Towards Solving the Challenge of Minimal Overhead Monitoring

David Georg Reichelt
d.g.reichelt@lancaster.ac.uk
Lancaster University in Leipzig /
Universität Leipzig
Leipzig, Saxony, Germany

Stefan Kühne
stefan.kuehne@uni-leipzig.de
Universität Leipzig
Leipzig, Saxony, Germany

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de
Kiel University
Kiel, Schleswig-Holstein, Germany

ABSTRACT

The examination of performance changes or the performance behavior of a software requires the measurement of the performance. This is done via probes, i.e., pieces of code which obtain and process measurement data, and which are inserted into the examined application. The execution of those probes in a singular method creates overhead, which deteriorates performance measurements of calling methods and slows down the measurement process. Therefore, an important challenge for performance measurement is the reduction of the measurement overhead.

To address this challenge, the overhead should be minimized. Based on an analysis of the sources of performance overhead, we derive the following four optimization options: (1) Source instrumentation instead of AspectJ instrumentation, (2) reduction of measurement data, (3) change of the queue and (4) aggregation of measurement data. We evaluate the effect of these optimization options using the MooBench benchmark. Thereby, we show that these optimizations options reduce the monitoring overhead of the monitoring framework Kieker. For MooBench, the execution duration could be reduced from 4.77 μs to 0.39 μs per method invocation on average.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Software and its engineering** → **Software performance**; *Software maintenance tools.*

KEYWORDS

software performance engineering, benchmarking, performance measurement, monitoring overhead

ACM Reference Format:
David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2023. Towards Solving the Challenge of Minimal Overhead Monitoring. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584851

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584851

1 INTRODUCTION

The examination of performance changes, performance anomalies or the general performance behavior of a software usually requires the measurement of the performance.[1] The measurement in production environments is called monitoring [21].[2] Performance monitoring is possible using two options: (1) using instrumentation, i.e., by inserting measurement probes into the source code, or (2) using sampling, i.e., by frequently obtaining the current state of the application from the outside. Measurement probes are pieces of code which obtain and process measurement data, and which are inserted into the examined application. Frequently obtaining the current state of the application requires a suitable technical interface to do so, e.g. the JVM provides the tool `jstack`, which obtains the current stack of an application.

Both, instrumentation and sampling, create overhead. When measuring the performance, this overhead reduces the accuracy of the measurements itself and slows down the measurement process. If the accuracy of measurements decreases, this is often resulting in increased standard deviation of the measurement values. The detectability of small performance changes depends on the effect size, i.e., the relation between performance change size and standard deviation. Therefore, especially detection of small performance changes require minimal performance monitoring overhead. This is especially important when aiming for the identification of small performance changes [17]. Hence, an important *challenge* for performance monitoring is the reduction of the measurement overhead.

To address this challenge, the main options are: (1) the optimization of sampling intervals and configuration, (2) the configuration of adaptive monitoring supported by adaptive instrumentation and (3) the optimization of the measurement process itself. In this work, we present a first step addressing the challenge of minimal monitoring overhead by reducing the overhead of the measurement process itself (option 3). We demonstrate these overhead reductions using the application performance monitoring framework Kieker [6].

Based on an analysis of the sources of performance overhead in Kieker, we derive the following four optimizations: (1) Source code instrumentation, i.e., instrumenting the source code of the software instead of using AspectJ, (2) Reduced data storage, i.e., only measuring the duration and the method name, (3) Queue exchange, i.e., use a `CircularFifoQueue` instead a `LinkedBlockingQueue`,

[1] Alternative approaches require data sources with similarly detailed resolution, e.g. using data from existing logging [11] requires fine-grained log statements.
[2] Nowadays, the detailed measurement in production environments is often also called *observability*, since tool marketing wants to emphasize the comprehensiveness of obtained data and possible analysis. We stick to the classical term *monitoring*.

and (4) Aggregated writing, i.e., writing the aggregated measured performance data instead writing the measurement values of every singular method call. We apply the empirical standard of benchmarking [5] to compare the effect of these overhead reductions, and find that each optimization is able to significantly reduce the performance overhead. Using all overhead reductions, we can reduce the overhead from 4.77 μs to 0.39μs per method invocation.

The remainder of this paper is organized as follows: First, we give an introduction of the Kieker framework and the MooBench benchmark. Afterwards, we describe our optimizations for the Kieker probes. Subsequently, we provide measurement results of the overheads using MooBench. Our results are then compared to related work. Finally, we give a summary and outlook.

2 FOUNDATIONS

In this section, we describe the monitoring framework Kieker and the monitoring overhead benchmark MooBench.

2.1 Kieker

A variety of tools is able to perform performance monitoring, including Kieker, SPASSmeter [3], inspectIT,[3] OpenTelemetry[4] and Dynatrace APM.[5] The OpenAPM[6] initiative gives an overview over existing monitoring tools and their interoperability.

Two central components of a monitoring tool are the *library* and the *agent*. One of both or both are added to the execution of a software in order to obtain monitoring data. These monitoring data might be at infrastructure, application or business level; in this work, we focus on *application* monitoring. Application monitoring can be done either using instrumentation, i.e., adding monitoring code to the measured system, or using sampling, i.e., obtaining measurement data at given times by technical interfaces.

Sampling in the JVM is only able to obtain monitoring data at safepoints [7]. Therefore, it stops the execution of all threads and might obtain inaccurate measurement data. To avoid these pitfalls, we focus on monitoring using instrumentation. According to the MooBench benchmark, Kieker is the framework with the lowest monitoring overhead when compared to OpenTelemetry and inspectIT [16].

The measurement process of Kieker is depicted in Figure 1. At first, the instrumentation is executed, which is regularly done using AspectJ. This adds a monitoring probe, which creates the monitoring records, e.g., one instance of `OperationExecutionRecord` for every method invocation. The records contain metadata of the execution, e.g., execution order index and execution stack size, which enable reconstruction of the call tree afterwards. These records are passed to the queue afterwards, which is a `LinkedBlockingQueue` by default. In parallel to the program main thread, a writer is executed in a parallel thread. The writer thread awaits new records and writes the records to their destination. By default, this is a `FileWriter` writing the monitoring records to the hard disk.

[3]https://www.inspectit.rocks/
[4]https://opentelemetry.io/
[5]https://www.dynatrace.com/de/platform/application-performance-monitoring/
[6]https://openapm.io/

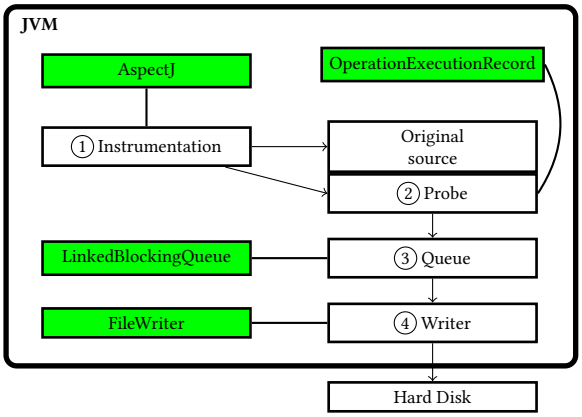

Figure 1: Monitoring Process in Kieker

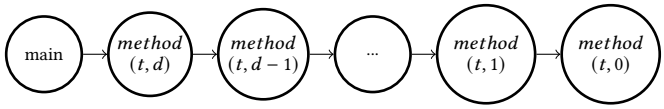

Figure 2: Call Tree of MooBench

2.2 MooBench

MooBench is a benchmark that aims for measuring the performance overhead of monitoring frameworks [22]. To measure the performance overhead, it calls a method recursively with a given call depth d. In the leaf node, it executes busy waiting until a specified amount of time t has passed. To avoid performance optimizations, the recursive method calls return the timestamp of the root method call. The call tree of this process is visualized in Figure 2.[7]

Performance measurement in Java is affected by non-determinism due to Just-in-Time Compilations, optimizations and garbage collections of the JVM. Therefore, a complex measurement process and a statistical analysis of measurements is required [4].

Moobench therefore contains scripts which automate the repetition of VM starts and benchmark iterations inside the VM, so performance differences can be detected in a statistically sound manner. Originally, it was built to monitor Kieker, inspectIT and SpassMETER [3]. For each of the supported frameworks, MooBench contains a definition of different measurement configurations, e.g., measuring the overhead of performance measurement in Kieker with serialization to hard disk or to a TCP receiver. This makes it possible to compare the performance of these variants. MooBench has been used in different execution environments [10]. Recently, it has been extended for measurement of OpenTelemetry [16].

3 PERFORMANCE OPTIMIZATIONS FOR KIEKER PROBES

All the parts of Kieker's monitoring process provide room for performance optimization: (1) To avoid the AspectJ overhead, **source code instrumentation** can be used, (2) to avoid metadata creation

[7]Original source code: https://github.com/kieker-monitoring/moobench/blob/ef9ca00259a8546e6fa9cdda47ac4cca3f116fe5/tools/benchmark/src/main/java/moobench/application/MonitoredClassSimple.java

steps, only the necessary data to create a **DurationRecord** can be obtained, (3) to decrease waiting times for queue inserts, a **CircularFifoQueue** can be used and (4) to decrease waiting times for data storage, only **aggregated data** can be stored. A summary of the optimizations is visualized in Figure 4. The details of these optimizations are described in the following subsections.

3.1 Source Code Instrumentation Instead AspectJ

Monitoring the program execution, e.g., measuring the execution time of methods, requires changes to the executed source. This can be done using instrumentation libraries, like ByteBuddy for OpenTelemetry[8] or AspectJ in Kieker.[9] Kiekers AspectJ instrumentation is configured via the `aop.xml`. It defines which aspect should be used, i.e., whether `OperationExecutionRecords`, capturing start and end time of a method execution, or `BeforeOperationRecords`, capturing only the start of a method cexecution, is used. When starting the application, the monitoring is started by passing the parameter `-javaagent:kieker-1.15.2-aspectj.jar` to the JVM.

Using instrumentation libraries creates overhead, because it creates `JoinPointImpl.proceed` calls to the stack trace. The overhead can be avoided by directly inserting the monitoring code into the monitored application. Therefore, we created the tool *kieker-source-instrumentation*[10] which automatically adds measurement code to all called methods. This requires two automated changes to the monitored application's source code: (1) Adding the required variables to the monitored class and (2) adding the monitoring source code to all methods that should be monitored.

The first step is necessary, since monitoring requires at least a reference to the `MonitoringController`, to pass the created monitoring record to the queue, and the currently used `TimeSource`, to get the current time. These are both singletons; however, obtaining the instances on the fly using `getInstance` creates overhead and should be avoided. Therefore, the source instrumentation creates `static final` fields for every class that is instrumented.

For the second step, the monitoring source code is inserted into the method. This always requires definition of the signature, determining start and end time of the method, creating a monitoring record and writing this record to the queue. Optionally, other metadata are obtained, like the execution stack size. All variables are created with a prefix, e.g. `_kieker`, so collisions with existing method names are avoided. A simplified example of instrumentation with determining the current stack size is depicted in Listing 1.

3.2 DurationRecord Instead of OperationExecutionRecord

For monitoring using the `OperationExecutionRecord`, the execution order index, the execution stack size, the current hostname and an id of the current session are stored. This requires storing the current count of executions and the current stack size. Additionally, the data are stored in the record and are written to the hard disk.

[8]https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/javaagent-extension-api/build.gradle.kts

[9]https://kieker-monitoring.readthedocs.io/en/1.15.2/getting-started/AspectJ-Instrumentation-Example.html#gt-aspectj-instrumentation-example

[10]https://github.com/kieker-monitoring/kieker-source-instrumentation

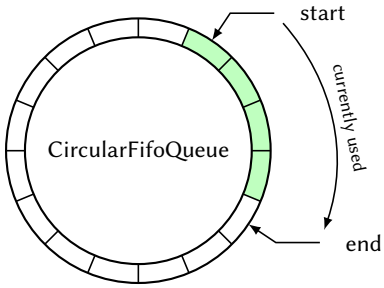

Figure 3: CircularFifoQueue Structure

Therefore, we created the `DurationRecord`, which is a minimal Kieker record that only stores the duration. To enable monitoring this record, we adapted the instrumentation process accordingly.

3.3 CircularFifoQueue Instead of LinkedBlockingQueue

Currently, Kieker uses a `LinkedBlockingQueue`. It is a singly linked list that stores a capacity. If there are more elements in the queue than the capacity, the queue blocks on every new insert. Blocking slows down the execution of the application thread significantly, and should therefore never happen. A downside of the `LinkedBlockingQueue` is, that it needs to create new queue elements, which requires reserving space, and link the new element by setting the pointer of the last queue element.

Using a `CircularFifoQueue`, the queue speed can be increased. The `CircularFifoQueue` is a ring buffer for queue elements. Its structure is visualized in Figure 3: It contains an array of a given size, the index of the current start element and the index of the current end element. If a new element is added, the element at the end index is set and the end index is increased (modulo the queue size). If an element is taken from the queue, the element at the start index is returned, it is set to null and the start index is increased (modulo the queue size). Thereby, no new memory needs to be allocated while handling the queue. The current implementation of Apache Commons Collections does not react if the queue is full, i.e., if all elements are used, the new elements overwrite old elements.

Another problem of the basic `CircularFifoQueue` is, that it is not directly usable in parallel. Since Kieker reading and writing is tone by different threads, this is necessary. Therefore, we use a tweaked version of the `CircularFifoQueue` which is synchronized.[11] To use it, we configured the queue using the Java properties that Kieker reads.

3.4 Storing Aggregated Data Instead of Method Executions

Every insertion into the queue and every write to the hard disk is time-consuming. Additionally, the execution time depends on the current state of the hard disk. Therefore, we reduce the stored data by only obtaining and storing aggregated data, e.g., the average

[11]Synchronized version of `CircularFifoQueue`: https://github.com/DaGeRe/KoPeMe/blob/main/kopeme-core/src/main/java/de/dagere/kopeme/collections/SynchronizedCircularFifoQueue.java

Listing 1: Example Instrumented Source

```
public void myMethod(){
  final String _kieker_signature = "public␣void␣net.kieker.Class.myMethod()";
  [...]
  final int _kieker_ess;
  long _kieker_traceId = _kieker_controlFlowRegistry.recallThreadLocalTraceId();
  [...]
  if (_kieker_traceId == -1) {
    _kieker_ess = _kieker_controlFlowRegistry.recallAndIncrementThreadLocalESS();
    [...]
  } else {
    _kieker_ess = _kieker_controlFlowRegistry.recallAndIncrementThreadLocalESS();
    [...]
  }
  final long _kieker_tin = C0_0._kieker_TIME_SOURCE.getTime();
  try {
    // Execute method original code
  } finally {
    final long _kieker_tout = C0_0._kieker_TIME_SOURCE.getTime();
    _kieker_controller.newMonitoringRecord(
      new OperationExecutionRecord(_kieker_signature, _kieker_tin, _kieker_tout, ...));
}
```

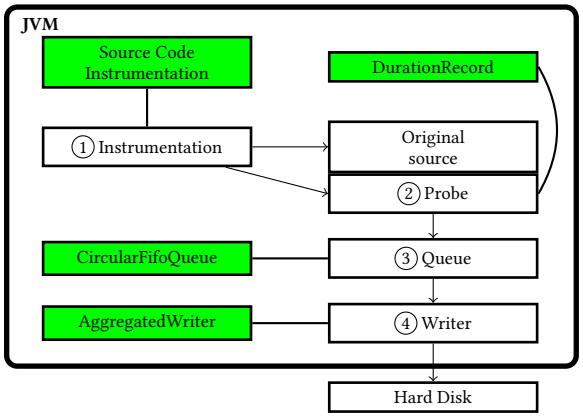

Figure 4: Possible Monitoring Optimizations

of 1000 method execution durations, instead of every method execution duration. This is achieved by adding two counters to each examined class, one which contains the sum execution time and one which contains the count of executions. For every iteration, the current execution time is added to the sum and the counter is increased by one. If the counter is equal to the parameterized execution count, for example 1000, a monitoring record is created. Additionally, the sum and the counter are reset.

This optimization makes it impossible to examine fine-grained properties of method invocations, e.g., the frequency of outliers. Therefore, it can only be applied if performance behaviour in the long run should be examined. For example, if a method became slower on 2 % on average, this could be examined using aggregated data storage. If outliers occur every 500 invocations, that cause an average slowdown of 0.5 %, this cannot be examined with aggregated data storage; in this case, the fine-grained data are required.

4 BENCHMARKING RESULTS OF THE PERFORMANCE OPTIMIZATIONS

To measure the overhead of our performance optimizations, we used the MooBench benchmark and executed the optimizations individually and the combination of them. In this section, we first describe our setup and afterwards the individual measurements.

4.1 Setup

For every optimization, we executed MooBench with and without the optimization. We used a call tree depth of 10, a method invocation count of 2 000 000 and a VM start count of 10, which are the default parameters of MooBench. Based on the technical suitability, we also compared the combination of different optimizations. Additionally, we compare the measurements to the execution without any instrumentation.

All measurements have been executed on an i7-4770 CPU @ 3.40GHz using the JVM from OpenJDK 1.8.0_352, i.e., the latest OpenJDK release at the execution time. The following subsections describe our measurement results. All values are, if not otherwise specified, in **microseconds** (μs) per **MooBench method call**. For example, a measured value of 4.77 for Kieker with AspectJ and a call tree depth of 10 means that it took 4.77μs for the monitoring of 10 nodes, i.e., a single node monitoring would have an average overhead of 0.477μs. The measurements can be repeated using the

	No instr.	AspectJ	Source Code Instr.
Mean	0.0548	4.7711	2.4169
95 %	± 0.0	± 0.0444	± 0.0038
Q_1	0.0520	4.1890	2.4380
Median	0.0530	5.0790	2.5730
Q_3	0.0580	5.5030	2.6880

Table 1: Statistics of Source Code Instrumentation

	No instr.	Source Code Instr.	DurationRecord
Mean	0.0548	2.4169	2.3426
95 %	± 0.0	± 0.0038	± 0.0037
Q_1	0.0520	2.4380	2.1720
Median	0.0530	2.5730	2.4720
Q_3	0.0580	2.6880	2.6080

Table 2: Statistics of `DurationRecord`

`optimizations-all` branch of the MooBench main repository.[12] The dataset of our measurement results is published.[13]

4.2 Source Code Instrumentation

The statistics of no instrumentation, AspectJ instrumentation and source code instrumentation are displayed in Table 1. Due to the low mean execution time without instrumentation, the 95 % confidence interval has technically a size of 0. It shows that AspectJ instrumentation roughly increases the execution time by a factor of 10. Based on the granularity of methods, this factor might be higher or lower in production workloads. Using source code instrumentation roughly decreases the execution time by a factor of 2 and is therefore a recommendable optimization. However, it is only usable if the source code is available; therefore, using the `-javaagent` to inject monitoring probes will in some cases stay necessary.

4.3 DurationRecord

Since the `DurationRecord` is currently only implemented for the source code instrumentation, we compared the source code instrumentation for Kiekers regular `OperationExecutionRecord` and the source code instrumentation using the `DurationRecord`. The results are displayed in Table 3. It shows that using `DurationRecord` also reduces the measurement duration statistically significant, but not with an effect size in the order of magnitude of the improvement using source code instrumentation.

4.4 CircularFifoQueue

The `CircularFifoQueue` can be used with source instrumentation and either `OperationExecutionRecord` or `DurationRecord`. Therefore, we compare the execution times of both. The results are displayed in Table 3. The table shows that the `CircularFifoQueue` provides a statistically significant performance improvement, both when using `OperationExecutionRecord` and `DurationRecord`. Surprisingly, the improvement is higher when using Kiekers default `OperationExecutionRecord` than when using `DurationRecord`. We assume that this is due to internal optimizations of the JVM; however, combining `CircularFifoQueue` and `DurationRecord` should therefore not be blindly applied.

	No instr.	CircularFifoQueue + OperationExecutionR.	CircularFifoQueue + DurationRecord
Mean	0.0548	1.5017	1.6503
95 %	± 0.0	± 0.0047	± 0.0039
Q_1	0.0520	1.2330	1.2670
Median	0.0530	1.3300	1.4190
Q_3	0.0580	1.4810	1.7420

Table 3: Statistics of `CircularFifoQueue`

Additionally, the `CircularFifoQueue` could *swallow* elements if more elements than the capacity of the queue are added. Therefore, we would only advise using `CircularFifoQueue` in specific settings, where it can be guaranteed that this does not happen. Another method for reduction of the queue overhead is the aggregation of data before they are inserted into the queue, is presented in the following.

4.5 Aggregated Writing

Aggregating is currently only implemented for source code instrumentation. Additionally, it is only implementable straightforward for `DurationRecord`, since metadata like the execution stack size might not be aggregated easily. Therefore, we compare the aggregated writing and usage of `DurationRecord`. There might be mechanisms for aggregating also metadata, e.g., by storing a mapping from the execution stack size to the current data. The performance characteristics of this would heavily depend on the tree structure. Examining this could be a part of future work.

Table 4 shows the statistics of aggregated data reading (when aggregating always 1000 method invocations). It shows that aggregating the data also significantly decreases the monitoring overhead. Therefore, for use cases where aggregated data can be used, we would advise using aggregated writing.

	No instr.	DurationRecord	Aggregated Writing
Mean	0.0548	2.3426	0.4014
95 %	± 0.0	± 0.0037	± 0.0003
Q_1	0.0520	2.1720	0.3810
Median	0.0530	2.4720	0.3860
Q_3	0.0580	2.6080	0.3870

Table 4: Statistics of aggregated data reading

4.6 Combination

Since aggregated writing reduces the amount of created monitoring records, it also makes sense to combine aggregated writing and the usage of the `CircularFifoQueue`. Table 5 shows the statistics of this approach. In this setting, the difference in the measured values is not statistically significant. Therefore, we cannot make a statement about whether using the `CircularFifoQueue` leads to an overhead reduction in this setting.

[12] https://github.com/kieker-monitoring/moobench/tree/optimizations-all
[13] https://doi.org/10.5281/zenodo.7566677

	No instr.	Aggregated Writing	Combination
Mean	0.0548	0.4014	0.3897
95 %	± 0.0	± 0.0003	± 0.0002
Q_1	0.0520	0.3810	0.3830
Median	0.0530	0.3860	0.3880
Q_3	0.0580	0.3870	0.3890

Table 5: Statistics of aggregated data reading

In real-world use cases, the overhead is not only relevant for a call tree depth of 10. Therefore, we measured how the combination of these optimizations perform for different call tree sizes. Figure 5 shows how the overhead evolves with growing call tree depth. For better visibility, the measurement points are connected, even if the call tree depth is a concrete value. The areas around the curves marks the area of $\mu + \sigma$ and $\mu - sigma$, where μ is the mean and σ is the standard deviation. We see adding the use of `DurationRecord` only slightly reduces the overhead for the examined call tree sizes. Since using the `DurationRecord` also reduces the overhead if we do not use the `CirclarFifoQueue`, and since the metadata are not usable when aggregating operation calls, we keep using the `DurationRecord`. Overall, the figure shows that for a call tree size of 128, all optimizations reduce the overhead significantly in comparison to the regular monitoring using AspectJ.

5 RELATED WORK

Related work can be grouped into work that only *measures the monitoring overhead*, work that examines ways to *reduce the monitoring overhead* and *work that reduces the monitoring* itself. In the following, we give an overview over works from these fields.

5.1 Measurement

Horky et al. [8] examine the overhead of dynamic monitoring in Java, i.e., monitoring that is inserted and removed by demand. They state that three effects–the mere presence of the probe, the manipulation of the probes execution and the code optimizations possibly happening because of the probe–influence the execution time of the monitored method. By using the SPECjbb2015™, they find that the activation and deactivation of probes in realistic environments might both slow down or speed up the execution.

Bara et al. [1] develop hardware support for performance monitoring in an processor core and show that performance monitoring also increases the energy consumption of a processor. In contrast to this work, they focus on hardware level performance monitors.

A main requirement for performance measurement is reproducability. Eichelberger et al. [2] research how MooBench measurements of SPASS-meter can be reproduced. Additionally, Knoche and Eichelberger [9, 10] discuss how a Raspberry Pi can be used for reproducable performance measurement. In contrast to their work, we present measurement results of an established benchmark on a typical Desktop PC. In order to facilitate reproducibility of our result, it would make sense to repeat our measurements on different execution hardware, e.g., on a Raspberry Pi.

Furtheremore, there exist measurement implementations that aim for minimal overhead using different techniques. JPortal [25] achieves minimal overhead by using hardware-based tracing using Intel processor trace. Since this requires decoding the stored metadata of maschine code executions, the obtained traces are not completely correct. However, they report an overhead of at maximum 16.5 %. In contrast to our work, they used different workloads for benchmarking and their goal was not to get measure accurate runtimes but obtain accurate traces.

Schardl et al. [18] develop the CSI framework, which is able to insert code at compile time. This can–amongst other use cases–be utilized to measure the performance of executed code. They evaluate the CSI framework by instrumenting the Apache Server and the bzip2 data compressor and benchmarking them by appropriate benchmarks. They find that the overhead is less than 70 % of the programs execution time. While they pursue a similiar approach for instrumentation, e.g., instrumenting at compile time, they use a different technology (software written in c) and evaluate benchmarks measuring real-world use cases. Therefore, their results are not directly compareable to ours.

5.2 Reduction of the Monitoring Overhead

In the following, we first describe the research on overhead reduction in Kieker chronologically and afterwards the work on using a different instrumentation technique.

To *reduce the the monitoring overhead*, Waller and Hasselbring [24] first researched how the use of multi-core processors influence the runtime overhead of Kieker. They find that asynchronous monitoring writers have lower overhead when multi-core processor systems are used. Waller et al. [23] state that the monitoring overhead consists of the instrumentation overhead, the data collection time and the time for writing the data to a queue. They examine four possible optimizations: (1) Internal optimizations, e.g., reusing the method `String` definition, (2) using an `ArrayBlockingQueue` from the disruptor framework, (3) sending `ByteBuffer` instances to the monitoring queue, instead of the record, and (4) reducing the amount of source used in Kieker by providing a minimal Kieker project. They find that the first two optimizations provide significant improvements and are feasible from a maintainability point of view. Optimization (3) reduces the overhead, but requires the definition of individual serializations for each record type, and is therefore not considered further. Optimization (4) does slightly reduce the overhead, but also reduces the maintainability and extendability of Kieker and is therefore also not considered further.

Finally, Strubel and Wulf [20] further reduce the monitoring overhead. Formerly, string attributes were serialized using a registry records having an id, so strings are not fully serialized. This transformation of registry records was done in the application thread; by moving it to the writer thread, both the code complexity and the overhead inside of the application thread could be reduced.

Since this work builds on the current Kieker version, the feasible optimizations created by Waller and Hasselbring [24], Waller et al. [23] and Strubel and Wulf [20] are already included. Nevertheless, they had the focus of preserving the architecture discovery functionality of focus, whereas this work focusses on monitoring method execution duration without preserving the architecture discovery functionality.

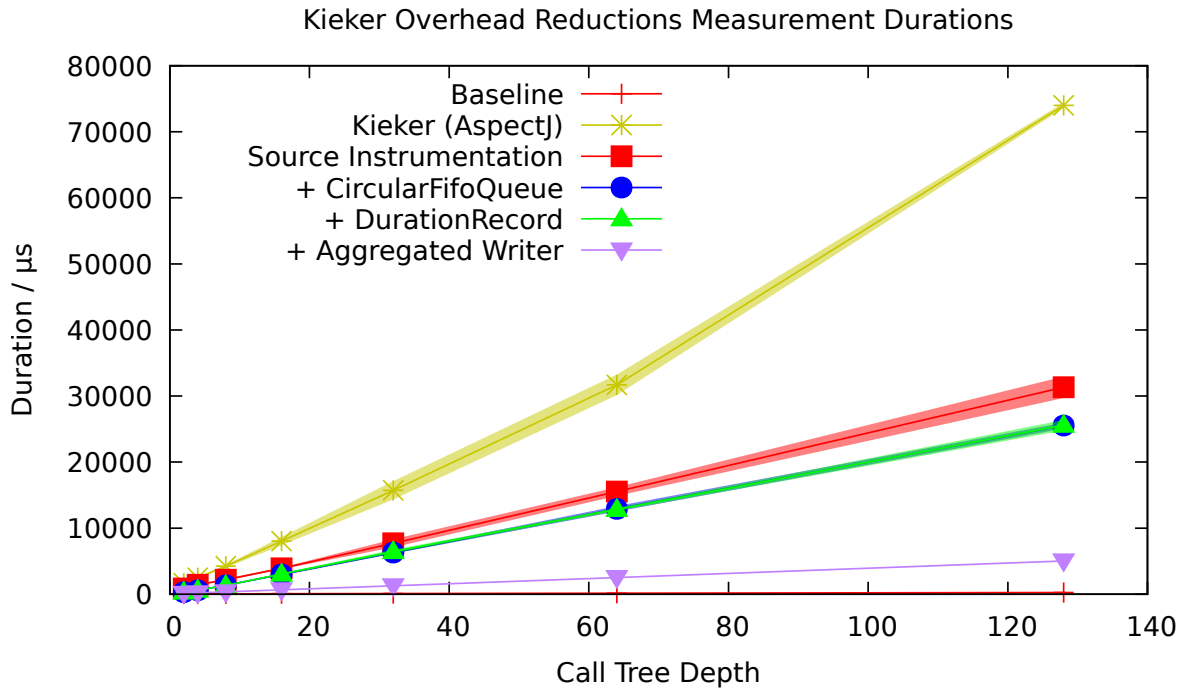

Figure 5: Growth of the Overhead Based on Call Tree Size

The domain-specific language for bytecode instrumentation (DiSL) enables a more fine-grained instrumentation than AspectJ [12]. Since it relies on ASM, it is very likely to have different performance characteristics than Kieker's default AspectJ instrumentation. Therefore, it would be an alternative for our source code instrumentation and could therefore be evaluated against it. However, there is currently no usable implementation provided, therefore we cannot compare them.

Okanovic et al. [14] examined how to *use different instrumentation techniques* in order to reduce the monitoring overhead. Therefore, they used DiSL instead of AspectJ for their monitoring system DProf. They find that using DiSLs reduced the monitoring overhead by 1.2 % for their use case. In contrast to our work, they used their system DProf to evaluate the performance overhead. Their performance improvement by 1.2 % indicates that DiSL very likely does not offer a overhead reduction that is as big as the improvement of our source instrumentation.

5.3 Reduction of the Monitoring

For reduction of monitoring itself, Popiolek et al. [15] research how to reduce the overhead of performance monitoring by reducing the performance monitors in cloud infrastructures. They do so by clustering the counters using the Pearson correlation coefficient, i.e., by determining which counters are either high or low at the same time. Thereby, they reduce the monitoring overhead by up to one third. Similarly, Shang et al. [19] determine the performance counters also by using Pearson correlation of the measured values. Afterwards, they hierarchically cluster the performance counters using Calinski-Harabasz stopping rule. In contrast to these works, we do not try to select where to monitor but to reduce the monitoring overhead at the methods where monitoring happens.

Mertz and Nunes [13] reduce the overhead by adaptive configuration: In the first step, they perform a lightweight monitoring. Based on the results of this monitoring, they decide which parts of the source code should be monitored in more detail. In contrast to this work, we do not perform an adaptive configuration but try to reduce monitoring overhead with static configuration.

6 SUMMARY AND OUTLOOK

We described how we reduced the monitoring overhead of the Kieker framework. This was done using four changes to the monitoring process: Instrumenting the source code directly instead of instrumenting using AspectJ, reducing the stored monitoring data, using `CircularFifoQueue` instead `LinkedBlockingQueue` and using an aggregated writer instead of Kiekers `FileWriter`.

Reduction of the stored monitoring data and usage of an aggregated writer is only feasible if the long-term performance should be examined. If individual method invocations and outliers for these invocations should be examined, only source code instrumentation and the queue exchange can be used.

Using the monitoring overhead benchmark MooBench, we showed that all of these optimizations decrease the monitoring overhead. Therefore, it is planned to make them available for regular monitoring. This requires the following adaptions: (1) For Kieker source instrumentation, this has been already done by publishing the code as part of the Kieker GitHub organization. (2) For the `DurationRecord`,

the process of including records with reduced monitoring information into the Kieker infrastructure is currently ongoing. Since Kieker supports a variety of languages, the inclusion of such records requires defining a basic record in the Kieker Instrumentation Language and generating the records for every language out of it. (3) Switching the queue is a functionality that will stay up to the user. While using the `CircularFifoQueue` improves the performance statistically significant, it is unclear whether production systems might produce too many records for the queue and therefore records get lost. Future research might examine whether adding additional checks for integrity could be added without disturbing the monitoring overhead reduction. (4) It is also planned to add the `AggregatedWriter` as part of Kieker itself, but not set it as default. Thereby, users can switch to the `AggregatedWriter` if they use the `DurationRecord` and aim for only measuring the performance, but not doing architecture recovery tasks. In addition to adapting Kieker itself, it should also be checked to which degree aggregating the data improves the measurement quality. While we assume that lower overhead always results in improved distinguishability of measurement results, this thesis requires checking.

This work is a first step towards minimal overhead monitoring: In addition to optimize the process itself, the following further optimizations should be researched: (1) Our optimized monitoring process should be compared to sampling. While sampling is only able to obtain the stack trace at safepoints, it has a very low overhead. Therefore, a comparison between our optimized Kieker probe and sampling should compare both, the overhead and the ability to detect performance changes. (2) Usage of adaptive monitoring (like [13]): It was shown that deactivated probes nearly reduce the monitoring overhead to 0. To obtain usable data while preserving measurement accuracy, it might be promising to activate probes only part-time. Additionally, our approach is only able to measure the performance and not the creation of the call tree, since no monitoring metadata are present. If monitoring metadata are required, it would also be possible to add multiple probes to the same source code, one regular probe for creation of `OperationExecutionRecord` and one probe for `DurationRecord`. Afterwards, the necessary probes could be used. By this work and the examination of sampling and adaptive monitoring, the definition of minimal overhead processes for performance monitoring can be created. This could be the basis for understanding the performance behavior of software on a low level in production systems.

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

REFERENCES

[1] Lucian Bara, Oana Boncalo, and Marius Marcu. 2015. Hardware support for performance measurements and energy estimation of OpenRISC processor. In *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics*. 399–404. https://doi.org/10.1109/SACI.2015.7208237

[2] Holger Eichelberger, Aike Sass, and Klaus Schmid. 2016. From reproducibility problems to improvements: a journey. In *SSP 2016, Softwaretechnik-Trends*, Vol. 36. 43–45.

[3] Holger Eichelberger and Klaus Schmid. 2014. Flexible resource monitoring of Java programs. *Journal of Systems and Software* 93 (2014), 163–186. https://doi.org/10.1016/j.jss.2014.02.022

[4] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76. https://doi.org/10.1145/1297027.1297033

[5] Wilhelm Hasselbring. 2021. Benchmarking as Empirical Standard in Software Engineering Research, In EASE 2021: Evaluation and Assessment in Software Engineering. *Evaluation and Assessment in Software Engineering*, 365–372. https://doi.org/doi:10.1145/3463274.3463361

[6] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (2020), 100019. https://doi.org/10.1016/j.simpa.2020.100019

[7] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2015. Lightweight Java profiling with partial safepoints and incremental stack tracing. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. 75–86.

[8] Vojtěch Horký, Jaroslav Kotrč, Peter Libič, and Petr Tůma. 2016. Analysis of Overhead in Dynamic Java Performance Monitoring. In *7th ACM/SPEC ICPE* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2851553.2851569

[9] Holger Knoche and Holger Eichelberger. 2017. The Raspberry Pi: A Platform for Replicable Performance Benchmarks? *Softwaretechnik-Trends* 37, 3 (2017), 14–16.

[10] Holger Knoche and Holger Eichelberger. 2018. Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper. In *Proceedings of the 2018 ICPE*. 305–316. https://doi.org/10.1145/3184407.3184431

[11] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2022. Locating Performance Regression Root Causes in the Field Operations of Web-Based Systems: An Experience Report. *IEEE Transactions on Software Engineering* 48, 12 (2022), 4986–5006. https://doi.org/10.1109/TSE.2021.3131529

[12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: a domain-specific language for bytecode instrumentation. In *11th International Conference on Aspect-oriented Software Development*. 239–250. https://doi.org/10.1145/2162049.2162077

[13] Jhonny Mertz and Ingrid Nunes. 2019. On the Practical Feasibility of Software Monitoring: A Framework for Low-Impact Execution Tracing. In *14th SEAMS* (Montreal, Quebec, Canada). IEEE Press, 169–180. https://doi.org/10.1109/SEAMS.2019.00030

[14] Dušan Okanović, Milan Vidaković, and Zora Konjović. 2013. Towards performance monitoring overhead reduction. In *2013 IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*. 135–140. https://doi.org/10.1109/SISY.2013.6662557

[15] Pedro Freire Popiolek, Karina dos Santos Machado, and Odorico Machado Mendizabal. 2021. Low overhead performance monitoring for shared infrastructures. *Expert Systems with Applications* 171 (2021), 114558. https://doi.org/10.1016/j.eswa.2020.114558

[16] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2021. Overhead Comparison of OpenTelemetry, inspectIT and Kieker. In *SSP 2021*.

[17] David Georg Reichelt, Stefan Kühne, and Willhelm Hasselbring. 2019. PeASS: A Tool for Identifying Performance Changes at Code Level. In *Proceedings of the 33rd ACM/IEEE ASE*. ACM. https://doi.org/10.1109/ASE.2019.00123

[18] Tao B Schardl, Tyler Denniston, Damon Doucet, Bradley C Kuszmaul, I-Ting Angelina Lee, and Charles E Leiserson. 2017. The CSI Framework for Compiler-Inserted Program Instrumentation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 2 (2017), 1–25. https://doi.org/10.1145/3154502

[19] Weiyi Shang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2015. Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 15–26. https://doi.org/10.1145/2668930.2688052

[20] Hannes Strubel and Christian Wulf. 2016. Refactoring Kieker's Monitoring Component to further Reduce the Runtime Overhead. In *Symposium on Software Performance 2016 (SSP '16)*.

[21] Jan Waller. 2015. *Performance Benchmarking of Application Monitoring Frameworks*. BoD–Books on Demand.

[22] Jan Waller, Nils Christian Ehmke, and Wilhelm Hasselbring. 2015. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *ACM SIGSOFT Software Engineering Notes* 40, 2 (3 2015), 1–4. https://doi.org/10.1145/2735399.2735416

[23] Jan Waller, Florian Fittkau, and Wilhelm Hasselbring. 2014. Application performance monitoring: Trade-off between overhead reduction and maintainability. *Proceedings of the Symposium on Software Performance* (2014).

[24] Jan Waller and Wilhelm Hasselbring. 2012. A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. In *International Conference on Multicore Software Engineering, Performance, and Tools*. Springer, 42–53. https://doi.org/10.1007/978-3-642-31202-1_5

[25] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: Precise and efficient control-flow tracing for JVM programs with Intel Processor Trace. In *Proceedings of the 42nd ACM SIGPLAN ICPLDI*. 1080–1094. https://doi.org/10.1145/3453483.3454096