to these modifications as well as to the prepared experimental configurations and finally to all results of our experiments are available online [Waller et al. 2014b].

AspectJ release 1.7.3 with load-time weaving is used to insert the particular `MonitoringProbes` into the Java bytecode. Kieker is configured to use a queue with 10,000 entries to synchronize the communication between the `MonitoringWriter` and `WriterThread`. If the capacity of the `WriterThread` is insufficient to empty the queue, Kieker is configured to block. In the case of a disk writer, an additional buffer of 16 MiB is used to reduce disk accesses. In the case of the TCP writer, the additional buffer is sized to 64 KiB. Furthermore, Kieker is configured to use event records from the `kieker.common.record.flow` package and the respective probes.

We use a single benchmark thread and repeat the experiments on ten identically configured JVM instances with a sleep time of 30 seconds between all executions. In all experiments using a disk writer, we call the

11.5. Performance Tunings of Kieker

monitoredMethod() 2,000,000 times on each run with a configured `methodTime` of 0 μ s and a stack depth of ten. We discard the first 1,000,000 measured executions as warm-up and use the second 1,000,000 steady state executions to determine our results. In all experiments using the TCP writer, we increase the number of method executions to 20,000,000 and discard the first half.

In all cases, a total of 21 records are produced and written per method execution: a single `TraceMetaData` record, containing general information about the trace, e. g., the thread ID or the host name, and ten `BeforeOperationEvent` and `AfterOperationEvent` records each, containing information on the monitored method, e. g., time stamps and operation signatures. This set of records is named a *trace*.

To benchmark different scenarios, we select four different writers available for Kieker. First, we use the Kieker default ASCII writer (HDD CSV), i. e., all records are written in human-readable CSV-format to disk, resulting in 4484 bytes written per method execution (with a recursion depth of ten). Second, we use the binary disk writer (HDD bin), i. e., all records are written directly into a binary file, reducing the log size to 848 bytes per trace. Third, we additionally compress the binary log files (HDD zip) to further reduce the amount of written data to 226 bytes per trace. Finally, we use the TCP writer, intended for online analysis of monitoring data, i. e., the `MonitoringRecords` are transported to a remote system, e. g., a storage cloud, to be analyzed while the monitored system is still running. In the case of our experiments, we used the local loopback device for communication, to avoid further perturbation and capacity limits by the local network. The transmitted bytes correspond to the size of the binary monitoring log.

We perform our benchmarks under controlled conditions on a system exclusively used for the experiments. Aside from this, the server machine is held idle and is not utilized.

11.5.2 Base Evaluation

In this section, we present the results of our base evaluation of the Kieker framework. We use the Kieker 1.8 code base without the performance tunings mentioned in the following sections. The results of this base

11. Evaluation of Kieker with MooBench

Table 11.2. Throughput for the base evaluation (traces per second)

	No instr.	Deactiv.	Collect.	HDD csv	HDD bin	HDD zip	TCP
Mean	1 176.5k	757.6k	63.2k	11.5k	33.7k	14.5k	16.6k
95% CI	± 25.9k	± 5.5k	± 0.1k	± 0.8k	± 0.4k	± 0.1k	± 0.02k
Q ₁	1 189.2k	756.6k	63.0k	0.0k	30.8k	14.0k	16.2k
Median	1 191.2k	765.9k	63.6k	6.8k	33.8k	14.4k	16.8k
Q ₃	1 194.6k	769.8k	63.9k	23.5k	36.6k	15.5k	17.2k

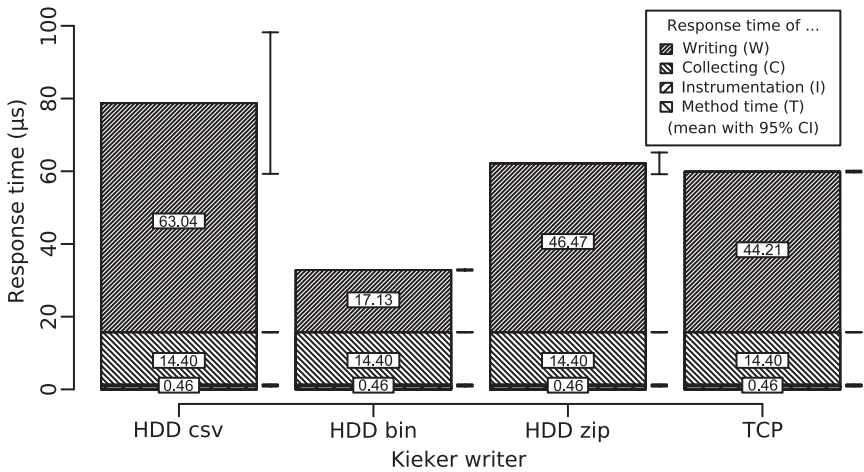


Figure 11.9. Overview of base results in response time [Waller et al. 2014a]

evaluation are used to form a baseline for our tuning experiments. The measured throughput is presented in Table 11.2 and the response times for each writer are visualized in Figure 11.9.

For the uninstrumented benchmark system (first experiment run to measure the method time (T)), we measured an average of 1,176.5k traces per second. Adding deactivated Kieker probes (second experiment run to measure ($T + I$)) resulted in an average of 757.6k traces per second.

11.5. Performance Tunings of Kieker

Activating the probes and collecting the monitoring records without writing them (third experiment run to measure $(T + I + C)$) further reduced the average throughput to 63.2 k traces per second. The fourth experiment run with the addition of an active monitoring writer (measuring $(T + I + C + W)$) is repeated four times to compare the four mentioned writers. For all four monitoring writers, our workload during the experiments exceeds the writer thread's capacity, thus causing blocking behavior.

Due to the amount of data written, the HDD CSV writer shows the worst results. Furthermore, as evident by the quartiles, there actually are periods with a duration of more than a second, when no monitored methods are executed. This is caused by garbage collection activity and pauses for actually writing the data.

The binary disk writer (HDD bin) produces the best results with an average of 33.7 k traces per second. But, as the `WriterThread` is already at capacity, the addition of a compression algorithm (HDD zip) has no positive impact.

The TCP writer for online analysis produces the second-best results with an average of 31.6 k traces per second. Furthermore, the 95% confidence interval and the quartiles suggest very stable results, caused by the static stream of written data, compared to the write bursts of the disk writers.

As expected, the response times for each monitoring writer differ only in the writing phase. The garbage collection activity and actually writing the data to the disk also cause a high confidence interval in the response time of the HDD CSV writer. The HDD zip and HDD writer perform slightly better than the HDD CSV writer. Similar to the throughput, the HDD bin writer shows the best response time.

11.5.3 PT1: Caching & Cloning

As is evident by our analysis of the base evaluation, i. e., by the response times presented in Figure 11.9 and by the throughputs in Table 11.2, the main causes of monitoring overhead are the collection of data (C) and the act of actually writing the gathered data (W).

Thus, we first focus on general performance tunings in these areas. We identified four possible performance improvements:

11. Evaluation of Kieker with MooBench

Table 11.3. Throughput for PT1 (traces per second)

	No instr.	Deactiv.	Collect.	HDD csv	HDD bin	HDD zip	TCP
Mean	1 190.5k	746.3k	78.2k	11.4k	38.5k	15.1k	31.6k
95% CI	± 4.1k	± 4.1k	± 0.1k	± 0.8k	± 0.4k	± 0.1k	± 0.1k
Q ₁	1 191.0k	728.1k	78.3k	0.0k	35.8k	14.3k	28.1k
Median	1 194.1k	756.6k	78.5k	7.8k	38.6k	15.2k	32.5k
Q ₃	1 195.1k	763.7k	78.7k	22.6k	40.8k	15.5k	34.7k

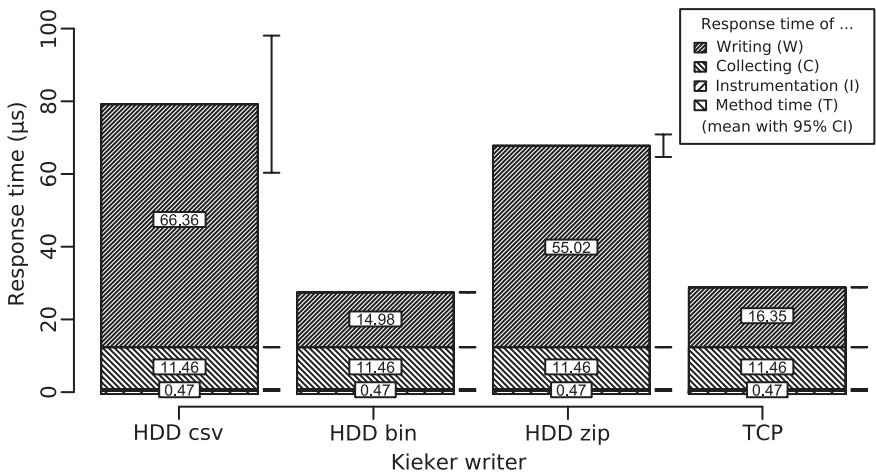


Figure 11.10. Overview of PT1 results in response time [Waller et al. 2014a]

1. Our preliminary tests showed that certain Java reflection API calls, like constructor and field lookup, are very expensive. These calls are used by Kieker to provide an extensible framework. Instead of performing these lookups on every access, the results can be cached in HashMaps.
2. The signatures of operations are stored in a specific String format. Instead of creating this signature upon each request, the resulting String can be stored and reused.

11.5. Performance Tunings of Kieker

3. A common advice when developing a framework is not to expose internal data structures, such as arrays. Instead of directly accessing the array, users of the framework should only be able to access cloned data structures to prevent the risk of accidentally modifying internal data. However, in most cases internal data is only read from a calling component and not modified. For all these cases, copying data structures is only a costly effort without any benefit. We omit this additional step of cloning internal data structures and simply provide a hint in the documentation.
4. Finally, some internal static fields of classes were marked private and accessed by reflection through a `SecurityManager` to circumvent this protection. These fields were changed to be public to avoid these problems when accessing the fields.

The resulting throughput is shown in Table 11.3 and the response time is visualized in Figure 11.10. As can be expected, the changes in the uninstrumented benchmark (*T*) and with deactivated probes (*I*) are not significant. However, the response times and the throughput of the collecting phase (*C*) has been improved. The main difference is visible in the writing phase (*W*).

The influence on the simple implementation of the HDD CSV writer is minimal and not significant. The binary disk writer (HDD bin) achieves a throughput of an additional 5 k traces per second. As the `WriterThread` is still at its capacity, the additional compression (HDD zip) behaves worse than the binary writer, even decreasing its throughput. The results of the TCP and HDD bin writers are similar and have achieved significant improvements.

However, the greatest improvement is achieved with the TCP writer. Its throughput almost doubled while still providing very stable measurements (small confidence interval). As a result, we will focus our further improvements on this writer. In addition to its evident potential for performance improvements, it is also capable of handling live analysis, i. e., performing an analysis concurrent to monitoring, in contrast to the disk writers. Thus, only the TCP writer will be discussed in PT2–PT4.

The improvements discussed in this section are used in recent Kieker versions because of their minimal influence on the maintainability and the great improvements in the area of performance efficiency.

11. Evaluation of Kieker with MooBench

Table 11.4. Throughput for PT2 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	757.6k	78.2k	56.0k
95% CI	± 3.6k	± 6.2k	± 0.1k	± 0.2k
Q ₁	1 190.5k	760.0k	78.1k	52.3k
Median	1 191.6k	766.8k	78.4k	53.9k
Q ₃	1 194.2k	771.4k	78.7k	61.0k

11.5.4 PT2: Inter-Thread Communication

As concluded in PT1, the writer queue is still saturated. That is, the monitoring thread blocks while waiting for available space inside the queue, i. e., until the writer thread has finished its writing. With sufficient available space inside the queue, the monitoring thread would pass the records directly into the queue and proceed with the method without further delays. Thus, our next goal is to further decrease the response time of writing, resulting in more available queue space. In this section, we aim to improve the communication between the monitoring and the writing thread.

Currently, the internal communication between the monitoring and the writing thread is realized with the `ArrayBlockingQueue` Java class. To improve the performance of this inter-thread communication, we employ the disruptor framework [Thompson et al. 2011], which provides a more efficient implementation for our communication scenario.

The resulting throughput of this series of experiments is presented in Table 11.4 and the response time is displayed in the overview diagram Figure 11.11. As expected, our changes have no significant influence on the measured method time (T), the deactivated probe (I), and the data collection (C). However, we are able to decrease the response time overhead of writing from $16.35\ \mu\text{s}$ to $6.18\ \mu\text{s}$. The decrease in the response time is accompanied with an increase in the average throughput rate from 31.6k to 56.0k traces per second.

This improvement is scheduled to be included in future Kieker versions, as the maintainability of Kieker is not hindered by this optimization. All

11.5. Performance Tunings of Kieker

Table 11.5. Throughput for PT3 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 176.5k	729.9k	115.7k	113.2k
95% CI	± 2.1k	± 4.4k	± 0.2k	± 0.5k
Q ₁	1 186.0k	726.5k	115.8k	113.1k
Median	1 187.1k	734.5k	116.2k	114.3k
Q ₃	1 189.2k	739.7k	116.5k	115.0k

required changes can be abstracted into already existing abstract classes. Furthermore, the measured performance improvements are significant enough to warrant such an inclusion.

11.5.5 PT3: Flat Record Model

Although we have improved the overhead of writing, our experiments still suggest that the monitoring thread is waiting for buffer space. Hence, we intend to further decrease the response time of writing.

An analysis of the TCP writer's source code reveals the serialization of incoming `MonitoringRecords` into a `ByteBuffer` as the writer thread's main task besides actually sending the data. A possible solution would be the creation of an additional thread handling the object serialization. However, this approach would only be applicable with sufficient free cores (see Section 11.4).

Object serialization is required to transfer the `MonitoringRecords` into the TCP data stream. However, the `MonitoringRecords` are created within the monitoring probes solely for passing them to the monitoring writer. That is, no calculations are performed with these records. Therefore, we can skip the object creation and write the gathered monitoring data directly into a `ByteBuffer` that is passed to the writer thread. This optimization grants several performance advantages: First, the object creation and garbage collection for those objects is avoided. Additionally, a `ByteBuffer` can store the contents of several `MonitoringRecord`, causing less exchange within the disruptor framework.

11. Evaluation of Kieker with MooBench

The resulting throughput is presented in Table 11.5 and the response time is visualized in Figure 11.11. Due to the changes, the deactivated probe (*I*) behaves slightly worse. However, this can be fixed with additional adjustments, as is evident from PT4 in the next section. In the collecting phase (*C*), we are able to decrease the response time from 11.44 μs to 7.23 μs . The main advantage can be measured in the writing phase (*W*). Its response time has decreased from 6.18 μs to only 0.2 μs . Similarly, the average throughput for the writing phase has increased from 56.0k to 113.2k traces per second.

The drastic decrease in the response time overhead of writing (*W*) is caused by the significant reduction of the work of the writer thread. Its work is reduced to sending the already prepared ByteBuffers to the network. Therefore, the employed buffer provides sufficient space most of the time. The remaining 0.2 μs overhead for the writing phase is caused by putting the ByteBuffers into the queue. We also improved the response time overhead of the collecting phase. This is caused by no longer creating MonitoringRecord objects and consequently reducing required garbage collections.

Although providing great performance benefits, this improvement will not be used in the next versions of Kieker. Our proposed changes would hinder the modularity, modifiability, and reusability of the framework. However, the changes are part of a high-throughput optimized version of Kieker used in the ExplorViz project [Fittkau et al. 2013a]. Furthermore, recent developments in Kieker aim at the automatic generation of probes and records [Jung et al. 2013]. This also enables the tunings of PT3.

11.5.6 PT4: Minimal Monitoring Code

In PT3, we have been able to optimize the writing phase, resulting in minimal overhead. Now, the main cause of monitoring overhead is found in the collecting phase. Therefore, we will focus on this phase in the following.

To achieve our intended optimization, we design a minimal monitoring tool from scratch without any code not directly related to monitoring. For instance, we remove interface definitions, consistence checks, and configurability. Furthermore, we only provide five hard coded types of MonitoringRecords, limiting extensibility.

11.5. Performance Tunings of Kieker

Table 11.6. Throughput for PT4 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	763.3k	145.1k	141.2k
95% CI	± 2.0k	± 4.0k	± 0.2k	± 0.3k
Q ₁	1 187.4k	747.0k	144.2k	139.4k
Median	1 191.4k	762.5k	146.1k	142.7k
Q ₃	1 195.2k	778.4k	146.8k	144.2k

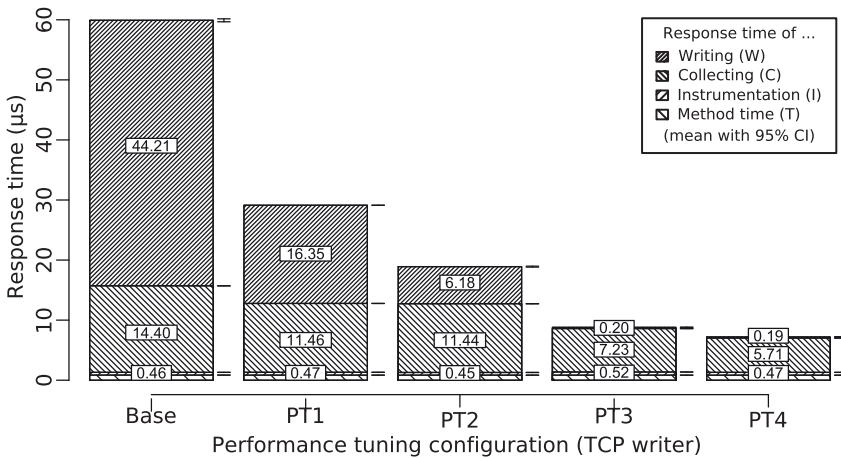


Figure 11.11. Overview of the tuning results in response time [Waller et al. 2014a]

The resulting response time is visualized in Figure 11.11 and the throughput is shown in Table 11.6. The measured response times of the no instrumentation (*T*), deactivated probe (*I*), and writing (*W*) phases do not significantly differ from our previous experiments. However, we have fixed the performance regression for the deactivated probes. In our targeted collecting phase (*C*), the response time has decreased to 5.71 µs and the throughput has increased from 115.7k to 145.1k traces per second.

11. Evaluation of Kieker with MooBench

Similar to PT3, this improvement will not be used in future Kieker versions. With these changes, the monitoring tool lacks important features for a framework, e.g., configurability and reusability. However, these optimizations are also part of the high-throughput optimized version of Kieker used in the ExplorViz project [Fittkau et al. 2013a].

11.5.7 Conclusions

With the help of the series of experiments presented in this section, we have been able to demonstrate the use of MooBench in a structured performance tuning approach. We employed our benchmark to identify parts of our monitoring framework under test that are promising for performance tunings. Furthermore, we have validated the success of these tunings with additional benchmark runs. Finally, we have demonstrated in PT4 the applicability of the MooBench micro-benchmark to another monitoring tool that is not Kieker.

11.6 Benchmarking the Analysis Component

In this section, we extend the already described MooBench micro-benchmark approach to the Kieker.Analysis component of the Kieker monitoring framework (see Section 4.5.1). In addition to the method time (T), deactivated probe (I), data collection (C), and writing (W) phases, we add two additional phases for a simple analysis project: trace reconstruction (R) and trace reduction (D). Note that in these experiments writing (W) already includes the transmission to the analysis component and the deserialization of the transmitted data in MonitoringRecords. Parts of these results have already been presented in Fittkau et al. [2013b,c].

In addition to the Kieker.Analysis component, we also repeat all experiments with the high-throughput tuned version of Kieker, already used in PT4 of the previous section. This version of Kieker is part of the ExplorViz project [Fittkau et al. 2013a]. Similarly, the performed simple analysis is taken from the context of this project. The analysis consists of three steps. The first step is included in our writing overhead (W) and performs a

11.6. Benchmarking the Analysis Component

deserialization of the transmitted monitoring data into `MonitoringRecords`. In the trace reconstruction (*R*) step, all `MonitoringRecords` belonging to a single trace, e. g., the ten recursive calls, are gathered together. Finally, in the trace reduction (*D*) step, several of these collected traces are clustered together to prepare a future visualization. Refer to Fittkau et al. [2013a,b] for further details.

11.6.1 Experimental Setup

We employ two Virtual Machines (VMs) in our OpenStack private cloud for our experiments. Each physical machine in our private cloud contains two eight-core Intel Xeon E5-2650 (2 GHz) processors, 128 GiB RAM, and a 500 Mbit network connection. When performing our experiments, we reserve the whole cloud and prevent further access in order to reduce perturbation. The two used VMs are each assigned 32 virtual CPU cores and 120 GiB RAM. Thus, both VMs are each fully utilizing a single physical machine. For our software stack, we employ Ubuntu 13.04 as the VMs' operating system and an Oracle Java 64-bit Server VM in version 1.7.0_45 with up to 12 GiB of assigned memory.

In our experiments, we use Kieker release 1.8 with AspectJ 1.7.3. Kieker is configured to use a queue with 10,000 entries to synchronize the communication between the `MonitoringWriter` and `WriterThread`. If the capacity of the `WriterThread` is insufficient to empty the queue, Kieker is configured to block. The employed TCP writer uses an additional buffer sized to 64 KiB. Furthermore, Kieker is configured to use event records from the `kieker.common.record.flow` package and the respective probes. In addition, we employ the high-throughput optimized version of Kieker in its default configuration, as provided by the `ExplorViz` project [Fittkau et al. 2013a].

We use a single benchmark thread and repeat the experiments on ten identically configured JVM instances with a sleep time of 30 seconds between all executions. We call the `monitoredMethod()` 4,000,000 times on each run with a configured `methodTime` of 0 μ s and a stack depth of ten. For our high-throughput tuned version, we increased the number of measured executions to 100,000,000. In each case, we discard the first half of the executions as a warm-up period.

11. Evaluation of Kieker with MooBench

Table 11.7. Throughput for Kieker 1.8 (traces per second)

	No instr.	Deactiv. Collecting	Writing	Reconst.	Reduction	
Mean	2 500.0k	1 176.5k	141.8k	39.6k	0.5k	0.5k
95% CI	\pm 371.4k	\pm 34.3k	\pm 2.0k	\pm 0.4k	\pm 0.001k	\pm 0.001k
Q ₁	2 655.4k	1 178.0k	140.3k	36.7k	0.4k	0.4k
Median	2 682.5k	1 190.2k	143.9k	39.6k	0.5k	0.5k
Q ₃	2 700.4k	1 208.0k	145.8k	42.1k	0.5k	0.5k

Table 11.8. Throughput for our high-throughput tuned version (traces per second)

	No instr.	Deactiv. Collecting	Writing	Reconst.	Reduction	
Mean	2 688.2k	770.4k	136.5k	115.8k	116.9k	112.6k
95% CI	\pm 14.5k	\pm 8.4k	\pm 0.9k	\pm 0.7k	\pm 0.7k	\pm 0.8k
Q ₁	2 713.6k	682.8k	118.5k	102.5k	103.3k	98.4k
Median	2 720.8k	718.1k	125.0k	116.4k	116.6k	114.4k
Q ₃	2 726.8k	841.0k	137.4k	131.9k	131.3k	132.4k

11.6.2 Results and Discussion

The throughput for each phase is displayed in Table 11.7 for Kieker 1.8 and in Table 11.8 for our high-throughput tuned version. In addition, the measured response times are visualized in Figure 11.12.

As expected, we measure no significant difference in the uninstrumented run of our benchmark. Adding the deactivated instrumentation, Kieker version 1.8 performs significantly better with 1 176 k traces per second compared to 770 k. However, the executed code for both versions is identical.

We attribute this measured (and reproducible) difference to the change in the number of measured executions with each version. Our tuned version runs 20 times longer than Kieker 1.8 which might have resulted in different memory utilization. A similar result can be measured in the collecting phase: Kieker 1.8 handles 141.8 k traces per second whereby our high-throughput tuned version achieves 136.5 k traces per second which

11.6. Benchmarking the Analysis Component

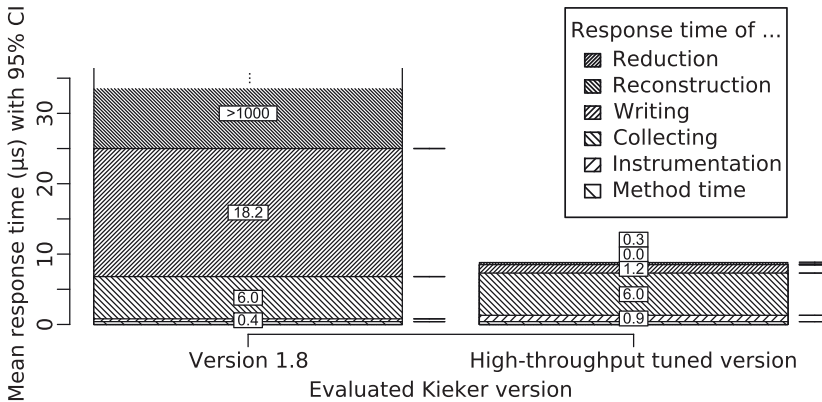


Figure 11.12. Comparison of analysis response times [Fittkau et al. 2013b]

is roughly the same with regards to the different number of measured executions of both experiments.

The main difference can be measured in the following phases. During writing, our high-throughput tuned version reaches 115.8 k traces per second while Kieker 1.8 only achieves 39.6 k traces per second. Note that the trace amount is actually limited by the available network bandwidth in the case of our high-throughput tuned version. This is also evident in the response times. Kieker 1.8 shows 18.2 µs while our tuned version achieves 1.2 µs in the writing phase. The comparatively high response times in Kieker 1.8 suggest that the TCP writer fails to keep up with the generation of MonitoringRecords and therefore, the buffer to the writer fills up, resulting in higher response times.

In the trace reconstruction phase, Kieker 1.8 performs 466 traces per second and our tuned version still reaches 116.9 k traces per second. We attribute the increase of 1.1 k traces per second in our tuned version to measuring inaccuracy which is confirmed by the overlapping confidence intervals.

Our high-throughput tuned version performs about 250 times faster than Kieker 1.8. This has historical reasons since performing live trace processing is a rather new requirement for Kieker. Furthermore, the results

11. Evaluation of Kieker with MooBench

suggest that the pipes and filters architecture of Kieker 1.8 has a bottleneck in handling the pipes, resulting in poor throughput.

Kieker 1.8 achieves 461 traces per second and our tuned version achieves 112.6k traces per second in the reduction phase. Compared to the previous phase, the throughput slightly decreased for both versions which is reasonable considering the additional work.

In the reconstruction and reduction phases, Kieker 1.8 has over 1,000 μs (in total: 1,714 μs and 1,509 μs), and our high-throughput tuned version achieves 0.0 μs and 0.3 μs . The response times of our tuned version suggest that the filter are efficiently implemented in way that the additional buffers are not filling up.

11.6.3 Conclusions

This series of experiments demonstrated the portability and configurability of the MooBench micro-benchmark. We have been able to adjust the benchmark to the new scenario of additionally measuring the influence of an online analysis on the monitoring overhead.

Note that we have not measured the cost of analysis on the analysis node, but rather the additional overhead caused by waiting on the monitoring node. We performed a series of benchmark evaluations of the actual analysis node of Kieker in Ehmke et al. [2013]. However, these evaluations are using a set of custom micro-benchmarks developed in the context of the master thesis of Ehmke [2013].

In addition, we demonstrated the feasibility of a (private) cloud environment for performance benchmarks. Although an additional virtualization layer is employed, the results are stable and comparable to our previous results on native hardware. In accordance with our requirement on robust execution (R9) of the benchmark (see Section 8.3), we recommend the usage of the `MonitoredClassSimple` implementation of the Monitored Application (see Section 8.2.1) for virtualized environments.

Finally, we again demonstrated the comparability of two different monitoring tools with the help of our benchmark. Refer to Chapter 12 for further comparisons of different monitoring frameworks.

11.7 Using MooBench in Continuous Integration

In this section, we describe our inclusion of MooBench into the continuous integration system used by Kieker. Parts of the section will also be published in Waller et al. [2015b].

Generally speaking, micro-benchmarks, such as MooBench, can be included in continuous integration setups [Fowler 2006; Duvall et al. 2007; Meyer 2014] to automatically record performance improvements and regressions [e. g., Bulej et al. 2005; Kalibera 2006; Weiss et al. 2013; Reichelt and Braubach 2014]. First attempts to include MooBench into the Kieker setup have been performed in a co-supervised bachelor thesis by Zloch [2014] and in a supervised term paper by Beitz [2014].

Any inclusion of MooBench into the Kieker continuous integration setup is motivated by our findings that are presented in Section 11.2. Due to irregularly performed manual benchmarks of the monitoring overhead of Kieker, we have detected several performance regressions after the releases of new versions. This has enabled us to further investigate the regressions and to provide bug fixes for future releases.

When patching performance regressions, the main challenge is identifying the source code changes that have triggered the regressions. With irregular manual benchmarks, lots of source code commits can contain the possible culprit. Ideally, our benchmark would have been executed automatically with each nightly build to provide immediate hints on performance problems with each change. Thus, the continuous integration of benchmarks provides the benefit of an immediate and automatic feedback to the developers.

Rather than including long running benchmarks directly into the continuous integration system, an integration of their execution into the release process might be sufficient. However, the continuous integration approach provides the mentioned additional benefit of earlier feedback. For instance, in the case of nightly builds, the feedback is provided within a single day. Integration into a truly continuous process, such as a build with each commit, might be challenging due to the time cost of each benchmark execution (e. g., currently about 60 minutes for MooBench within Kieker's continuous integration setup).

11. Evaluation of Kieker with MooBench

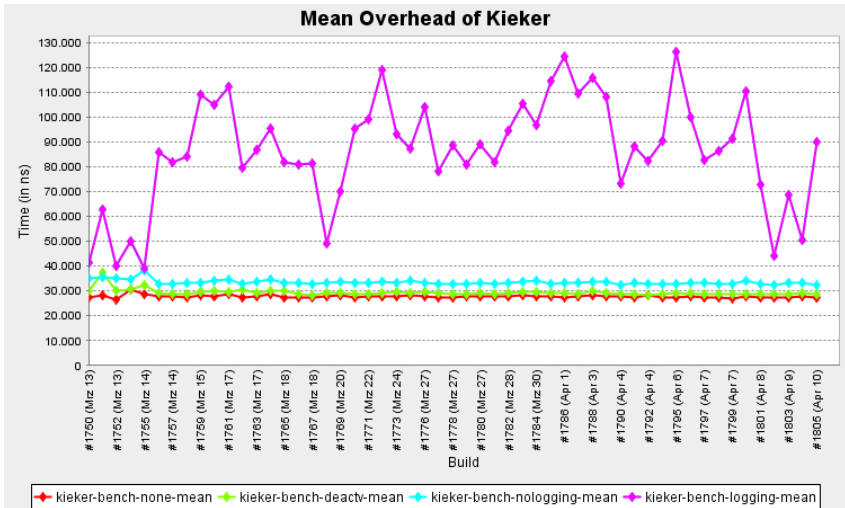


Figure 11.13. First approach of including MooBench into Jenkins [Waller et al. 2015b]

11.7.1 Integrating MooBench into the Kieker Setup

The continuous integration setup that is employed by Kieker is based upon Jenkins [Jenkins].

In our first approach [Zloch 2014], we have developed a Jenkins plugin to control and execute the benchmark. In its simplest form, this plugin replaces the MooBench’s External Controller (see Section 8.2) and directly executes the benchmark from within a build in Jenkins. However, this kind of execution violates the requirement for an idle environment (R12): Jenkins and its periodical tasks, as well as the running plugin itself, influence the benchmark results. This can be seen in Figure 11.13. Note that the changes in the response times of the pink graph are not caused by actual changes in the source code but rather by background tasks within Jenkins. Even remote execution with the help of a Jenkins master/slave setup, i. e., executing the benchmark within an otherwise idle Jenkins instance on a separate server, has only provided fluctuating results.

11.7. Using MooBench in Continuous Integration

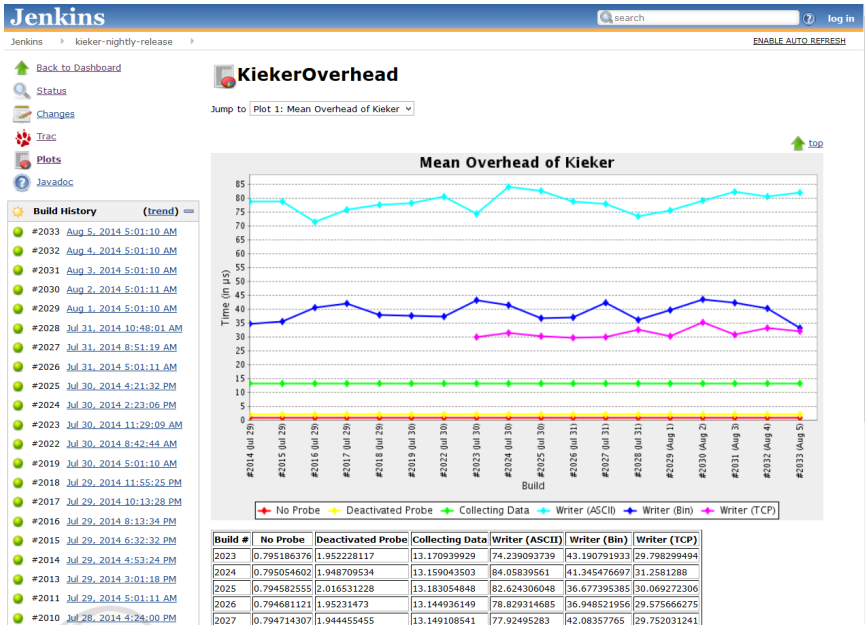


Figure 11.14. Performance measurements of MooBench within Kieker’s Jenkins
source: <http://build.kieker-monitoring.net/job/kieker-nightly-release/plot/>

For our next approach [Beitz 2014], we have employed the KoPeMe framework [Reichelt and Braubach 2014]. KoPeMe has been developed to provide continuous performance tests. Furthermore, it directly provides an integration into Jenkins. However, similar to our own Jenkins plugin, we have not been able to produce satisfying results.

Finally, we have chosen a simpler approach: Instead of using complex plugins, we simply call a shell script at the end of each nightly build on Jenkins. This script copies the benchmark and the created Kieker nightly jar-file to an idle, pre-configured remote server (e. g., onto a cloud instance). There, the benchmark gets executed while Jenkins waits for the results. In

11. Evaluation of Kieker with MooBench

addition to the usual analyses performed by MooBench, e. g., calculating mean and median with their confidence intervals and quartiles, we also create a CSV file with the mean measurement results. This file can be read and interpreted by a plot plugin within Jenkins. An example of such a generated plot based upon the Kieker nightly builds is presented in Figure 11.14.

As is evident by the display of the data in Figure 11.14, the plot plugin is rather limited. For instance, it is only capable of displaying the measured mean response times that still contain some variations. The display of additional statistical method, such as confidence intervals, would be beneficial to their interpretation.

In addition, we currently only display the collected results rather than automatically notifying the developers when a performance anomaly occurs. The actual detection of the anomalies has to be performed manually. However, previous work on anomaly detection within Kieker results [e. g., Ehlers et al. 2011; Frotscher 2013] can be adapted for this scenario.

Finally, as is common with dynamic analysis approaches, the detection and visualization of performance regressions is only possible within benchmarked areas of Kieker. As a consequence, any performance regression caused by combinations of, for instance, unused probes or writers cannot be found. However, a more thorough benchmark requires a higher time cost. Thus, a balance has to be found between benchmark coverage and time spent for benchmarking.

Despite these remaining challenges, the inclusion of MooBench into the continuous integration setup of Kieker already provides huge benefits. Especially considering our motivation for the inclusion, similar performance regressions are now detected immediately. Furthermore, the regressions can be directly linked to a small sets of changes. Thus, diagnosis of performance problems is aided.

The current and future implementations of our integration of benchmarks into Jenkins are available as open source software with MooBench.³ Furthermore, the current state of our implementation is available with our continuous integration setup.⁴

³<http://kieker-monitoring.net/MooBench/>

⁴<http://build.kieker-monitoring.net/job/kieker-nightly-release/plot/>

11.7. Using MooBench in Continuous Integration

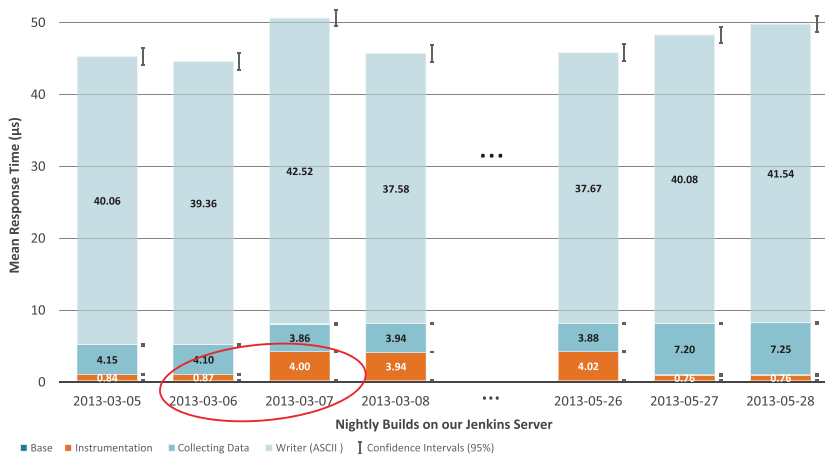


Figure 11.15. Post-mortem analysis of a performance regression [Waller et al. 2015b]

11.7.2 Evaluation of MooBench in Continuous Integration

Since our recent installation of MooBench into the continuous integration setup of Kieker, no additional major performance regressions have occurred. Instead of artificially creating an anomaly to demonstrate the capabilities of our setup, we have recreated earlier nightly builds and executed the benchmark as it would have been included. This post-mortem benchmarking also allows for an outlook on a more advanced visualization and anomaly detection than is currently realized within our Jenkins implementation.

Specifically, we have selected the first performance regression that is described in Section 11.2: In Kieker release version 1.7, we have detected an unintended increase of the overhead of instrumentation (*I*) that has been related to a bug in our implementation of adaptive monitoring. This regression is also described in ticket #996⁵ in the Kieker issue tracking system. To further narrow down the cause of this regression, we have taken a look at the nightly builds between Kieker releases 1.6 and 1.7. For each build,

⁵Ticket #996: <http://trac.kieker-monitoring.net/ticket/996>

11. Evaluation of Kieker with MooBench

we have run the MooBench benchmark in a configuration identical to the one used in our continuous integration setup. The resulting visualization of a few of the relevant benchmark results of the nightly builds is presented in Figure 11.15.

In this figure, the mean benchmark results are depicted as stacked bars. Each bar is annotated to the right with its respective 95%-CI. The lowest bar is barely visible and represents the method time (T) of the benchmark without any monitoring overhead. The other three bars correspond to the three causes of monitoring overhead: instrumentation (I), data collection (C), and writing (W) (see also Section 6.2 on page 90). Our focus of this analysis is on the orange bar, representing the instrumentation overhead of Kieker. The actually found performance anomaly is highlighted with a red ellipse.

The first four nightly builds presented here are part of our analysis: two builds before and after the performance regression occurred are presented. The final three builds demonstrate our bug fixing two and a half months after the performance regression. With the help of our presented vision of including benchmarks into continuous integration and performing automated anomaly detections on the results, the time to fix performance regressions should be reduced.

11.7.3 Conclusions

With our inclusion of MooBench into the continuous integration setup of Kieker, we have demonstrated its capabilities as a tool for continuous performance evaluation. Furthermore, we have demonstrated, as already shown in Section 11.2 and in Section 11.5, that MooBench can be employed to detect performance regressions and to steer performance optimizations. Finally, the additional challenges that are still present in our integration, e. g., lacking visualization or no automatical anomaly detection, remain as future work.

The code used for the inclusion of MooBench into continuous integration systems is available with MooBench (see Section 8.1). Furthermore, all results from our evaluation as well as all scripts used to create the results are available for download [Waller et al. 2015a].

Comparing Monitoring Frameworks with MooBench

In this chapter, we investigate the capabilities of MooBench to *evaluate further monitoring frameworks*. In the previous chapter, we have already performed a detailed analysis of several aspects of Kieker. Additionally, we have compared the performance of Kieker to the performance of ExplorViz's monitoring tool that is inspired by Kieker. In Section 12.1, we evaluate several aspects of the performance of the *inspectIT monitoring tool* and perform a brief comparison to the performance of Kieker. In Section 12.2, we similarly evaluate the performance of the *SPASS-meter monitoring tool*.

When you are content to be simply yourself and don't compare or compete, everyone will respect you.

—Laozi (founder of philosophical Taoism), Tao Te Ching

12. Comparing Monitoring Frameworks with MooBench

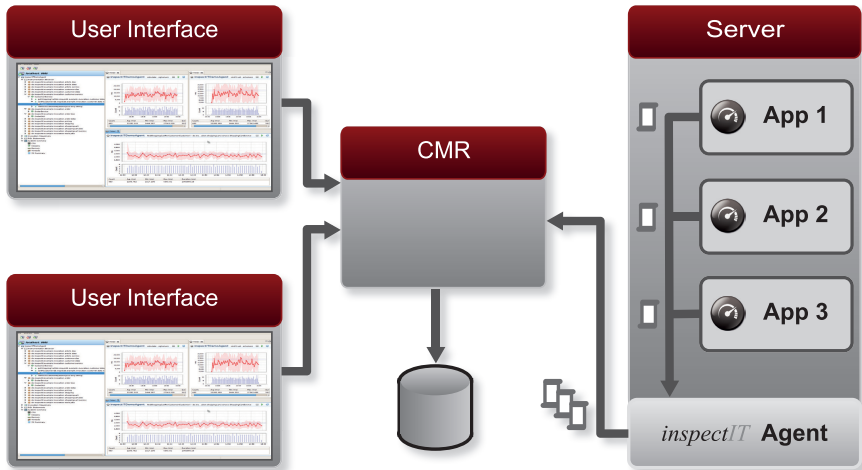


Figure 12.1. Overview on the inspectIT components [Siegl and Bouillet 2011]

12.1 The inspectIT Monitoring tool

The inspectIT monitoring tool (website: <http://www.inspectit.eu/>) has been developed by NovaTec.¹ Although the tool is not open source, it is provided free of charge. A short introduction to the tool is provided in a white paper by Siegl and Bouillet [2011]. Further details are provided through the official documentation.²

InspectIT is an Application Performance Management (APM) tool for application-level monitoring of Java business applications (see Figure 4.2). Thus, it provides probes, called *sensors*, for application-level tracing and timing of method calls, as well as probes gathering information from the used libraries, middleware containers, or JVMs. Generally speaking, it is a typical representative of the class of application-level monitoring frameworks and a typical System Under Test (SUT) for our MooBench micro-benchmark.

¹<http://www.novatec-gmbh.de/>

²<http://documentation.novatec-gmbh.de/display/INSPECTIT>

12.1. The inspectIT Monitoring tool

On a high abstraction level, inspectIT consists of three components: (1) the inspectIT Agent, handling the monitoring of the target applications, (2) the Centralized Measurement Repository (CMR), receiving all monitoring data and performing analyses of this data, and (3) the Eclipse-based graphical user interface, displaying the data and the results of the CMR. All three components and their interconnection are displayed in Figure 12.1.

When benchmarking the monitoring overhead of the inspectIT monitoring framework, the inspectIT Agent is the focus of our interest. Compared to the monitoring component of the Kieker monitoring framework, this inspectIT Agent is less configurable. For instance, it is not possible to instrument a target method without collecting any monitoring data, e. g., by deactivating the probe.

The second inspectIT component employed in most of our benchmark experiments is the CMR, since a running CMR is required by the monitoring agent. The monitoring agent transmits all gathered monitoring data to this repository using a TCP connection. Via this TCP connection, the CMR can either be deployed on the same machine as the inspectIT Agent or on a different remote machine. Depending on the selected sensors, analyses of the gathered data are performed and visualizations for the user interface are prepared within the CMR. Furthermore, it is possible to activate a storage function to collect all recorded monitoring data on the hard disk.

The third component of inspectIT, the graphical user interface, is not directly relevant for our intended benchmark scenarios, as it is solely involved with displaying the gathered data. Furthermore, due to its graphical nature and its realization as an Eclipse-based tool, its execution is hard to automate for the command-line-based External Controller of our micro-benchmark. So, although its use might put additional load on the CMR and thus indirectly on the monitoring agent, it is outside of the scope of the presented benchmarking experiments.

12.1.1 Specialties and Required Changes for MooBench

In order to gather more detailed information on our three causes of monitoring overhead (see Section 6.2) despite the limited configuration possibilities of the inspectIT Agent, we make minimal alterations to the provided class

12. Comparing Monitoring Frameworks with MooBench

files of inspectIT. That is, we add the possibility to deactivate two parts of the employed sensors: first the whole data collection and second the data sending. These changes are implemented with a simple if-clause checking a binary variable set in the respective sensor's constructor. We aim to minimize the perturbation caused by our changes with this implementation.

Furthermore, we encounter additional challenges with high workloads: InspectIT employs a hash function when collecting its monitoring records. Due to the present implementation, it is only capable of storing one monitoring record per millisecond per monitored method. Thus, the MooBench approach of rapidly calling the same method in a loop is not easily manageable. In order to benchmark higher workloads, we also change this behavior to one monitoring record per nanosecond per monitored method. At least with our employed hardware and software environment, this small change proves to be sufficient. Otherwise, the hash function could easily be adapted to use a unique counter instead of a timestamp.

It is important to note that inspectIT focusses on low overhead instead of monitoring data integrity. That is, in cases of high load, the framework drops monitoring records to reduce the load. This behavior can to some extent be countered by providing larger queues and memory buffers to the agent and CMR. For instance, instead of using the default `SimpleBufferStrategy`, the configurable `SizeBufferStrategy` strategy can be employed. However, it is not possible to entirely prevent this behavior in a reliable way. Even with sufficiently large buffers, loss of monitoring data occasionally occurs.

Additionally, the sending behavior can be configured: With the default `TimeStrategy`, monitoring data is collected and sent in regular intervals. Contrary to that, the `ListSizeStrategy` collects a specified amount of data and then sends it. However, in our experiments, the `TimeStrategy` provides superior performance under high load. The `ListSizeStrategy` either sends many small messages (small list size) or sending each message takes a much longer time compared to similar messages sizes of the `TimeStrategy` (large list sizes).

Finally, we encounter challenges with threaded benchmarks. On our test systems, with more than six active benchmark worker threads providing stress workloads, inspectIT crashes due to excessive garbage collection activity.

12.1.2 General Benchmark Parameters

In the following, we describe a series of benchmark experiments that we have performed to evaluate the feasibility of MooBench to determine the monitoring overhead of the application-level monitoring framework inspectIT. All of these experiments utilize, except as noted below, the same sets of environment, workload, and system parameters. Specifically, we conduct our experiments with the Oracle Java 64-bit Server JVM in version 1.7.0_45 with up to 12 GiB of heap space provided to the JVM. Furthermore, we utilize X6270 Blade Servers 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 is otherwise held idle. The instrumentation of the monitored application is performed through load-time weaving using the (modified) inspectIT Agent.

The configuration of the MooBench parameters is, except as noted, left at default values, as described in Listing 8.1 on page 133. That is, we use 2,000,000 calls of the monitored method with a recursion depth of ten. Our experiments suggest discarding the first 1,500,000 monitored executions as warm-up period for executions of inspectIT (see Figure 12.2). Each part of each experiment is repeated ten times. The configured method time depends on the intended experiment and is either configured to 500 μ s or to 0 μ s for ten recursive calls.

InspectIt is configured with all default sensors active, including the platform sensors collecting data concerning the CPU or memory usage. Depending on the benchmark scenario, the `isequence` sensor and/or the `timer` sensor are employed with a targeted instrumentation of the monitored `Method()`. Furthermore, the `TimeStrategy` is used to send all monitoring data every second to the CMR. To provide additional buffers, the `SizeBufferStrategy` is configured to a size of 1,000,000. Otherwise, the default configuration of inspectIT is employed.

An exemplary time series diagram for our initial experiments with inspectIT is presented in Figure 12.2. Our suggested warm-up period of 1,500,000 monitored executions is evident in this graph. Especially with the full-blown instrumentation experiment, massive garbage collector activity is visible. Besides the visible spikes in the presented graph, this is also

12. Comparing Monitoring Frameworks with MooBench

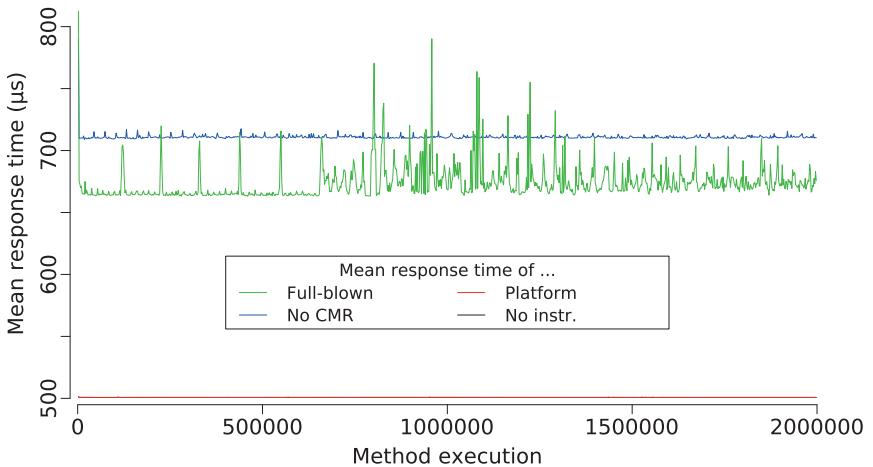


Figure 12.2. Time series diagram for the initial experiment with inspectIT

documented in the logs of each benchmark run. For instance, during some benchmark executions the JVM blocks with garbage collection activity for up to 0.5s per 100,000 monitored executions. Similarly, our observations of the environment demonstrate between two and ten active and fully-loaded threads during these experiments. Finally, our observation of JIT compiler activity also suggest the warm-up period of 1,500,000 monitored executions. Although most compilations in our benchmark runs occur earlier, especially Java functions associated with sending the collected data are often compiled late in the runs.

Our configurations of the inspectIT Agent and the CMR as well as of the accompanying loggers are provided with MooBench.³ Additionally, we provide the preconfigured External Controllers used to run our benchmarking experiments. However, due to licensing constraint, we are unable to provide the modified versions of the class files. In order to configure MooBench for inspectIT, the ant build target build-inspectit can be used. This way, a ready to use benchmark environment is provided.

³<http://kieker-monitoring.net/MooBench>

12.1. The inspectIT Monitoring tool

Table 12.1. Response times for the initial inspectIT experiment (in μ s)

	No instr.	Platform	Full-blown	No CMR
Mean	500.73	500.76	674.16	708.54
95% CI	± 0.00	± 0.02	± 1.43	± 0.07
Q ₁	500.73	500.73	664.20	707.07
Median	500.73	500.77	666.58	707.97
Q ₃	500.73	500.77	673.57	708.82
Min	500.55	500.55	661.12	703.41
Max	546.51	6021.37	320297.39	7464.32

12.1.3 Initial Performance Benchmark

In this section, we present the results of our first overhead evaluation of the inspectIT monitoring tool. Our goal is a direct evaluation of inspectIT, without any of our modifications for more detailed studies (see Section 12.1.1). However, in order to measure the overhead of monitoring all method executions, we have to tune the employed hash function for these experiments as well. Our evaluation consists of four experiment runs:

1. First, we measure the uninstrumented benchmark system to establish a baseline for the response time of executing the `monitoredMethod()`.
2. Next, we instrument the benchmark system with inspectIT using only platform sensors. That is, the `monitoredMethod()` is not instrumented. However, infrastructure data is collected from the JVM and sent to the CMR. This experiment run established the basic costs of instrumenting a system with inspectIT.
3. In the third experiment run, a full-blown instrumentation is employed. That is, in addition to the platform sensors, the `monitoredMethod()` is instrumented with `isequence` and `timer` probes, which collect tracing and time information from each execution. All collected data is in turn sent to the CMR.
4. We again employ a full-blown instrumentation. However, we do not provide an available CMR.

12. Comparing Monitoring Frameworks with MooBench

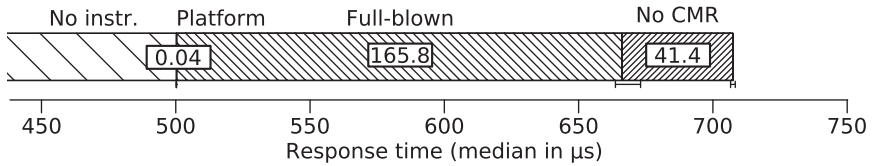


Figure 12.3. Median response time for the initial experiment with inspectIT (including the 25% and 75% quartiles)

Experimental Results & Discussion

The results of our evaluation are presented in Table 12.1. Additionally, we visualize the median response times in Figure 12.3. Finally, we have already presented a time series diagram for this experiment in Figure 12.2.

For the uninstrumented benchmark system, we measure an average response time of $500.73 \mu\text{s}$. Adding inspectIT with platform instrumentation minimally increases this average response time to $500.76 \mu\text{s}$. However, due to the added activity, the measured maximal execution was $6021.37 \mu\text{s}$. Such high values are rare, as is evident by the quartiles and by the mean being very similar to the median. Activating a full-blown instrumentation further increases the measured average response time to $674.16 \mu\text{s}$. The difference to the median, as well as the quartiles, hint at a skewness in the measured distribution. Of special note is the measured maximum of more than 300 ms. By deactivating the CMR, the average response time further increases to $708.54 \mu\text{s}$, albeit with less skewness.

The full-blown instrumentation of our benchmark results in a rather high overhead of about $17 \mu\text{s}$ per monitored method execution (we utilize a stack depth of ten). Furthermore, the memory usage and garbage collection activity are very high. Our original intention when deactivating the CMR was to deactivate the sending of monitoring data from the inspectIT Agent to the CMR. However, as is evident in our results, deactivating the CMR increases the monitoring overhead instead of reducing it. This is probably caused by timeouts in the sending mechanism.

In summary, we are able to determine the monitoring overhead of a full-blown instrumentation of inspectIT compared to an uninstrumented

12.1. The inspectIT Monitoring tool

benchmark run. However, we are not able to further narrow down the actual causes of monitoring overhead with this experiment. In the following experiments, we add our already discussed modifications to inspectIT in order to perform finer-grained experiments

12.1.4 Comparing the `isequence` and `timer` Probes

In our second overhead evaluation of inspectIT, we target two new aspects: First, we evaluate our three causes of monitoring overhead in the context of inspectIT (see Section 6.2). Second, we compare the provided `isequence` and `timer` probes. The `isequence` probe provides trace information, similar to the common Kieker probes. The `timer` probe, on the other hand, provides more detailed timings of each monitored execution, e. g., actual processor time spent, rather than trace information. If both probes are active, as in the previous experiment, these data points are combined in the CMR.

Our experiment runs correspond to the default series of monitoring overhead evaluations with MooBench:

1. In the first run, only the execution time of the chain of recursive calls to the `monitoredMethod()` is determined (T).
2. In the second run, the `monitoredMethod()` is instrumented with either a `isequence` or a `timer` probe, that is deactivated for the `monitoredMethod()`. Thus, the duration $T + I$ is measured.
3. The third run adds the respective data collection for the chosen probe without sending any collected data ($T + I + C$).
4. The fourth run finally represents the measurement of full-blown monitoring with the addition of an active `Monitoring Writer` sending the collected data to the CMR ($T + I + C + W$).

An active CMR is present in all runs except the first. Similarly, platform sensors collect data in all three runs containing inspectIT probes.

12. Comparing Monitoring Frameworks with MooBench

Table 12.2. Response times for the isequene probe (in μs)

	No instr.	Deactiv.	Collecting	Writing
Mean	500.60	504.07	527.98	530.82
95% CI	± 0.00	± 0.05	± 0.06	± 0.30
Q ₁	500.58	503.95	527.80	529.18
Median	500.58	503.98	527.86	529.34
Q ₃	500.58	504.01	527.94	529.60
Min	500.55	503.86	527.46	528.61
Max	560.66	8433.44	8571.24	21116.22

Table 12.3. Response times for the timer probe (in μs)

	No instr.	Deactiv.	Collecting	Writing
Mean	500.60	505.20	661.96	672.87
95% CI	± 0.00	± 0.04	± 0.06	± 2.23
Q ₁	500.58	505.09	660.68	662.60
Median	500.58	505.12	661.40	664.38
Q ₃	500.58	505.16	662.20	671.66
Min	500.55	505.02	658.30	659.02
Max	560.66	7350.89	4791.38	472453.11

Experimental Results & Discussion

The results of our benchmark experiments are presented in Tables 12.2 and 12.3. Additionally, we provide a direct comparison of the median monitoring overhead of the two different probes in Figure 12.4.

For our uninstrumented benchmark, we measure an average response time of $500.60\ \mu\text{s}$ (T), as expected for the configured $500.0\ \mu\text{s}$. Adding a deactivated isequene probe increases the average response time to $504.07\ \mu\text{s}$ ($T + I$). Similarly, a deactivated timer probe increases the average response time to $505.20\ \mu\text{s}$ ($T + I$). As expected, both values are very similar and the slightly higher overhead of the timer probe can be explained by differences in the employed instrumentation of the two probes.

12.1. The inspectIT Monitoring tool

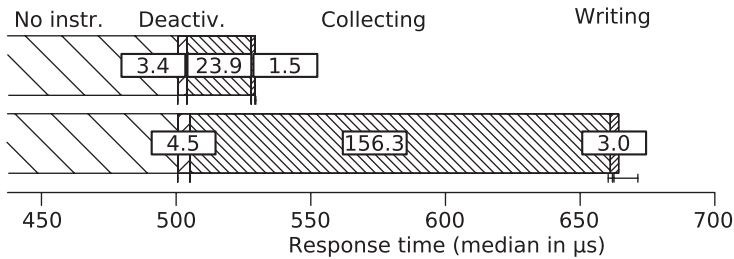


Figure 12.4. Comparison of the median response times (including the 25% and 75% quartiles) of the isequence (upper bar) and the timer probe (lower bar))

Adding data collection, the behavior of both probes starts to differ ($T + I + C$). The isequence probe further increases the average response time to $527.98 \mu\text{s}$. However, the timer probe raises the measured response time to $661.96 \mu\text{s}$. In both cases, the results are very stable, as is evident by the confidence intervals and the quartiles. Furthermore, the timer probe seems to be the main cause of monitoring overhead in a full-blown instrumentation.

Finally, we add sending of data ($T + I + C + W$). Again, the isequence probe causes fewer additional overhead than the timer probe. The former raises the response time to an average total of $530.82 \mu\text{s}$, while the latter reaches $672.87 \mu\text{s}$. This can be explained by the amount of monitoring data sent. In the case of the isequence probe, a monitoring record is sent with each trace. In the case of the timer probe, a record is sent with each method execution. This is further evident by the higher difference between measured median and mean values with the timer probe, as well as with the high maximal response time (comparable overhead to the full-blown instrumentation of the previous experiment).

Overall, the isequence probe causes less overhead and more stable results. Furthermore, the monitoring overhead of the two probes alone cannot simply be added to calculate the monitoring overhead of the combination of both probes. Especially, the writing part (sending of monitoring records to the CMR) causes higher overhead under the combined load. However, any performance tuning of inspectIT should focus on the collection of monitoring data.

12. Comparing Monitoring Frameworks with MooBench

Table 12.4. Response times for the isequence probe under high load (in μs)

	No instr.	Deactiv.	Collecting	Writing
Mean	0.85	3.38	28.46	36.34
95% CI	± 0.00	± 0.02	± 0.04	± 4.92
Q ₁	0.83	3.20	28.18	30.29
Median	0.83	3.22	28.24	30.65
Q ₃	0.83	3.24	28.31	31.65
Min	0.83	3.17	28.01	29.12
Max	346.59	3769.13	5095.64	974849.69

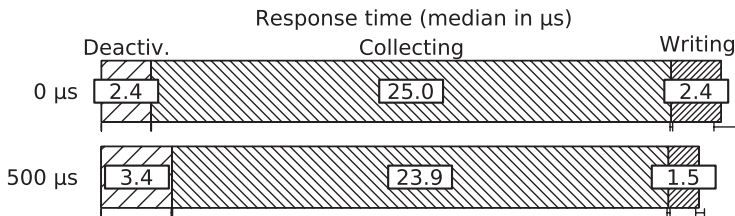


Figure 12.5. Comparison of the median response time overhead of inspectIT probes with workloads of 0 μs and 500 μs (including the 25% and 75% quartiles)

12.1.5 Comparing Different Workloads

In our third evaluation of inspectIT, we compare different workloads. Specifically, we compare our results from the previous evaluation, that used a `methodTime` of 500 μs , with a similar benchmark experiment using a configured `methodTime` of 0 μs . Due to its similarity with the common Kieker probes, we focus the rest of our evaluations on the isequence probe. Furthermore, high workloads in combination with the timer probe regularly result in errors due to excessive garbage collection.

The results of our benchmark experiments are presented in Table 12.4. Additionally, we provide a direct comparison of the median monitoring overhead of the two different workloads in Figure 12.5.

Discussion of the Results

Overall, the results of our benchmark experiment with a configured method time of $0\ \mu\text{s}$ are similar to our previous experiment with a configured method time $500\ \mu\text{s}$. The overhead of the deactivated probe (*I*) is slightly lower (from $3.4\ \mu\text{s}$ to $2.4\ \mu\text{s}$). This reduction is probably caused by different JIT optimizations, e. g., inlining, due to the changed runtime behavior (shorter execution path) of the `monitoredMethod()`. The overhead of collecting (*C*) and sending data (*W*) are slightly increased. The former increases from $23.9\ \mu\text{s}$ to $25\ \mu\text{s}$, while the latter increases from $1.5\ \mu\text{s}$ to $2.4\ \mu\text{s}$. This raise is caused by the higher workload of our benchmark. It is especially visible in the difference between the mean and median response times of writing (hinting in a skewness in our measurement distribution), as well as in the measured maximum response time of almost one second.

Again, the experiment hints at the collection of monitoring data as a main target for future performance tunings of inspectIT. A major cause of the skewness and the high maxima is garbage collector activity (with garbage collection regularly taking more than 0.5 s. Furthermore, the burst behavior (also affecting the high maximal response time) of sending data could be improved. However, this could also be caused by our employed buffering and sending strategy. By adjusting these parameters in additional experiments, such performance tunings could be guided. However, these experiments as well as the corresponding tunings are outside of the scope of our evaluation of MooBench with inspectIT.

12.1.6 Comparing Local CMR to Remote CMR

In our next evaluation of inspectIT, we compare a locally deployed CMR to a CMR deployed on a remote server. The remote server is an identically configured machine to our primary server running the benchmark. It is connected with a common 1 GBit/s local area network to the primary server. Furthermore, we employ the configured `methodTime` of $0\ \mu\text{s}$ from our last experiment and an `isequence` probe. Thus, we can reuse our previous measurements for the local CMR.

12. Comparing Monitoring Frameworks with MooBench

Table 12.5. Response times for the isequance probe with a remote CMR (in μs)

	No instr.	Deactiv.	Collecting	Writing
Mean	0.84	3.28	28.58	36.55
95% CI	± 0.00	± 0.03	± 0.04	± 5.01
Q ₁	0.83	3.23	28.08	32.74
Median	0.83	3.24	28.13	33.35
Q ₃	0.83	3.26	28.20	34.29
Min	0.83	3.20	27.93	29.14
Max	128.02	4275.52	5857.00	1269495.52

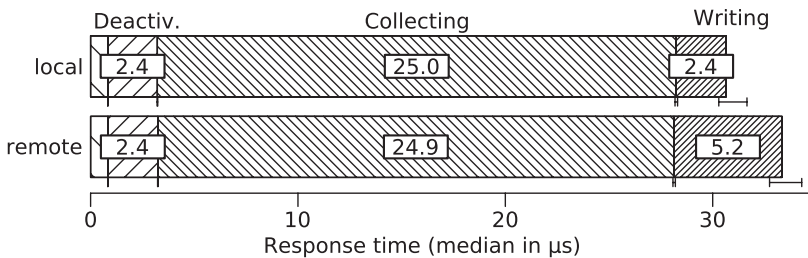


Figure 12.6. Comparison of the median response time overhead of inspectIT probes with local and remote CMR (including the 25% and 75% quartiles)

Experimental Results & Discussion

The results of our benchmark experiments are presented in Table 12.5. Additionally, we provide a direct comparison of the median monitoring overhead of the local CMR with the remote CMR in Figure 12.6.

The measured response times for the deactivated probe (*I*), as well as for the data collection (*C*) are very similar and in both cases slightly below the local CMR. This is probably caused by freeing the available CPUs from several additional CMR threads. In contrast, the additional response time overhead of sending the monitoring data (*W*) increases from an average of 7.9 μs to 8.0 μs and from a median value of 2.4 μs to 5.2 μs . The fact that

12.1. The inspectIT Monitoring tool

mostly the median is affected rather than the mean hints at a slightly higher overhead while sending each monitoring record, while the garbage collector behavior (causing a few very high response times) is unaffected.

12.1.7 Conclusions

Our performed experiments demonstrate the applicability of MooBench as a benchmark to determine the monitoring overhead of inspectIT. However, some adjustments to inspectIT, as detailed in Section 12.1.1, are required. The gravest adjustment is required in the employed hash function. Otherwise, at least a baseline evaluation of the monitoring tool is feasible.

With additional minor adjustments, more detailed experiments are possible to determine the actual causes of monitoring overhead. These experiments provide first hints at possible targets for future performance optimizations of inspectIT.

Compared to Kieker, the total monitoring overhead is higher, similar to earlier versions of Kieker before benchmark aided performance evaluations were performed (see Section 11.2 and Section 11.5). This higher overhead is mainly caused by the collection of monitoring data. On the other hand, the actual sending of monitoring data is similar to or slightly faster than with Kieker. This can be caused by Kieker's focus on data integrity contrary to inspectIT's focus on throughput. That is, inspectIT discards gathered data to reduce overhead while Kieker (in its default configuration) either blocks or terminates under higher load.

As mentioned before, our configuration of inspectIT is provided with MooBench. Additionally, we provide ready to run experiments in order to facilitate repeatability and verification of our results. Similarly, our raw benchmark results as well as produced log files and additional diagrams are available for download [Waller 2014b].

Addendum After presenting these results to the developers of inspectIT, we have received very encouraging responses. For instance, it is planned to integrate the adjustments to the inspect-it class files into future releases. Furthermore, they have replicated our performance tests and they are planning to integrate MooBench into automated weekly load tests of inspectIT.

12. Comparing Monitoring Frameworks with MooBench

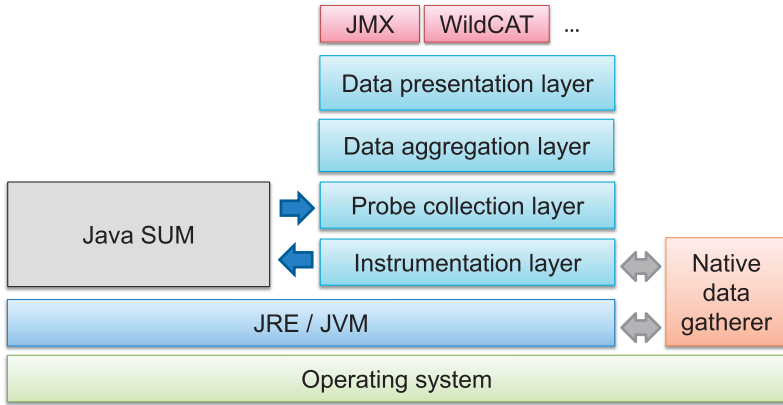


Figure 12.7. Architecture of SPASS-meter [Eichelberger and Schmid 2014]

12.2 The SPASS-meter Monitoring tool

The SPASS-meter monitoring tool (website: <http://sse.uni-hildesheim.de/spass-meter/>) has been developed by the Software Systems Engineering working group at University of Hildesheim [Eichelberger and Schmid 2012; 2014]. It is provided as open source software [Apache License, Version 2.0]. For our evaluation, we employ release version 0.75⁴ that, as of this writing, also corresponds to the most recent source code available in its git repository.⁵

The monitoring focus of SPASS-meter is on the resource consumption of Java applications. This includes resources such as CPU-time and response time, but also memory consumption or file and network operations. These details are collected and aggregated for so-called monitoring groups, i. e., groups of classes or methods. The collected resource details are finally presented in absolute values as well as relative to the system, the JVM, or other monitoring groups.

A general overview on the static architecture of SPASS-meter is presented in Figure 12.7. The System Under Monitoring (SUM) can be instrumented

⁴<http://projects.sse.uni-hildesheim.de/SPASS-meter/>

⁵<http://github.com/SSEHUB/spassMeter>

12.2. The SPASS-meter Monitoring tool

on two different layers. Native data gatherers, i. e., probes written in C++, are inserted into the JVM as well as into the SUM with the help of the Java Virtual Machine Tool Interface (JVM TI) (see Section 4.3.4). In addition, Javassist [Javassist] or ASM [ASM] are employed to insert Java-based probes into the SUM (see Section 4.3.5). Finally, this instrumentation can be performed during different times, i. e., dynamic during the load time or the runtime, static before the runtime, or mixed in a combination of both.

All gathered monitoring data, regardless of the source, is then collected and pre-processed in the Probe collection layer. The next steps are depending on the configured analysis technique: local or remote. In the case of local analysis, the collected information is aggregated into the configured monitoring groups. Then, the data is prepared for presentation, e. g., as a simple textual summary, or via more advanced JMX or WildCAT plugins. In the case of remote analysis, the data is instead sent to a remote server via TCP. The remote server then performs the requested analyses instead of the local monitoring system.

The configuration of SPASS-meter is usually performed with a combination of JVM parameters and XML configuration files. Furthermore, it is possible to utilize Java annotations instead of the configuration file. However, this approach is more feasible for development projects than benchmark experiments. We provide all configuration files and document all required parameters for our experiments with our releases of MooBench.

12.2.1 General Benchmark Parameters

With this series of benchmark experiments we evaluate the feasibility of MooBench to determine the monitoring overhead of the resource monitoring tool SPASS-meter.

To enhance the comparability of our experiments, we try to utilize similar environments. However, due to the use of native code within SPASS-meter, we are unable to use our common Solaris testbed. Hence, we utilize an identically configured X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM, this time running Debian 3.2.57 with an additional upgrade of the required libraries `libc`, `libdl`, and `libpthread` to version 2.15. Furthermore, we employ the Oracle Java 64-

12. Comparing Monitoring Frameworks with MooBench

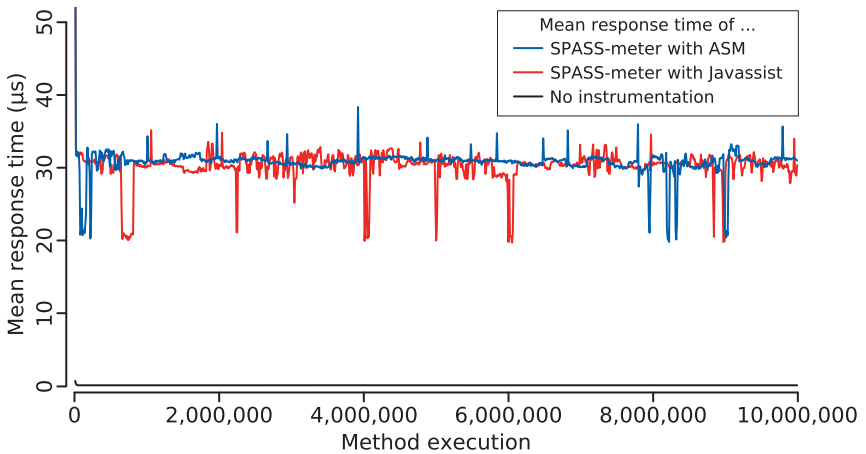


Figure 12.8. Extended time series diagram for experiments with SPASS-meter

bit Server JVM in version 1.7.0_55 with up to 4 GiB of heap space provided to the JVM. This hardware and software system is used exclusively for the experiments and is otherwise held idle.

The instrumentation of the monitored application is performed through load-time weaving using either Javassist or ASM, as noted with each experiment. This selection is performed with the command-line switch `-Dspass-meter.iFactory`. Besides this, SPASS-meter can be configured to either perform an analysis on the fly concurrent to the running SUM, or to connect via TCP to a remote instance, where this analysis is performed. The chosen configuration is noted with each experiment. Otherwise, all of the experiments utilize the same sets of environment, workload, and system parameters.

The configuration of the MooBench parameters is, except as noted, left at default values, as described in Listing 8.1 on page 133. That is, we use 2,000,000 calls of the monitored method with a recursion depth of ten and discard the first half of executions as warm-up period. Each part of each experiment is repeated ten times on fresh JVMs. The configured method time is set to 0 μ s.

12.2. The SPASS-meter Monitoring tool

We have confirmed this configuration for a warm-up period with long running experiments. For instance, in Figure 12.8 a timeseries diagram with results from a series of three experiments with 10,000,000 calls of the monitored method with a recursion depth of ten is displayed. Even with this longer runtime, the measured response times behave erratic. However, this behavior can not be attributed to JIT compilation or garbage collection activity, but is rather caused by the online analysis performed by SPASS-meter. Furthermore, our measurements suggest that an initialized state is usually reached within the first 100,000 monitored method calls. Thus, our configured warm-up period is sufficient for our experiments.

Finally, we employ the `MonitoredClassSimple` implementation for our experiments rather than the more advanced implementation querying the `ThreadMXBean` (see Section 8.2.1). Due to our testbed with Debian 3.2.57, the rapid JMX queries of the advanced implementation cause a high overhead and distort our measurements, e.g., an average response time for the uninstrumented system of 28.90 μs compared to 0.09 μs with the simple implementation.

Our configurations of SPASS-meter is provided with MooBench.⁶ Additionally, we provide the preconfigured External Controllers used to run our benchmarking experiments. In order to configure MooBench for SPASS-meter, the ant build target `build-spasmeter` can be used. This way, a ready to use benchmark environment is provided.

12.2.2 Initial Performance Benchmark

In this initial performance evaluation of the SPASS-meter monitoring tool, we evaluate the influence of the two provided instrumentation mechanisms on the response time overhead of monitoring. In doing so, we can demonstrate the feasibility of our MooBench micro-benchmark for performing these kinds of comparisons with different monitoring tools or frameworks.

In our first series of experiments, we focus on the default configuration of SPASS-meter, including on-the-fly data analysis concurrent to the execution of the SUM. In the next section, we evaluate the influence of remote analysis via TCP in comparison to these base results (Section 12.2.4). Finally, a more

⁶<http://kieker-monitoring.net/MooBench>

12. Comparing Monitoring Frameworks with MooBench

Table 12.6. Response times for the initial SPASS-meter experiment (in μs)

	No instr.	SPASS-meter	SPASS-meter (ASM)
Mean	0.09	31.10	27.95
95% CI	± 0.00	± 0.07	± 0.05
Q ₁	0.09	29.38	26.27
Median	0.09	31.60	28.76
Q ₃	0.09	33.47	30.60
Min	0.08	12.20	11.47
Max	10.41	36060.56	23350.49

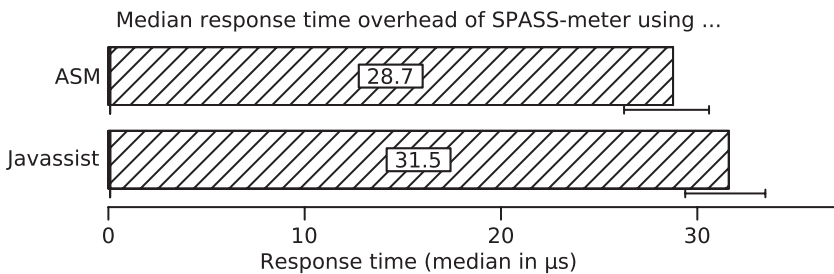


Figure 12.9. Median response time for the initial SPASS-meter experiment (including the 25% and 75% quartiles)

complex experiment, analyzing the common causes of monitoring overhead, is presented in Section 12.2.4.

We start our evaluation of SPASS-meter with a set of three different benchmark experiments, employing MooBench:

1. a baseline experiment with the uninstrumented benchmark (to compare the other results against),
2. an experiment with the default configuration of SPASS-meter, i. e., employing Javassist for the instrumentation and performing an on-the-fly analysis of the gathered data, and
3. a similar experiment employing ASM instead of Javassist.

Experimental Results & Discussion

The results of our benchmark experiments are presented in Table 12.6 and in Figure 12.9. Additional results are available in Figure 12.10.

For the default configuration with Javassist, we measure an average response time overhead (in comparison to our baseline experiment) of 31.01 μs . The measured median overhead is very similar at 31.51 μs . The closeness of these measured values, as well as the determined confidence interval and the quartiles hint at rather stable results with only few outliers. However, in comparison to our previous evaluations of other monitoring frameworks, the median is above the mean. This hints at a response time distribution with a lot of values below this average, rather than the usual case of lots of outliers with higher values. This behavior is also evident in the respective time series diagrams, for instance in Figure 12.8.

Similar results can be observed with the experiment employing ASM instead of Javassist. The measured mean and median values of the response time overhead in comparison to our baseline are slightly lower with 27.86 μs and 28.67 μs , respectively. However, otherwise, the same behavior as above can be observed.

If we compare these findings to the experiments conducted with the SPECjvm[®]2008 by Eichelberger and Schmid [2014], the observed results are similar. For instance, in both cases the ASM-based implementation has a slightly better performance. Eichelberger and Schmid also compared the performance of SPASS-meter to Kieker, resulting in a better measured performance with Kieker 1.5. However, no additional details have been given on the actual configuration of Kieker.

Assuming a standard configuration, we can compare our results to the results presented in Figure 11.2 on page 186. Our measured average response time overhead for Kieker 1.5 is 30.36 μs . This value is comparable to our measurements with SPASS-meter. Contrary to the experiments of Eichelberger and Schmid, Kieker provides no definite performance improvements in our experiments.

However, as is evident by, e. g., Figure 11.3 on page 187, Kieker's performance greatly improves with lower workloads, e. g., resulting in an average overhead of 17.85 μs . A similar experiment with SPASS-meter (con-

12. Comparing Monitoring Frameworks with MooBench

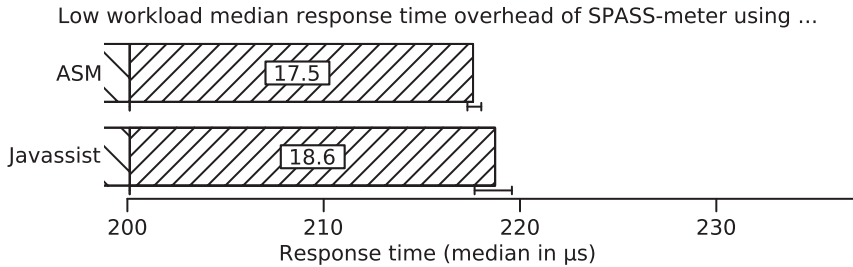


Figure 12.10. Low workload median response time for SPASS-meter experiments (including the 25% and 75% quartiles)

figured method time of 200 μs) results in an average overhead of 19.96 μs or 19.58 μs , respectively. The results of this additional benchmark experiment are illustrated in Figure 12.10. Thus, with the help of a low workload benchmark, the performance benefit of Kieker, as determined by Eichelberger and Schmid [2014], can be detected in our results as well, albeit less pronounced.

As is mentioned by Eichelberger and Schmid, direct comparisons between the performance of Kieker and SPASS-meter are usually misleading. Both frameworks collect different data and perform different analyses with these data points. Thus, different workloads can have a huge impact on the results, i. e., the workloads of MooBench compared to the workload of SPEC_{jvm}[®]2008.

However, the results of our MooBench experiments for SPASS-meter are promising. We can confirm the findings of Eichelberger and Schmid, concerning the different performance of the two implementations within SPASS-meter. Furthermore, our MooBench micro-benchmark has successfully been applied to SPASS-meter without any problems.

12.2.3 Performance Benchmark of Remote Analysis

In our next series of benchmark experiments, we evaluate the influence of the concurrent online analysis of SPASS-meter. Thus, we repeat the experiments from our initial series with an active TCP connection to an

12.2. The SPASS-meter Monitoring tool

Table 12.7. Response times for the SPASS-meter TCP experiment (in μs)

	No instr.	SPASS-meter	SPASS-meter (ASM)
Mean	0.09	50.14	46.64
95% CI	± 0.00	± 43.16	± 41.49
Q ₁	0.09	16.04	13.33
Median	0.09	16.96	13.56
Q ₃	0.09	17.56	14.16
Min	0.08	11.64	11.99
Max	8.47	16055711.23	15957040.17

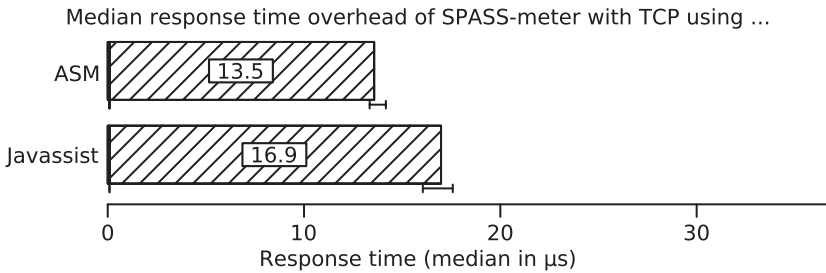


Figure 12.11. Median response time for the SPASS-meter TCP experiment (including the 25% and 75% quartiles)

analysis server. This server runs locally on the same machine as the SUT. So, instead of performing the analysis of the monitoring data concurrent to the execution of the SUM in the same JVM, it is executed on the same physical machine, albeit in a different JVM with a less strict coupling between the monitoring and the analysis.

Note that two bugs in SPASS-meter 0.75 prevent the execution of experiments with the TCP connection. The first bug causes a stack overflow by erroneously creating an infinite loop, the second bug prevents the termination of SPASS-meter. Both bugs are trivial to fix. The bugfixes are included in the SPASS-meter library available with MooBench and will be fixed in future releases of SPASS-meter.

12. Comparing Monitoring Frameworks with MooBench

Experimental Results & Discussion

The results of these additional benchmark experiments are presented in Table 12.7 and in Figure 12.11.

In comparison to our previous series of experiments, the median overhead has been greatly reduced to 16.87 μ s or 13.47 μ s, respectively. However, the measured mean values are very high with extremely large confidence intervals. This is caused by few very high outliers, as also evident by the measured maximal response times of about 16 seconds.

With an analysis of our benchmark execution logs, we can attribute this behavior to garbage collections taking this huge amount of time. Thus, the splitting into two JVMs greatly improved most of the executions, but also caused extreme garbage collections due to massive amounts of additional required memory. To further evaluate this memory consumption scenario, we increase the heap size of the employed JVM in the next series of experiments to 12 GiB.

Apart from that, our previous findings hold. That is, the ASM-based implementation of SPASS-meter still provides a slightly better performance. Furthermore, these experiments again demonstrate the feasibility of MooBench as a benchmark for different monitoring frameworks and especially for comparing different scenarios and configurations for a single framework or tool.

12.2.4 Causes of Monitoring Overhead

In our final series of benchmark experiments, we investigate the three common causes of monitoring overhead, as introduced in Section 6.2. Similar to our earlier experiments, we perform a total of four benchmark experiments to determine each individual portion of monitoring overhead as well as the base method time T .

The first experiment measures the uninstrumented benchmark system, i. e., T . In the next experiment, we perform an instrumentation with SPASS-meter, but we never actually start the monitoring activity by setting the provided command line parameter `mainDefault=NONE`. Thus, we can measure $T + I$. For the third experiment, we deactivate the configured TCP

12.2. The SPASS-meter Monitoring tool

Table 12.8. Response times for the third SPASS-meter experiment (in μs)

	No instr.	Deactiv.	Collecting	Writing
Mean	0.09	0.12	12.07	16.58
95% CI	± 0.00	± 0.00	± 0.00	± 0.06
Q ₁	0.09	0.12	11.90	15.99
Median	0.09	0.12	12.02	16.66
Q ₃	0.09	0.12	12.13	17.19
Min	0.08	0.11	11.46	12.33
Max	9.91	162.55	108.63	27740.66

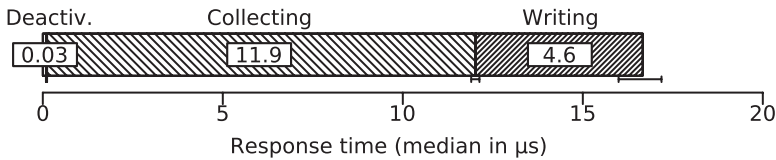


Figure 12.12. Median response time for the third SPASS-meter experiment (including the 25% and 75% quartiles)

writer. In order to do so, a minor code modification is necessary, adding a new command line switch `-DspassmeterNoWriter=true`. In this way, we can measure $T + I + C$. Finally, we perform full monitoring with an active TCP connection, resulting in $T + I + C + W$. This final experiment is a replication of our experiment from the previous series, albeit with a higher memory allocation.

All experiments are performed with the default instrumentation technology of Javassist. In addition, as mentioned with the previous set of experiments, we increase the allocated amount of heap memory to 12 GiB to avoid excessive garbage collector activity. The necessary benchmark configurations, as well as the required changes to SPASS-meter, are included in our releases of MooBench.

12. Comparing Monitoring Frameworks with MooBench

Experimental Results & Discussion

The results of these final benchmark experiments are presented in Table 12.8 and in Figure 12.12.

First of all, the results are very stable. Even with our final experiment, which is a replication of our experiments from the previous section, we get a very small confidence interval and a mean that is very similar to the median. Furthermore, the measured mean value is very similar to the results of the previous section. Thus, the higher available heap space fixes performance problems due to excessive garbage collector activity.

The measured performance overhead cost of instrumenting (*I*) a series of ten recursive method calls is very low with a median of 0.03 μ s. The major cause of monitoring overhead is the act of collecting monitoring data (*C*). Here, we can determine a median overhead of 11.9 μ s. Finally, the act of sending these collected data with the available TCP connection (*W*) adds the final 4.6 μ s of overhead.

Regarding the three common causes of monitoring overhead, the main cause is the collection of monitoring data. This corresponds to our experience with the performance tunings of Kieker in Section 11.5 and with our experiments with inspectIT in Section 12.1. The typical cost of instrumentation is rather low. The cost of sending the data to a remote analysis can also be tuned to very low, as long as sufficient idle cores are available. However, the actual collection of monitoring data cannot be avoided.

Furthermore, as is evident from our results in the previous two subsections, the main cause of overhead in SPASS-meter is caused by the concurrent analysis of the gathered data. This result is also backed by the findings of Eichelberger and Schmid [2014].

12.2.5 Conclusions

Our three performed series of benchmark experiments again demonstrate the capabilities of MooBench as a benchmark for monitoring frameworks or tools. In all three series of experiments, MooBench has been employed without any complications. Only in the final experiment, a minor modification to the SPASS-meter code has been required.

12.2. The SPASS-meter Monitoring tool

Similar to Kieker or inspectIT, MooBench could easily be employed to further guide the development of SPASS-meter. For instance, further experiments could be used to provide a more detailed analysis of the overhead of the concurrent analysis of monitoring data and to provide hints for performance optimizations.

Finally, as mentioned before, all configurations of SPASS-meter are provided with MooBench.⁷ Additionally, we provide ready to run experiments in order to facilitate repeatability and verification of our results. Similarly, our raw benchmark results as well as produced log files and additional diagrams are available for download [Waller 2014b].

⁷<http://kieker-monitoring.net/MooBench>

Evaluation of Kieker with Macro-Benchmarks

In this chapter, we evaluate the performance of Kieker with several macro-benchmarks. The results of these benchmarks can be used to validate the results of our MooBench micro-benchmarks. First, we employ the *Pet Store* (Section 13.1). Next, we perform measurements with the more complex *SPECjvm*[®]2008 in Section 13.2. In Section 13.3, we benchmark Kieker with the *SPECjbb*[®]2013 that simulates a complex landscape of systems. Finally, we present a *summary* and an outlook on *experiments with further macro-benchmarks* in Section 13.4.

An experiment is a question which science poses to Nature, and a measurement is the recording of Nature's answer.

— Max Planck, recipient of the Nobel price

13. Evaluation of Kieker with Macro-Benchmarks

Previous Publications

Parts of this chapter are already published in the following works:

1. 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)*. ACM, Apr. 2012, pages 247–248

13.1 Experiments with the Pet Store

We have performed a total of four benchmark experiments with our Pet Store macro-benchmark (see Section 9.2). These four experiments correspond to the four experiments to determine the causes of monitoring overhead, as introduced in Section 6.2.

Experimental Setup

For our experiments, we utilize two X6270 Blade Servers each with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM. For the software environment, we utilize Solaris 10 with the Oracle Java 64-bit Server JVM in version 1.7.0_45. The employed Jetty version is 7.6.10, while the jMeter is used in version 2.11.

The first server runs our workload generator, while the second server runs the JPetStore application. Both servers are connected with a common 1 GBit/s local area network. Besides the execution of the experiments, both systems are held idle.

Kieker is used in release version 1.9 with event-based records and (in the final experiment) a binary writer. For the instrumentation, AspectJ in version 1.7.4 is employed. We utilize a full instrumentation without getters or setters. This way, in our final experiment a total amount of 2.7 GiB of binary monitoring log files is produced within its 24 minute runtime.

The full configuration of our macro-benchmark and all results of our experiments are available for download [Waller 2014b].

Benchmark results

The average results of our four benchmark experiments are presented in Figure 13.1. The results are ordered from light to dark, where the lightest results correspond to a measurement of T and the darkest results correspond to a measurement of $T + I + C + W$.

Considering only our measurement of T or $T + I + C + W$, we can determine an average monitoring overhead between 0 and 3 ms. However, adding the remaining two experiments, the results become less clear. For instance, with signoff all experiments except the third provide identical results,

13. Evaluation of Kieker with Macro-Benchmarks

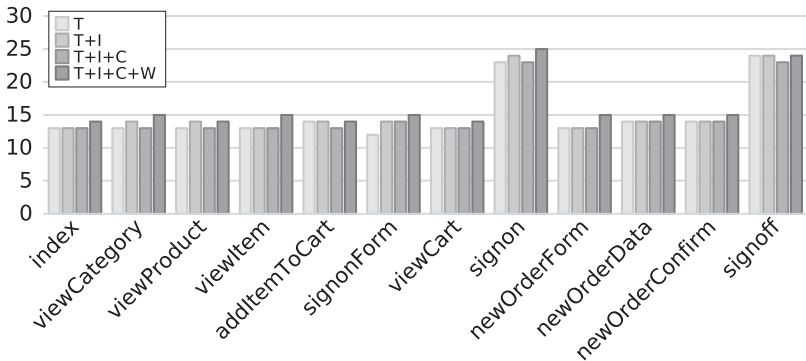


Figure 13.1. Average benchmark results for our JPetStore macro-benchmark

while the third even provides a better performance. These discrepancies are mostly caused by the imprecise measurements of jMeter.

Additional replications of the experiments have provided similar results. However, the discrepancies are found in different cases with each repetition.

Furthermore, in all experiments the CPU of the SUT is not fully utilized. This is a typical behavior for enterprise systems. With full utilization, the system would be beyond its capacity (see Figure 2.2 on page 23). In the case of our experiments, we can measure a maximal CPU utilization of 30% in the first experiment and a maximum of 40% in the final experiment. Thus, as long as we are below capacity, there usually are sufficient available resources to minimize the influence of monitoring.

The benchmark results are only of limited use to validate our MooBench results. We can mostly confirm the binary writer as the main cause of monitoring overhead (see also Figure 11.4 on 189). However, the division into the three common causes of monitoring overhead cannot be validated.

In summary, we can employ our Pet Store macro-benchmark to determine the percentage monitoring overhead o of Kieker in this scenario and environment at between 0% and 25%. However, the provided measures of jMeter are too imprecise to perform a more detailed evaluation of the causes of monitoring overhead. Thus, in the following, we will utilize more refined macro-benchmarks.

13.2 Experiments with the SPECjvm2008

Similar to our experiments with the Pet Store in the previous section, we have performed a series of four benchmark experiments with the SPECjvm[®]2008 macro-benchmark (see Section 9.3). These four runs of the benchmark are used to determine our previously introduced causes of monitoring overhead (Section 6.2).

Experimental Setup

Our experimental environment consists of an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM. It is running Solaris 10 with the Oracle Java 64-bit Server JVM in version 1.7.0_60 with 10 GiB assigned memory. Besides the execution of the experiments, the system is held idle.

The SPECjvm[®]2008 benchmark is employed in release version 1.01. It is configured to ignore the validation of its class files to prevent problems caused by the instrumentation with Kieker. Furthermore, we restrict the set of executed benchmarks according to Section 9.3. Otherwise the default configuration of the benchmarks is employed. Thus, each benchmark begins with a 120 s warmup period and is then scheduled for an execution of 240 s.

Kieker is used in release version 1.9 with AspectJ 1.7.4, event-based records in full instrumentation without getters and setters, and (in the final experiment) a TCP writer. This way, in our final experiment a total amount of about 160 GiB of binary monitoring data is written by the TCP writer.

The full configuration of our macro-benchmark experiments and our results of the experiments are available for download [Waller 2014b].

Benchmark results

The results of each sub-benchmark as well as the total score of the SPECjvm[®] 2008 are presented in Table 13.1 and Figure 13.2. Depending on the actual sub-benchmark, the monitoring stages have a greater or lesser impact on the measured throughput.

For instance, in the case of the Crypto benchmark, mostly unmonitored methods of the Java API are executed. Thus, only minimal percentage

13. Evaluation of Kieker with Macro-Benchmarks

Table 13.1. Benchmark results of the SPECjvm2008 (in operations per minute)

	No instr.	Deactiv.	Collecting	Writing
Crypto	739.05	741.48	739.36	733.93
Derby	611.69	526.69	117.20	2.48
Mpegaudio	246.07	240.50	237.51	224.60
Scimark (large)	47.19	36.27	24.11	1.11
Scimark (small)	422.38	373.96	105.54	2.30
Serial	281.97	269.36	237.98	19.79
Sunflow	148.34	147.98	147.17	147.92
Xml	842.69	849.80	824.80	770.87
Total (ops/min)	306.63	284.69	187.27	35.19

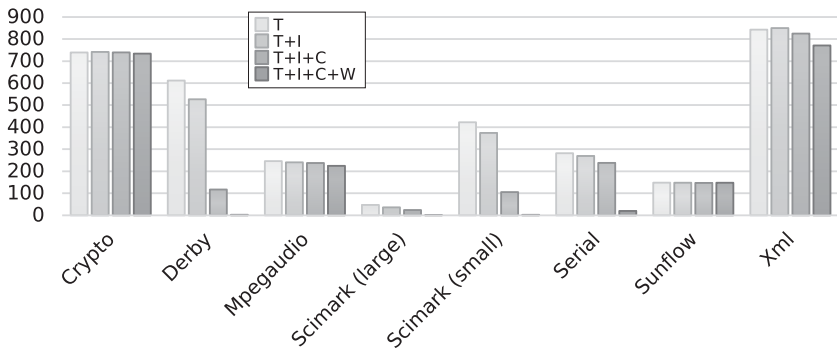


Figure 13.2. Benchmark results of the SPECjvm2008 (in operations per minute)

monitoring overhead o is measurable when writing data ($o = 1\%$). The apparent speedup with deactivated monitoring or data collection in comparison to the uninstrumented benchmark is within the typical variation of measurement results for this benchmark and not significant.

In comparison, within the Derby sub-benchmark we can detect a major impact of monitoring on the measured throughput. While a deactivated probe has some impact, collecting the data and especially writing data has catastrophic impact on the throughput. This is mainly caused by sev-

eral very small helper methods within the benchmark, e. g., constructing prepared database statements, that are called very often. In the uninstrumented benchmark, these methods can be optimized by the JIT, for instance, by inlining. Due to the added monitoring code, these optimizations are no longer possible. Furthermore, due the number of calls to very small and fast methods, the monitoring writer effectively blocks, forcing all 16 active benchmark threads to wait and to synchronize with each other (cf., Section 8.2.4 on page 145 (high workload) and Section 11.4 on page 194 (no available cores for monitoring)). In total, the percentage monitoring overhead o for the derby sub-benchmark is almost 25 k%.

In all cases with significant measurements, we can identify collecting and especially writing as the main causes of monitoring overhead. This corresponds to our previous experiments with the MooBench micro-benchmark (see Chapter 11). Contrary to these previous experiments with the TCP writer (see, for instance, PT1 in Section 11.5), the influence of writing is greater. This is caused by the benchmark starting 16 active threads, effectively blocking all available cores. Similar experiments performed with MooBench confirm this behavior (see A1 in Section 11.4).

Additional details on the results, including additional diagrams and the raw measurement data, are available for download [Waller 2014b].

Discussion

We have repeated the experiment described above with an additional parameter to repeat all executions ten times (-i 10). These result are available for download, too [Waller 2014b]. Although the measured values for each benchmark differed, the general trend of the results remains. In several of the sub-benchmarks, monitoring has only minimal or no measurable impact. In other cases, the monitoring overhead is catastrophic, e. g., with the Derby benchmark a reduction from an average of 600.93 operations/minute down to 2.45 operations/minute.

The actual impact of monitoring on the sub-benchmarks is largely dependent upon the structure of these benchmarks. Benchmarks consisting of many small Java methods that are called in rapid succession are expensive to monitor, while benchmarks mostly consisting of unmonitored API calls

13. Evaluation of Kieker with Macro-Benchmarks

measure almost no overhead. However, this behavior is also present in real systems. Thus, the uneven instrumentation can be argued as increasing the relevance of the benchmark.

The importance of the choice of instrumentation points is also evident in the benchmark results published by Eichelberger and Schmid [2014]. The authors employed the SPECjvm[®]2008 to compare several different monitoring tools. However, the chosen instrumentation of the sub-benchmarks has been minimal, i. e., only a small subset of the available methods has been instrumented. In extreme cases, the instrumentation has been restricted to a few helper methods and the methods starting and stopping the execution of the benchmark itself. Thus, the measured minimal overhead of all investigated monitoring tools can be explained.

We have encountered several challenges within the SPECjvm[®]2008 benchmark suite itself. For instance, the set of benchmarks is executed within the same JVM instance. Thus, the sub-benchmarks influence each others via their JIT optimizations or garbage collections. Furthermore, the warm-up period and the measurement period are rather short. Especially in the case of active monitoring, a steady state may not be reached. For instance, with the Xml.validation workload, a performance degradation with each additional run is evident [Waller 2014b].

Additionally, although a measurement period of 240s is configured and displayed, each monitored execution of a sub-benchmark has taken up to 30 minutes. This behavior is caused by the internal construction of employed benchmark harness and could potentially further influence the results. Finally, similar to our benchmarking experiments with MooBench (see Chapter 11), several repetitions of the benchmark suite with fresh JVM instances might be beneficial to reduce the measurement jitter due to different optimization paths.

In summary, our experiments with the SPECjvm[®]2008 can confirm the three causes of monitoring overhead. Furthermore, we can identify collecting and especially writing as the major causes, which corresponds, as mentioned above, to our experiments with MooBench. However, we cannot perform any direct comparisons, such as comparing the exact ratios of overhead causes between both benchmarks. Furthermore, depending on the actual sub-benchmark, the results differ greatly.

13.3 Experiments with the SPECjbb2013

Similar to our previous experiments, we have performed a series of four benchmark experiments with the SPECjbb[®]2013 macro-benchmark (see Section 9.4). These four runs of the benchmark are used to determine our previously introduced causes of monitoring overhead (Section 6.2).

Experimental Setup

The SPECjbb[®]2013 benchmark simulates a large enterprise system. We employ our OpenStack private cloud infrastructure to provide a powerful environment for our experiments.

We employ two Virtual Machines (VMs) in the cloud: The first VM contains the controller and the transaction injector. The second VM contains the backend in its default configuration of two supermarkets, two suppliers, and one headquarter. Refer to Section 9.4 and the benchmark's documentation [SPEC 2013c; d] for further details.

Each used physical machine in our private cloud contains two eight-core Intel Xeon E5-2650 (2 GHz) processors, 128 GiB RAM, and a 10 Gbit network connection. When performing our experiments, we reserve two physical machines of the cloud and prevent further access in order to reduce perturbation. The two used VMs are each assigned 32 virtual CPU cores and 121 GiB RAM. Thus, both VMs are each fully utilizing a single physical machine. For our software stack, we employ Ubuntu 14.04 as the VMs' operating system and an Oracle Java 64-bit Server VM in version 1.7.0_65 with up to 64 GiB of assigned memory.

We employ Kieker release version 1.9 with AspectJ 1.7.4. The backend is instrumented with event-based records and full instrumentation but without getters and setters. Kieker is configured to use a blocking queue with 100,000,000 entries to synchronize the communication between the MonitoringWriter and WriterThread. The employed TCP writer produces a total amount of about 108 GiB of binary monitoring data in a single experiment run.

The full configuration of our macro-benchmark experiments is available for download [Waller 2014b].

13. Evaluation of Kieker with Macro-Benchmarks

Table 13.2. Benchmark results of the SPECjbb2013

	No instr.	Deactiv.	Collect.	Writing	ExplorViz
max-jOPS	268705	19490	2013	303	5783
critical-jOPS	7066	2938	806	147	2298

Benchmark results

For our first benchmarking experiment with the SPECjbb[®]2013 benchmark, we employ the benchmark in its default distributed configuration. That is, the benchmark automatically searches for the maximal workload that the SUT can handle. At the end of a two hour run, the benchmark performs a detailed analysis of its results. Two values serve as the benchmark score: The maximal sustainable throughput (max-jOPS) and the throughput under typical SLO constraints (critical-jOPS). Refer to the benchmark documentation for details on the calculation of the scores [SPEC 2013c; d].

The results of our four benchmark runs are presented in Table 13.2. We have also executed an additional run of the SPECjbb[®]2013 benchmark instrumented with a current build of the high-throughput optimized version of Kieker used in the ExplorViz project [Fittkau et al. 2013a].

As is evident by the table, the main factors for monitoring overhead are data collection and writing the monitoring data. This result confirms our expectations from our previous experiments. However, as is the case with our other macro-benchmark experiments, we cannot directly compare these results to the absolute numbers of our micro-benchmark experiments.

An analysis of the results is further hindered by the varying workloads of the experiments that are also reported as the respective scores. For instance, comparing the max-jOPS scores to the critical-jOPS scores, the deactivated probe seems to have a greater impact on the monitoring overhead in the seconds case. However, a comparison of all four scores for the critical-jOPS indicates that our division into three causes of monitoring overhead with collecting and writing having the major impact remains valid. The actual differences are caused by the second score being more sensitive to even small changes in the response times.

13.3. Experiments with the SPECjbb2013

Table 13.3. Benchmark results of the SPECjbb2013 (mean response times in ms)

	No instr.	Deactiv.	Collect.	Writing	ExplorViz
Response time	2.4 ms	2.6 ms	3.6 ms	7.2 ms	4.1 ms

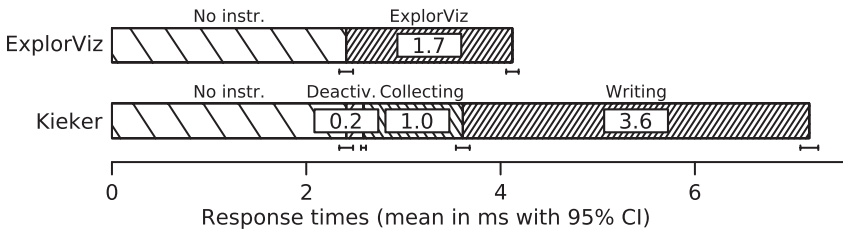


Figure 13.3. Benchmark results of the SPECjbb2013 (mean response times in ms)

Concerning our additional benchmark run with ExplorViz, we achieved better scores compared to Kieker (5783 instead of 303). This huge improvement is as expected from our previous experiments in Sections 11.5 and 11.6. Despite this huge performance, the more recent version of ExplorViz used in this experiment collects additional data (e. g., object identifiers for created class instances) compared to Kieker and also compared to the previously employed versions: The collected monitoring log contains 1.7 TiB of binary data compared to 108 GiB collected by Kieker.

Additional details on the results, including all results and many diagrams generated by the benchmark, are available for download [Waller 2014b]. To further validate our findings, we repeat the experiments with an adjusted benchmark configuration. Specifically, instead of varying workloads for each experiment, we employ a fixed workload of 100 (which would correspond to a score of 100) for ten minutes and compare the measured response times for the tasks of the benchmark.

The results of our four benchmark runs and of our additional experiment with the high-throughput optimized Kieker version of ExplorViz are presented in Table 13.3 and visualized in Figure 13.3. The SPECjbb[®]2013 benchmark reports eighteen median response times for the ten minute

13. Evaluation of Kieker with Macro-Benchmarks

period. We have discarded the first half as warm-up period and taken the average of the remaining nine response times to calculate our presented mean values.

Comparing these results to the results of PT1 (roughly Kieker 1.9) and PT4 (earlier version of ExplorViz) from Figure 11.11 on page 209, we can see very similar results: Instrumentation causes only minimal overhead while the main contributing factors to the monitoring overhead are data collection as well as writing. Furthermore, writing is more expensive than data collection. Although the performance of ExplorViz is not as good as indicated by our previous experiments, it is still considerably better than the performance of Kieker. However, as mentioned above, the current version collects additional data compared to the version used in our previous experiments.

Discussion

To improve the comparability of our benchmark results, we have run an additional series of Benchmark experiments with MooBench using the same environment and configuration of Kieker as in our SPECjbb[®]2013 experiments. The results are presented in Table 13.4 and visualized in Figure 13.4.

As is evident by the results, the micro-benchmark results are very similar to the results of our macro-benchmark. In Figure 13.5, we provide a normalized comparison of the monitoring overheads that have been measured within the MooBench micro-benchmark and within the SPECjbb[®]2013 macro-benchmark. The ratio between the three causes of monitoring overhead is very similar with both benchmarks. The higher impact of writing in the case of the macro-benchmark can be explained by the multi-threaded implementation of the benchmark that employs all available cores (refer to Section 11.4 for the influence of available cores on monitoring overhead).

The results and raw benchmark data of all our experiments with the SPECjbb[®]2013 are available for download [Waller 2014b]. This package also includes the necessary scripts and configuration parameters for the macro-benchmark as well as additional data on our presented results of our comparison to MooBench.

13.3. Experiments with the SPECjbb2013

Table 13.4. Response times (in μs) for Kieker release 1.9 with MooBench (using the TCP writer and event-based record in the cloud)

	No instr.	Deactiv.	Collecting	Writing
Mean	0.25	0.83	5.31	18.98
95% CI	± 0.00	± 0.03	± 0.07	± 0.47
Q ₁	0.25	0.66	5.02	10.16
Median	0.25	0.71	5.05	17.54
Q ₃	0.25	0.78	5.14	26.11
Min	0.24	0.59	4.93	5.66
Max	17.03	12593.89	29439.02	136899.97

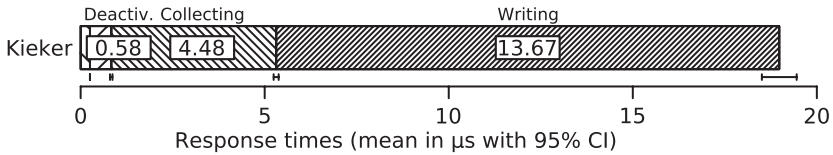


Figure 13.4. Response times for Kieker release 1.9 with MooBench (mean in μs) (using the TCP writer and event-based record in the cloud)

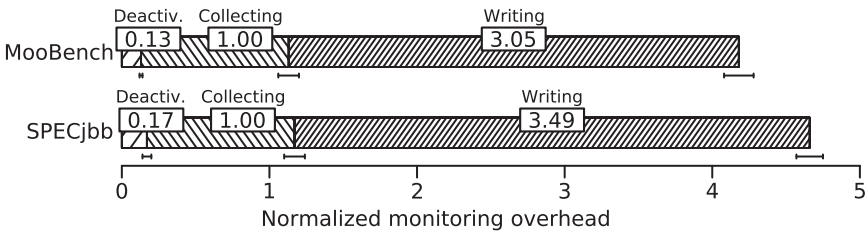


Figure 13.5. Normalized comparison of monitoring overhead between the SPECjbb 2013 macro-benchmark and the MooBench micro-benchmark

13.4 Summary and Further Macro-Benchmarks

As mentioned in Section 9.1, further macro-benchmarks are often employed to measure the performance of monitoring tools or frameworks. We have selected three representatives as a small subset of the possible set of Java-based macro-benchmarks.

Our first benchmark, the Pet Store (Section 13.1), is a basic sample application that has not originally been designed as a benchmark. Several similar applications, i. e., small open source software systems, can be used for similar benchmark scenarios. Although our measurement results are too imprecise to perform a detailed comparison to our micro-benchmark results, we can confirm writing as the major cause of monitoring overhead. Similar benchmark applications would probably provide very similar results. However, with the help of more complex benchmark applications and more precise measurements, additional insights might be possible.

Our second benchmark, the SPECjvm[®]2008 (Section 13.2), consists of a collection of smaller sub-benchmarks. Other commonly used collections of small benchmarks are, for instance, the DaCapo benchmark suite [Blackburn et al. 2006] or the Qualitas Corpus [Tempero et al. 2010]. Depending on the actually used sub-benchmarks, the experiment results differ significantly. However, we have been able to confirm the three sources of monitoring overhead with collecting and writing as the major causes. Further benchmark suites are expected to produce similar results. Depending on the actual implementation of the sub-benchmarks, monitoring overhead might be more or less severe.

Our third benchmark, the SPECjbb[®]2013 (Section 13.3), simulates a large enterprise system. In its default configuration, the results of the benchmark are difficult to compare due to their varying workloads. However, in a configuration with a fixed workload, we have retrieved monitoring overhead results that have a very similar ratio to the one expected from our MooBench benchmarks. Thus, these results validate our micro-benchmarking approach.

Further similar macro-benchmarks to the SPECjbb[®]2013 benchmark are, for instance, the SPECjEnterprise[®]2010 [SPEC 2012] and its predecessor [Kounev 2005]. Van Hoorn et al. [2012] report on a basic performance evaluation of

13.4. Summary and Further Macro-Benchmarks

the Kieker monitoring framework with the help of the SPECjEnterprise[®]2010 macro-benchmark. In the following, we present these results in greater detail. However, additional experiments with further similar macro-benchmarks are expected to result in similar findings as our presented ones.

Performance of Kieker with the SPECjEnterprise2010 Benchmark

Parts of this section on a monitoring overhead evaluation of Kieker with SPECjEnterprise[®]2010 benchmark have already been published in van Hoorn et al. [2012]. Note that the level of reporting of this evaluation is lacking in detail compared to our other experiments. For instance, the raw benchmark results are no longer available. Similarly, the exact details of the configuration of the environment are lost.

The SPECjEnterprise[®]2010 macro-benchmark has been developed to measure the performance of Java EE servers. It simulates a typical enterprise application, in this case an automobile manufacturer with web-based access for automobile dealers and service-based access for manufacturing sites and suppliers. In our experiment, we have deployed the components of the benchmark (load driver, application, database, and external supplier) onto four blade servers. Refer to the descriptions of the rest of our experiments for details on the typical hardware and software configuration of our blade servers. The used setup corresponds to a typical configuration for the SPECjEnterprise[®]2010 benchmark [SPEC 2012].

The first benchmark run is performed without any involvement of the Kieker framework. After a reinitialization of the database and the servers, a second run, monitored by Kieker release version 1.4, is started. The instrumentation is an AspectJ-based full instrumentation omitting getters, setters, and enhancement methods, i. e., all method calls, except methods with names starting with `get`, `set`, `is`, or `_persistence` are monitored, resulting in 40 instrumented classes with 138 instrumented methods. The Kieker asynchronous file system writer is used to persist the recorded data.

We use the Manufacturing Driver and the Dealer Driver with an injection rate parameter of 20, resulting in 260 concurrent threads accessing the benchmark. In each run a total number of approximately 115,000 enterprise operations is performed. Each enterprise operation consists of several

13. Evaluation of Kieker with Macro-Benchmarks

Table 13.5. Mean response times (in ms with 95% CI) of the SPECjEnterprise2010

	No Monitoring	Monitoring
CreateVehicleEJB	36 (± 0.6)	36 (± 0.6)
CreateVehicleWS	16 (± 0.3)	18 (± 1.2)
Purchase	10 (± 0.3)	12 (± 1.2)
Manage	8 (± 0.3)	9 (± 0.3)
Browse	23 (± 0.2)	26 (± 1.2)

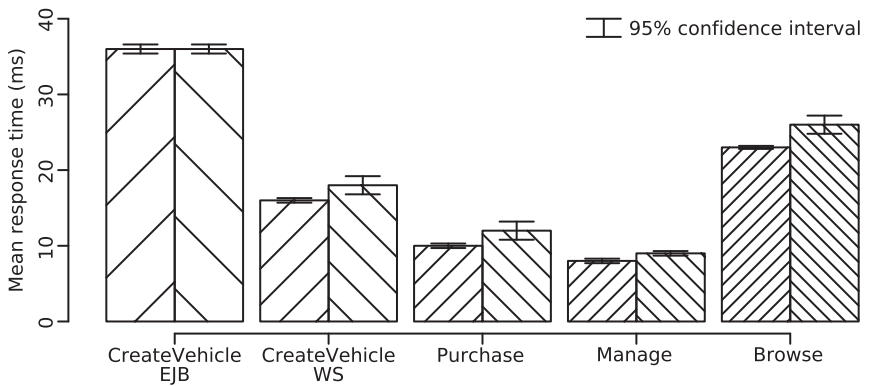


Figure 13.6. Benchmark results of the SPECjEnterprise2010 [van Hoorn et al. 2012]

method calls, totaling in over 1,000,000 individual transactions per experiment run. Each experiment run instrumented with Kieker produces 7 GiB of Kieker monitoring log files.

The results of our benchmark experiments are presented in Table 13.5 and Figure 13.6. The response times of the monitored system are slightly above the response times of the unmonitored system. Similarly to our experiments with the Pet Store, the measured percentage monitoring overhead is between 0% and 20% with an average percentage monitoring overhead of 9.0%.

13.4. Summary and Further Macro-Benchmarks

Compared to other large SPEC benchmarks, such as the SPECjvm[®]2008 or the SPECjbb[®]2013, the overhead is rather low. This is caused by the database reliance of the SPECjEnterprise[®]2010 benchmark. The unmonitored database is under extremely high load, while the monitored benchmark components regularly wait for database responses. Thus, there usually are sufficient spare resources available for Kieker.

We omit additional experiments with the SPECjEnterprise[®]2010 benchmark. They would probably not yield any additional insights into the monitoring overhead of Kieker beyond the results of the three already presented other benchmarks of this chapter. Furthermore, in addition to its problems with a database bottleneck, the SPECjEnterprise[®]2010 has not been widely distributed in the community. Since its publication in 2013, the similar SPECjbb[®]2013 benchmark has already received more published results than the SPECjEnterprise[®]2010 since its publication in 2009.

Meta-Monitoring of Kieker

In this chapter, we describe the results of our meta-monitoring experiments with Kieker and Kicker (see Section 10.1). The goal of these experiments is to validate our previous performance experiments, especially with respect to MooBench, with additional results. First, we describe our general *experimental setup* (Section 14.1). Then, we present *results of our experiments* with the help of the analysis functions included in Kieker (Section 14.2). Finally, we briefly present results from our analyses with *SynchroVis* (Section 14.3) and *ExplorViz* (Section 14.4).

Know thyself

— Inscription on the Temple of Apollo at Delphi

14. Meta-Monitoring of Kieker

14.1 Experimental Setup

To enhance comparability, we have performed our experiments on the same hardware and software infrastructure as most of our previous benchmark experiments, specifically an Oracle Java 64-bit Server VM in version 1.7.0_45 running on an X6270 Blade Server with two Intel Xeon 2.53 GHz E5540 Quadcore processors and 24 GiB RAM with Solaris 10 and up to 4 GiB of available heap space for the JVM.

We employ Kieker in a snapshot version of the (currently) upcoming Kieker release 1.10 as the monitoring framework under test. Similarly, our modified version of Kieker used for meta-monitoring, named Kicker, is based on the same release. The complete setup of our experiment, including all required binary and configuration files, is available with MooBench.¹

Kieker is configured to its default behavior with a single instrumentation point within the monitored method. To avoid problems caused by AspectJ instrumenting AspectJ-generated code, we employ a manual instrumentation of our `monitoredMethod()` that simulates the code insertions normally performed by AspectJ. This way, we can instrument the `monitoredMethod()` as well as all methods of Kieker with the help of Kicker and AspectJ release 1.8.0. The exact configuration of Kicker and the implementation of our manually instrumented `monitoredMethod()` are available with MooBench. However, note that we employ a binary writer for Kicker to reduce its monitoring log size.

We employ MooBench as a workload generator for our experiments. It is configured to employ a single thread, performing recursive calls of the `monitoredMethod()` with a stack depth of ten and a configured method time of 0 μ s. A total of 20,000 calls is performed and the first half gets discarded as warm-up period.

A single run of the experiment takes about four minutes and results in 79 MiB of Kieker logs, containing monitoring traces of the `monitoredMethod()` in an ASCII format. In the same time, a total of 1.15 GiB of Kicker logs in a binary format are created. The size of the Kicker logs and the resulting storability and analyzability is our main reason for restricting our experiment to the rather low runtime of 20,000 benchmark calls.

¹<http://kieker-monitoring.net/MooBench/>

14.2. Analyzing our Results with Kieker

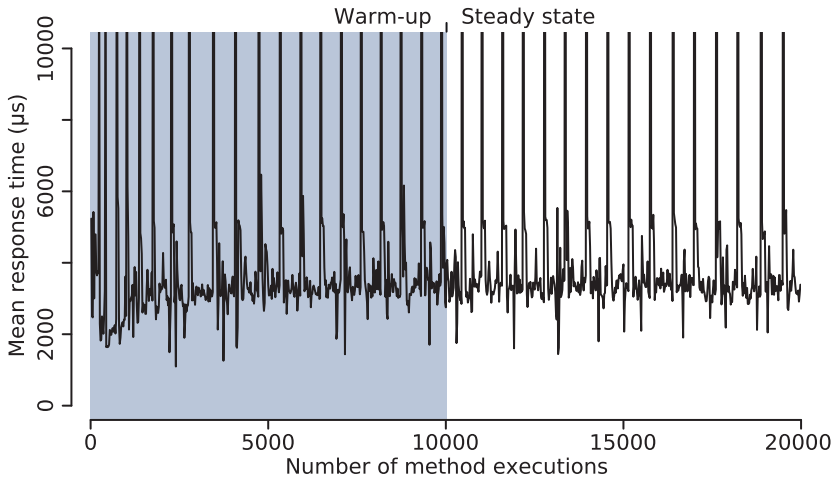


Figure 14.1. Timeseries diagram for Kieker instrumented with Kicker

A preliminary experiment with our setup suggests that the performed warm-up period is sufficiently long. This is also evident by the timeseries diagram in Figure 14.1. Although the results remain chaotic, a stable trend is visible. The regular spikes in the mean response time are caused by garbage collection activity. A detailed analysis reveals that in our meta-monitoring experiments, each garbage collection takes about half a second of runtime. Furthermore, JIT activity still has some influence on the runtime, as the experimental duration is not sufficient to complete all compilations. However, as mentioned above, a preliminary experiment suggest that longer runtimes have only minimal influence on our results.

14.2 Analyzing our Results with Kieker

We employ the analysis tools provided by Kieker to analyze the monitoring logs generated by our instrumentation of Kieker. Specifically, we use the trace analysis tool to generate dependency graphs of a subset of our monitored traces.

14. Meta-Monitoring of Kieker

A dependency graph provides an aggregated view upon a set of traces [van Hoorn et al. 2009b]. The nodes in this graph represent operations or methods of our monitored system. Methods belonging to the same class are further aggregated into a parent node, representing their class. Directed edges in the dependency graph represent method calls. These edges are annotated with the number of calls within the set of traces. Finally, special nodes exist, e. g., an entry node, representing the start of the dependency graph.

An example of such an dependency graph is presented in Figure 14.2. It is based upon our meta-monitoring experiment. In this dependency graph, we present a slightly simplified representation of a monitoring trace that is caused by monitoring a set of ten recursive method calls with Kieker. Although the annotated numbers of calls correspond to a single trace, the mean and median response times accompanying each node are calculated from the set of all traces within our steady state. Note that the annotated response times of the nodes include the execution times of all nodes that lie within the same trace on a deeper stack depth.

Starting with the Entry node, the `monitoredMethod()` is called, resulting in a total of ten recursive calls. Each execution of the method causes calls to the `triggerBefore()` and `triggerAfter()` parts of the employed monitoring probe. Within the probe, several additional calls are performed, for instance, `isMonitoringEnabled()` checks whether the monitoring framework is enabled or disabled. Besides `getTrace()`, these would be the only calls executed, when benchmarking a disabled monitoring framework (to determine *I*). With the exception of the final call to `newMonitoringRecord()` contained in `AsyncFsWriter`, all further calls are usually part of collecting monitoring data (*C*). However, the majority of work spent collecting this data is part of the `triggerBefore()` and `triggerAfter()` parts of the probe. Finally, `newMonitoringRecord()` of `AsyncFsWriter` is part of writing the monitoring data (*W*). Here, the collected data is inserted into the queue to exchange it with the active writer thread. A possible waiting for free space in this queue is the major cause of overhead due to writing.

Concerning the development of Kieker, this meta-monitoring experiment provides very interesting results. For instance, for a total of ten monitored method calls, `isMonitoringEnabled()` is executed 31 times. Similarly,

14.2. Analyzing our Results with Kieker

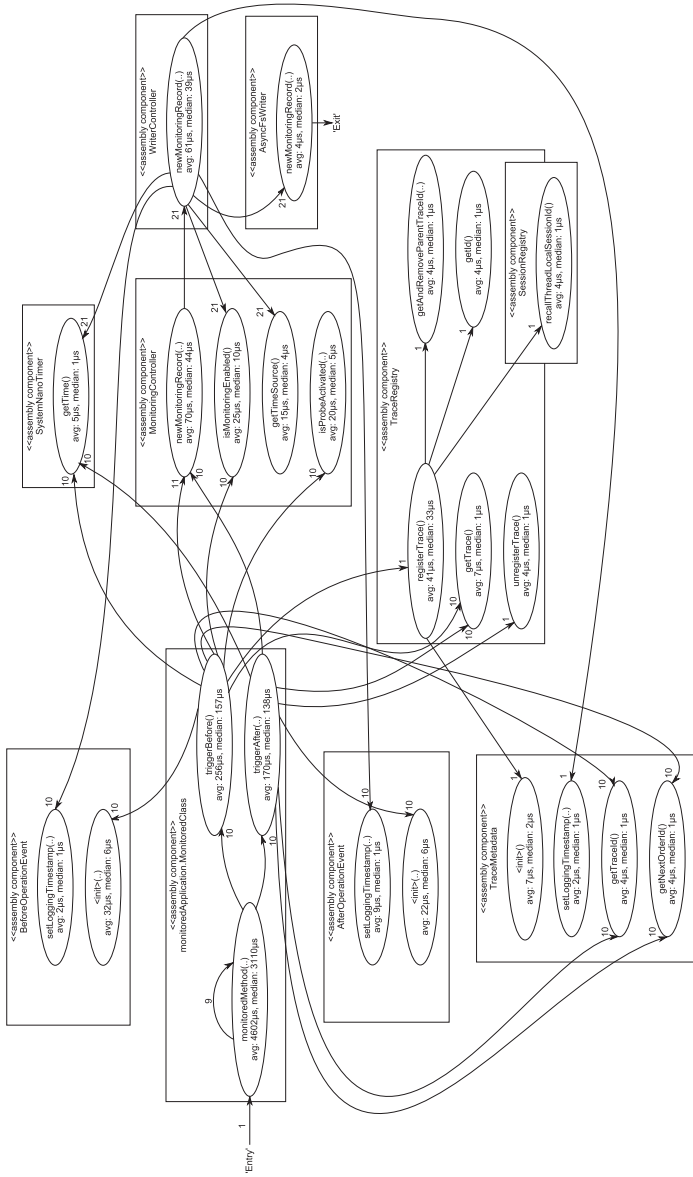


Figure 14.2. Dependency graph of monitoring a method call with Kieker

14. Meta-Monitoring of Kieker

the method `getTimeSource()` could probably be called once at the initialization of the monitoring framework instead of with each record that is written. Furthermore, several of the measured response times warrant an additional investigation. For instance, `isMonitoringEnabled()` has a rather long execution time for essentially checking a boolean variable. Another candidate for a more detailed analysis and optimization might be the `WriterController`.

The main goal of this meta-monitoring experiment is to further validate our previous micro-benchmarks performed with `MooBench`. As an initial analysis of the results, we can cluster the response times of method calls into groups associated with instrumentation (*I*), collecting data (*C*), and writing data (*W*). For instance, the instrumentation cluster contains the response times of calls to `isMonitoringEnabled()` originating in `triggerBefore()` and the response times of calls to `getTrace()` originating in `triggerAfter()`. Similarly, the writing cluster only contains the response times of the final `newMonitoringRecord()` method. The data collection cluster contains the response times of `triggerBefore()` and `triggerAfter()`, thus containing all other response times. We then subtract the response times of the other two clusters to estimate the total time.

This simple clustering results in an average response time overhead for the instrumentation of $320\ \mu\text{s}$ with a median of $110\ \mu\text{s}$. The respective values for collecting and writing are as follows: average/median response time collecting $3856\ \mu\text{s}/2798\ \mu\text{s}$ and average/median response time writing $84\ \mu\text{s}/42\ \mu\text{s}$. According to these measurements, the main cause of monitoring overhead lies in collecting the monitoring data. This result is similar to our findings in the tuning experiments with Kieker (see Section 11.5).

Due to the low workload in the meta-monitoring experiments (only 20,000 repetitions), the buffer between the monitored application and the writer thread never blocks, similar to experiments PT3 and PT4. In these earlier experiments, we have determined collecting monitoring data as the major cause of monitoring overhead, with writing data having less influence than the instrumentation itself.

Although these results are encouraging for our `MooBench` micro-benchmark, they are only of anecdotal quality. The influence of the second monitoring framework is very high, especially due to its disk accesses and due to additional garbage collection (as evident by Figure 14.1). Further-

more, due to our complete instrumentation of Kieker, we prevent many optimizations a JIT compiler normally performs, e. g., inlining very simple methods like `isMonitoringEnabled()`. In addition, it is always hard to directly compare results of a micro-benchmark with a larger, more realistic scenario. Having said that, our measurements are replicable and additional runs produce very similar results on our infrastructure.

Additional Meta-Monitoring Experiments

We have performed additional experiments with monitoring a disabled monitoring framework and with monitoring Kieker with a disabled writer. The results of these experiments are presented in Figure 14.3. The main difference to our previous meta-monitoring experiment is the overall better performance in both cases.

In the case of a disabled Kieker monitoring framework, only one additional method call is performed inside the `triggerBefore` and `triggerAfter()` methods. This has a huge impact on our Kicker meta-monitoring framework, as fewer methods are monitored and the resulting monitoring log is smaller (100 MiB). Thus, the actual meta-monitoring overhead is greatly reduced, also resulting in a reduction of the overhead measured in Kieker.

In the case of a disabled monitoring writer, the structure of the dependency graph is identical to our first experiment. However, the performance annotated on each node is again greatly improved. The performance of the Kicker meta-monitoring framework again has a major impact on our measurement results. First, the resulting monitoring log is still smaller (730 MiB). Second, Kicker has sole access to the hard disk instead of sharing its access with Kieker. Thus, less blocking occurs inside of Kicker, resulting in less influence on Kieker.

If we directly compare the response times measured at the probe (`triggerBefore` and `triggerAfter()`) for all three experiments, i. e., the overhead Kicker determines for Kieker, we can determine that the act of writing monitoring data has the greatest influence on the monitoring overhead. This result would be fitting to most of our micro-benchmark results. However, especially in our primary experiment, the influence of Kicker on the measurements is high. Thus, the significance of these results is very limited.

14. Meta-Monitoring of Kieker

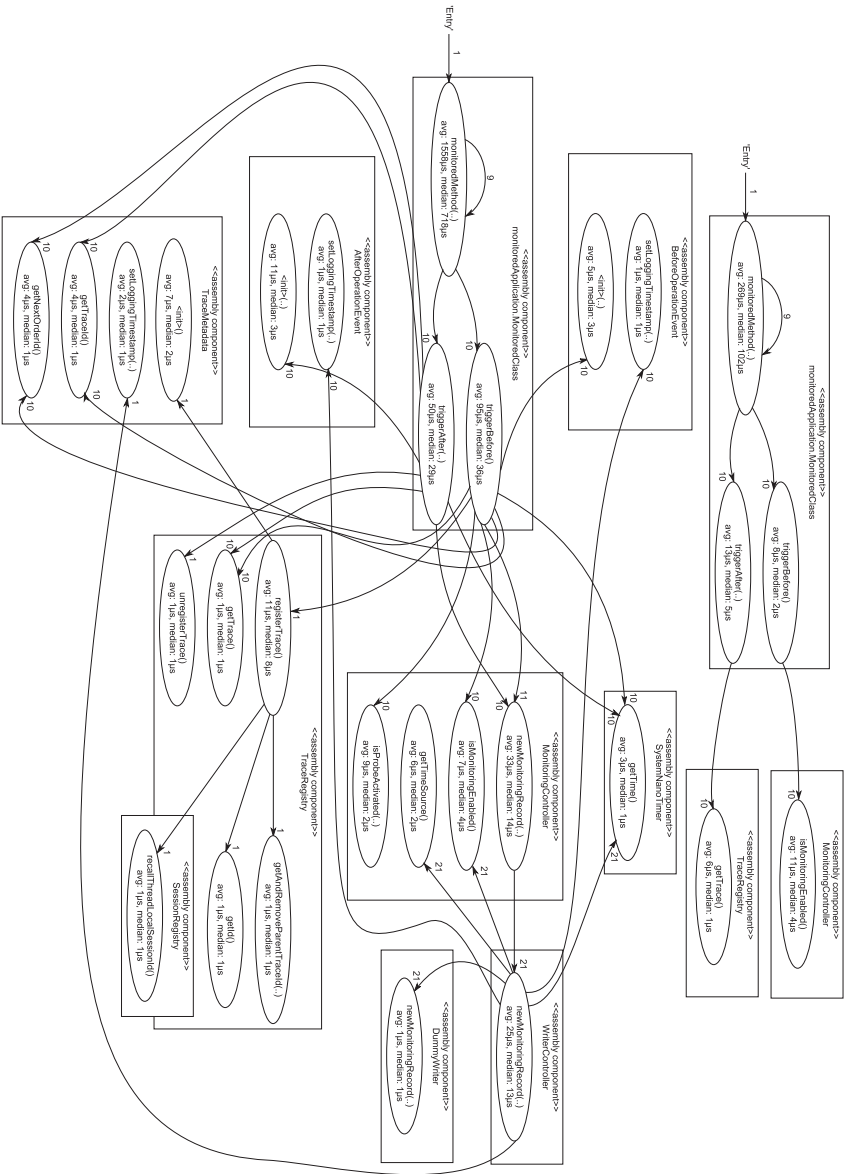


Figure 14.3. Dependency graph of monitoring deactivated Kieker or a disabled monitoring writer

Conclusions

In summary, the results of our meta-monitoring experiments provide results comparable to our micro-benchmarks. However, due to the significant additional overhead of the meta-monitoring, it is not possible to directly correlate results from micro-benchmark experiments to meta-monitoring experiments. Thus, although these results confirm our previous findings and the feasibility of MooBench for determining the monitoring overhead of Kieker, the actual results of these experiments are of limited expressiveness.

The complete sets of monitoring and meta-monitoring logs, that are the result of our experiments, are available for download [Waller 2014b]. Furthermore, our Kicker meta-monitoring framework for Kieker is available for download with MooBench. Thus, additional replications and studies of Kieker with the help of meta-monitoring are possible and highly encouraged. Especially, studies of areas not yet researched within Kieker, e.g., the analysis component, can provide insights for the further development and tuning of these areas.

14.3 Visualizing our Results with SynchroVis

Analyzing our meta-monitoring results with SynchroVis poses further challenges. For instance, besides the monitoring log of Kieker, SynchroVis requires additional input, such as a special meta-model of the static structure of the system under observation. Even when providing this model, the main feature of SynchroVis is visualizing the synchronization and blocking behavior within Java threads. Namely, synchronization with the synchronized keyword.

Within Kieker, more modern synchronization mechanisms are employed and the main feature of SynchroVis cannot be applied. All other visualization features are also a part of the ExplorViz visualization (see below). Thus, we only present a simple example of the 3D city visualization of SynchroVis in Figure 14.4. Refer to the next subsection for possible additional insights when employing these kinds of visualizations for performance evaluations.

14. Meta-Monitoring of Kieker

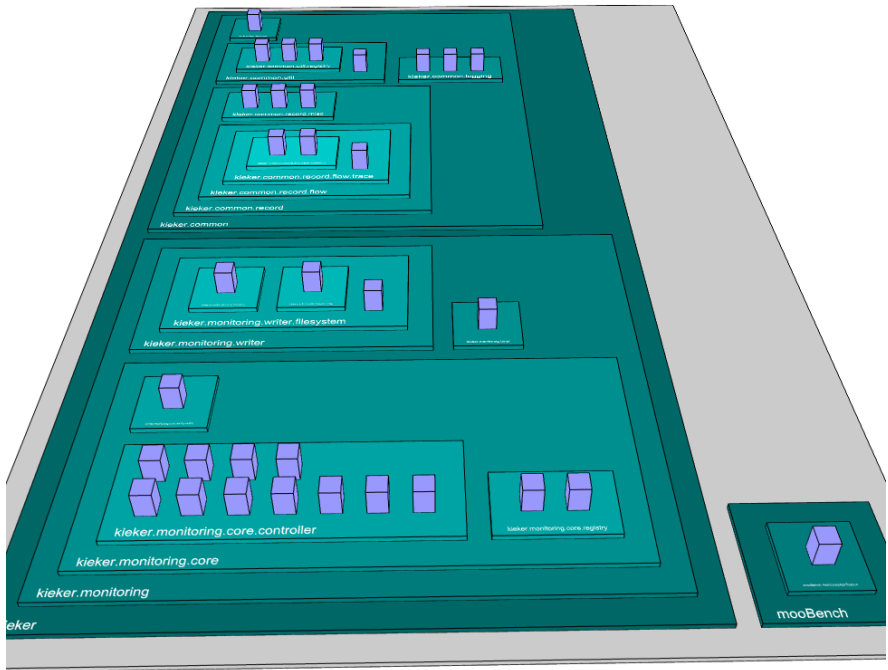


Figure 14.4. 3D city visualization of Kieker with SynchroVis [Waller et al. 2013]

14.4 Visualizing our Results with ExplorViz

An analysis of our meta-monitoring logs with the help of ExplorViz (see Section 10.3) only leads to minimal additional insights. This is caused by ExplorViz' focus on large software landscapes, while our results are based upon a single software system. Furthermore, ExplorViz has been developed to enhance the comprehension of a system, while we are interested in a performance analysis of the system. Nevertheless, a 3D city view of Kieker, as monitored with Kicker, is presented in Figure 14.5.

14.4. Visualizing our Results with ExplorViz



Figure 14.5. 3D city visualization of Kieker with ExplorViz [Fittkau et al. 2013a]

In this visualization, the green boxes represent Java packages containing the classes. The purple boxes correspond to the aggregated class nodes of our visualization with Kieker. The height of these purple boxes displays the number of active object instances within the set of visualized monitoring traces. Orange lines represent method calls from one class to another. The width of these lines represents the call frequency. Thus, calls occurring only once for the initialization or termination of monitoring are barely visible, while calls belonging to the main trace of monitoring the `monitoredMethod()` are obvious. For instance, we can see the accesses from the `monitoredMethod()` to the monitoring controller or to the used monitoring records.

In its current state, ExplorViz cannot provide any further insights into our meta-monitoring results. For instance, the load displayed at the bottom of Figure 14.5 is the load caused by reading our monitoring log from disk, instead of the load as it occurred during experiment time. Furthermore, all kinds of actual performance analyses are currently under development inside the ExplorViz project.

Related Work

In this chapter, we discuss the related work for this thesis. In accordance with our main contributions, we first discuss publications concerning *benchmark engineering methodologies* and requirements (Section 15.1). Next, we survey publications concerned with the *performance evaluation of monitoring frameworks* (see Section 15.2). Our main focus is on definitions of monitoring overhead and the measurement of it.

If I have seen further it is by standing on the shoulders of giants.

—Isaac Newton

15.1 Benchmark Engineering Methodology

In this section, we present related work to the benchmark engineering methodology that we have introduced in Chapter 7. A good overview on typical requirements for benchmarks in the literature has already been given in that chapter:

In Tables 7.1 and 7.2 on pages 104–105, we have given an overview on our eight primary and two secondary benchmark design requirements in the literature. Similarly, in Tables 7.3 and 7.4 on pages 116–117, we have dealt with our four benchmark execution requirements in the literature. Finally, in Tables 7.5 and 7.6 on pages 120–121, we have surveyed our three primary and one secondary benchmark analysis and presentation requirements in the literature.

In total, we have evaluated 50 different publications for these tables. All benchmark requirements mentioned by the authors have been collected and mapped to our set of fifteen primary and three secondary requirements. In contrast to our work, no single publication discusses all of the benchmark requirements. Furthermore, in many publications, the requirements are only handled as side notes for descriptions of presented benchmarks.

Only a few of these publications are directly concerned with general guidelines for designing benchmarks. Even fewer are concerned with actual benchmark engineering methodologies that include the design of benchmarks as well as their execution, analysis, and presentation. We detail these publications in the following:

The focus of Dongarra et al. [1987], Hinnant [1988], and Price [1989] is on comparing existing benchmark results. To aid in this process, each author gives a small set of benchmark requirements that benchmarks should adhere to.

Gray [1993] provides four important and often cited criteria for benchmark design. Similar criteria are presented by Bull et al. [2000], Sim et al. [2003] and Brebner et al. [2005]. Sim et al. [2003] also stress the idea of benchmark design as a community process.

In the thesis of Kounev [2005], several requirements for all three phases of a benchmark engineering methodology are given. Contrary to other publications, the author gives special attention to the execution of benchmarks.

15.1. Benchmark Engineering Methodology

A detailed description of several guidelines and requirements for all three phases from a practitioner's point of view is given by Huppler [2009].

Kuperberg et al. [2010] present an approach for the automatic generation of benchmarks for Java APIs. Although this approach cannot be directly transferred to benchmarks for monitoring overhead, the presented requirements for the generated benchmarks can be adapted.

In the thesis of Sachs [2011], the term benchmark engineering has been introduced. The author also states the lack of an established benchmark engineering methodology that encompasses the design, execution, analysis, and presentation of benchmarks. Additionally, several requirements for each phase are presented.

Gil et al. [2011] present several benchmark requirements for Java-based micro-benchmarks with a special emphasis of the execution phase. On the other hand, Folkerts et al. [2012] give a detailed analysis of several requirements for macro-benchmarks in the cloud. A further description of some common benchmark requirements is given by Vieira et al. [2012].

Saller et al. [2013] give a detailed description of a benchmarking methodology focussing on the design and presentation phase. Especially their generic model of a benchmark can form the basis of a detailed description in the analysis and presentation phase.

Contrary to all these mentioned publications, we have presented a encompassing benchmark engineering methodology that covers all phases of benchmark engineering: design, execution, analysis, and presentation. Furthermore, we have incorporated all fifteen primary and all three secondary requirements into our methodology.

We will not discuss publications detailing further benchmarks as related work. Several of these publications are already mentioned in the Tables 7.1–7.6 in Chapter 7. The actual details of the benchmarks, besides their requirements for a benchmark engineering methodology, are not within the scope of this chapter.

15. Related Work

15.2 Measuring Monitoring Overhead

To the best of our knowledge, there are currently no benchmarks that are specifically designed to measure or benchmark software monitoring tools and frameworks. Thus, we present publications that determine the monitoring overhead or perturbation for specific frameworks or tools. The list of publications is not restricted to benchmarks of monitoring tools, but also includes several profilers (see Section 4.1).

In order to enhance clarity, we group the publications according to similarity: First, we present a short overview on publications that only perform a basic analysis of monitoring overhead (Section 15.2.1). Next, we discuss publications containing a more detailed evaluation of causes of overhead in Section 15.2.2. In Section 15.2.3, we present monitoring tools that measure their own overhead and adapt accordingly. Further performance analyses of the Kieker monitoring framework are discussed in Section 15.2.4. Finally, we present further publications that do not fit into the other groups (Section 15.2.5).

15.2.1 Basic Analysis of Monitoring Overhead

The group of publications that perform a basic analysis of monitoring overhead employ a similar approach: An existing benchmark or application is executed twice, once in its native form and once with monitoring active.

Table 15.1. Notes for the overview on basic analyses of overhead (Table 15.2)

Notes:

-
- | | |
|-----|--|
| (a) | monitoring real-time system; evaluation of scalability of monitoring |
| (b) | evaluation of other effects of monitoring (e. g., cache misses) |
| (c) | monitoring with hardware performance counters |
| (d) | compensating profiler results with the measured overhead |
| (e) | detailed analysis of the employed benchmarks |
| (f) | evaluation of overhead for online analysis |
| (g) | comparison of four profilers: hprof, jprofile, xprof, and yourkit |
| (h) | monitoring for embedded real-time systems |
| (i) | evaluation of other effects of overhead (e. g., garbage collection) |
| (j) | instrumentation of JavaScript |

15.2. Measuring Monitoring Overhead

Table 15.2. Overview on basic analyses of monitoring overhead in the literature

Publication	Benchmark	Application	Case study	Tool (notes)
Dodd and Ravishankar [1992]			✓	HMON (a)
Patil and Fischer [1995]	✓		✓+	Shadow
Jeffery [1996]			✓	MT Icon
Ammons et al. [1997]	✓			Path Profiler (b)
Kranzlmüller et al. [1999]	✓			MPI Monitor
Gao et al. [2000, 2001]		✓		Beantracker
Sweeney et al. [2004]	✓			Perf. Explorer (c)
Mos [2004]			✓	COMPAS
Chawla and Orso [2004]			✓	PTrace
Malony and Shende [2004]	✓			TAU (d)
Hauswirth et al. [2004]	✓+			Vertical profiling (e)
Moon and Chang [2006]			✓	Thread Monitor
Parsons et al. [2006]; Parsons [2007]		✓		COMPAS
Moseley et al. [2007]	✓			Shadow Profiling
Schneider et al. [2007]	✓+			HPM
Zhu et al. [2009]	✓+		✓	Embedded Gossip
Bach et al. [2010]	✓		✓+	PIN
Goodstein et al. [2010]	✓+			Butterfly (f)
Vlachos et al. [2010]	✓+			ParaLog
AppDynamics [2010]			✓	AppDynamics Lite
Mytkowicz et al. [2010]	✓			four profilers (g)
Zhao et al. [2010]	✓+			PiPA
Bonakdarpour and Fischmeister [2011]			✓	Time-aware (h)
Brüseke et al. [2013]			✓	Blame Analysis
Marek et al. [2013]	✓			ShadowVM (i)
Zhou et al. [2014]			✓	MTracer
Lavoie et al. [2014]			✓+	Photon (j)
$\sum 27$	15	3	12	

15. Related Work

The results of both runs are compared to determine the monitoring overhead. Thus, these experiments are similar to our experiments described in Chapter 13. However, no further analyses are performed to validate these results or to determine the causes of monitoring overhead.

An overview on the employed technique by each publications of this group is given in Tables 15.1 and 15.2. Each employed technique is marked with a check mark. If, for instance, multiple benchmarks are employed, the check mark is annotated with a plus sign.

An entry of *benchmark* refers to the use of an usually already existing benchmark or benchmark suite, e. g., a SPEC benchmark or the DaCapo suite. These evaluations are similar to our experiments presented in Section 13.2 or Section 13.3.

Application corresponds to mostly small applications that are combined with a workload driver to act as a benchmark. A typical example is the Pet Store with JMeter, but more complex applications are employed as well. These evaluations are similar to our experiments presented in Section 13.1.

Finally, *case study* refers to employing existing applications or scenarios that are used to benchmark the monitoring. For instance, a test suite gets executed and its execution time is measured. Depending on the actually used application or scenario, the resulting benchmark is either a small macro-benchmark or even a micro-benchmark. This is usually the case in publications describing specialized monitoring solutions that are written for a single scenario or application, e. g., specific embedded systems.

Contrary to most of the publications mentioned in Table 15.2, we have evaluated multiple scenarios for our monitoring frameworks under test. Furthermore, we have performed additional experiments to validate our results. However, our major unique feature compared to all publications is a detailed analysis of possible causes of monitoring overhead.

15.2.2 Analyzing Causes of Monitoring Overhead

In this section, we present a group of publications that discusses or measures different causes of monitoring overhead. In most cases, the actually employed technique is similar to the ones presented in the previous group of publications. Thus, all of these publications are more closely related to

15.2. Measuring Monitoring Overhead

our approach. The publications are discussed below in chronological order. The first publication detailing our own approach of analyzing causes of monitoring overhead is by van Hoorn et al. [2009b].

Anderson et al. [1997] describe a hardware performance counter based profiling tool that gets performance evaluated with the help of several existing benchmark collections. They further analyze the cause of monitoring overhead by splitting it into two components: First, the time spent handling the actual performance counter event. Second, the time spent collecting and aggregating all incoming events. Thus, this approach is similar to our experiments to determine the cost of online analysis (Section 11.6). Contrary to our approach, the investigation of the costs of both components is done by adding additional measurement code to the tool itself, while we deactivate parts of our tools to perform the analysis.

Bellavista et al. [2002] evaluate the monitoring overhead of the MAPI tool. Their evaluation is performed using an otherwise unspecified Java benchmarking application in several different hardware environments. Similar to the previous authors, they have split the cost of monitoring overhead into a monitoring and an analysis (object tracing) part. However, similar to our own approach, the measurements are performed by deactivating the analysis and comparing the results. But, no further analysis of the possible causes of the remaining monitoring overhead part is performed.

[Barham et al. 2004] benchmark the monitoring overhead of the Magpie monitoring tool. Although the authors do not investigate the causes of overhead, they break down the additional cost of a performed online analysis into three sub-components, similar to our experiments in Section 11.6.

The Javana system has been evaluated with several benchmark collections by Maebe et al. [2006]. The authors identify four possible causes of monitoring overhead for their framework: The overhead (1) of the used binary instrumentation that runs below the JVM, (2) of linking high-level Java constructs to low-level binary code, (3) of similarly linking memory accesses, and (4) of actually executing the instrumentation code. Similar to our own approach, these causes are quantified by deactivating parts of Javana and then comparing the results of the performed benchmark. Contrary to our own approach, the identified causes of monitoring overhead are specific for Javana and not transferable to further monitoring frameworks.

15. Related Work

Bulej and Bureš [2006] and Bulej [2007] discuss differences in overhead for a disabled monitoring tool compared to an enabled one. However, no actual measurements or benchmarks are performed.

Wallace and Hazelwood [2007] describe their SuperPin approach to use massive parallelization to reduce the overhead of monitoring. In their performance evaluation using an existing benchmark suite, the authors also perform a breakdown of the monitoring overhead into four components. However, this breakdown is specific for their parallel implementation and no further analysis of the part actually doing the monitoring exists.

Mohror and Karavanic [2007, 2012] investigate the overhead of profiling or monitoring tools for high-performance computing and high-end parallel systems. They split the overhead into two distinct causes: (1) collecting the data and (2) writing the data to disk. Similar to our own experiments, they repeat three benchmark runs with different existing benchmarks to determine the individual portion by calculating the differences between each run. Furthermore, the authors have employed this division of monitoring overhead to evaluate different configurations for the tools under test. In contrast to the authors, we have further split the data collection into an instrumentation and a collection part to enable the investigation of different instrumentation technologies. Furthermore, we provide a more general approach that can be employed for different monitoring tools, while the authors are focused on the evaluation of their own tools.

The performance analysis tool Vampir for high-performance computing has been evaluated with the help of a sample application and a benchmark collection by Müller et al. [2007]. Although different causes of monitoring overhead are discussed (initialization at start-up, collecting data per event, writing data, and post-processing), no detailed analysis of the all four given causes is performed. Furthermore, depending on the executed sub-benchmarks, the authors' approach is not capable of measuring the given portions of overhead.

Ha et al. [2009] evaluate the overhead of a concurrent online analysis. Similar to our experiment in Section 11.6, they perform a breakdown into several analysis components and measure the performance overhead of each component with several benchmarks. However, the authors perform no further analysis of the causes of overhead for the monitoring component.

15.2. Measuring Monitoring Overhead

Sheng et al. [2011] investigate a race detection tool for multi-core systems with the help of a number of sample applications and custom benchmarks. Although the authors provide a breakdown of the causes of monitoring overhead for their tool, these causes are specific for their implementation and for race detection. In contrast, our approach provides a more general partition of monitoring overhead.

Although Kanstrén et al. [2011] provide a discussion of possible causes of monitoring overhead (e. g., additional network latency or access of system resources), no actual evaluation of these causes or the monitoring tool is performed.

Trümper et al. [2012] evaluate the memory and time overhead of a monitoring tool for embedded systems using a case study system. The authors compare an uninstrumented system to a system containing a deactivated or activated monitoring tool. Similarly, partial instrumentation is compared to full instrumentation of the case study system. However, no further evaluations of the causes of monitoring overhead in the active monitoring solution are performed.

The Senseo tool is evaluated with the help of a benchmark collection by Röthlisberger et al. [2012]. The authors break the total monitoring overhead down into three parts: (1) creating a calling context tree for their analysis, (2) collection of dynamic information, and (3) serialization and transmission of collected data. Similar to other approaches, this breakdown is specific for the monitoring tool under test. Furthermore, no additional investigation of the causes of data collection or of the influence of the chosen instrumentation is performed.

15.2.3 Adapting Monitoring to the Overhead

The publications in this group are concerned with adaptive monitoring solutions. More specifically, the presented monitoring tools measure their own overhead during their runtime and adapt their instrumentation accordingly. Thus, these publications usually are only marginally related to our approach. Contrary to our approach, the publications perform no analysis of the causes of monitoring overhead (similar to the first group of publications).

15. Related Work

Callanan et al. [2008] and Huang et al. [2012] describe their High-Confidence Software Monitoring approach as well as their software monitoring with controllable overhead approach. In both cases, monitoring probes are enabled or disabled to achieve a certain target monitoring overhead. The approach is evaluated using case study systems and small benchmark applications.

The DYPER monitoring framework by Reiss [2008] determines three portions of its own overhead: monitoring, analysis, and reporting. Then, depending on its configuration, it assigns priorities and time budgets to each portion. This approach is evaluated using a case study system. However, besides these three rough portions, no further analysis of the overhead causes is performed.

Arnold et al. [2011] utilize a specialized JVM to monitor running systems. Similar to the previous approaches, their Quality Virtual Machine can be configured with a target monitoring overhead. Depending on the portion of time spent executing monitoring code compared to application or JVM code, the sampling-based monitoring instrumentation can be adjusted. Their approach is evaluated using two benchmark collections. Furthermore, the authors investigate the impact of the reduced monitoring coverage caused by reducing the monitoring overhead.

Katsaros et al. [2012] utilize a slightly different strategy for the cloud monitoring approach. Instead of directly measuring the caused monitoring overhead, an indirect measuring technique is employed: Their case study system is a distributed video converter. Dropped frames are detected in this system and the monitoring is adjusted accordingly.

15.2.4 Performance Evaluations of Kieker

Our next group of related publications contains further performance evaluations of the Kieker monitoring framework that have been performed independently of our approach.

The first performance evaluations have been performed in the diploma thesis of Focke [2006], which has laid the groundwork for the development of Kieker. Several case study systems have been used to investigate the impact of including a monitoring tool into different systems. However, no

15.2. Measuring Monitoring Overhead

further analysis of different workloads or causes of monitoring overhead has been performed.

Rohr et al. [2008] have performed an evaluation with the help of the Pet Store. The experimental setup is similar to the experiment described in Section 13.1. However, contrary to our approach, no further investigation of overhead components is done.

Okanović et al. [2013] have evaluated a modified version of Kieker for the DProf tool. Although changes and performance tunings to reduce the monitoring overhead are mentioned, these have not been evaluated. The single performance test is based upon a custom micro-benchmark to compare the impact of two different instrumentation techniques for DProf. However, no further investigation of the impact of these techniques on the different causes of overhead has been performed. Especially considering that in a single benchmark run out of nine the performance benefits have been reversed, a further validation of the benchmark with additional experiments would have been beneficial.

Eichelberger and Schmid [2012, 2014] have employed the SPECjvm[®]2008 benchmark to compare the performance of their SPASS-meter tool to other monitoring or profiling tools, among others Kieker. We have already discussed these publications in the context of our evaluation of SPASS-meter in Section 12.2 and in the context of our experiments with the used benchmark (see Section 13.2). To summarize these discussions: The used workload is rather low compared to our own experiments. However, using a low workload with MooBench, we have been able to measure a similar performance when comparing Kieker to SPASS-meter. Although the performed evaluation is quite detailed, no further investigations into the causes of monitoring overhead have been performed. Furthermore, no additional benchmarks or workloads have been used to validate the findings.

15.2.5 Further Evaluations of Monitoring Overhead

In this final group we have gathered the remaining publications that do not fit into the other four groups. Thus, in the following, we briefly discuss further publications concerned with measuring monitoring overhead.

15. Related Work

Malony et al. [1992] have evaluated the monitoring overhead of a simple performance instrumentation for their target system with the help of a small micro-benchmark. Then, they utilize their measurements to predict the overhead when instrumenting macro-benchmarks. Thus, their approach is similar to our validation of micro-benchmark results with macro-benchmarks. However, their approach of predicting the exact performance cost is limited to rather simple environments. The optimizations performed by modern processors or environments such as the JVM limit the general applicability.

Instead of measuring the overhead of program instrumentation, Kumar et al. [2005] propose an instrumentation optimizer that acts similar to a Just-In-Time (JIT) compiler. The executed instrumentation is analyzed during the runtime and certain optimizations are performed, e.g., inlining or combining multiple probes into a single one. Then, the authors compare the optimized instrumentation of several profilers with the original one using benchmarks. However, no comparison to an instrumented benchmark is done.

Rather than evaluating the overhead of a profiler or monitoring tool, Avgustinov et al. [2005b, 2007] investigate optimizations for the overhead caused by AspectJ [AspectJ] using the *abc* compiler. Similar further optimizations are performed and benchmarked by Bodden et al. [2007].

He and Zhai [2011] perform similar experiments to the ones described in Section 11.4. They separate the monitoring logic from the monitored system and assign each an own core. Furthermore, they split the measured monitoring overhead for their solution into up to three causes: actual monitoring, communication, and additional local calculations. However, no details on measuring these portions are given.

Diwan et al. [2011] are focussed on evaluating the cost of analyzing the results of monitoring. Contrary to our own evaluation in Section 11.6, the authors are not concerned with live analysis but rather with offline analysis. Thus, they do not determine the influence an analysis has on monitoring but rather the flat cost of performing this analysis.

Meng and Liu [2013] describe a cloud-based monitoring approach for monitoring as a service. Thus, their evaluation is mostly focussed on the communication overhead introduced by their monitoring solution. They

15.2. Measuring Monitoring Overhead

determine the influence of several environmental parameters, such as workload or number of systems, on their solution in several experiments.

Similarly to the earlier presented publication on optimizing AspectJ, Marek et al. [2015] and Sarimbekov et al. [2014] are focussed on the overhead induced by employing DiSL instead of other instrumentation technologies. The authors change the employed instrumentation of several existing monitoring or profiling tools and compare the resulting performance with the original one.

Vierhauser et al. [2014] measure the performance of their monitoring framework for systems-of-systems. However, they do not provide a baseline for an unmonitored system. Thus, although the authors demonstrate the scalability of their approach, they do not measure the actual overhead of their framework.

Goers and Popma [2014] measure the achievable throughput of using the LOG4j logging framework. Although the framework is not primarily used for monitoring or profiling purposes, it can easily be adapted for such tasks (see Section 4.3.5). However, no actual evaluation of monitoring overhead is performed.

Of special note is an upcoming publication by Wert et al. [2015]. The authors utilize MooBench to evaluate and compare the performance of their monitoring tool to an uninstrumented system and to Kieker. In addition, they validate these results with a macro-benchmark and a case study system. Preliminary results on these experiments have already been published by Flaig [2014] and Schulz et al. [2014].

Part IV

**Conclusions and Future
Work**

Conclusions and Future Work

In this chapter, we summarize the findings and contributions of this thesis and of our experimental evaluation (Section 16.1). In addition, we provide an outlook on possible future work in Section 16.2.

Everything not saved will be lost.

—Nintendo “Quit Screen” message

16.1 Conclusions

In this thesis, we have presented our approach for measuring the monitoring overhead for application-level monitoring frameworks with the help of benchmarks. In addition, we determine and quantify common causes of monitoring overhead. To realize our measurement with a benchmark, we provide a general benchmark engineering methodology. This methodology is applied to develop and execute the MooBench micro-benchmark and to analyze and present its results.

This approach provides the answer to our main research question (*RQ*) that we have posed in Section 5.2: *What is the performance influence an application-level monitoring framework has on the monitored system?* In the following, we detail our contributions and their evaluation as well as the execution of our research plan and the answers to our research questions.

Causes of Monitoring Overhead

This first main contribution of our thesis is a common definition of monitoring overhead with respect to the response times of monitored applications. The detailed description of this contribution is provided in Chapter 6. Our analysis has led to three common causes of monitoring overhead: (I) *instrumentation* of the system under monitoring, (C) *collecting* monitoring data within the system, and (W) *writing* the collected data into a monitoring log or stream.

This contribution provides the answer to our research question *RQ1*: *What are the causes for observed changes in the response time of a monitored method?* We have evaluated this contribution in Chapters 11 and 12 using multiple *lab experiments* with benchmarks. With the help of these experiments, we have demonstrated the presence and measurability of these three overhead causes within the Kieker, ExplorViz, inspectIT, and SPASS-meter monitoring frameworks.

Furthermore, in Section 11.6, we have expanded upon our general approach of three common causes of monitoring overhead by introducing additional causes of monitoring overhead due to the live analysis of monitoring data concurrent to the act of collecting the data. Thus, we have

demonstrated the applicability of our approach to quantify the causes of monitoring overhead as well as its extendibility for further scenarios such as live analysis of monitoring data.

Benchmark Engineering Methodology

The second main contribution of this thesis is providing an encompassing benchmark engineering methodology. The detailed description of this methodology is given in Chapter 7. In its essence, we have split the benchmark engineering process into three phases: (1) The first phase is the *design and the implementation of the benchmark*. (2) The second phase is the *execution of the benchmark*. (3) The third phase is the *analysis and presentation of the benchmark results*. For each of these phases, we have given a set of requirements that a benchmark should adhere to.

This benchmark engineering methodology provides the answer to our research question *RQ2.1: How to develop a benchmark?* We have evaluated our methodology in Chapter 7 using an extensive *literature review* of 50 different publications concerned with benchmarking. Furthermore, in Chapter 8, we have provided a *proof-of-concept implementation* of the MooBench micro-benchmark that has employed our benchmark engineering methodology for. Thus, we have demonstrated the feasibility of our approach.

Benchmarks of Monitoring Overhead & Evaluation of MooBench

The third main contribution of this thesis is the *MooBench micro-benchmark*, which is used to measure and quantify the previously established portions of monitoring overhead in an application-level monitoring framework. The details of the benchmark are presented in Chapter 8. It has been created in accordance to our benchmark engineering methodology and enables benchmark experiments for the investigation of monitoring overhead and its causes.

In addition to the use of our micro-benchmark, we have proposed the additional use of *established macro-benchmarks*, such as the Pet Store, the SPECjvm[®]2008 or the SPECjbb[®]2013. We have presented an adaptation of these benchmarks to measure monitoring overhead in Chapter 9.

16. Conclusions and Future Work

Finally, we have proposed a *meta-monitoring* approach of using the monitoring framework on itself in Chapter 10. Thus, we are able to further investigate the monitoring overhead and its causes with the help of the analyses provided by the monitoring framework.

These three approaches provide the answer to our research question *RQ2.2: How to measure and quantify monitoring overhead using benchmarks?* The focus of our evaluation lies on the employment of our MooBench micro-benchmark. The other two approaches are used to validate our results. The detailed description of our evaluations has been presented in Part III (Chapters 11 – 14). In the following, we present the application of the GQM approach for our GQM goal G that has been given in Table 5.1.

The first metric (M1) is concerned with the monitoring frameworks that MooBench is capable of benchmarking. Due to the nature of the metric, it is not possible to provide a final value. However, we have evaluated our MooBench micro-benchmark with with four frameworks: Kieker, ExplorViz, inspectIT, and SPASS-meter. In all cases, no changes have been necessary for a basic analysis of the monitoring overhead (M2). In the case of a detailed analysis of overhead causes, only minimal to no changes have been necessary (M3). For instance, in the case of Kieker, the modular nature of the framework has enabled all experiments. In the cases of inspectIT or SPASS-meter, minimal changes to the code have been made to deactivate parts of the framework. Refer to Chapter 12 for details. Thus, Q1 can be answered with these four monitoring frameworks. However, further application-level monitoring frameworks typically provide similar possibilities.

The required run-time for our benchmark experiments (M4) depends on the actual benchmarking scenario and the respectively required warm-up period and the employed number of repetitions. We have provided the used configuration with each experiment in Chapters 11 and 12. Typically, a run of benchmark experiments with MooBench can take multiple hours due to the number repetitions. Thus, Q2 can be answered with minimal initial effort to prepare the benchmarks. However, a rather long experiment run-time with several repetitions is required to provide representative results.

A selection of different scenarios for our benchmark (M5) has been presented in Chapters 11 and 12: For instance, we have compared different versions of the same framework with each other (Section 11.2), we have

evaluated the scalability of monitoring overhead with increasing workloads (Section 11.3), and we have compared the impact of different environments on the monitoring overhead (Section 11.4). Refer to the mentioned chapters for a detailed overview on our scenarios. Within these scenarios, we have also demonstrated the configurability and adaptability of our benchmark (M6). Similarly, we have demonstrated the reproducibility of our benchmark results (M7) by comparing the results of experiments with slightly different environments in our scenarios. Furthermore, we have repeated each experiment multiple times with stable results. Thus, with respect to our presented scenarios and monitoring frameworks, Q3 can be answered with yes.

We have compared our micro-benchmark results to the results of three macro-benchmarks in Chapter 13 for the metric M8. While the Pet Store and the SPEC_{vm}[®]2008 benchmark have only confirmed parts of our results, the detailed analysis with the more advanced SPEC_{bb}[®]2013 benchmark has confirmed our division of monitoring overhead and the relative portions of overhead for each cause. In addition, we have compared our findings with MooBench to meta-monitoring experiments with Kieker in Chapter 14. Although these experiments confirm the feasibility of MooBench for determining the monitoring overhead of Kieker, the actual results of these experiments are of limited expressiveness. Thus, Q4 can also be answered with yes.

In summary, we have demonstrated the capabilities of our MooBench micro-benchmark to measure and quantify the performance overhead of application-level monitoring frameworks.

Performance Evaluations of Monitoring Frameworks

The final contribution of this thesis is an extensive evaluation of the Kieker, ExplorViz, inspectIT, and SPASS-meter monitoring frameworks. These evaluations have led to new insights into the inner workings of these frameworks and have already steered parts of their further development.

16.2 Future Work

As mentioned in the previous section, we have evaluated four application-level monitoring frameworks in Chapters 11 and 12. However, our MooBench micro-benchmark is not restricted to these tools. Thus, an interesting field of further studies is the evaluation of additional monitoring frameworks. Primarily, the focus would be on application-level tools. However, evaluations of low-level tools could also provide valuable insights.

The focus of this thesis has been on Java-based tools and frameworks. Although the current implementation of MooBench is limited to Java, the concepts can easily be transferred to further environments and programming languages.

We have evaluated our micro-benchmark with the help of comparisons to existing and established macro-benchmarks in Chapter 13. Further experiments and validations with additional benchmarks, such as the CoCoME or DaCapo benchmarks (see Section 9.1), can provide further confidence to our results.

Of special interest is the inclusion of MooBench into continuous integration environments as described in Section 11.7. This currently rather rudimentary inclusion already provides huge benefits to the Kieker project by facilitating an early detection of performance regressions. However, the currently presented visualization of the analyses is very simple. Thus, additional work can be put into providing more meaningful visualizations of our benchmark results. Furthermore, currently no automatical notifications are performed in the case of performance anomalies. So, future work could include the development and integration of automatical performance anomaly detection mechanisms into our inclusion of MooBench.

As mentioned before, we have generalized our division of causes of monitoring overhead for live analysis in Section 11.6. This generalization can be further evaluated with different analysis scenarios and additional monitoring and analysis frameworks. Furthermore, our benchmarking approach can be adapted to quantify costs of analysis frameworks. For instance, an adapted version of MooBench will be employed for extensive performance evaluations of the TeeTime pipes & filters framework [Wulf et al. 2014].

Benchmarking is considered to be a community effort. Thus, additional steps are required to establish and further develop our benchmark within the larger monitoring and performance evaluation community. First steps are provided by additional researchers or practitioners that employ our benchmark independently of this thesis, for instance, the usage of MooBench in the further development of inspectIT (see Section 12.1) or the performance evaluations performed by Flaig [2014], Schulz et al. [2014] and Wert et al. [2015].

Finally, of particular interest are replications and validations of our findings and experiments:

Replication and Validation of our Experiments

The developed MooBench micro-benchmark is provided as open source software [Apache License, Version 2.0] with each release of the Kieker framework and on the Kieker home page.¹ In addition, a snapshot of the MooBench git repository has been archived at ZENODO [Waller 2014a]. Furthermore, the results of all experiments, including the raw result data, the results of additional experiments, and the configuration of our benchmarks, are available for download. These data packages have been published in several data publications [Waller and Hasselbring 2012b; 2013, b; Fittkau et al. 2013c; Waller et al. 2014b; Waller 2014b; Waller et al. 2015a]. Thus, we facilitate further validations and replications of our experiments.

¹<http://kieker-monitoring.net/MooBench/>

List of Figures

2.1	A common partitioning of the response time	21
2.2	Response time, throughput, and resource utilization	23
2.3	A simplified domain model for a converged SPE process	25
2.4	Integration of SPE in the software development process	27
3.1	Different kinds of performance testing	34
4.1	The monitoring process	46
4.2	Monitoring the abstraction layers of a software system	49
4.3	Screenshot of the Windows Process Explorer tool	53
4.4	Screenshot of the Java VisualVM tool	56
4.5	Aspect-Oriented Programming (AOP)	61
4.6	A top-level view on the Kieker framework architecture	70
6.1	UML sequence diagram for monitoring overhead	91
6.2	UML sequence diagram for monitoring overhead	92
7.1	Classification of benchmark engineering	101
7.2	The three phases of a benchmark engineering methodology	102
7.3	A generic model of a benchmark	111
7.4	Reproducibility spectrum for scientific publications	125
8.1	The general architecture of the benchmark setup	132
8.2	UML activity diagram for a typical External Controller	134
8.3	UML activity diagram for the Benchmark Driver	138
8.4	Exemplary time series diagram of measured timings	149
9.1	Workload on a typical enterprise web site	156
9.2	Architecture of the SPECjbb2013 benchmark	160

List of Figures

10.1	Schematic view of our SynchroVis approach	168
10.2	Visualizing the static structure of a large software	169
10.3	Visualizing the dining philosophers problem	171
10.4	Visualizing a deadlock in the dining philosophers problem	172
10.5	Example of the ExplorViz landscape level	174
10.6	Example of the ExplorViz system level	175
11.1	Exemplary time series diagram of measured timings	184
11.2	Performance comparison of twelve different Kieker versions	186
11.3	Performance comparison of twelve different Kieker versions	187
11.4	Performance comparison of five different Kieker versions	189
11.5	Exemplary time series diagram of measured timings	193
11.6	Linear Increase of Monitoring Overhead	193
11.7	Single-Threaded Monitoring Overhead	195
11.8	A comparison of different multi-core architectures	199
11.9	Overview of base results in response time	202
11.10	Overview of PT1 results in response time	204
11.11	Overview of the tuning results in response time	209
11.12	Comparison of analysis response times	213
11.13	First approach of including MooBench into Jenkins	216
11.14	MooBench within Kieker's Jenkins	217
11.15	Post-mortem analysis of a performance regression	219
12.1	Overview on the inspectIT components	222
12.2	Time series diagram for an experiment with inspectIT	226
12.3	Response time for an experiment with inspectIT	228
12.4	Comparison of different inspectIT probes	231
12.5	Comparison of different workloads with inspectIT	232
12.6	inspectIT: local and remote CMR	234
12.7	Architecture of SPASS-meter	236
12.8	Time series diagram for experiments with SPASS-meter	238
12.9	Median response time for a SPASS-meter experiment	240
12.10	Low workload response time for SPASS-meter experiments	242
12.11	Response time for the SPASS-meter TCP experiment	243
12.12	Response time for the third SPASS-meter experiment	245

List of Figures

13.1	Benchmark results for our JPetStore macro-benchmark	252
13.2	Benchmark results of the SPECjvm2008	254
13.3	Benchmark results of the SPECjbb2013	259
13.4	Response times for Kieker release 1.9 with MooBench	261
13.5	Normalized comparison of monitoring overhead	261
13.6	Benchmark results of the SPECjEnterprise2010	264
14.1	Timeseries diagram for Kieker instrumented with Kicker	269
14.2	Dependency graph of monitoring a method call with Kieker	271
14.3	Dependency graph of monitoring a method call with Kieker	274
14.4	3D city visualization of Kieker with SynchroVis	276
14.5	3D city visualization of Kieker with ExplorViz	277

List of Tables

4.1	Differences between profiling tools and monitoring tools . . .	45
5.1	Definition of the GQM goal	86
7.1	Benchmark design requirements in the literature	104
7.2	Benchmark design requirements in the literature (cont.) . . .	105
7.3	Benchmark execution requirements in the literature	116
7.4	Benchmark execution requirements in the literature (cont.) . .	117
7.5	Benchmark analysis requirements in the literature	120
7.6	Benchmark analysis requirements in the literature (cont.) . . .	121
11.1	Response times for Kieker release 1.9	185
11.2	Throughput for the base evaluation	202
11.3	Throughput for PT1 (traces per second)	204
11.4	Throughput for PT2	206
11.5	Throughput for PT3	207
11.6	Throughput for PT4	209
11.7	Throughput for Kieker 1.8	212
11.8	Throughput for our high-throughput tuned Kieker version . .	212
12.1	Response times for the initial inspectIT experiment	227
12.2	Response times for the isequance probe	230
12.3	Response times for the timer probe	230
12.4	Response times for the isequance probe under high load . . .	232
12.5	Response times for the isequance probe with a remote CMR . .	234
12.6	Response times for the initial SPASS-meter experiment	240
12.7	Response times for the SPASS-meter TCP experiment	243
12.8	Response times for the third SPASS-meter experiment	245
13.1	Benchmark results of the SPECjvm2008	254

List of Tables

13.2	Benchmark results of the SPECjbb2013	258
13.3	Benchmark results of the SPECjbb2013	259
13.4	Response times for Kieker release 1.9 with MooBench	261
13.5	Benchmark results of the SPECjEnterprise2010	264
15.1	Notes for the overview on basic analyses	282
15.2	Overview on basic analyses of monitoring overhead	283

List of Listings

4.1	Querying the CPU time of the current thread in Java with JMX	55
4.2	Manual instrumentation of Java source code	57
4.3	A Java agent for bytecode instrumentation	59
4.4	AspectJ aspect for the instrumentation of Java source code . .	63
4.5	AspectJ: simple aop.xml file for Java instrumentation	64
4.6	AspectJ pointcut for the instrumentation of Java servlets . . .	66
6.1	Simplified Java source code for a Kieker monitoring probe . .	93
8.1	Excerpt of an External Controller script	133
8.2	Required Java interface for the Monitored Application	136
8.3	Basic implementation of the MonitoredClass interface	136
8.4	Excerpt of the Benchmark Thread's run method	139
8.5	Configuration parameters for the Benchmark	140

Acronyms

ADM	Architecture-Driven Modernization
AOP	Aspect-Oriented Programming
API	Application Programming Interface
APM	Application Performance Management
APM	Application Performance Monitoring
ARM	Application Response Measurement
ASCII	American Standard Code for Information Interchange
BCEL	Byte Code Engineering Library
BPMN	Business Process Model and Notation
CERN	European Organization for Nuclear Research
CI	Confidence Interval
CLR	Common Language Runtime
CMR	Centralized Measurement Repository
CPU	Central Processing Unit
CSV	Comma-Separated Values
DARPA	Defense Advanced Research Projects Agency
DFG	Deutsche Forschungsgemeinschaft
DSL	Domain-Specific Language
EE	Enterprise Edition (of the Java platform)

Acronyms

GQM	Goal, Question, Metric
HDD	Hard Disk Drive
HPET	High Precision Event Timer
HSQLDB	Hyper SQL Database
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ID	Identifier
ISO	International Organization for Standardization
JCGM	Joint Committee for Guides in Metrology
JIT	Just-In-Time Compilation
JMS	Java Message Service
JMX	Java Management Extensions
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JSP	JavaServer Page
JSR	Java Specification Request
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
JVM TI	Java Virtual Machine Tool Interface
NIST	National Institute of Standards and Technology
OMG	Object Management Group
OS	Operating System

Acronyms

RVM	Research Virtual Machine
SLO	Service Level Objective
SMM	Software Metrics Meta-Model
SPE	Software Performance Engineering
SPEC	Standard Performance Evaluation Corporation
SPEC RG	SPEC Research Group
SPEM	Software & Systems Process Engineering Metamodel
SUM	System Under Monitoring
SUT	System Under Test
TCP	Transmission Control Protocol
TPC	Transaction Processing Performance Council
TSC	Time Stamp Counter
UML	Unified Modeling Language
VM	Virtual Machine
WDC	World Data Center
XML	eXtensible Markup Language

Bibliography

- [Adamson et al. 2007] A. Adamson, D. Dagastine, and S. Sarne. SPECjbb2005 – A year in the life of a benchmark. In: *Standard Performance Evaluation Corporation (SPEC) Benchmark Workshop*. SPEC, Jan. 2007, pages 1–4. (Cited on pages 3, 38, 105, 117, 127, and 159)
- [Almaer 2002] D. Almaer. Making a Real World PetStore. TheServerSide.com J2EE Community. Apr. 2002. URL: <http://www.theserverside.com/articles/article.tss?l=PetStore>. (Cited on page 37)
- [Alpern et al. 2005] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44.2 (Jan. 2005), pages 399–417. (Cited on page 51)
- [Ammons et al. 1997] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In: *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI 97)*. ACM, June 1997, pages 85–96. (Cited on pages 51 and 283)
- [Anderson et al. 1997] J.-A. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems* 15.4 (Nov. 1997), pages 357–390. (Cited on pages 51 and 285)
- [Apache JMeter] Apache JMeter. The Apache Software Foundation. URL: <http://jmeter.apache.org/>. (Cited on pages 29 and 155)
- [Apache License, Version 2.0] Apache License, Version 2.0. The Apache Software Foundation. URL: <http://www.apache.org/licenses/LICENSE-2.0.html>. (Cited on pages 131, 236, and 301)

Bibliography

- [AppDynamics 2010] AppDynamics. AppDynamics Lite Performance Benchmark Report. May 2010. URL: <http://www.appdynamics.com/learn-more>. (Cited on page 283)
- [Araiza et al. 2005] R. Araiza, T. Pham, M. G. Aguilera, and P. J. Teller. Towards a cross-platform microbenchmark suite for evaluating hardware performance counter data. In: *Richard Tapia Celebration of Diversity in Computing (Tapia '05)*. IEEE Computer Society, Oct. 2005, pages 36–39. (Cited on page 105)
- [Arnold and Grove 2005] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In: *Proceedings of International Symposium on Code Generation and Optimization (CGO 05)*. IEEE Computer Society, Mar. 2005, pages 51–62. (Cited on page 54)
- [Arnold et al. 2011] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. *ACM Transactions on Software Engineering and Methodology* 21.1 (Dec. 2011), 2:1–2:35. (Cited on page 288)
- [ASM] ASM framework. OW2 Consortium. URL: <http://asm.ow2.org/>. (Cited on pages 60, 65, and 237)
- [AspectJ] AspectJ language extension. Eclipse Foundation. URL: <http://www.eclipse.org/aspectj/>. (Cited on pages 62, 65, and 290)
- [Avgustinov et al. 2005a] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In: *Proceedings of the 4th international conference on Aspect-oriented software development (AOSD)*. ACM, Mar. 2005, pages 87–98. (Cited on page 64)
- [Avgustinov et al. 2005b] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, June 2005, pages 117–128. (Cited on page 290)

- [Avgustinov et al. 2006] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. In: *First Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV)*. Springer, Aug. 2006, pages 20–39. (Cited on page 64)
- [Avgustinov et al. 2007] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, Oct. 2007, pages 589–608. (Cited on page 290)
- [Bach et al. 2010] M. Bach, M. Charney, R. Cohn, E. Demikhovskiy, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *IEEE Computer* 43.3 (Mar. 2010), pages 34–41. (Cited on page 283)
- [Ball 1999] T. Ball. The concept of dynamic analysis. In: *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-7)*. Springer, Oct. 1999, pages 216–234. (Cited on pages 29 and 44)
- [Balsamo et al. 2003] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Software Performance: State of the Art and Perspectives. Technical report CS-2003-1. Dipartimento di Informatica, Ca' Foscari University of Venice, Jan. 2003. (Cited on page 28)
- [Balsamo et al. 2004] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30.5 (May 2004), pages 295–310. (Cited on pages 2 and 28)
- [Barber 2004] S. Barber. Beyond performance testing, parts 1-14. IBM developerWorks, Rational Technical Library. June 2004. URL: <http://www.perftestplus.com/pubs.htm>. (Cited on page 29)

Bibliography

- [Barham et al. 2004] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*. USENIX, Dec. 2004, pages 259–272. (Cited on page 285)
- [Basili et al. 1994] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric Approach. In: *Encyclopedia of Software Engineering*. John Wiley & Sons, Feb. 1994, pages 528–532. (Cited on page 82)
- [BCEL] The Byte Code Engineering Library (BCEL). The Apache Software Foundation. URL: <http://commons.apache.org/bcel/>. (Cited on pages 60 and 62)
- [Becker 2008] S. Becker. Performance-Related Metrics in the ISO 9126 Standard. In: *Dependability Metrics*. Springer, June 2008, pages 204–206. (Cited on page 20)
- [Becker et al. 2004] S. Becker, L. Grunske, R. Mirandola, and S. Overhage. Performance prediction of component-based systems: A survey from an engineering perspective. In: *Architecting Systems with Trustworthy Components*. Springer, Dec. 2004, pages 169–192. (Cited on page 28)
- [Becker et al. 2009] S. Becker, H. Koziolok, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82.1 (Jan. 2009), pages 3–22. (Cited on page 28)
- [Beitz 2014] P. Beitz. KoPeMe, MooBench, Jenkins, Ant, and more. Term paper. Department of Computer Science, Kiel University, Germany, Aug. 2014. (Cited on pages 215 and 217)
- [Bell et al. 2009] G. Bell, T. Hey, and A. Szalay. Beyond the data deluge. *Science* 323.5919 (Mar. 2009), pages 1297–1298. (Cited on page 126)
- [Bellavista et al. 2002] P. Bellavista, A. Corradi, and C. Stefanelli. Java for on-line distributed monitoring of heterogeneous systems and services. *The Computer Journal* 45.6 (June 2002), pages 595–607. (Cited on pages 54, 89, and 285)
- [Bertolino et al. 2013] A. Bertolino, E. Marchetti, and A. Morichetta. Adequate monitoring of service compositions. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, Aug. 2013, pages 59–69. (Cited on page 46)

- [Berube and Amaral 2007] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In: *Standard Performance Evaluation Corporation (SPEC) Benchmark Workshop*. SPEC, Jan. 2007, pages 1–7. (Cited on page 38)
- [Beye 2013] J. Beye. Evaluation of Technologies for Communication between Monitoring and Analysis Component in Kieker. Bachelor thesis. Department of Computer Science, Kiel University, Germany, Sept. 2013. (Cited on page 10)
- [Binnig et al. 2009] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow? Towards a benchmark for the cloud. In: *Proceedings of the Second International Workshop on Testing Database Systems (DBTest '09)*. ACM, July 2009, 9:1–9:6. (Cited on page 105)
- [Blackburn et al. 2006] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiederemann. The DaCapo benchmarks: Java benchmarking development and analysis. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, Oct. 2006, pages 169–190. (Cited on pages 38, 105, 117, 121, 127, 155, and 262)
- [Blackburn et al. 2012] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, V. Babka, W. Binder, T. Brecht, L. Bulej, L. Eeckhout, S. Fischmeister, D. Frampton, R. Garner, A. Georges, L. J. Hendren, M. Hind, A. L. Hosking, R. Jones, T. Kalibera, P. Moret, N. Nystrom, V. Pankratius, and P. Tuma. Can you trust your experimental results? Technical report #1. Evaluate Collaboratory, Feb. 2012. (Cited on pages 38, 81, 105, 117, and 121)
- [Bloch 2009] J. Bloch. Performance Anxiety. Talk at JavaOne conference. June 2009. (Cited on pages 3, 127, 128)
- [Bodden and Havelund 2008] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In: *Proceedings of the 2008 international*

Bibliography

- symposium on Software testing and analysis (ISSTA)*. ACM, July 2008, pages 155–166. (Cited on page 64)
- [Bodden et al. 2007] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In: *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007)*. Springer, Aug. 2007, pages 525–549. (Cited on page 290)
- [Bodkin 2005] R. Bodkin. AOP@Work: Performance monitoring with AspectJ, parts 1–2. IBM developerWorks, Java Technical Library. Sept. 2005. (Cited on page 64)
- [Böhme and Freiling 2008] R. Böhme and F. Freiling. On Metrics and Measurements. In: *Dependability Metrics*. Springer, June 2008, pages 7–13. (Cited on page 20)
- [Bonakdarpour and Fischmeister 2011] B. Bonakdarpour and S. Fischmeister. Runtime monitoring of time-sensitive systems. In: *Proceedings of the 2nd International Conference on Runtime Verification (RV'11)*. Springer, Sept. 2011, pages 19–33. (Cited on page 283)
- [Bond and McKinley 2005] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Nov. 2005, pages 130–140. (Cited on page 54)
- [Bošković and Hasselbring 2009] M. Bošković and W. Hasselbring. Model driven performance measurement and assessment with MoDePeMART. In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*. Springer, Oct. 2009, pages 62–76. (Cited on page 67)
- [Boyer 2008] B. Boyer. Robust Java benchmarking, parts 1–2. IBM developerWorks, Java Technical Library. June 2008. URL: <https://www.ibm.com/developerworks/java/library/j-benchmark1/>. (Cited on page 127)
- [Brauer and Hasselbring 2012] P. C. Brauer and W. Hasselbring. Capturing provenance information with a workflow monitoring extension for the Kieker framework. In: *Proceedings of the 3rd International Workshop on Semantic Web in Provenance Management*. CEUR Workshop Proceedings, May 2012, pages 1–6. (Cited on page 67)

- [Brauer and Hasselbring 2013] P. C. Brauer and W. Hasselbring. PubFlow: a scientific data publication framework for marine science. In: *Proceedings of the International Conference on Marine Data and Information Systems (IMDIS 2013)*. OGS, Sept. 2013, pages 29–31. (Cited on page 126)
- [Brauer et al. 2014] P. C. Brauer, F. Fittkau, and W. Hasselbring. The aspect-oriented architecture of the CAPS framework for capturing, analyzing and archiving provenance data. In: *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW 2014)*. Springer, June 2014, pages 1–3. (Cited on page 67)
- [Brebner et al. 2005] P. Brebner, E. Cecchet, J. Marguerite, P. Tůma, O. Ciuhandu, B. Dufour, L. Eeckhout, S. Frénot, A. S. Krishna, J. Murphy, and C. Verbrugge. Middleware benchmarking: Approaches, results, experiences. *Concurrency and Computation: Practice and Experience* 17.15 (June 2005), pages 1799–1805. (Cited on pages 105, 117, 121, and 280)
- [Bruneton et al. 2002] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In: *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002: Adaptable and extensible component systems*. ACM, Nov. 2002. (Cited on pages 60 and 65)
- [Brüseke et al. 2013] F. Brüseke, G. Engels, and S. Becker. Decision support via automated metric comparison for the Palladio-based performance blame analysis. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, Apr. 2013, pages 77–88. (Cited on page 283)
- [Buble et al. 2003] A. Buble, L. Bulej, and P. Tůma. CORBA benchmarking: A course with hidden obstacles. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Apr. 2003, pages 279–285. (Cited on pages 104, 116, and 120)
- [Büge et al. 2010] V. Büge, V. Mauch, G. Quast, A. Scheurer, and A. Trunov. Site specific monitoring of multiple information systems – the HappyFace Project. *Journal of Physics: Conference Series* 219.6 (May 2010), pages 1–9. (Cited on page 165)

Bibliography

- [Bulej 2007] L. Bulej. Connector-based Performance Data Collection for Component Applications. PhD thesis. Charles University in Prague, Czech Republic, July 2007. (Cited on pages 45, 48, 49, 51, 52, 57, 60, 89, and 286)
- [Bulej and Bureš 2006] L. Bulej and T. Bureš. Eliminating execution overhead of disabled optional features in connectors. In: *Proceedings of the the 3rd European Workshop on Software Architectures (EWSA 2006)*. Springer, Sept. 2006, pages 50–65. (Cited on page 286)
- [Bulej et al. 2005] L. Bulej, T. Kalibera, and P. Tůma. Repeated results analysis for middleware regression benchmarking. *Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems* 60.4 (May 2005), pages 345–358. (Cited on pages 35, 36, 105, 117, 121, and 215)
- [Bull et al. 2000] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency and Computation: Practice and Experience* 12.6 (Aug. 2000), pages 375–388. (Cited on pages 35, 36, 104, 116, 120, 127, and 280)
- [Callanan et al. 2008] S. Callanan, D. J. Dean, M. Gorbvitski, R. Grosu, J. Seyster, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with bounded overhead. In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*. IEEE Computer Society, Apr. 2008, pages 1–8. (Cited on page 288)
- [Cappelli and Kowall 2011] W. Cappelli and J. Kowall. Magic Quadarant for Application Performance Monitoring (2011). Gartner, Inc., Sept. 2011. (Cited on page 47)
- [Carzaniga and Wolf 2002] A. Carzaniga and A. L. Wolf. A Benchmark Suite for Distributed Publish/Subscribe Systems. Technical report CU-CS-927-02. Department of Computer Science, University of Colorado, CO, USA, Apr. 2002. (Cited on pages 38 and 104)
- [Chawla and Orso 2004] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. *ACM SIGSOFT Software Engineering Notes* 29.5 (Sept. 2004), pages 1–4. (Cited on page 283)

- [Chen and Roşu 2005] F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Apr. 2005, pages 546–550. (Cited on page 64)
- [Chiba 2000] S. Chiba. Load-time structural reflection in Java. In: *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*. Springer, June 2000, pages 313–336. (Cited on pages 60 and 65)
- [Chiba et al. 2010] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*. ACM, Oct. 2010, pages 539–554. (Cited on page 65)
- [Christensen 2006] L. B. Christensen. *Experimental Methodology*. 10th edition. Pearson, June 2006. (Cited on page 81)
- [Click 2009] C. Click. The Art of (Java) Benchmarking. Talk at JavaOne conference. June 2009. (Cited on pages 127, 128)
- [Coady et al. 2001] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In: *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9)*. ACM, Sept. 2001, pages 88–98. (Cited on page 65)
- [Colyer et al. 2003] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for component integration in middleware. In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*. ACM, Oct. 2003, pages 339–344. (Cited on pages 64, 65)
- [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. *IEEE Transactions on Software Engineering* 99.5 (Apr. 2009), pages 684–702. (Cited on pages 29, 44, and 89)

Bibliography

- [Crick et al. 2014] T. Crick, B. A. Hall, and S. Ishtiaq. “Can I implement your algorithm?”: A model for reproducible research software. In: *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE2)*. arXiv, Nov. 2014, pages 1–4. (Cited on pages 81 and 126)
- [Cybenko et al. 1990] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. In: *Proceedings of the 4th international conference on Supercomputing (ICS '90)*. ACM, Sept. 1990, pages 254–266. (Cited on pages 35, 36, 104, 116, and 120)
- [Dahm 2001] M. Dahm. Byte Code Engineering with the BCEL API. Technical report B-17-98. Department of Computer Science, Free University of Berlin, Germany, Apr. 2001. (Cited on page 60)
- [De Oliveira et al. 2013] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous performance evaluation made easy. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, Apr. 2013, pages 137–148. (Cited on page 40)
- [Debusmann and Geihs 2003] M. Debusmann and K. Geihs. Efficient and transparent instrumentation of application components using an aspect-oriented approach. In: *Self-Managing Distributed Systems, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*. Springer, Oct. 2003, pages 209–220. (Cited on pages 64 and 66)
- [Denaro et al. 2004] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In: *Proceedings of the 4th international workshop on Software and performance (WOSP '04)*. ACM, Jan. 2004, pages 94–103. (Cited on pages 29, 105, 117, and 121)
- [DFG 2013] Proposals for Safeguarding Good Scientific Practice: Recommendations of the Commission on Professional Self Regulation in Science. Deutsche Forschungsgemeinschaft (DFG), Dec. 2013. (Cited on page 126)
- [DiSL] DiSL Java instrumentation language and framework. URL: <http://disl.ow2.org/>. (Cited on page 65)

- [Diwan et al. 2011] A. Diwan, M. Hauswirth, T. Mytkowicz, and P. F. Sweeney. TraceAnalyzer: A system for processing performance traces. *Software: Practice and Experience* 41.3 (Mar. 2011), pages 267–282. (Cited on page 290)
- [Dodd and Ravishankar 1992] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed realtime programs. *Software: Practice and Experience* 22.10 (Oct. 1992), pages 863–877. (Cited on page 283)
- [Döhring 2012] P. Döhring. Visualisierung von Synchronisationspunkten in Kombination mit der Statik und Dynamik eines Softwaresystems. German. Master thesis. Department of Computer Science, Kiel University, Germany, Oct. 2012. (Cited on pages 7, 10, 166, and 168)
- [Dongarra et al. 1987] J. Dongarra, J. L. Martin, and J. Worlton. Computer benchmarking: Paths and pitfalls. *IEEE Spectrum* 24.7 (July 1987), pages 38–43. (Cited on pages 35, 104, 120, and 280)
- [Dumke et al. 2013] R. Dumke, C. Ebert, J. Heidrich, and C. Wille. Messung und Bewertung von Software. German. *Informatik-Spektrum* 36.6 (Dec. 2013), pages 508–519. (Cited on pages 3 and 82)
- [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 pages 35 and 215)
- [Easterbrook et al. 2008] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting Empirical Methods for Software Engineering Research. In: *Guide to Advanced Empirical Software Engineering*. Springer, Jan. 2008, pages 285–311. (Cited on page 81)
- [Ehlers 2012] J. Ehlers. Self-Adaptive Performance Monitoring for Component-Based Software Systems. Kiel Computer Science Series 2012-1. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, Apr. 2012. (Cited on pages 20, 21, 27, 49, 72, and 94)

Bibliography

- [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)*. ACM, June 2011, pages 197–200. (Cited on pages 6, 88, 130, and 218)
- [Ehmke 2013] N. C. Ehmke. Development of a Concurrent and Distributed Analysis Framework for Kieker. Master thesis. Department of Computer Science, Kiel University, Germany, Oct. 2013. (Cited on pages 9, 10, and 214)
- [Ehmke et al. 2013] N. C. Ehmke, J. Waller, and W. Hasselbring. Development of a concurrent and distributed analysis framework for Kieker. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 79–88. (Cited on pages 8, 10, and 214)
- [Eichelberger and Schmid 2012] H. Eichelberger and K. Schmid. Erhebung von Produkt-Laufzeit-Metriken: Ein Vergleich mit dem SPASS-Meter-Werkzeug. German. In: *Proceedings of the DASMA Metrik Kongress (MetriKon '12)*. Shaker Verlag, Nov. 2012, pages 171–180. (Cited on pages 236 and 289)
- [Eichelberger and Schmid 2014] H. Eichelberger and K. Schmid. Flexible resource monitoring of Java programs. *Journal of Systems and Software* 93 (July 2014), pages 163–186. (Cited on pages 2, 46, 54, 89, 136, 236, 241, 242, 246, 256, and 289)
- [Fenton and Pfleeger 1998] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. 2nd edition. PWS Publishing Co., Feb. 1998. (Cited on pages 20, 40, 81, and 114)
- [Fidge 1996] C. J. Fidge. Fundamentals of distributed system observation. *IEEE Software* 13.6 (Nov. 1996), pages 77–83. (Cited on page 89)
- [Fittkau et al. 2013a] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In: *1st IEEE International Working Conference on Software Visualization (VISOFT 2013)*. IEEE Computer Society, Sept.

2013, pages 1–4. (Cited on pages 7, 164, 173–175, 208, 210, 211, 258, and 277)

- [Fittkau et al. 2013b] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with Kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 89–98. (Cited on pages 8, 88, 125, 130, 180, 210, 211, and 213)
- [Fittkau et al. 2013c] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Data for: Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. Nov. 2013. DOI: 10.5281/zenodo.7622. (Cited on pages 125, 210, and 301)
- [Fittkau et al. 2015] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing trace visualizations for program comprehension through controlled experiments. In: *International Conference on Program Comprehension (ICPC 2015)*. Submitted. 2015, pages 1–11. (Cited on page 9)
- [Flaig 2014] A. Flaig. Dynamic Instrumentation in Kieker Using Runtime Bytecode Modification. Bachelor thesis. Institute of Software Technology, University of Stuttgart, Germany, Nov. 2014. (Cited on pages 291 and 301)
- [Fleming and Wallace 1986] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM* 29.3 (Mar. 1986), pages 218–221. (Cited on page 40)
- [Focke 2006] T. Focke. Performance Monitoring von Middleware-basierten Applikationen. German. Diploma thesis. University of Oldenburg, Mar. 2006. (Cited on pages 69 and 288)
- [Focke et al. 2007] T. Focke, W. Hasselbring, J.-G. Schute, and M. Rohr. Ein Vorgehensmodell für Performance-Monitoring von Informationssystemlandschaften. German. *EMISA Forum* 27.1 (Jan. 2007), pages 26–31. (Cited on page 46)

Bibliography

- [Folkerts et al. 2012] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In: *Proceedings of the 4th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC '12)*. Springer, Aug. 2012, pages 173–188. (Cited on pages 3, 101, 105, 121, and 281)
- [Fowler 2006] M. Fowler. Continuous Integration. May 2006. URL: <http://www.martinfowler.com/articles/continuousIntegration.html>. (Cited on pages 35 and 215)
- [Frey 2013] S. Frey. Conformance Checking and Simulation-based Evolutionary Optimization for Deployment and Reconfiguration of Software in the Cloud. Kiel Computer Science Series 2013-1. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, Aug. 2013. (Cited on page 69)
- [Frotscher 2013] T. Frotscher. Architecture-Based Multivariate Anomaly Detection for Software Systems. Master thesis. Department of Computer Science, Kiel University, Germany, Oct. 2013. (Cited on pages 10 and 218)
- [Gait 1986] J. Gait. A probe effect in concurrent programs. *Software: Practice and Experience* 16.3 (Mar. 1986), pages 225–233. (Cited on page 89)
- [Gao et al. 2000] J. Gao, E. Y. Zhu, S. Shim, and L. Chang. Monitoring software components and component-based software. In: *The 24th Annual International Computer Software and Applications Conference (COMP-SAC 2000)*. IEEE, Oct. 2000, pages 403–412. (Cited on pages 66, 68, and 283)
- [Gao et al. 2001] J. Gao, E. Y. Zhu, and S. Shim. Tracking software components. *Journal of Object-oriented Programming* 14.4 (May 2001), pages 13–22. (Cited on pages 66, 68, and 283)
- [García et al. 2006] F. García, M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruíz, M. Piattini, and M. Genero. Towards a consistent terminology for software measurement. *Information and Software Technology* 48.8 (Aug. 2006), pages 631–644. (Cited on page 20)

- [Georges et al. 2004] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*. ACM, Oct. 2004, pages 270–287. (Cited on pages 51 and 127)
- [Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, Oct. 2007, pages 57–76. (Cited on pages 3, 38, 40–42, 105, 117, 121, 127, 150, and 184)
- [Georges et al. 2008] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, Oct. 2008, pages 367–384. (Cited on pages 127, 128)
- [Gil et al. 2011] J. Y. Gil, K. Lenz, and Y. Shimron. A microbenchmark case study and lessons learned. In: *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages (VMIL '11)*. ACM, Oct. 2011, pages 297–308. (Cited on pages 105, 117, 127, 128, and 281)
- [GluonJ] GluonJ aspect-oriented programming system for Java. URL: <http://www.csg.ci.i.u-tokyo.ac.jp/projects/gluonj/>. (Cited on page 65)
- [Goers and Popma 2014] R. Goers and R. Popma. LOG4j – Asynchronous Loggers for Low-Latency Logging. The Apache Software Foundation, July 2014. URL: <http://logging.apache.org/log4j/2.x/manual/async.html>. (Cited on page 291)
- [Goetz 2004] B. Goetz. The perils of benchmarking under dynamic compilation. IBM developerWorks, Java Theory and Practice. Dec. 2004. URL: <http://www.ibm.com/developerworks/Library/j-jtp12214/>. (Cited on page 127)
- [Goodstein et al. 2010] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring. In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages*

Bibliography

- and Operating Systems (ASPLOS 2010)*. ACM, Mar. 2010, pages 257–270. (Cited on page 283)
- [Graham et al. 1982] S. L. Graham, P. B. Kessler, and M. K. Mckusick. gprof: A call graph execution profiler. In: *Proceedings of the 1982 SIGPLAN symposium on Compiler construction (SIGPLAN '82)*. ACM, June 1982, pages 120–126. (Cited on page 60)
- [Gray 1993] J. Gray, editor. *The Benchmark Handbook: For Database and Transaction Systems*. 2nd edition. Morgan Kaufmann, May 1993. (Cited on pages 39, 101, 104, and 280)
- [Gray 2009] J. Gray. Jim Gray on eScience: A Transformed Scientific Method (Based on the transcript of a talk given by Jim Gray in 2007). In: *The Fourth Paradigm*. Edited by T. Hey, S. Tansley, and K. Tolle. Microsoft Research, Oct. 2009, pages xvii–xxxi. (Cited on page 126)
- [Ha et al. 2009] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, Oct. 2009, pages 155–174. (Cited on page 286)
- [Harms 2013] B. Harms. Reverse-Engineering und Analyse einer Plugin-basierten Java-Anwendung. German. Master thesis. Department of Computer Science, Kiel University, Germany, Nov. 2013. (Cited on page 10)
- [Hauswirth et al. 2004] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*. ACM, Oct. 2004, pages 251–269. (Cited on pages 51, 53, 89, and 283)
- [He and Zhai 2011] G. He and A. Zhai. Efficient dynamic program monitoring on multi-core systems. *Journal of Systems Architecture (Special Issue On-Chip Parallel And Network-Based Systems)* 57.1 (Jan. 2011), pages 121–133. (Cited on page 290)

- [Herold et al. 2008] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolok, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME – The Common Component Modeling Example. In: *The Common Component Modeling Example*. Springer, Oct. 2008, pages 16–53. (Cited on page 155)
- [Hinnant 1988] D. F. Hinnant. Accurate Unix benchmarking: Art, science, or black magic? *IEEE Micro* 8.5 (Oct. 1988), pages 64–75. (Cited on pages 3, 33, 36, 37, 101, 104, 116, 120, and 280)
- [Hinsen 2012] K. Hinsen. Caring for your data. *Computing in Science & Engineering* 14.6 (Nov. 2012), pages 70–74. (Cited on page 126)
- [Höfer and Tichy 2006] A. Höfer and W. F. Tichy. Status of empirical research in software engineering. In: *Proceedings of the International Workshop on Empirical Software Engineering Issues (Critical Assessment and Future Directions)*. Springer, June 2006, pages 10–19. (Cited on page 81)
- [Hohenstein and Jäger 2009] U. D. Hohenstein and M. C. Jäger. Using aspect-orientation in industrial projects: Appreciated or damned? In: *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD)*. ACM, Mar. 2009, pages 213–222. (Cited on page 65)
- [Hohpe and Woolf 2004] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, Aug. 2004. (Cited on page 73)
- [Huang et al. 2012] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer* 14.3 (June 2012), pages 327–347. (Cited on pages 89 and 288)
- [Hunt and John 2011] C. Hunt and B. John. *Java™ Performance*. Addison-Wesley, Sept. 2011. (Cited on pages 28, 29, 45, 52, 54, 55, 127, 128)
- [Huppler 2009] K. Huppler. The art of building a good benchmark. In: *First TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009)*. Springer, Aug. 2009, pages 18–30. (Cited on pages 101, 105, 117, 121, and 281)

Bibliography

- [Ihaka and Gentleman 1996] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5.3 (May 1996), pages 299–314. (Cited on page 150)
- [Iqbal and John 2010] M. F. Iqbal and L. K. John. Confusion by all means. In: *Proceedings of the Workshop on Unique Chips and Systems (UCAS 2010)*. Dec. 2010, pages 105–114. (Cited on page 40)
- [Isaacs et al. 2014] K. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. In: *Proceedings of the 16th annual Eurographics Conference on Visualization (EuroVis 2014)*. Blackwell Publishing, June 2014, pages 1–20. (Cited on pages 45, 51, and 166)
- [ISO/IEC 25010] ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO/IEC, Mar. 2011. (Cited on pages 18–20, and 32)
- [ISO/IEC 9126] ISO/IEC 9126-1:2001 – Software Engineering – Product Quality – Part 1: Quality Model. ISO/IEC, June 2001. (Cited on pages 18 and 20)
- [ISO/IEC/IEEE 24765] ISO/IEC/IEEE 24765:2010 – Systems and software engineering – Vocabulary. ISO/IEC/IEEE, Dec. 2010. (Cited on pages 18, 32, 44, and 48)
- [Itzkowitz 2010] M. Itzkowitz. Oracle Solaris Studio Performance Tools. White paper. Oracle Corporation, Nov. 2010. (Cited on page 60)
- [Jain 1991] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Apr. 1991. (Cited on pages 19–21, 23, 28, 33, 40, 48, 51, 68, 89, 104, 113, 114, 116, and 120)
- [Java Pet Store] Java Pet Store. Sun Microsystems, Inc., 2005. URL: <http://java.sun.com/developer/releases/petstore/>. (Cited on page 37)
- [Javassist] S. Chiba. Javassist (Java programming assistant). URL: <http://www.javassist.org/>. (Cited on pages 60, 65, and 237)

- [JCGM 200:2008] International vocabulary of metrology – Basic and general concepts and associated terms (VIM). Joint Committee for Guides in Metrology (JCGM), Jan. 2008. (Cited on page 20)
- [Jeffery 1996] C. L. Jeffery. *Program Monitoring and Visualization: An Exploratory Approach*. Springer, June 1996. (Cited on pages 2 and 283)
- [Jendrock et al. 2012] E. Jendrock, R. Cervera-Navarro, I. Evans, D. Gollapudi, K. Haase, W. M. Oliveira, and C. Srivathsa. *The Java EE 6 Tutorial*. Oracle Corporation, July 2012. URL: <http://docs.oracle.com/javaee/6/tutorial/doc/>. (Cited on page 66)
- [Jenkins] Jenkins – An extendable open source continuous integration server. Jenkins CI community. URL: <http://jenkins-ci.org/>. (Cited on page 216)
- [Jikes RVM] Jikes Research Virtual Machine (RVM). The Jikes RVM Project. URL: <http://jikesrvm.org>. (Cited on pages 51 and 54)
- [JMX] Java Management Extensions (JMX) Technology. URL: <http://java.sun.com/products/JavaManagement/>. (Cited on page 54)
- [John 2005a] L. K. John. Benchmarks. In: *Performance Evaluation and Benchmarking*. Taylor & Francis, Sept. 2005. (Cited on pages 35, 39, and 105)
- [John 2005b] L. K. John. Performance Modeling and Measurement Techniques. In: *Performance Evaluation and Benchmarking*. Taylor & Francis, Sept. 2005. (Cited on pages 28 and 45)
- [Johnson 2004] M. W. Johnson. Monitoring and diagnosing applications with ARM 4.0. In: *Proceedings of the Computer Measurement Group Conference*. Computer Measurement Group, Dec. 2004, pages 473–484. (Cited on page 57)
- [Jones 2010] D. Jones. *The Five Essential Elements of Application Performance Monitoring*. Quest Software. Nov. 2010. (Cited on page 2)
- [Joslin 1965] E. O. Joslin. Application benchmarks: The key to meaningful computer evaluations. In: *Proceedings of the 20th National Conference (ACM '65)*. ACM, Aug. 1965, pages 27–37. (Cited on pages 33 and 104)

Bibliography

- [JSR 77] JSR 77: J2EE™ Management. Oracle Corporation, July 2002. URL: <http://www.jcp.org/en/jsr/detail?id=77>. (Cited on page 55)
- [Jung et al. 2013] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In: *Proceedings Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDAYS 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 99–108. (Cited on pages 67 and 208)
- [Juristo and Gómez 2012] N. Juristo and O. S. Gómez. Replication of Software Engineering Experiments. In: *Empirical Software Engineering and Verification*. Springer, Jan. 2012, pages 60–88. (Cited on page 81)
- [JVM TI] Java Virtual Machine Tool Interface (JVM TI). URL: <http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/>. (Cited on pages 54, 58, 59, and 65)
- [Kähkipuro et al. 1999] P. Kähkipuro, A. Buble, A. Gopinath, A. Kaushal, C. Liyanaarachchi, C. Readler, D. Flater, D. Niehaus, F. Caruso, F. Plášil, K. Raatikainen, L.-F. Pau, M. Chung, P. Tůma, S. Nimmagadda, and S. Tockey. Benchmark PSIG – White Paper on Benchmarking. White paper. Object Management Group (OMG), Dec. 1999. (Cited on pages 33, 104, 116, and 120)
- [Kalibera 2006] T. Kalibera. Performance in Software Development Cycle: Regression Benchmarking. PhD thesis. Charles University in Prague, Czech Republic, Sept. 2006. (Cited on pages 35, 36, 105, 117, and 215)
- [Kalibera and Jones 2013] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In: *Proceedings of the International Symposium on Memory Management (ISMM '13)*. ACM, June 2013, pages 63–74. (Cited on pages 40, 105, 117, 118, and 121)
- [Kalibera et al. 2004] T. Kalibera, L. Bulej, and P. Tůma. Generic environment for full automation of benchmarking. In: *Proceedings of the First International Workshop on Software Quality (SOQUA 2004)*. GI, Sept. 2004, pages 125–132. (Cited on pages 105, 117, 121, and 126)
- [Kalibera et al. 2005] T. Kalibera, L. Bulej, and P. Tůma. Benchmark precision and random initial state. In: *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005)*. SCS, July 2005, pages 853–862. (Cited on page 40)

- [Kanstrén et al. 2011] T. Kanstrén, R. Savola, S. Haddad, and A. Hecker. An adaptive and dependable distributed monitoring framework. *International Journal On Advances in Security* 4.1&2 (Sept. 2011), pages 80–94. (Cited on pages 3 and 287)
- [Katsaros et al. 2012] G. Katsaros, G. Kousiouris, S. V. Gogouvitis, D. Kyriazis, A. Menychtas, and T. Varvarigou. A self-adaptive hierarchical monitoring mechanism for clouds. *Journal of Systems and Software* 85.5 (May 2012), pages 1029–1041. (Cited on page 288)
- [Khaled et al. 2003] R. Khaled, J. Noble, and R. Biddle. InspectJ: Program monitoring for visualisation using AspectJ. In: *Proceedings of the 26th Australasian computer science conference (ACSC)*. Australian Computer Society, Inc., Feb. 2003, pages 359–368. (Cited on page 64)
- [Kiczales et al. 1996] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect-Oriented Programming. Position Paper from the Xerox PARC Aspect-Oriented Programming Project. Xerox Paolo Alto Research Center, 1996. (Cited on pages 61, 62)
- [Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*. Springer, June 1997, pages 220–242. (Cited on page 61)
- [Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*. Springer, June 2001, pages 327–353. (Cited on page 62)
- [Kieker] Kieker Monitoring and Analysis Framework. Kieker Project. URL: <http://kieker-monitoring.net/>. (Cited on page 69)
- [Kieker 2014] Kieker Project. Kieker User Guide. Oct. 2014. URL: <http://kieker-monitoring.net/documentation/>. (Cited on pages 37, 69, and 155)
- [Kim 2002] H. Kim. AspectC#: An AOSD implementation for C#. Master’s thesis. Trinity College Dublin, Sept. 2002. (Cited on page 65)

Bibliography

- [Kitchenham et al. 2002] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002), pages 721–734. (Cited on page 81)
- [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)*. GI, May 2012, pages 33–34. (Cited on pages 65, 69, 74, and 97)
- [Konarski 2012] B. Konarski. Ein 3D-Ansatz zur Visualisierung der Kernauslastung in Multiprozessorsystemen. German. Diploma thesis. Department of Computer Science, Kiel University, Germany, July 2012. (Cited on page 10)
- [Kounev 2005] S. Kounev. Performance Engineering of Distributed Component-Based Systems – Benchmarking, Modeling and Performance Prediction. PhD thesis. TU Darmstadt, Germany, Dec. 2005. (Cited on pages 105, 117, 121, 155, 262, and 280)
- [Kowall and Cappelli 2012a] J. Kowall and W. Cappelli. Criteria Are Maturing for the Magic Quadrant for Application Performance Monitoring. Gartner, Inc., Feb. 2012. (Cited on page 47)
- [Kowall and Cappelli 2012b] J. Kowall and W. Cappelli. Magic Quadrant for Application Performance Monitoring (2012). Gartner, Inc., Aug. 2012. (Cited on page 47)
- [Kowall and Cappelli 2013] J. Kowall and W. Cappelli. Magic Quadrant for Application Performance Monitoring (2013). Gartner, Inc., Dec. 2013. (Cited on pages 47, 48)
- [Koziolok 2010] H. Koziolok. Performance evaluation of component-based software systems: A survey. *Performance Evaluation* 67.8 (Aug. 2010), pages 634–658. (Cited on page 28)

- [Kranzlmüller et al. 1999] D. Kranzlmüller, R. Reussner, and C. Schaub-schläger. Monitor overhead measurement with SKaMPI. In: *Proceedings of the 6th European PVM/MPI Users' Group Meeting*. Springer, Sept. 1999, pages 43–50. (Cited on page 283)
- [Kroll 2011] L. Kroll. Performance Monitoring for a Web-based Information System. Bachelor thesis. Department of Computer Science, Kiel University, Germany, June 2011. (Cited on page 10)
- [Kuck and Sameh 1987] D. J. Kuck and A. H. Sameh. A supercomputing performance evaluation plan. In: *Proceedings of the First International Conference on Supercomputing (ICS '87)*. Springer, June 1987, pages 1–17. (Cited on pages 104 and 120)
- [Kumar et al. 2005] N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, Sept. 2005, pages 28–34. (Cited on page 290)
- [Kunze et al. 2011] J. A. Kunze, T. Cruse, R. Hu, S. Abrams, K. Hastings, C. Mitchell, and L. Schiff. Practices, Trends, and Recommendations in Technical Appendix Usage for Selected Data-Intensive Disciplines. CDL Staff Publications. California Digital Library, Jan. 2011. (Cited on page 126)
- [Kuperberg et al. 2010] M. Kuperberg, F. Omri, and R. Reussner. Automated benchmarking of Java APIs. In: *Proceedings of the Software Engineering 2010 (SE 2010)*. Köllen Verlag, Feb. 2010, pages 57–68. (Cited on pages 105, 117, 121, 127, and 281)
- [Lange 2009] K.-D. Lange. Identifying shades of green: The SPECpower benchmarks. *IEEE Computer* 42.3 (Mar. 2009), pages 95–97. (Cited on page 105)
- [Lavoie et al. 2014] E. Lavoie, B. Dufour, and M. Feeley. Portable and efficient run-time monitoring of JavaScript applications using virtual machine layering. In: *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP 2014)*. Springer, July 2014, pages 541–566. (Cited on page 283)

Bibliography

- [Leadbetter et al. 2013] A. Leadbetter, L. Raymond, C. Chandler, L. Pikula, P. Pissierssens, and E. Urban. *Ocean Data Publication Cookbook*. Intergovernmental Oceanographic Commission, Manuals and Guides. UNESCO, Mar. 2013. (Cited on page 126)
- [Lilja 2000] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, Aug. 2000. (Cited on pages 35, 36, 39, 40, 67, 68, 104, and 120)
- [Lilja and Yi 2005] D. J. Lilja and J. J. Yi. *Statistical Techniques for Computer Performance Analysis*. In: *Performance Evaluation and Benchmarking*. Taylor & Francis, Sept. 2005. (Cited on page 40)
- [Lindholm et al. 2014] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java™ Virtual Machine Specification*. Oracle Corporation, Mar. 2014. URL: <http://docs.oracle.com/javase/specs/>. (Cited on pages 58 and 60)
- [Liu et al. 2013] K. Liu, H. B. K. Tan, and X. Chen. Binary code analysis. *IEEE Computer* 46.8 (Aug. 2013), pages 60–68. (Cited on page 60)
- [Lucas 1971] H. C. Lucas Jr. Performance evaluation and monitoring. *ACM Computer Surveys* 3.3 (Sept. 1971), pages 79–91. (Cited on pages 2, 35, 45, 89, and 104)
- [Maebe et al. 2006] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. *Javana: A system for building customized Java program analysis tools*. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, Oct. 2006, pages 153–168. (Cited on page 285)
- [Magedanz 2011] F. Magedanz. *Dynamic analysis of .NET applications for architecture-based model extraction and test generation*. Diploma thesis. Department of Computer Science, Kiel University, Germany, Oct. 2011. (Cited on page 69)
- [Malony and Shende 2004] A. D. Malony and S. S. Shende. Overhead compensation in performance profiling. In: *Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par 2004)*. Springer, Sept. 2004, pages 119–132. (Cited on page 283)

- [Malony et al. 1992] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems* 3.4 (July 1992), pages 433–450. (Cited on pages 89 and 290)
- [Marek 2014] L. Marek. Instrumentation and Evaluation for Dynamic Program Analysis. PhD thesis. Charles University in Prague, Czech Republic, Sept. 2014. (Cited on pages 29, 48, 54, 55, 65, and 89)
- [Marek et al. 2012] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A domain-specific language for bytecode instrumentation. In: *Proceedings of the 11th annual international conference on Aspect-oriented software Development (AOSD)*. ACM, Mar. 2012, pages 239–250. (Cited on page 65)
- [Marek et al. 2013] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. In: *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, Sept. 2013, pages 105–114. (Cited on page 283)
- [Marek et al. 2015] L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and P. Tůma. Introduction to dynamic program analysis with DiSL. *Science of Computer Programming* 98.1 (Feb. 2015), pages 100–115. (Cited on pages 65 and 291)
- [Maxwell 2006] K. D. Maxwell. What You Need To Know About Statistics. In: *Web Engineering*. Springer, Jan. 2006, pages 365–408. (Cited on page 40)
- [McCanne and Torek 1993] S. McCanne and C. Torek. A randomized sampling clock for CPU utilization estimation and code profiling. In: *Proceedings of the Winter 1993 USENIX Conference (USENIX '93)*. USENIX Association, Jan. 1993, pages 387–394. (Cited on page 48)
- [McCoy 2002] D. W. McCoy. Business Activity Monitoring: Calm Before the Storm. Gartner, Inc., Apr. 2002. (Cited on page 67)

Bibliography

- [Menascé et al. 2004] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. Performance by Design: Computer Capacity Planning by Example. Prentice Hall, Jan. 2004. (Cited on pages 20, 21, 23, 49, and 51)
- [Menascé 2002] D. A. Menascé. Load testing, benchmarking, and application performance management for the web. In: *Proceedings of the Computer Measurement Group Conference*. Computer Measurement Group, Dec. 2002, pages 271–282. (Cited on pages 29, 33, 38, 47, 104, and 112)
- [Menascé et al. 1999] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of E-commerce sites. In: *Proceedings of the 1st ACM conference on Electronic commerce (EC '99)*. ACM, Nov. 1999, pages 119–128. (Cited on pages 29 and 113)
- [Meng and Liu 2013] S. Meng and L. Liu. Enhanced monitoring-as-a-service for effective cloud management. *IEEE Transactions on Computers* 62.9 (Sept. 2013), pages 1705–1720. (Cited on page 290)
- [Merriam et al. 2013] N. Merriam, P. Gliwa, and I. Broster. Measurement and tracing methods for timing analysis. *International Journal on Software Tools for Technology Transfer* 15.1 (Feb. 2013), pages 9–28. (Cited on page 28)
- [Meyer 2014] M. Meyer. Continuous integration and its tools. *IEEE Software* 31.3 (May 2014), pages 14–16. (Cited on pages 35 and 215)
- [Mogul 1992] J. C. Mogul. SPECmarks are leading us astray. In: *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS '92)*. IEEE Computer Society, Apr. 1992, pages 160–161. (Cited on page 35)
- [Mohror and Karavanic 2007] K. Mohror and K. L. Karavanic. Towards scalable event tracing for high end systems. In: *Proceedings of the 3rd International Conference on High Performance Computing and Communications (HPCC'07)*. Springer, Sept. 2007, pages 695–706. (Cited on pages 45, 48, 68, and 286)
- [Mohror and Karavanic 2012] K. Mohror and K. L. Karavanic. Trace profiling: Scalable event tracing on high-end parallel systems. *Parallel Computing* 38.4–5 (Apr. 2012), pages 194–225. (Cited on pages 45 and 286)
- [Molyneaux 2009] I. Molyneaux. The Art of Application Performance Testing. O'Reilly Media, Jan. 2009. (Cited on pages 21, 29, and 113)

- [Momm et al. 2008] C. Momm, T. Detsch, and S. Abeck. Model-driven instrumentation for monitoring the quality of web service compositions. In: *Proceedings of the 12th Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC 2008)*. IEEE Computer Society, Sept. 2008, pages 58–67. (Cited on page 67)
- [Montgomery and Runger 2010] D. C. Montgomery and G. C. Runger. Applied statistics and probability for engineers. 5th edition. John Wiley & Sons, June 2010. (Cited on pages 40 and 182)
- [Moon and Chang 2006] S. Moon and B.-M. Chang. A thread monitoring system for multithreaded Java programs. *ACM SIGPLAN Notices* 41.5 (May 2006), pages 21–29. (Cited on page 283)
- [Mos 2004] A. Mos. A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications. PhD thesis. University College Dublin, Ireland, Aug. 2004. (Cited on page 283)
- [Moseley et al. 2007] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IRISA-CNRS, Mar. 2007, pages 198–208. (Cited on page 283)
- [Müller et al. 2007] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In: *Parallel Computing: Architectures, Algorithms and Applications*. IOS Press, Sept. 2007, pages 637–644. (Cited on page 286)
- [MyBatis JPetStore]. MyBatis JPetStore. MyBatis Project. URL: <http://code.google.com/p/mybatis/>. (Cited on pages 37 and 155)
- [Mytkowicz et al. 2008a] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We have it easy, but do we have it right? In: *Proceedings of 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '08)*. IEEE Computer Society, Apr. 2008, pages 1–7. (Cited on pages 51, 105, 117, and 121)

Bibliography

- [Mytkowicz et al. 2008b] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Observer Effect and Measurement Bias in Performance Analysis. Technical report CU-CS 1042-08. Department of Computer Science, University of Colorado, CO, USA, June 2008. (Cited on pages 105, 117, and 121)
- [Mytkowicz et al. 2009] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. ACM, Mar. 2009, pages 265–276. (Cited on page 127)
- [Mytkowicz et al. 2010] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In: *Proceedings of the 31th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, May 2010, pages 187–197. (Cited on pages 48, 89, 105, 117, 121, 127, 128, and 283)
- [NIST 2004] National Institute of Standards and Technology (NIST). SciMark 2.0. Mar. 2004. URL: <http://math.nist.gov/scimark2/>. (Cited on page 159)
- [Nusayr and Cook 2009] A. Nusayr and J. Cook. Using AOP for detailed runtime monitoring instrumentation. In: *Proceedings of the Seventh International Workshop on Dynamic Analysis (WODA)*. ACM, July 2009, pages 8–14. (Cited on page 64)
- [Okanović et al. 2013] D. Okanović, M. Vidaković, and Z. Konjović. Towards performance monitoring overhead reduction. In: *Proceedings of the IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*. IEEE Computer Society, Sept. 2013, pages 135–140. (Cited on page 289)
- [Oliner et al. 2012] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM* 55.2 (Feb. 2012), pages 55–61. (Cited on page 58)
- [OMG 2006] Architecture-Driven Modernization (ADM) Task Force – Glossary. Object Management Group (OMG), Jan. 2006. (Cited on page 20)
- [OMG 2008] Software & Systems Process Engineering Metamodel (SPEM) Specification, version 2.0. Object Management Group (OMG), Apr. 2008. (Cited on pages 24, 25)

- [OMG 2011a] Business Process Model and Notation (BPMN), version 2.0. Object Management Group (OMG), Jan. 2011. (Cited on page 67)
- [OMG 2011b] Unified Modeling Language (UML) Specification, version 2.4.1. Object Management Group (OMG), Aug. 2011. (Cited on pages 70, 73, 91, 132, 135, and 137)
- [OMG 2012] Software Metrics Meta-Model (SMM) Specification, version 1.0. Object Management Group (OMG), Jan. 2012. (Cited on page 20)
- [Ortin et al. 2014] F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo. The runtime performance of invokedynamic: An evaluation with a Java library. *IEEE Software* 31.4 (July 2014), pages 82–90. (Cited on pages 117 and 121)
- [Parsons et al. 2006] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end-to-end runtime path tracing for J2EE systems. *IEEE Software* 153.4 (Aug. 2006), pages 149–161. (Cited on pages 66 and 283)
- [Parsons 2007] T. Parsons. Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems. PhD thesis. University College Dublin, Ireland, Nov. 2007. (Cited on pages 29, 54, 58, 66, and 283)
- [Patil and Fischer 1995] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In: *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG 1995)*. IRISA-CNRS, May 1995, pages 119–132. (Cited on page 283)
- [Patterson 2002] D. Patterson. How to Have a Bad Career in Research/Academia. Talk at CRA Academic Careers Workshop. Feb. 2002. (Cited on page 38)
- [Pearce et al. 2002] D. Pearce, P. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the linux kernel. In: *Proceedings of the 12th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS 2002)*. Springer, Apr. 2002, pages 220–226. (Cited on page 60)
- [Pearce et al. 2007] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience* 37.7 (June 2007), pages 747–777. (Cited on pages 29, 48, and 64)

Bibliography

- [Peng 2011] R. D. Peng. Reproducible research in computational science. *Science* 334.6060 (Dec. 2011), pages 1226–1227. (Cited on pages 124, 125)
- [Petcu 2013] D. Petcu. Multi-cloud: Expectations and current approaches. In: *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds (MultiCloud '13)*. ACM, Apr. 2013, pages 1–6. (Cited on page 165)
- [Pfleeger 1995] S. L. Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering* 1 (Dec. 1995), pages 219–253. (Cited on page 39)
- [Plattner and Nievergelt 1981] B. Plattner and J. Nievergelt. Special feature: Monitoring program execution: A survey. *IEEE Computer* 14.11 (Nov. 1981), pages 76–93. (Cited on pages 2, 44, and 89)
- [Pogue et al. 2014] C. Pogue, A. Kumar, D. Tollefson, and S. Realmuto. SPECjbb2013 1.0: An overview. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*. ACM, Mar. 2014, pages 231–232. (Cited on page 159)
- [Price 1989] W. J. Price. A benchmark tutorial. *IEEE Micro* 9.5 (Oct. 1989), pages 28–43. (Cited on pages 3, 33, 39, 101, 104, 116, 120, and 280)
- [R] The R Project for Statistical Computing. GNU Operating System. URL: <http://www.r-project.org/>. (Cited on page 150)
- [Reichelt and Braubach 2014] D. G. Reichelt and L. Braubach. Sicherstellung von Performanzeigenschaften durch kontinuierliche Performanztests mit dem KoPeMe Framework. German. In: *Software Engineering 2014: Fachtagung des GI-Fachbereichs Softwaretechnik*. Köllen Verlag, Mar. 2014, pages 119–124. (Cited on pages 35, 215, and 217)
- [Reimer 2013] S. Reimer. Architekturzentriertes Monitoring für den Betrieb. Talk at Softwareforen Leipzig. Nov. 2013. (Cited on page 3)
- [Reiss 2008] S. P. Reiss. Controlled dynamic performance analysis. In: *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*. ACM, June 2008, pages 43–54. (Cited on page 288)

- [Reussner et al. 1998] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In: *Proceedings of the 5th European PVM/MPI Users' Group Meeting*. Springer, Sept. 1998, pages 52–59. (Cited on pages 104, 116, and 120)
- [Richters and Gogolla 2003] M. Richters and M. Gogolla. Aspect-oriented monitoring of UML and OCL constraints. In: *Proceedings of the 4th Workshop on Aspect-Oriented Modeling with UML on the 6th International Conference on the Unified Modeling Language (UML)*. Oct. 2003. (Cited on page 64)
- [Rohr et al. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In: *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*. ACTA Press, Feb. 2008, pages 80–85. (Cited on pages 68, 69, and 289)
- [Rohr et al. 2010] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In: *Proceedings of the Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW'10)*. ACM, Jan. 2010, pages 87–92. (Cited on page 69)
- [Röthlisberger et al. 2012] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Computer* 38.3 (May 2012), pages 579–591. (Cited on page 287)
- [Saavedra et al. 1993] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Characterizing the performance space of shared memory computers using Micro-Benchmarks. Technical report USC-CS-93-547. Department of Computer Science, University of Southern California, CA, USA, July 1993. (Cited on pages 37 and 154)
- [Saavedra-Barrera et al. 1989] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Transactions on Computers* 38.12 (Dec. 1989), pages 1659–1679. (Cited on pages 35, 104, 114, 116, and 120)

Bibliography

- [Sabetta and Koziolok 2008] A. Sabetta and H. Koziolok. Measuring Performance Metrics: Techniques and Tools. In: *Dependability Metrics*. Springer, June 2008, pages 226–232. (Cited on pages 29, 39, 45, 51, 52)
- [Sachs 2011] K. Sachs. Performance Modeling and Benchmarking of Event-Based Systems. PhD thesis. TU Darmstadt, Germany, Aug. 2011. (Cited on pages 3, 38, 101, 105, 117, 121, and 281)
- [Saller et al. 2013] K. Saller, K. Panitzek, and M. Lehn. Benchmarking Methodology. In: *Benchmarking Peer-to-Peer Systems*. Springer, June 2013, pages 19–45. (Cited on pages 105, 111–113, 121, and 281)
- [Sarimbekov et al. 2014] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tůma, and Z. Qi. Dynamic program analysis – Reconciling developer productivity and tool performance. *Science of Computer Programming* 95.3 (Dec. 2014), pages 344–358. (Cited on pages 65 and 291)
- [Schneider et al. 2007] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*. ACM, June 2007, pages 373–382. (Cited on pages 51 and 283)
- [Schulz et al. 2014] H. Schulz, A. Flaig, A. Wert, and A. van Hoorn. Adaptive Instrumentation of Java-Applications for Experiment-Based Performance Analysis. Talk at Symposium on Software Performance. Nov. 2014. (Cited on pages 291 and 301)
- [Seltzer et al. 1999] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The case for application-specific benchmarking. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*. IEEE Computer Society, Mar. 1999, pages 102–107. (Cited on pages 34 and 104)
- [Shao et al. 2010] J. Shao, H. Wei, Q. Wang, and H. Mei. A runtime model based monitoring approach for cloud. In: *Proceedings of the 3rd International Conference on Cloud Computing (CLOUD'10)*. IEEE Computer Society, July 2010, pages 313–320. (Cited on page 3)

- [Sheng et al. 2011] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A lightweight and non-invasive race detection tool for production applications. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, May 2011, pages 401–410. (Cited on page 287)
- [Shiv et al. 2009] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm 2008 performance characterization. In: *Computer Performance Evaluation and Benchmarking*. Springer, Jan. 2009, pages 17–35. (Cited on page 157)
- [Shull et al. 2002] F. Shull, V. Basili, J. Carver, J. C. Maldonado, G. H. Travassos, M. Mendonca, and S. Fabbri. Replicating software engineering experiments: Addressing the tacit knowledge problem. In: *Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE 2002)*. IEEE Computer Society, Oct. 2002, pages 7–16. (Cited on page 81)
- [Siegl and Bouillet 2011] S. Siegl and P. Bouillet. inspectIT ...because performance matters! White paper. NovaTec, June 2011. (Cited on pages 2, 3, 136, and 222)
- [Sim et al. 2003] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. IEEE Computer Society, May 2003, pages 74–83. (Cited on pages 3, 32, 38, 39, 104, and 280)
- [Sjøberg et al. 2007] D. I. K. Sjøberg, T. Dybå, and M. Jørgensen. The future of empirical methods in software engineering research. In: *International Conference on Software Engineering, Workshop on the Future of Software Engineering (FOSE 2007)*. IEEE Computer Society, May 2007, pages 358–378. (Cited on page 81)
- [Smith 1990] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, May 1990. (Cited on pages 2, 24, and 28)
- [Smith 1994] C. U. Smith. Performance Engineering. In: *Encyclopedia of Software Engineering*. John Wiley & Sons, Feb. 1994. (Cited on page 24)

Bibliography

- [Smith and Williams 2001] C. U. Smith and L. G. Williams. Performance Solutions – A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley, Sept. 2001. (Cited on pages 19, 22–24, 28, 33, 44, 48, 50, 52, 57, 67, 89, 104, 113, 114)
- [Smith and Williams 2003] C. U. Smith and L. G. Williams. Best practices for software performance engineering. In: *Proceedings of the Computer Measurement Group Conference*. Computer Measurement Group, Dec. 2003, pages 83–92. (Cited on page 104)
- [Snatzke 2008] R. G. Snatzke. Performance Survey 2008. codecentric GmbH. Oct. 2008. (Cited on page 2)
- [SPEC 2008a] Standard Performance Evaluation Corporation (SPEC). SPEC jvm2008 Benchmark. Apr. 2008. URL: <http://spec.org/jvm2008/>. (Cited on page 157)
- [SPEC 2008b] Standard Performance Evaluation Corporation (SPEC). SPEC jvm2008 Benchmark User’s Guide. Apr. 2008. URL: <http://spec.org/jvm2008/docs/UserGuide.html>. (Cited on page 157)
- [SPEC 2012] Standard Performance Evaluation Corporation (SPEC). SPEC jEnterprise2010 Benchmark. Apr. 2012. URL: <http://spec.org/jEnterprise2010/>. (Cited on pages 155, 262, 263)
- [SPEC 2013a] Standard Performance Evaluation Corporation (SPEC). SPEC Glossary. Feb. 2013. URL: <http://spec.org/spec/glossary/>. (Cited on pages 20, 29, 33, 105, 117, and 121)
- [SPEC 2013b] Standard Performance Evaluation Corporation (SPEC). SPECjbb 2013 Benchmark. Jan. 2013. URL: <http://spec.org/jbb2013/>. (Cited on pages 124, 159, 160)
- [SPEC 2013c] Standard Performance Evaluation Corporation (SPEC). SPECjbb 2013 Design Document. Jan. 2013. URL: <http://spec.org/jbb2013/docs/designdocument.pdf>. (Cited on pages 160, 257, 258)
- [SPEC 2013d] Standard Performance Evaluation Corporation (SPEC). SPECjbb 2013 User’s Guide. Jan. 2013. URL: <http://spec.org/jbb2013/docs/userguide.pdf>. (Cited on pages 160, 257, 258)

- [Spinczyk et al. 2002] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In: *Proceedings of the Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*. Australian Computer Society, Inc., Jan. 2002, pages 53–60. (Cited on page 65)
- [Spring] Spring Framework Reference Documentation. SpringSource. URL: <http://www.springsource.org/>. (Cited on page 64)
- [Srivastava and Eustace 1994] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*. ACM, June 1994, pages 196–205. (Cited on page 60)
- [Stantchev 2009] V. Stantchev. Performance evaluation of cloud computing offerings. In: *Proceedings of the 3rd International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP'09)*. IEEE Computer Society, Oct. 2009, pages 187–192. (Cited on page 81)
- [Subraya and Subrahmanya 2000] B. M. Subraya and S. V. Subrahmanya. Object driven performance testing of web applications. In: *Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS 2000)*. IEEE Computer Society, Oct. 2000, pages 17–26. (Cited on page 29)
- [Sweeney et al. 2004] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In: *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*. USENIX, May 2004, pages 57–72. (Cited on pages 51, 54, and 283)
- [Sydor 2010] M. J. Sydor. APM Best Practices: Realizing Application Performance Management. Apress, Dec. 2010. (Cited on page 47)
- [Szyperski et al. 2002] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. 2nd edition. Addison-Wesley, Nov. 2002. (Cited on page 66)

Bibliography

- [Tempero et al. 2010] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In: *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC '10)*. IEEE Computer Society, Dec. 2010, pages 336–345. (Cited on pages 155 and 262)
- [Thompson et al. 2011] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Technical paper. LMAX, May 2011. URL: <http://lmax-exchange.github.io/disruptor/>. (Cited on page 206)
- [Tichy 1998] W. F. Tichy. Should computer scientists experiment more? *IEEE Computer* 31.5 (May 1998), pages 32–40. (Cited on pages 3, 37, 38)
- [Tichy 2014] W. F. Tichy. Ubiquity symposium: The science in computer science – Where’s the science in software engineering? *Ubiquity* 2014.March (Mar. 2014), 1:1–1:6. (Cited on pages 3, 37, 38, and 81)
- [Tichy and Padberg 2007] W. F. Tichy and F. Padberg. Empirische Methodik in der Softwaretechnik im Allgemeinen und bei der Software-Visualisierung im Besonderen. German. In: *Proceedings of the Software Engineering 2007 - Beiträge für Workshops (SE 2007)*. Gesellschaft für Informatik, Mar. 2007, pages 211–222. (Cited on page 81)
- [Treibig et al. 2012] J. Treibig, G. Hager, and G. Wellein. Performance patterns and hardware metrics on modern multicore processors: Best practices for performance engineering. In: *Proceedings of the 5th Workshop on Productivity and Performance (PROPER 2012) at Euro-Par 2012*. Springer, Aug. 2012, pages 451–460. (Cited on page 51)
- [Trümper et al. 2012] J. Trümper, S. Voigt, and J. Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In: *Proceedings of the 2nd International Workshop on Software Engineering for Embedded Systems (SEES 2012)*. IEEE Computer Society, June 2012, pages 58–64. (Cited on page 287)
- [Utting and Legeard 2007] M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach. Morgan-Kaufmann, Mar. 2007. (Cited on pages 33, 34)

- [Van Hoorn 2014] A. van Hoorn. Model-Driven Online Capacity Management for Component-Based Software Systems. Kiel Computer Science Series 2014-6. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, Oct. 2014. (Cited on pages 69, 73, and 156)
- [Van Hoorn et al. 2008] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In: *Performance Evaluation – Metrics, Models and Benchmarks: Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW 2008)*. Springer, June 2008, pages 124–143. (Cited on pages 29, 107, 155, 156)
- [Van Hoorn et al. 2009a] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In: *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010 (WUP 2009)*. ACM, Apr. 2009, pages 41–44. (Cited on page 156)
- [Van Hoorn et al. 2009b] 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. (Cited on pages 6, 69, 73, 88, 90, 125, 130, 191, 270, and 285)
- [Van Hoorn et al. 2009c] A. van Hoorn, M. Rohr, and W. Hasselbring. Engineering and continuously operating self-adaptive software systems: Required design decisions. In: *Proceedings of the 1st Workshop of the GI Working Group Long-Living Software Systems (L2S2): Design for Future*. CEUR Workshop Proceedings, Oct. 2009, pages 52–63. (Cited on page 46)
- [Van Hoorn et al. 2011] A. van Hoorn, H. Knoche, W. Goerigk, and W. Hasselbring. Model-driven instrumentation for dynamic analysis of legacy software systems. In: *Proceedings of the 13. Workshop Software-Reengineering (WSR 2011)*. GI, May 2011, pages 26–27. (Cited on pages 65 and 69)

Bibliography

- [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)*. ACM, Apr. 2012, pages 247–248. (Cited on pages 2, 7, 69, 250, 262–264)
- [Van Solingen and Berghout 1999] R. van Solingen and E. Berghout. The Goal/Question/Metric Method. McGraw-Hill, Jan. 1999. (Cited on page 82)
- [Vieira et al. 2012] M. Vieira, H. Madeira, K. Sachs, and S. Kounev. Resilience Benchmarking. In: *Resilience Assessment and Evaluation of Computing Systems*. Springer, Oct. 2012, pages 283–301. (Cited on pages 3, 33, 38, 101, 105, 121, and 281)
- [Vierhauser et al. 2014] M. Vierhauser, R. Rabiser, P. Grünbacher, C. Danner, S. Wallner, and H. Zeisel. A flexible framework for runtime monitoring of system-of-systems architectures. In: *Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture (WICSA 2014)*. IEEE Computer Society, Apr. 2014, pages 57–66. (Cited on page 291)
- [Visser 2005] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation* 40.1 (July 2005), pages 831–873. (Cited on page 60)
- [VisualVM] Java VisualVM tool. Oracle Corporation. URL: <http://visualvm.java.net>. (Cited on pages 55, 56, and 60)
- [Viswanathan and Liang 2000] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal* 39.1 (Jan. 2000), pages 82–95. (Cited on pages 45 and 54)
- [Vlachos et al. 2010] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*. ACM, Mar. 2010, pages 271–284. (Cited on page 283)

- [Vokolos and Weyuker 1998] F. I. Vokolos and E. J. Weyuker. Performance testing of software systems. In: *Proceedings of the 1st International Workshop on Software and Performance (WOSP '98)*. ACM, Nov. 1998, pages 80–87. (Cited on pages 3, 29, 33, 104, 113, and 116)
- [Wallace and Hazelwood 2007] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. ACM, Mar. 2007, pages 209–220. (Cited on page 286)
- [Waller 2013] J. Waller. Benchmarking the Performance of Application Monitoring Systems. Technical report TR-1312. Department of Computer Science, Kiel University, Germany, Nov. 2013. (Cited on pages 9, 80, 88, 100, 130, and 164)
- [Waller 2014a] J. Waller. Benchmark for: Performance Benchmarking of Application Monitoring Frameworks. Aug. 2014. DOI: 10.5281/zenodo.11515. (Cited on pages 131, 145, and 301)
- [Waller 2014b] J. Waller. Data for: Performance Benchmarking of Application Monitoring Frameworks. Aug. 2014. DOI: 10.5281/zenodo.11425. (Cited on pages 157, 159, 161, 182, 190, 235, 247, 251, 253, 255–257, 259, 260, 275, and 301)
- [Waller and Hasselbring 2012a] J. Waller and W. Hasselbring. A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring. In: *Multicore Software Engineering, Performance, and Tools (MSEPT)*. Springer, June 2012, pages 42–53. (Cited on pages 7, 10, 88, 90, 91, 130, 180, 191, 193–195, and 199)
- [Waller and Hasselbring 2012b] J. Waller and W. Hasselbring. Data for: A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. June 2012. DOI: 10.5281/zenodo.7619. (Cited on pages 191, 194, and 301)
- [Waller and Hasselbring 2012c] J. Waller and W. Hasselbring. Benchmark for: A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. June 2012. DOI: 10.5281/zenodo.7620. (Cited on pages 191 and 194)

Bibliography

- [Waller and Hasselbring 2013a] J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (KPDays 2013)*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68. (Cited on pages 8, 88, 90, 92, 100, 102, 125, 130, 180, 182, 184, 186, 187, and 189)
- [Waller and Hasselbring 2013b] J. Waller and W. Hasselbring. Data for: A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. Nov. 2013. DOI: 10.5281/zenodo.7615. (Cited on pages 125, 126, 182, 188, 190, and 301)
- [Waller and Hasselbring 2013c] J. Waller and W. Hasselbring. Benchmark for: A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. Nov. 2013. DOI: 10.5281/zenodo.7616. (Cited on pages 125, 182, and 190)
- [Waller et al. 2013] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and W. Hasselbring. SynchroVis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. IEEE Computer Society, Sept. 2013, pages 1–4. (Cited on pages 7, 10, 164, 166, and 276)
- [Waller et al. 2014a] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 46–69. (Cited on pages 9, 88, 90, 100, 130, 180, 200, 202, 204, and 209)
- [Waller et al. 2014b] J. Waller, F. Fittkau, and W. Hasselbring. Data for: Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability. Nov. 2014. DOI: 10.5281/zenodo.11428. (Cited on pages 200 and 301)
- [Waller et al. 2015a] J. Waller, N. C. Ehmke, and W. Hasselbring. Data for: Including Performance Benchmarks into Continuous Integration to Enable DevOps. 2015. DOI: 10.5281/zenodo.11409. (Cited on pages 220 and 301)

- [Waller et al. 2015b] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes* (2015). Submitted, pages 1–4. (Cited on pages 9, 180, 215, 216, and 219)
- [Wang et al. 2014] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A big data benchmark suite from internet services. In: *Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture (HPCA 2014)*. IEEE, Feb. 2014, pages 1–12. (Cited on page 105)
- [WDC-MARE] World Data Center for Marine Environmental Sciences. Alfred Wegener Institute for Polar and Marine Research. URL: <http://www.wdc-mare.org/>. (Cited on page 126)
- [Weicker 1990] R. P. Weicker. An overview of common benchmarks. *IEEE Computer* 23.12 (Dec. 1990), pages 65–75. (Cited on pages 35, 36, 39, 104, and 116)
- [Weiss et al. 2013] C. Weiss, D. Westermann, C. Heger, and M. Moser. Systematic performance evaluation based on tailored benchmark applications. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. ACM, Apr. 2013, pages 411–420. (Cited on pages 35, 105, 117, and 215)
- [Wert et al. 2015] A. Wert, H. Schulz, C. Heger, and R. Farahbod. AIM: Adaptable Instrumentation and Monitoring for automated software performance analysis. In: *International Conference on Performance Engineering (ICPE 2015)*. Submitted. 2015. (Cited on pages 291 and 301)
- [Westermann et al. 2010] D. Westermann, J. Happe, M. Hauck, and C. Heupel. The Performance Cockpit approach: A framework for systematic performance evaluations. In: *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*. IEEE Computer Society, Sept. 2010, pages 31–38. (Cited on page 29)

Bibliography

- [Wettel and Lanza 2007] R. Wettel and M. Lanza. Visualizing software systems as cities. In: *Proceedings of the 4th International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007)*. IEEE Computer Society, Apr. 2007, pages 92–99. (Cited on page 166)
- [Weyuker and Vokolos 2000] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering* 26.12 (Dec. 2000), pages 1147–1156. (Cited on page 29)
- [Wilson and Kesselman 2000] S. Wilson and J. Kesselman. Java platform performance: Strategies and tactics. Prentice Hall, June 2000. (Cited on pages 28 and 34)
- [Wohlin et al. 2006] C. Wohlin, M. Höst, and K. Henningsson. Empirical Research Methods in Web and Software Engineering. In: *Web Engineering*. Springer, Jan. 2006, pages 409–430. (Cited on page 81)
- [Woodside et al. 2007] C. M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In: *International Conference on Software Engineering, Workshop on the Future of Software Engineering (FOSE 2007)*. IEEE Computer Society, May 2007, pages 171–187. (Cited on pages 2, 24, 25, 28, 45, and 48)
- [Wulf 2010] C. Wulf. Runtime Visualization of Static and Dynamic Architectural Views of a Software System to identify Performance Problems. Bachelor thesis. Department of Computer Science, Kiel University, Germany, Mar. 2010. (Cited on pages 7, 10, and 166)
- [Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework. In: *Proceedings of the Symposium on Software Performance: Joint Descartes/Kieker/Palladio Days (SoSP 2014)*. University of Stuttgart, Technical Report Computer Science No. 2014/05, Nov. 2014, pages 70–82. (Cited on page 300)
- [Xu et al. 2011] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM, June 2011, pages 270–282. (Cited on page 54)

- [ZENODO] ZENODO — Research. Shared. European Organization for Nuclear Research (CERN). URL: <https://zenodo.org/>. (Cited on page 126)
- [Zhao et al. 2010] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: Pipelined Profiling and Analysis on multicore systems. *ACM Transactions on Architecture and Code Optimization* 7.3 (Dec. 2010), 13:1–13:29. (Cited on page 283)
- [Zhou et al. 2014] J. Zhou, Z. Chen, H. Mi, and J. Wang. MTracer: A trace-oriented monitoring framework for medium-scale distributed systems. In: *Proceedings of the 8th International Symposium on Service Oriented System Engineering (SOSE 2014)*. IEEE Computer Society, Apr. 2014, pages 266–271. (Cited on page 283)
- [Zhu et al. 2009] J. W. Zhu, P. G. Bridges, and A. B. Maccabe. Lightweight online performance monitoring and tuning with Embedded Gossip. *IEEE Transactions on Parallel and Distributed Systems* 20.7 (July 2009), pages 1038–1049. (Cited on page 283)
- [Zloch 2014] M. Zloch. Automatisierte Durchführung und Auswertung von Microbenchmarks in Continuous Integration Systemen. German. Bachelor thesis. Department of Computer Science, Kiel University, Germany, Mar. 2014. (Cited on pages 11, 35, 135, 144, 215, 216)

Monitoring of software systems provides insights into their dynamic behavior, helping to maintain their performance and availability during runtime. While many monitoring frameworks claim to have minimal impact on the performance, these claims are usually not backed up with a detailed performance evaluation determining the actual cost of monitoring.

This PhD thesis introduces a benchmark approach to measure the performance overhead of application-level monitoring frameworks. Extensive experiments demonstrate the feasibility and practicality of the approach and validate the benchmark results. The developed benchmark is available as open source software and the results of all experiments are available for download to facilitate further validation and replication of the results.



Jan Waller has received his diploma in computer science at Kiel University. Afterwards, he has been a Ph.D. student and researcher with the Software Engineering Group at Kiel University where he has worked on the Kieker monitoring framework. His primary research interest is software performance engineering. Of particular interest are benchmarking, performance testing, monitoring, and the evaluation of software systems.

The Kiel Computer Science Series (KCSS) is published by the Department of Computer Science of the Faculty of Engineering at Kiel University. The scope of this open access publication series includes dissertations, habilitation theses, and text books in computer science.

ISBN 978-3-7357-7853-6



9 783735 778536

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)



Software Engineering Group