Application Performance Monitoring: Trade-Off between Overhead Reduction and Maintainability

Jan Waller, Florian Fittkau, and Wilhelm Hasselbring

Department of Computer Science, Kiel University, Kiel, Germany
{jwa,ffi,wha}@informatik.uni-kiel.de

Abstract: Monitoring of a software system provides insights into its runtime behavior, improving system analysis and comprehension. System-level monitoring approaches focus, e.g., on network monitoring, providing information on externally visible system behavior. Application-level performance monitoring frameworks, such as Kieker or Dapper, allow to observe the internal application behavior, but introduce runtime overhead depending on the number of instrumentation probes.

We report on how we were able to significantly reduce the runtime overhead of the Kieker monitoring framework. For achieving this optimization, we employed micro-benchmarks with a structured performance engineering approach. During optimization, we kept track of the impact on maintainability of the framework. In this paper, we discuss the emerged trade-off between performance and maintainability in this context. To the best of our knowledge, publications on monitoring frameworks provide none or only weak performance evaluations, making comparisons cumbersome. However, our micro-benchmark, presented in this paper, provides a basis for such comparisons.

Our experiment code and data are available as open source software such that interested researchers may repeat or extend our experiments for comparison on other hardware platforms or with other monitoring frameworks.

1 Introduction

Software systems built on and around internal and external services are complex. Their administration, adaptation, and evolution require a thorough understanding of their structure and behavior at runtime. Monitoring is an established technique to gather data on runtime behavior and allows to analyze and visualize internal processes.

System monitoring approaches, such as Magpie [BIMN03] or X-Trace [FPK+07], are minimal invasive and target only network and operating system parameters. Although these approaches have the advantage of minimal performance impact, they are not able to provide a view of internal application behavior.

A solution to these limitations is application level monitoring, as provided by SPASSmeter [ES14], Dapper [SBB+10], or Kieker [vHWH12]. However, application-level monitoring introduces monitoring overhead depending on the number of monitored operations and the efficiency of the monitoring framework. For a detailed view of a software system's internal behavior, monitoring must be all-embracing, causing significant performance im-

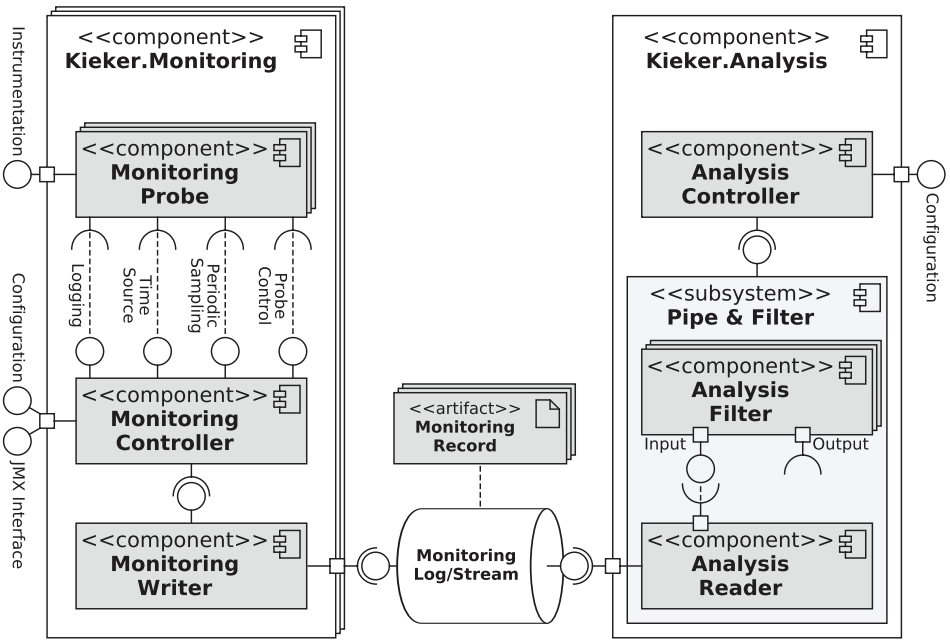

Figure 1: UML component diagram of a top-level view on the Kieker framework architecture

pact. While many monitoring frameworks are claiming to have minimal impact on the performance, these claims are often not backed up with a concise performance evaluation determining the actual cost of monitoring.

Two of our recent projects, ExplorViz [FWWH13] and iObserve [HHJ+13], use monitoring to evaluate distributed cloud systems. They need detailed information on the internal behavior of these systems. Therefore, they rely on a high-throughput monitoring framework which is easy to adapt and maintain. Furthermore, the framework must be able to transport logged data from the observed system components and aggregate it on a remote analysis system.

The Kieker monitoring framework has been developed in our research group over the past years. We evaluated this monitoring framework with a micro-benchmark to assess its capability as high-throughput framework and used a structured performance engineering approach to minimize its overhead while keeping the framework maintainable.

In this paper, we present our monitoring micro-benchmark MooBench and its application in a structured performance engineering approach 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 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 optimization useless.

In summary, our main contributions are:

- a micro-benchmark for monitoring frameworks,
- an example of a structured performance engineering activity for tuning the throughput of monitoring frameworks using a micro-benchmark,
- and an evaluation of the trade-off between performance and maintainability in a high-throughput monitoring framework.

Besides our approach and evaluation, we provide the research community with all sources to repeat and validate our benchmarks and our results in our download area.[1] These downloads include our results in a raw data format, statistical analyses, and generated diagrams.

The rest of the paper is organized as follows. In Section 2, we introduce software quality terms and the Kieker monitoring framework. Our benchmark developed for overhead evaluation of monitoring frameworks is presented in Section 3. Our performance tunings and our evaluations are described in Section 4, Finally, related work is discussed in Section 5, while we draw the conclusions and present future work in Section 6.

2 Foundations

This paper evaluates performance advances realized for the Kieker framework in conjunction with their impact on maintainability. For a better understanding of these two characteristics we provide a brief description of the used software quality attributes (Section 2.1) and an overview of the Kieker framework (Section 2.2) in this section.

2.1 Software Quality

The ISO/IEC 25010 [ISO11] standard, the successor of the well-known ISO/IEC 9126 [ISO01] standard, defines software quality over eight distinct characteristics and 23 subcharacteristics. In this paper we mainly focus on *performance efficiency* and *maintainability* supplemented by *functional completeness*, as these characteristics are used to evaluate different modifications of our monitoring framework.

Performance efficiency, as defined in [ISO11], comprises *time behavior*, *resource utilization*, and *capacity*, as sub-characteristics for the degree of requirement fulfillment. For our evaluation we focus on processing and response time in combination with throughput.

Maintainability is very important for a monitoring framework to be applicable in different software projects. It is characterized by *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability*. However, maintainability, especially if realized by modularization or generalization, can lead to a reduction in performance. Therefore, optimizations for these two characteristics are conflicting requirements.

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

2.2 Kieker Monitoring Framework

The Kieker[2] framework [vHWH12, vHRH+09] 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. Analysis plug-ins operate on the gathered data, and extract and visualize architectural models, augmented by quantitative observations.

In 2011, the Kieker framework was reviewed, accepted, and published as a recommended tool for quantitative system evaluation and analysis by multiple independent experts of the SPEC RG. Since then, the tool is also distributed as part of SPEC RG's tool repository.[3] Although originally developed as a research tool, Kieker has been evaluated in several industrial systems [vHRH+09].

Kieker Architecture

The Kieker framework provides components for software instrumentation, collection of information, logging of collected data, and analysis of this monitoring data. A top-level view of its architecture is presented in Figure 1. Each Kieker component is extensible and replaceable to support specific project contexts.

The general Kieker architecture is divided into two parts: The Kieker.Monitoring component for monitoring software systems and the Kieker.Analysis component for analysis and visualization of gathered data. These two components are connected by a Monitoring Log or Stream, decoupling the analysis from the monitoring and providing the means to perform the analysis on a separate system resulting in a reduced performance impact on the monitored system.

The focus of this paper is on the Kieker.Monitoring component, see the left side in Figure 1. It is realized by three subcomponents for data collection, monitoring control, and data delivery. Data collection is performed by MonitoringProbes which are technology dependent, as they are integrated into the monitored software system. The MonitoringController plays a central role in the monitoring side of Kieker. To handle record data, it accepts MonitoringRecords from probes and delivers them to the MonitoringWriter. Kieker comes with different MonitoringWriters addressing different needs for monitoring data handling, like logs and streams. Asynchronous MonitoringWriters contain a separate WriterThread to decouple the overhead of writing from the monitored software system. In this case, MonitoringRecords are exchanged and buffered via a configurable (blocking) queue. All interfaces of Kieker.Monitoring are well defined, allowing to add new components or to augment functionality in a transparent way.

[2] http://kieker-monitoring.net
[3] http://research.spec.org/projects/tools.html

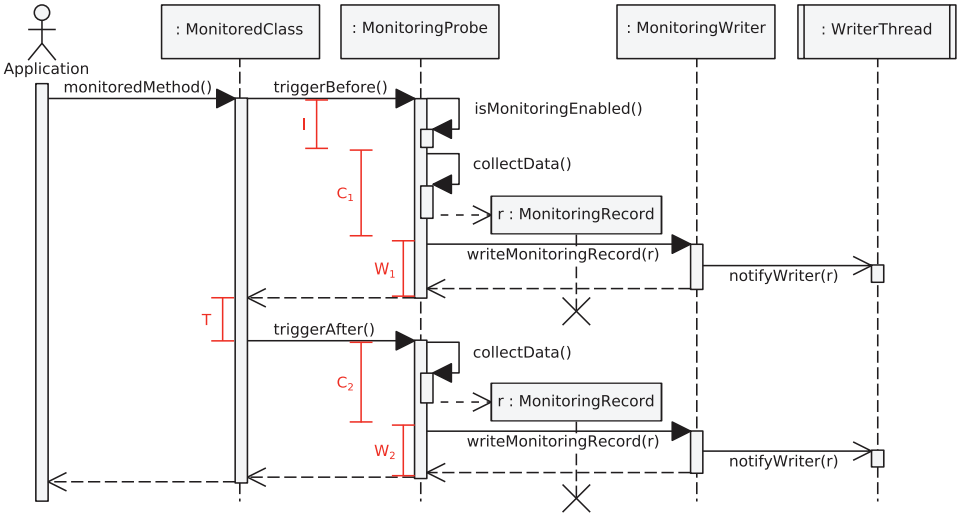

Figure 2: UML sequence diagram for method monitoring with the Kieker framework [WH13]

3 MooBench Benchmark

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, e. g., the performance of CPU, database management systems, or information retrieval algorithms [SEH03]. In this paper, we use the MooBench benchmark to measure the overhead caused by monitoring a method, to determine relevant performance tuning opportunities, and to finally evaluate the impact of the optimization in the Kieker framework.

In this section, we first introduce the notion of monitoring overhead and provide a partition into three causes of monitoring overhead (Section 3.1). Next, we provide a benchmark to measure these portions of monitoring overhead (Section 3.2).

3.1 Monitoring Overhead

A monitored software system has to share some of its resources with the monitoring framework (e. g., CPU-time or memory), resulting in the *probe effect* [Jai91]. The probe effect is the influence on the behavior of the system by measuring it. 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 (not monitored) execution of the program. In

Listing 1: MonitoredClass with monitoredMethod()

```
class MonitoredClass {
  ThreadMXBean threadMXBean = ManagementFactory
    .getThreadMXBean();
  long monitoredMethod(long methodTime, int recDepth) {
    if (recDepth > 1) {
      return this.monitoredMethod(methodTime, recDepth-1);
    } else {
      final long exitTime = this.threadMXBean
        .getCurrentThreadUserTime() + methodTime;
      long currentTime;
      do {
        currentTime = this.threadMXBean
          .getCurrentThreadUserTime();
      } while (currentTime < exitTime);
      return currentTime;
    }
  }
}
```

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.

3.1.1 Causes of Monitoring Overhead

In Figure 2, we present a simplified UML sequence diagram representation of the control flow for monitoring a method execution in the Kieker monitoring framework.Although this diagram is tailored to the Kieker framework, other application level monitoring frameworks usually have a similar general behavior. As annotated in red color in the diagram, we propose a separation into three different portions of monitoring overhead while monitoring an application [WH12].

These portions are formed by three common causes of application level monitoring overhead: (1) the instrumentation of the monitored system itself (I), (2) collecting data within the system, e. g., response times or method signatures, (C), and finally (3) either writing the data into a monitoring log or transferring the data to an analysis system (W). These three causes and the normal execution time of a monitored method (T) are further detailed below:

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 2 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.

I Before the code of the monitoredMethod() in the MonitoredClass is executed, the triggerBefore() part of the MonitoringProbe is executed. Within the probe, isMonitoringEnabled() determines whether monitoring is activated or deactivated for the monitoredMethod(). If monitoring is currently deactivated for the method, no further probe code will be executed and the control flow immediately returns to the monitoredMethod(). Besides these operations of the monitoring framework, I also includes any overhead caused by the used instrumentation. For instance, when performing aspect-oriented instrumentation with AspectJ, similar calls to our triggerBefore() are performed internally.

In Figure 2, I indicates the execution time of the instrumentation of the method including the time required to determine whether monitoring of this method is activated or deactivated.

C If monitoring of the monitoredMethod() is active, the MonitoringProbe will collect some initial data with its collectData() method, such as the current time and the method signature, and create a corresponding MonitoringRecord in memory (duration C_1). After this record is forwarded to the MonitoringWriter, the control flow is returned to the monitoredMethod().

When the execution of the actual code of the monitoredMethod() finished with activated monitoring, the triggerAfter() part of the MonitoringProbe is executed. Again, some additional data, such as the response time or the return values of the method, is collected and another corresponding MonitoringRecord is created in main memory. Finally, this record is forwarded to the MonitoringWriter, too.

In Figure 2, the time needed to collect data of the monitoredMethod() and to create the MonitoringRecords in main memory is $C = C_1 + C_2$.

W Each created and filled MonitoringRecord r is forwarded to a MonitoringWriter with the method writeMonitoringData(r). The MonitoringWriter in turn stores the collected data in an internal buffer, that is processed asynchronously by the WriterThread 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, 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. In other cases, with sufficient resources available, the additional overhead of the writer might be barely noticeable [WH12].

Listing 2: Benchmark thread calling monitoredMethod()

```
MonitoredClass mc; //initialized before
long start_ns, stop_ns;
for (int i = 0; i < totalCalls; i++) {
  start_ns = System.nanoTime();
  mc.monitoredMethod(methodTime, recDepth);
  stop_ns = System.nanoTime();
  timings[i] = stop_ns - start_ns;
}
```

In Figure 2, $W = W_1 + W_2$ is the amount of overhead caused by placing the monitoring data in an exchange buffer between the MonitoringWriter and the WriterThread as well as possibly the time of actually writing the collected monitoring data into a monitoring log or into a monitoring stream.

3.1.2 Measures of Monitoring Overhead

In this paper, we are focussed on improving the monitoring *throughput*, i. e., the number of MonitoringRecords sent and received per second, instead of the flat monitoring cost imposed per MonitoringRecord, i. e., the actual change in a monitored method's *response time*. This response time and the monitoring throughput are related: improving one measure usually also improves the other one. However, with asynchronous monitoring writers (as in the case of Kieker and 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 records between the MonitoringWriter and the WriterThread blocks, resulting in an increase of the monitored method's *response time*.

3.2 Our Monitoring Overhead Benchmark

The MooBench micro-benchmark has been developed to quantify the three portions of monitoring overhead under controlled and repeatable conditions. It is provided as open source software.[4] Although the benchmark and our experiments are originally designed for the Kieker framework, they can be adapted to other monitoring frameworks by exchanging the used monitoring component and its configuration.

[4] http://kieker-monitoring.net/MooBench

Figure 3: Benchmark engineering phases [WH13]

In order to achieve representative and repeatable performance statistics for a contemporary software system, benchmarks have to eliminate random deviations. For instance, software systems running on managed runtime systems, such as the Java VM (JVM), are hard to evaluate because of additional parameters influencing the performance, such as class loading, just-in-time compilation (JIT), or garbage collection [GBE07]. Therefore, a benchmark engineering process with guidelines to produce good benchmarks is required.

Our benchmark engineering process is partitioned into three phases [WH13] (see Figure 3). For each phase, a good benchmark should adhere to several common guidelines. For instance, it should be designed and implemented to be *representative* and *repeatable* [Gra93, Kou05, Hup09]. Similar guidelines should be followed during the execution and analysis/presentation of the benchmark [GBE07], i. e., using *multiple executions*, a *sufficient warm-up period*, and an *idle environment*. Furthermore, a rigorous *statistical analysis* of the benchmark results and a *comprehensive reporting* of the experimental setup are required in the analysis and presentation phase. Refer to [WH13] for an overview on our employed guidelines.

In the following, we detail the three benchmark engineering phases of our micro-benchmark for monitoring overhead. First, we describe our design and implementation decisions to facilitate representativeness and repeatability. Next, we give guidelines on the execution phase of our benchmark, focussing on reaching a steady state. Finally, we describe the analyses performed by our benchmark in the analysis & presentation phase.

3.2.1 Design & Implementation Phase

The architecture of our benchmark setup is shown in Figure 4. It consists of the Benchmark System running in a JVM, and an External Controller initializing and operating the system. The Benchmark System is divided into two parts: First, the Monitored Application, consisting of the Application instrumented by the Monitoring component. Second, the Benchmark, consisting of the Benchmark Driver with one or more active Benchmark Threads accessing the Monitored Application. For benchmarking the Kieker framework, the Monitoring is realized by the Kieker.Monitoring component (see Figure 1). For benchmarking of other monitoring frameworks, this component would be replaced accordingly.

For our micro-benchmark, the Monitored Application is a basic application core, consisting of a single MonitoredClass with a single monitoredMethod() (see Listing 1). This method has a fixed execution time, specified by the parameter methodTime, and can simulate recDepth nested method calls (forming one *trace*) within this allocated execution time. During the execution of this method, busy waiting is performed, thus fully utiliz-

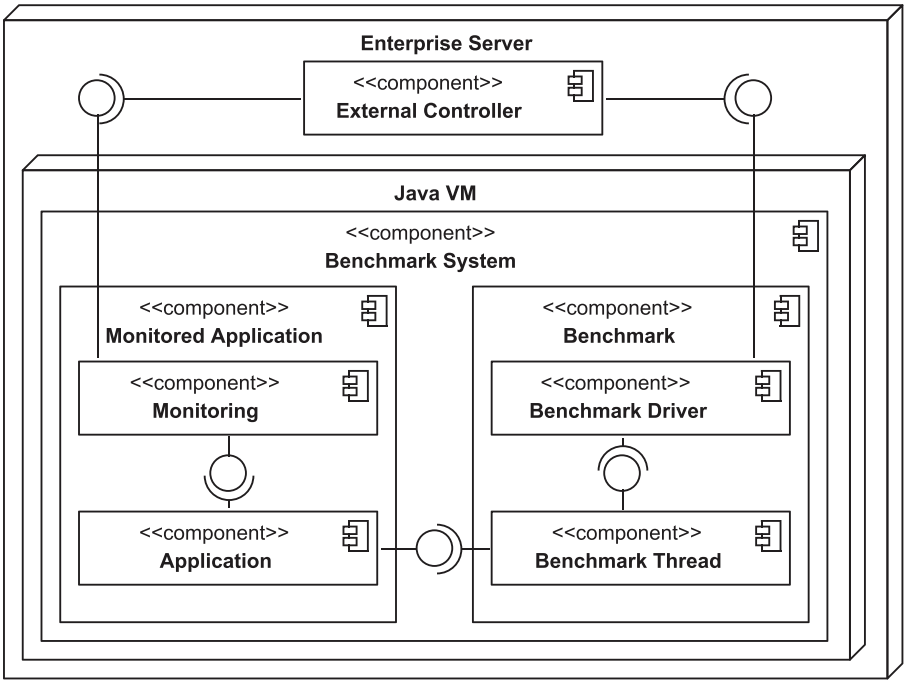

Figure 4: Architecture of the benchmark setup

ing the executing processor core. The loop of the method cannot be eliminated by JIT compiler optimizations, thus avoiding common pitfalls in benchmark systems. In order to correctly simulate a method using the CPU for a period of time despite the activities of other threads in the system, we use getCurrentThreadUserTime() of JMX's ThreadMXBean. The operating system and the underlying hardware of the Benchmark System have to provide a sufficient accuracy of the method in order to get stable and repeatable results. In the case of single threaded benchmarks on an otherwise unoccupied system we could use calls of System.nanoTime() or System.currentTimeMillis() instead.

The Benchmark Driver initializes the Benchmark System, then starts the required number of Benchmark Threads, and collects and persists the recorded performance data.

One or more concurrently executing Benchmark Threads call the monitoredMethod() while recording its response time with calls to System.nanoTime() (see Listing 2). Each thread is parameterized with a *total number of calls*, as well as the *method time* and the *recursion depth* of each call. The total number of calls has to be sufficiently large to include the warm-up period and a sufficient portion of the steady state. Execution time and recursion depth can be utilized to control the number of method calls the monitoring framework will monitor per second.

The External Controller calls the Monitored Application with the desired parameters and ensures that the Monitoring component is correctly initialized and integrated into the Monitored Application.

Each experiment consists of four independent runs, started by the external controller on a fresh JVM invocation. Each individual portion of the execution time is measured by one run (see T, I, C, and W in Figure 2). This way, we can incrementally measure the different portions of monitoring overhead as introduced in the previous Section 3.1. For instance, we can use this information to guide our optimizations.

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 a Monitoring Probe, that is deactivated for the monitoredMethod(). Thus, the duration $T + I$ is measured.
3. The third run adds the data collection with an activated Monitoring Probe without writing any collected data ($T + I + C$).
4. The fourth run finally represents the measurement of full monitoring with the addition of an active Monitoring Writer and an active Writer Thread ($T + I + C + W$).

In summary, this configurable benchmark design allows for repeatable measurements of monitoring overhead, that are representative for simple traces.

3.2.2 Execution Phase

The actual benchmark execution is controlled by the provided External Controller. Each independent experiment run to determine a portion of overhead can be repeated multiple times on identically configured JVM instances to minimize the influence of different JIT compilation paths. Furthermore, the number of method executions can be configured to ensure steady state.

In order to determine the steady state of experiment runs, the benchmark user can analyze the resulting data stream as a time series of averaged measured timings. Such a typical time series diagram for experiments with Kieker is presented in Figure 5. To visualize the warm-up phase and the steady state, invocations are bundled into a total of 1,000 bins. The benchmark calculates the mean values of each bin and uses the resulting values to generate the time series diagram.

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 to ensure a steady state in all cases (the grey-shaded part of Figure 5 illustrates this). We propose similar analyses with other monitoring frameworks, configurations, or hard- and software platforms to determine their respective steady states.

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. The regular spikes in the measured execution times, seen in Figure 5, correspond to additional garbage collections.

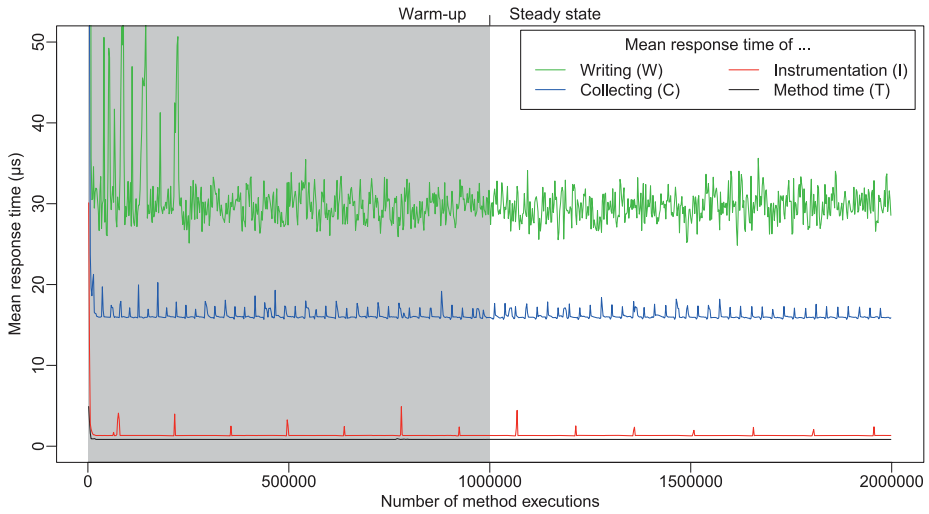

Figure 5: Time series diagram of measured timings

Finally, the benchmark user should ensure that the hard- and software environment, used to execute the experiments, is held idle during the experiments. Thus, there should be no perturbation by the execution of background tasks.

3.2.3 Analysis & Presentation Phase

In accordance with Georges et al. [GBE07], in the statistical analysis of the results of the benchmark runs, our benchmark 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 lower and upper quartile, as well as the 95% confidence interval of the mean value.

Of note is the calculation of the monitoring throughput within the benchmark. As mentioned in the benchmark design phase, our benchmark collects the response times of the MonitoredMethod(). These response times measurements are collected 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.

To facilitate repetitions and verifications of our experiments, the benchmark user has to provide a detailed description of the used configurations and environments.

3.3 Evaluations of MooBench

Our proposed micro-benchmark has already been evaluated with the Kieker framework in multiple scenarios. In [vHRH+09], we performed initial single-threaded measurements of the Kieker framework and demonstrated the linear scalability of monitoring overhead with

increasing recursion depths. In [WH12], we compared the monitoring overhead of several multi-threaded scenarios, of different writers, and of different hardware architectures with each other. In [vHWH12], we complemented additional results of our micro-benchmark by measurements of the SPECjEnterprise®2010 macro-benchmark. A detailed comparison of several different releases of Kieker as well as of different monitoring techniques and writers has been performed in [WH13]. Finally, in [FWBH13], we extended the benchmark to measure the additional overhead introduced by an online analysis of monitoring data concurrent to its gathering.

In order to broaden our evaluation basis, we intend to perform similar measurements with further available monitoring framework. Furthermore, we plan to validate our results by performing similar measurements with additional, more extensive benchmark suites and macro-benchmarks. Finally, it is of interest to compare further hard- and software configurations, e. g., different heap sizes, JVMs, or platforms as well as, e. g., different instrumentation techniques.

4 Overhead Reduction and its Impact on Maintainability

This section provides an evaluation of the monitoring overhead of Kieker with the MooBench micro-benchmark. The benchmark 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 how micro-benchmarks can be used to steer a structured performance engineering approach.

In the rest of the section, we first provide a description of our experimental setup (Section 4.1) to enable repeatability and verifiability for our experiments. Next, we describe our base evaluation of the Kieker framework without any tunings (Section 4.2). The next four sections (4.3 –4.6) describe our incremental performance tunings (PT). Finally, we discuss threats to the validity of our conducted experiments (Section 4.7).

4.1 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 Java-VM.

In our experiments, we use modified versions of Kieker 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 and to the prepared experimental configurations and finally to all results of our experiments are available online.[5]

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

Table 1: Throughput for basis (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 176.5k	757.6k	63.2k	16.6k
95% CI	± 25.9k	± 5.5k	± 0.1k	± 0.02k
Q_1	1 189.2k	756.6k	63.0k	16.2k
Median	1 191.2k	765.9k	63.6k	16.8k
Q_3	1 194.6k	769.8k	63.9k	17.2k

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 blocking queue with 10,000 entries to synchronize the communication between the MonitoringWriter and the WriterThread. The employed TCP writer uses an additional buffer of 64 KiB to reduce network accesses. 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 monitoredMethod() 20,000,000 times on each run with a configured methodTime of 0 µs and a stack depth of ten. We discard the first half of the measured executions as warm-up and use the second half as steady state executions to determine our results.

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*.

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.

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

However, the benchmark can be adapted to other scenarios, such as using replay compilation [GEB08] to avoid JIT compiler influence, a comparison of different JVMs [EGDB03], varying heap sizes [BGH+06], or different hardware combinations.

4.2 Base Evaluation

In this section, we present the results of our base evaluation of Kieker. We use the Kieker 1.8 code without the performance tunings mentioned in the following sections. The results from this base evaluation are used to form a baseline for our tuning experiments.

Table 2: Throughput for PT1 (traces per second)

	No instr.	Deactiv.	Collecting	Writing
Mean	1 190.5k	746.3k	78.2k	31.6k
95% CI	± 4.1k	± 4.1k	± 0.1k	± 0.1k
Q_1	1 191.0k	728.1k	78.3k	28.1k
Median	1 194.1k	756.6k	78.5k	32.5k
Q_3	1 195.1k	763.7k	78.7k	34.7k

The overhead of writing monitoring data (W) is measured in the fourth experiment run of each experiment. Writing requires the transport of the produced MonitoringRecords out of the probe into a remote site via a network connection.

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. Per method execution 848 bytes are transmitted.

Experimental Results & Discussion

The results of our base evaluation are presented in Table 1 and the response times for the four partitions of monitoring overhead of the TCP writer are shown in Figure 6.

For the uninstrumented benchmark system (first experiment run to measure the method time (T)), we measured an average of 1,176.5 k traces per second. Adding deactivated Kieker probes (second experiment run to measure ($T+I$)) resulted in an average of 757.6 k traces per second.

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 ($T+I+C+W$)) exceeds the writer thread's capacity (see Section 3.1.2), thus causing blocking behavior.

The TCP writer for online analysis produces 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.

4.3 PT1: Caching & Cloning

As is evident by our analysis of the base evaluation, i. e., by the response times presented in Figure 6 and by the throughputs in Table 1, 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:

First, 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 cashed in HashMaps.

Second, 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.

Third, 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.

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.

Experimental Results & Discussion

The resulting throughput is visualized in Table 2 and the response time is shown in Figure 6. As can be expected, the changes in the uninstrumented benchmark and with deactivated probes are not significant. However, when adding data collection (C) we measured an increase of 15 k additional traces per second compared to our previous experiments. Finally, the TCP writer's throughput almost doubled while still providing very stable measurements (small confidence interval).

The improvement discussed in this section will be used in Kieker because of their minimal influence on the maintainability and the great improvements in the area of performance efficiency.

4.4 PT2: Inter-Thread Communication

From PT1 we conclude that the queue is saturated and the monitoring thread waits for a free space in the queue, i. e., the writer thread to finish its work. Otherwise, the monitoring thread should be able to pass the records into the queue and proceed with the method. Therefore, our target is to decrease the writing response time. In this step, we want to achieve this goal by optimizing the communication between monitoring and writer thread.

Internal communication is presently modeled with the Java ArrayBlockingQueue class. It is used to pass monitoring records from the monitoring thread to the writer thread. To improve the inter-thread communication performance, we use the disruptor framework,[6] which provides an efficient implementation for inter-thread communication. It

[6] http://lmax-exchange.github.io/disruptor/

Table 3: 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

utilizes a ringbuffer to provide a higher throughput than, for instance, the Java ArrayBlockingQueue class.

The evaluation uses Kieker 1.8 with the designated disruptor ringbuffer. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting throughput is visualized in Table 3 and the response time is shown in Figure 6. Again, the no instrumentation and deactivated run is roughly the same from the previous experiments. Since we did not improve the collecting phase of Kieker, the response time and throughput for this part is approximately the same as in PT1. The writing response time decreased from 16.35 µs to 6.18 µs which is a decrease of about 62%. 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 (77%).

Therefore, our goal of decreasing the writing response time is achieved. The disruptor framework speeds up the transfer into the buffer and also the reading from the buffer resulting in the decreasing of the response time in the writing phase. However, the response time of the writing phase still suggests that the monitoring thread is waiting for the buffer to get an empty space for record passing.

The maintainability of Kieker is unharmed by this optimization because the disruptor framework can be abstracted to provide a single put method into the buffer and the writer components can be easily rewritten to the designated observer pattern of the disruptor framework. It even improves the maintainability of the code since the thread management and message passing is conducted by the disruptor framework. Therefore, this improvement will be used in Kieker.

4.5 PT3: Flat Record Model

As discussed in PT2, the monitoring thread is waiting for the writer thread to finish its work. Therefore, we want to further decrease the writing response time in this optimization. In contrast to PT2, we aim for reducing the work which has to be conducted by the writer thread.

Table 4: 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

The main work, which has to be conducted by the TCP writer, is the serialization of incoming MonitoringRecord objects into a byte representation and writing this byte representation into a ByteBuffer. The root of the object serialization challenge is the creation of the MonitoringRecords which are only created to pass them to the writer thread which serializes them. No calculation is conducted with those records. Therefore, we can also write the required monitoring information directly into a ByteBuffer and pass it into the writer thread. This optimization imposes several performance advantages. First, the object creation and thus garbage collection for those objects is avoided. Furthermore, the ByteBuffers passed into the disruptor framework are fewer objects than passing thousands of MonitoringRecord objects. Since less work has to be conducted by the disruptor framework, the inter-thread communication should be even faster.

For the evaluation we use Kieker 1.8 with the enhancement of directly writing the monitoring information into a ByteBuffer. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting throughput is shown in Table 4 and the response time is visualized in Figure 6. The no instrumentation and deactivated phase are the same as in the previous experiments. In the collecting phase, the response time decreased from 11.44 µs to 7.23 µs and the throughput increased from 78.2k to 115.7k traces per second. The writing response time decreased from 6.18 µs to 0.2 µs. The average throughput for the writing phase increased from 56.0k to 113.2k traces per second.

The decrease in the writing response time is rooted in the reduction of the work of the writer thread. This work is reduced to sending the ByteBuffer to the network interface and therefore the ringbuffer does not fill up most of the time. The remaining 0.2 µs overhead for the writing phase is mainly found in the putting of the ByteBuffer into the ringbuffer. The collecting phase also became more efficient. The response time decreased since no MonitoringRecord objects are created and therefore less garbage collection takes place.

This improvement will not be used in Kieker because it is harder for the framework user to add own MonitoringRecord types with this optimization. If she wants to add a MonitoringRecord type, she would be forced to write directly into the ByteBuffer instead of just using object-oriented methods. Thus, the optimization would hinder the modularity and reusability of the code.

Table 5: 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

4.6 PT4: Minimal Monitoring Code

In PT3, we optimized the writing phase such that the monitoring thread does not need to wait for the writer thread. Now, about 80% of the time consumed for monitoring the software execution is spent in the collecting phase. Therefore, we try to optimize this phase. As a further optimization, we deleted anything not directly related to pure monitoring. For instance, we deleted interface definitions, consistence checks, and configurability. Furthermore, we provide only five hard coded types of MonitoringRecords.

For the evaluation we use a newly created project, which only includes minimal code for the monitoring. Again, we measure the record throughput per second and the average response time.

Experimental Results & Discussion

The resulting response time is visualized in Figure 6 and the throughput is shown in Table 5. The response time of the no instrumentation, deactivated, and writing phase are roughly equal to the previous experiments. In the collecting phase, the response time decreased and the throughput increased from 115.7k to 145.1k traces per second.

We attribute the decrease in the response time in the collecting phase mainly to the hard coding of the monitoring since less generic lookups must be made. Furthermore, the consistence checks had also an impact.

This improvement will not be used in Kieker because the monitoring tool now lacks important features for a framework, e.g., configurability and reusability.

4.7 Threats to Validity

In the experiments at least one core was available for the monitoring. If all cores were busy with the monitored application, the results could be different. We investigated this circumstance in [WH12].

Further threats involve our benchmark in general. Common threats to validity of micro-benchmarks are their relevance and systematic errors. Our benchmark bases on repeatedly calling a single method. However, by performing recursive calls, the benchmark is able

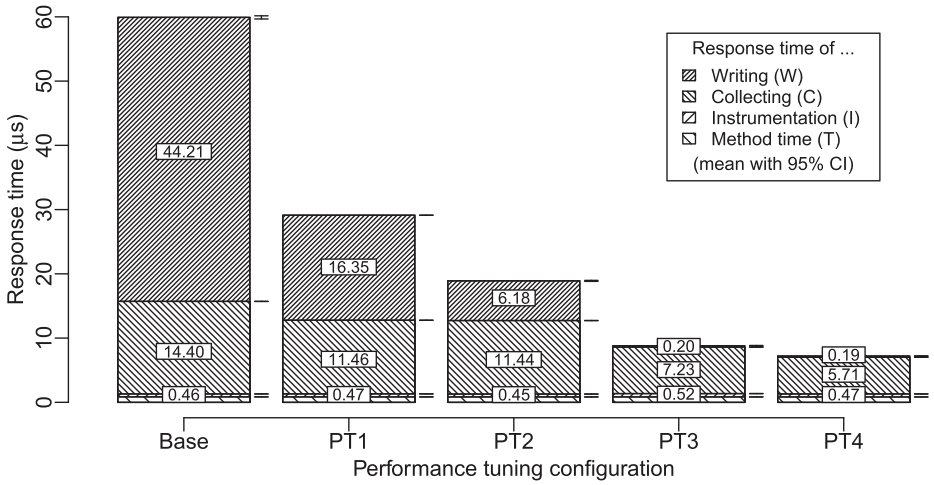

Figure 6: Overview of the tuning results in response time

to simulate larger call stacks. Additional comparisons of our results with more complex benchmarks or applications are required for validation. If the configured method execution time is negligible compared to the overhead, these comparison may be inappropriate. However, our experiments with different method execution times suggest that the results are still valid.

Further threats are inherent to Java benchmarks. 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. This has to be countered by further validation of our benchmark. 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.

5 Related Work

Monitoring is an important part of administration and operation software systems, especially in distributed systems. It allows to determine online or offline bottlenecks or other technical problems in a deployed software system or its neighboring systems which affect the operation. To provide monitoring capabilities, approaches have targeted different levels in the system. Early approaches monitor systems through network communication and services. X-Trace [BIMN03], for example, aggregates such monitoring information and builds system traces on that data. It is minimal invasive for the application, but cannot provide detailed information on machine or application internal behavior. Magpie [FPK+07]

is a framework for distributed performance monitoring and debugging which augments network information with host level monitoring of kernel activities, for example. Both approaches do not provide detailed information on their overhead, but data collection is provided by network infrastructure and the operating system. Therefore only the logging affects the overall system capacity. However, none of these approaches provided a detailed performance impact analyses.

Dapper [SBB+10] and SPASS-meter [ES14] provide, like Kieker, application level monitoring providing insights into internal behavior of applications and its components. Causing a significant impact on the overall performance of those software systems. To reduce the performance impact, Dapper uses sampling, which ignores traces during monitoring. The sampling mechanism of Dapper observes the system load and changes the sampling rate to limit the performance impact.

On systems, like search engines, the jitter of sampling may not have a big impact, as Sigelman et. al. [SBB+10] claim for their applications at Google. In such scenarios sampling can be an approach to balance the monitoring performance trade-off. However, in many scenarios, e.g., failure detection or performance forecasting, detailed monitoring data is necessary to produce a comprehensive view of the monitored system. Therefore, sampling, the reduction of probes, or limiting the scope of the observation to network or host properties is not always an option to limit performance overhead of monitoring. Therefore, we addressed the performance impact of the Kieker framework and minimized it.

Magpie provides a comparison of the impact of different observation methods in their scenario, but no detailed analysis of the overhead. Dapper and X-Trace address the issue of overhead as a characteristic of monitoring, which has to be considered when monitoring. But to the best of our knowledge, no other monitoring framework has been evaluated with a specialized monitoring benchmark targeting the overhead of monitoring itself.

6 Conclusions and Outlook

The paper presents our proposed micro-benchmark for monitoring frameworks. Furthermore, we introduce a structured performance engineering activity for tuning the throughput of the monitoring framework Kieker. The evaluation was conducted by utilizing our monitoring overhead benchmark MooBench. It demonstrates that high-throughput can be combined with maintainability to a certain degree. Furthermore, it shows that our TCP writer is the fastest writer and thus applicable for online analysis, which is required for our upcoming ExplorViz [FWWH13] project.

Our performance tuning shows an upper limit for the monitoring overhead in Kieker. However, in productive environments, monitoring probes are usually configured in a way to stay below the system's capacity. Furthermore, the execution time of most methods (T) is larger than the measured 0.85 µs. Refer to our previous publications for measurement of the lower limit of monitoring overhead [WH12, vHWH12, vHRH+09]. Note, that these previous measurements usually employed a stack depth of one compared to our used stack depth of ten.

As future work, we intend to reduce the impact of deactivated probes by utilizing other instrumentation frameworks than AspectJ, for instance, DiSL [MZA+12]. Furthermore, we will evaluate whether a generator can handle the monitoring record byte serialization and thus PT3 might become applicable in Kieker without loosing maintainability and usability for the framework users. In addition, we plan to measure Kieker with multi-threaded versions of our monitoring benchmark and test further configurations. We also plan to compare the benchmark results for Kieker to the resulting benchmark scores of other monitoring frameworks.

References

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 169–190, 2006.

[BIMN03] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS)*, pages 85–90, 2003.

[EGDB03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of 18th Conference on OO Programming Systems Languages and Applications (OOPSLA)*, pages 169–186, 2003.

[ES14] Holger Eichelberger and Klaus Schmid. Flexible Resource Monitoring of Java Programs. *Journal of Systems and Software (JSS-9296)*, pages 1–24, 2014.

[FPK+07] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked systems design & implementation (NSDI)*, pages 271–284, 2007.

[FWBH13] Florian Fittkau, Jan Waller, Peer Christoph Brauer, and Wilhelm Hasselbring. Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. In *Proceedings of the Syposium on Software Performance: Joint Kieker/Palladio Days (KPDays)*, pages 89–98, 2013.

[FWWH13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT)*, 2013.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. of 22nd Conf. on Object-oriented programming systems and applications (OOPSLA)*, pages 57–76, 2007.

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 367–384, 2008.

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

[HHJ+13] Wilhelm Hasselbring, Robert Heinrich, Reiner Jung, Andreas Metzger, Klaus Pohl, Ralf Reussner, and Eric Schmieders. iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems. Technical Report 1309, Kiel University, 2013.

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

[ISO01] Software Engineering – Product Quality – Part 1: Quality Model, 2001.

[ISO11] Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

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

[Kou05] Samuel Kounev. *Performance Engineering of Distributed Component-Based Systems – Benchmarking, Modeling and Performance Prediction*. PhD thesis, TU Darmstadt, Germany, 2005.

[MZA+12] Lukas Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Peter Tuma, and Zhengwei Qi. Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis. In *Proceedings of the 10th Asian Syposium on Programming Languages and Systems (APLAS)*, pages 256–263, 2012.

[SBB+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 74–83, 2003.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical Report 921, Kiel University, Germany, 2009.

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

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

[WH13] Jan Waller and Wilhelm Hasselbring. A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. In *Proceedings of the Syposium on Software Performance: Joint Kieker/Palladio Days (KPDays)*, pages 59–68, 2013.