Overhead Comparison of Instrumentation Frameworks

David Georg Reichelt
Lancaster University Leipzig
Leipzig, Saxony, Germany

Lubomír Bulej
Charles University
Prague, Czech Republic

Reiner Jung
Kiel University
Kiel, Germany

André van Hoorn
Universität Hamburg
Hamburg, Germany

ABSTRACT

Application Performance Monitoring (APM) tools are used in the industry to gain insights, identify bottlenecks, and alert to issues related to software performance. The available APM tools generally differ in terms of functionality and licensing, but also in monitoring overhead, which should be minimized due to use in production deployments. One notable source of monitoring overhead is the instrumentation technology, which adds code to the system under test to obtain monitoring data.

Because there are many ways how to instrument applications, we study the overhead of five different instrumentation technologies (AspectJ, ByteBuddy, DiSL, Javassist, and pure source code instrumentation) in the context of the Kieker open-source monitoring framework, using the MooBench benchmark as the system under test. Our experiments reveal that ByteBuddy, DiSL, Javassist, and source instrumentation achieve low monitoring overhead, and are therefore most suitable for achieving generally low overhead in the monitoring of production systems.

However, the lowest overhead may be achieved by different technologies, depending on the configuration and the execution environment (e.g., the JVM implementation or the processor architecture). The overhead may also change due to modifications of the instrumentation technology. Consequently, if having the lowest possible overhead is crucial, it is best to analyze the overhead in concrete scenarios, with specific fractions of monitored methods and in the execution environment that accurately reflects the deployment environment. To this end, our extensions of the Kieker framework and the MooBench benchmark enable repeated assessment of monitoring overhead in different scenarios.

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, Lubomír Bulej, Reiner Jung, and André van Hoorn. 2024. Overhead Comparison of Instrumentation Frameworks. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24 Companion), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3629527.3652269

1 INTRODUCTION

Understanding application performance at runtime requires collecting metrics such as response times, resource usage, or error rates during application execution. When applied to production environments, this process is referred to as performance monitoring [19]. While monitoring can be done at different levels, including business, application, and the infrastructure, we focus on the application level in this work.

Because performance monitoring is often orthogonal to features delivering value to application users, it is common to implement monitoring as a cross-cutting concern using instrumentation, i.e., by inserting instructions into software that explicitly create monitoring records associated with relevant performance events. An alternative method for obtaining information about application performance is *sampling*, in which the system periodically collects snapshots of generic (e.g., code location being executed, stack traces, memory consumption) or application-specific metrics (e.g., request queue depth, average response time) exposed through a suitable framework (e.g., JMX beans).

While sampling allows controlling the trade-off between (a fixed) overhead and accuracy by setting the sampling period, instrumentation producing explicit records of relevant events is generally more flexible, because it can produce data usable for both monitoring and tracing. However, the overhead of instrumentation-based monitoring is much more variable and potentially more difficult to control. In this paper, we focus on the baseline overhead due to the use of a particular instrumentation technology.

During instrumentation, monitoring code (*probe*) is inserted into the monitored application to create monitoring records corresponding to various events occurring during application execution. The injection of probes can be achieved using different instrumentation technologies. The instrumentation process, depicted in Figure 1, is generally driven by a application monitoring frameworks, such as OpenTelemetry or Kieker, which also determine the instrumentation technology used to insert probes into system under test.

ICPE '24 Companion, May 7–11, 2024, London, United Kingdom
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0445-1/24/05…$15.00
https://doi.org/10.1145/3629527.3652269

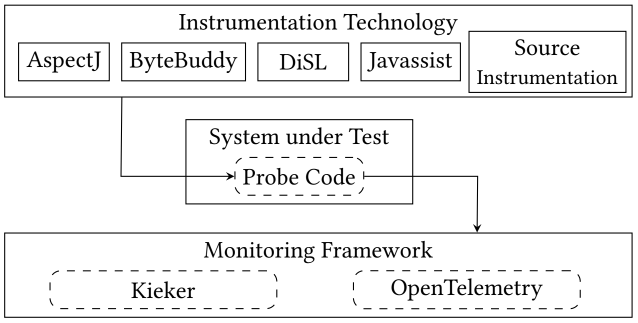

Figure 1: Instrumentation Process for Monitoring Frameworks

In this paper, we compare the overhead of different instrumentation technologies using the Kieker [5] monitoring framework. In addition, we compare the measured overhead to the overhead of OpenTelemetry using its default instrumentation technology ByteBuddy.

The instrumentation can be done at compile time, at application start time, using a `javaagent` with a `premain` method, or at application runtime, using an `agentmain` method. In this work, we compare the monitoring overhead that is introduced due to compile-time or application start-time instrumentation. The overhead of dynamic instrumentation has already been examined [6].

By comparing the monitoring overhead of the instrumentation technologies AspectJ[1], ByteBuddy[2], DiSL [11], Javassist[3], and source code instrumentation, we find that (1) There are significant differences using the instrumentation technologies, e.g., using AspectJ creates up to 30 % more overhead than directly instrumenting the source code, (2) source code instrumentation has the lowest overhead in our benchmarking executions, (3) OpenTelemetry has significantly higher overhead, that is not caused by its instrumentation technology ByteBuddy, and (4) all technologies scale linearly with the call tree depth. Furthermore, our extensions of the Kieker framework and the MooBench benchmark enable repeated assessment of monitoring overhead, e.g., in case the underlying JVM or the instrumentation technologies are changed.

This paper is structured as follows. We first provide a brief overview of the instrumentation technologies, the application monitoring framework, and the benchmark used. Then we present our experimental setup and results, which we subsequently compare to related work. Finally, we give a summary and an outlook.

2 FOUNDATIONS

In this section, we first provide an overview of the instrumentation technologies that are the subject of this study. We then describe the Kieker monitoring framework used to carry out the comparison and the information collected for monitoring purposes. Finally, we describe the benchmark used as the system under test.

[1]https://eclipse.dev/aspectj/
[2]https://bytebuddy.net/
[3]https://www.javassist.org/

2.1 Instrumentation Technologies

In this work, we use five instrumentation technologies: Source instrumentation, AspectJ, ByteBuddy, DiSL, and Javassist. Source instrumentation operates directly on the source code and transforms it into instrumented source code. Subsequent compilation produces instrumented bytecode which is executed by the JVM. Source code instrumentation requires access to the source code and is language-specific, and generally need to be done at application *compile time*. The other technologies employ *bytecode manipulation* and insert instrumentation code directly into the bytecode of application classes, producing instrumented bytecode to be executed by the JVM. This approach is (mostly) language-agnostic and offers increased flexibility compared to source instrumentation.

Frameworks such as AspectJ, ByteBuddy, and Javassist support *compile time* instrumentation, i.e., instrumentation is performed prior to application execution using bytecode that is either downloaded or produced by compilation. An alternative is to perform instrumentation at application *load time*, which completely decouples instrumentation from the build process. This is done using instrumentation API provided by the JVM, usually through a `javaagent` with a `premain` method which triggers bytecode transformation of the classes being loaded by the JVM. This approach is supported by all of the above mentioned frameworks. The different instrumentation processes are summarized in Figure 2.

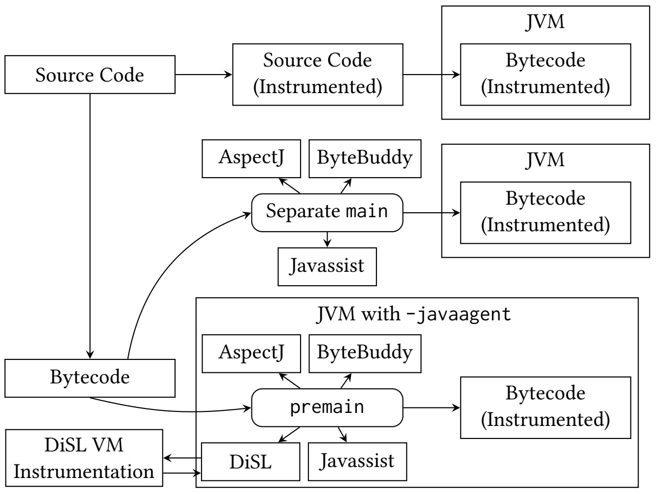

Figure 2: Instrumentation Options: Source Code, Compile Time and Load Time

Source Instrumentation. The most direct way of injecting monitoring probes is by adding them to the source code of an application. Listing 1 shows this in a simplified way: In the original code of `myMethod`, obtaining the start time, obtaining the end time, and passing this information to a `MonitoringController` is added.

Listing 1: Manual Instrumentation Example

```
public void myMethod() {
  long tin = System.nanoTime();
  ....
  long tout = System.nanoTime();
  MONITORING_CONTROLLER.newOperationExecution(
    tin, tout, "MyClass.myMethod()");
}
```

Manual instrumentation is time-consuming and error-prone, and reduces the maintainability of the source code. Therefore, we developed a tool for automated injection of monitoring probes in Java source code.[4] This reduces the monitoring overhead significantly [16]. While source code instrumentation can reduce the monitoring overhead, it cannot be used in cases where only the bytecode (and not the source code) is available. This includes cases where other programming languages have been used, e.g., Scala or Kotlin. This disadvantage could be fixed by implementing the source instrumentation for other languages.

AspectJ. AspectJ was developed to support the modular implementation of crosscutting concerns, i.e., concerns that need to be addressed in different modules. This is called aspect-oriented programming [8]. AspectJ builds on *advices* that consist of *pointcuts* (patterns describing when the pointcut is activated) and advice bodies (the code that should be executed on a matching pointcut). The usability of AspectJ for implementation of domain requirements that are crosscutting concerns is controversial [13]. Nevertheless, its usability for logging and monitoring is indisputable. Internally, AspectJ uses BCEL[5] for low-level adaptation of bytecode.

ByteBuddy. ByteBuddy aims to make the adaptation of byte code possible without knowledge of its format. Thereby, it makes it possible to support various use cases of bytecode manipulation for security, logging, or performance monitoring. Especially, the OpenTelemetry API reference implementation relies on ByteBuddy. Internally, AspectJ uses ASM[6] for low-level adaptation of bytecode.

DiSL. DiSL[7] (Domain-specific language for instrumentation) was developed primarily for dynamic program analysis [11]. DiSL allows selecting any region of the bytecode for instrumentation, in contrast to AspectJ's model, which only allows the selection of specific points, e.g., operation starts. DiSL also supports synthetic local variables, which facilitate passing of information between advice code in the scope of a single method, e.g., collecting operation start and end times, and passing the operation duration along with context information to the application performance monitoring framework for recording.

Internally, DiSL uses the ASM[8] library that provides a low-level interface for bytecode manipulation. During *load time* instrumentation, DiSL uses a separate VM to execute the instrumentation logic which processes the application classes loaded by the VM in which the application executes. This eliminates perturbations in the application VM caused by the instrumentation logic using common classes that may be subject to instrumentation (due to their use by the application).

Javassist. Javassist[9] is a Java bytecode manipulation library [2]. It supports changing the bytecode by the specification of adapted bytecode and source code. By adding a `javagent` and adding a `ClassTransformer` (which is a class from `java.lang`), monitoring libraries can interact with the class loading mechanism. Since bytecode manipulation needs a thorough knowledge of the structure of bytecode itself, libraries like Javassist can ease this process. However, the usage of Javassist also requires the (indirect) definition of bytecode manipulation. Checking the instrumentation source at compile-time (like with AspectJ aspects or DiSL instrumentation definitions) is not possible.

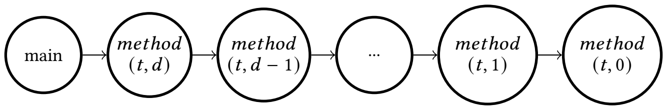

Figure 3: Call Tree of MooBench, from: [16]

2.2 Kieker

Application performance monitoring (APM) frameworks contain a library and an agent that are responsible for data acquisition.[10] There are both open-source APM tools, such as Kieker [5] and OpenTelemetry[11], and closed-source APM tools, such as Dynatrace APM[12] and the DataDog agent.[13] While their implementation details differ, they are all able to obtain operation execution data, such as operation start and end times, operation signature, and operation's position in the call tree.

Our prior study on the overhead of these frameworks shows that Kieker has the lowest overhead among the open-source tools [15]. Kieker's `OperationExecutionRecord` contains, next to the timing information, the execution operation index (`eoi`) and the execution stack size (`ess`), which make it possible to reconstruct the call tree. In this work, we compare how much overhead different instrumentation technologies cause to collect the data needed to create the `OperationExecutionRecord`.

2.3 MooBench

Moobench is a benchmark that compares the overhead of different APM frameworks and their configurations [20]. To measure the overhead for each operation execution, MooBench calls its `monitoredMethod` recursively for a given call tree depth. In the leaf node, a busy waiting is executed, which simulates calculations. The structure of the MooBench call tree is depicted in Figure 3.

Performance measurement is subject to non-determinism because of a number of factors. Some, such as other processes running on the same system or CPU frequency and voltage scaling can be mitigated by platform configuration. Other factors such as differences in memory layout between different runs of the workload (in different processes) need to be addressed by the measurement process. Experiments involving platforms such as the Java or JavaScipt VM need to account for the effects of just-in-time compilation and garbage collection. A rigorous measurement process must collect data from multiple experiment runs, each time using a new (managed language) VM process. Within each run, the measured operation needs to be repeated enough times to get past the warm-up

[4]https://github.com/kieker-monitoring/kieker-source-instrumentation
[5]https://commons.apache.org/proper/commons-bcel/
[6]https://asm.ow2.io/license.html
[7]https://gitlab.ow2.org/disl/disl
[8]https://asm.ow2.io/
[9]https://www.javassist.org/

[10]https://openapm.io/
[11]https://opentelemetry.io/
[12]https://www.dynatrace.com/de/platform/application-performance-monitoring/
[13]https://docs.datadoghq.com/agent/

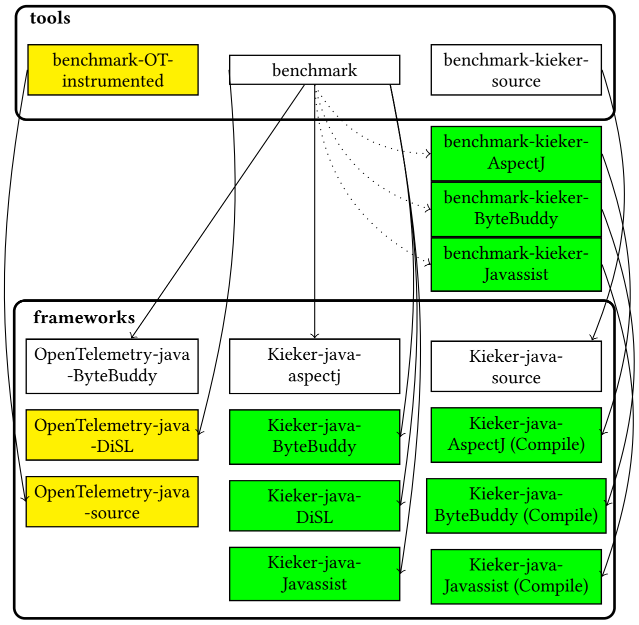
tools
benchmark-OT-instrumented
benchmark
benchmark-kieker-source
benchmark-kieker-AspectJ
benchmark-kieker-ByteBuddy
benchmark-kieker-Javassist
frameworks
OpenTelemetry-java-ByteBuddy
Kieker-java-aspectj
Kieker-java-source
OpenTelemetry-java-DiSL
Kieker-java-ByteBuddy
Kieker-java-AspectJ (Compile)
OpenTelemetry-java-source
Kieker-java-DiSL
Kieker-java-ByteBuddy (Compile)
Kieker-java-Javassist
Kieker-java-Javassist (Compile)

Figure 4: Architecture of MooBench (Green: Added by us, Yellow: Future extension options)

phase (dominated by just-in-time compilation) to collect samples representing the expected operation durations. The averages of operation durations obtained from individual runs then provide basis for statistical analysis, e.g., a comparison using the two-sided t-test [4].

MooBench allows comparing different frameworks and execution environments using a rigorous measurement process. Each framework/execution environment combination contains a single folder, e.g., Kieker-java and Kieker-Python with a script to execute measurements. Each script contains configuration parameters for the respective framework, including the name, the command line parameters, and the environment variables.

MooBench also aims to identify the causes of the overhead, assuming that monitoring overhead emerges from instrumentation overhead, data collection, and the writing of data [21]. To this end, MooBench supports performance measurements using multiple configurations: Baseline (without any instrumentation), instrumentation only (with *deactivated monitoring*), monitoring with *no logging* (data is not written to data sink), and full monitoring (writes data to a sink, e.g., binary file, TCP receiver, Zipkin server).

3 BENCHMARKING

In this section, we first describe how we implemented the benchmark changes. Afterwards, we describe our configurations for benchmark execution. Based on this, we describe the results of the MooBench performance comparison. Finally, we discuss the scalability of the overhead.

3.1 Benchmark Adaptation

To benchmark the different instrumentation technologies, load-time and compile-time instrumentations for each instrumentation technology, the decision of how to handle deactivated monitoring, and the adaptation of the benchmark were necessary. These are described in the following.

Load-time Instrumentation. Kieker already supports the AspectJ instrumentation and contains a subproject for automated source instrumentation. Therefore, we additionally implemented probes for ByteBuddy, DiSL, and Javassist. To do so, we started providing separate JARs for each instrumentation (`kieker-bytebuddy`, `kieker-disl`, and `kieker-javassist`), as a `kieker-aspectj`-jar was already provided before.

To create a `javaagent`, the `premain` needs to specify what should be done when the agent is started. AspectJ contains its own `premain` implementation which handles the instrumentation of classes according to joinpoint specification in `aop.xml`. Because ByteBuddy and Javassist require users to implement the instrumentation procedure on their own, we implemented a `javaagent` for each of them. Since the other technologies do not provide a method for joinpoint specification, we use the previously existing Kieker method pattern definition to specify methods that should be instrumented and pass it to the agents through the `KIEKER_SIGNATURES` environment variable. Both agents only support the `OperationExecutionRecord`, as this record is also used for the other MooBench implementations.

Compile-Time Instrumentation. Source instrumentation naturally happens at compile-time. To get to know whether the load-time weaving or the bytecode created by the instrumentation technology is causing overhead, we additionally created Kieker `main` methods that instrument an existing JAR using AspectJ, ByteBuddy and Javassist. Afterwards, these are used by specifically tailored benchmark configurations.

Deactivated Monitoring. One challenge is the need to support deactivation of monitoring at runtime, either fully or for selected methods. AspectJ and ByteBuddy allow to return directly from the instrumentation methods, and thereby execute the unchanged original method. Because that is not possible with the source instrumentation, we copy the original method body into a branch that is executed when monitoring is disabled or when the monitoring of the method is deactivated. In the case of Javassist and DiSL, we use a separate monitoring class (`OperationExecutionDataGatherer`) that the instrumentation code calls on operation start and operation end. When monitoring is disabled, the operation start invocation returns `null`, indicating to the instrumentation code that the operation end invocation is not necessary. Otherwise it returns a object (`FullOperationStartData`) representing operation data needed to create a monitoring record, which the instrumentation passes to the operation end invocation.

Listing 2: Exit Handling in DiSL and Javassist

```
@SyntheticLocal
static FullOperationStartData data;

@Before(marker = BodyMarker.class,
  scope = "MonitoredClass*.*")
public static void beforemain(
  final KiekerStaticContext c) {
  data = OperationExecutionDataGatherer
```

```
    .operationStart(c.operationSignature());
}

@After(marker = BodyMarker.class,
  scope = "MonitoredClass*.*")
public static void aftermain() {
  if (data != null) {
    OperationExecutionDataGatherer
      .operationEnd(data);
  }
}
```

For Javassist, another option is to obtain all monitoring data at the beginning of the method call (including the start time) and to only add the monitoring data if monitoring is enabled. This leads to heavily increased overhead for deactivated monitoring (0.55 μs, instead of 0.06μs) and to slightly decreased overhead for enabled monitoring (2.42 μs instead of 2.55μs). This is a good option if all instrumented classes have activated monitoring. If monitoring is deactivated (and might be activated again later), this increases the overhead. Since the other frameworks do not have this option, we did not consider this implementation variant further to have a fair comparison between all frameworks.

MooBench Adaptation. Based on these changes, we modified MooBench to support all instrumentation technologies. Before our refactoring, only framework-language combinations were supported, i.e., Kieker-java, Kieker-python, inspectIT-java, and OpenTelemetry-java. After our refactoring, we added the instrumentation technology for every framework, e.g., we renamed the old Kieker-java to Kieker-java-aspectj and additionally implemented Kieker-java-bytebuddy. Since ByteBuddy and Javassist provide their bytecode instrumentation directly, we could reuse most of the code. For DiSL, we needed to call the DiSL starter Python script in every benchmark call. The adapted architecture of MooBench is depicted in Figure 4.

3.2 Execution Configuration

After extending the benchmark, we executed it with two environment configurations: An i7-4770 running Ubuntu 22.04 and OpenJDK 11, and a Raspberry Pi 4 running Debian 11 with OpenJDK 11. While using the Raspberry Pi might yield measurement values that are different from typical business application deployments, they have the advantage of being reproducible for other researchers due to their standardization and affordability [9]. For each environment configuration, we ran two experiments: The measurement of the **monitoring overhead** and the measurement of the **overhead scalability**.

For the monitoring overhead, we ran the experiments with MooBench's default parametrization (2 000 000 calls, zero time for busy waiting, so the busy waiting part will only be two calls to `System.nanoTime()`, a call tree depth of 10 and 30 seconds sleep time), but set the number of VM starts to 30 (according to [4], who recommends at least 30 VM starts to gather enough observations for statistical testing).

For the overhead scalability, we also used the default configuration and set the recursion depth to 2, 4, …, 128 to examine the change of execution time with growing call tree size. To reduce the benchmarking time, we only executed the benchmark on the i7-4770. Our full measurement data are available as a dataset[14].

While executing the experiments, we noticed that the execution duration increased after a certain count of iterations. The effect occurred on every technology. Based on technology and execution environment, the threshold of iteration for this effect to start is between 500 000 and 2 000 000 iterations. After more iterations, the effect becomes stronger. We suspected memory and hard disk writing to be responsible for this effect. Therefore, we tried to change the memory settings (using different configurations between `-Xmx2g` and `-Xmx10g`). Regardless of the memory settings, the effect stayed the same. When reducing hard disk writing by setting a maximum file number that Kieker should write to,[15] we could eliminate this effect. The runtime for this effect is depicted in Figure 5 (with a very strong increase after 5 000 000, and a barely visible effect occurring earlier). Therefore, we set the `maxLogFiles` and will keep this for MooBench's future default configuration.

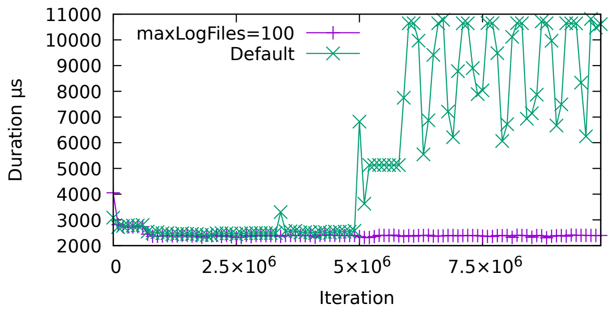

Figure 5: Average Duration per Iteration With and Without Setting `maxfiles`

3.3 Monitoring Overhead

Table 1 depicts the monitoring overhead for the different technology framework combinations for load-time instrumentation, and Table 2 depicts the monitoring overhead for Kieker for compile-time instrumentation. Overhead in the MooBench context means *execution time caused by monitoring*. Therefore, the measured duration is the measured duration of empty method executions with monitoring. For AspectJ, this would mean full monitoring causes roughly 3.30 μs overhead (=3.35μs − 0.05μs) on 10 method calls, indicating an overhead of 0.33 μs per node. In a production environment where methods themselves take time, the absolute overhead is expected to stay roughly the same (in the same execution environment), but the relative overhead will be significantly lower. The overhead depends on whether monitoring is deactivated, configured for execution without logging, or fully activated.

Deactivated Monitoring. For deactivated monitoring, OpenTelemetry creates a higher overhead than Kieker (at least factor 10). This overhead does not seem to be caused by ByteBuddy, since Kieker's ByteBuddy probe causes much lower overhead. The overhead for

[14]https://doi.org/10.5281/zenodo.10607598
[15]`-Dkieker.monitoring.writer.filesystem.FileWriter.maxLogFiles`

a deactivated AspectJ probe is notably higher than for ByteBuddy, DiSL, Javassist, and source instrumentation (roughly factor 3). ByteBuddy, DiSL, Javassist, and source instrumentation do not have statistically significant differences for compile-time weaving on the i7-4770. On Raspberry Pi, AspectJ compile-time instrumentation is slower than load-time. The remaining technologies differ only slightly.

No Logging. In this configuration the monitoring records are stored into a queue that discards them [14]. The performance in this configurations gives an indication how much of the overhead stems from the data collection itself. This configuration is only available for the Kieker probes. In this configuration, AspectJ has a significantly higher overhead than the other instrumentation technologies, and source instrumentation has a significantly lower overhead than all other instrumentation technologies. Surprisingly, we see that ByteBuddy has significantly higher overhead in its compile-time variant than its load-time variant on the Raspberry Pi. Since exactly the same final bytecode is executed, we assume that this is caused by different internal optimizations of the JVM, which won't necessarily occur in a production system.

Full Monitoring. In this configuration, the monitoring records are passed on to the application monitoring framework. Compared to OpenTelemetry, Kieker allows collecting traces with a lower overhead. Similarly to *no logging* and *deactivated monitoring* configurations, we see that AspectJ's full instrumentation creates higher overhead than source instrumentation, ByteBuddy, DiSL, and Javassist. As for *no logging*, we see that source instrumentation has the lowest overhead, which comes at the cost of requiring the source code. For the comparison of ByteBuddy, DiSL, and Javassist, we see that their ranking changes depending on whether load-time or compile-time instrumentation is used, and whether execution happens on a the Raspberry Pi or the i7-4700. Therefore, we assume that these differences are caused by different internal optimizations of the JVM, which might be different in production systems.

Looking at these values, we can infer that source instrumentation is the best variant if the source code is available, and that ByteBuddy, DiSL, and Javassist are good candidates for low overhead. Nevertheless, the overhead depends on the execution infrastructure, indicating that comparisons of instrumentation technologies for different software might differ.

It is also notable that the values measured for the baseline configuration are nearly the same as in our previous experiments [15], but the values for the full monitoring configuration changed (i7-4470 Kieker: Mean was 3.4, OpenTelemetry: 6.8). Since Kieker did not have significant changes in its code base, we assume that this is caused by optimizations that happened either in used libraries or in the execution environment, including the JVM (which we updated from OpenJDK 8 to OpenJDK 11) and the Linux Kernel.

3.4 Overhead Scalability

Figure 6 shows the overhead evolution with increasing call tree depth. It shows a nearly linear increase, which indicates that no instrumentation technology causes serious problems such as memory leaks. We observe that OpenTelemetry (with its ByteBuddy-based probes) has a significantly higher overhead. With Kieker, we observe that AspectJ is slower for all tested call tree depths. For source instrumentation, we see a slightly smaller standard deviation and slightly lower average duration than the bytecode instrumentation technologies.

For performance monitoring, warmup performance is also relevant, as a user might want to detect performance anomalies early in the runtime of an application. Figure 7 shows the evolution of the warmup performance. Here, the steady-state performance is

Benchmark	Mean	Standard Deviation	Mean	Standard Deviation
	i7-4770		Raspberry Pi	
Baseline	0.05	0.00	0.16	0.00
Kieker-java				
-aspectj (deactivated)	0.21	0.00	0.93	0.01
-aspectj (nologging)	1.68	0.02	5.63	0.12
-aspectj (full)	3.35	0.07	12.69	2.21
-bytebuddy (deactivated)	0.07	0.00	0.31	0.01
-bytebuddy (nologging)	1.08	0.01	3.42	0.09
-bytebuddy (full)	2.61	0.13	8.10	0.47
-disl (deactivated)	0.07	0.00	0.31	0.02
-disl (nologging)	1.21	0.01	3.94	0.11
-disl (full)	2.75	0.18	8.30	0.67
-javassist (deactivated)	0.06	0.00	0.27	0.01
-javassist (nologging)	1.16	0.01	4.01	0.14
-javassist (full)	2.58	0.17	8.25	0.58
OpenTelemetry-java				
-bytebuddy (deactivated)	3.28	0.15	14.29	0.52
-bytebuddy (full)	4.98	0.18	22.41	1.13

Table 1: Monitoring Overhead for Load Time Technologies (in μs)

Benchmark	Mean	Standard Deviation	Mean	Standard Deviation
	i7-4770		Raspberry Pi	
Baseline	0.05	0.00	0.16	0.01
Kieker-java				
-aspectj (deactivated)	0.22	0.00	1.17	0.05
-aspectj (nologging)	1.66	0.02	5.88	0.19
-aspectj (full)	3.35	0.12	13.75	3.63
-bytebuddy (deactivated)	0.06	0.00	0.27	0.01
-bytebuddy (nologging)	1.17	0.02	3.83	0.17
-bytebuddy (full)	2.53	0.12	8.44	0.89
-javassist (deactivated)	0.06	0.00	0.27	0.01
-javassist (nologging)	1.17	0.01	3.84	0.13
-javassist (full)	2.50	0.06	8.19	0.58
-sourceinstrumentation				
(deactivated)	0.07	0.00	0.28	0.01
(nologging)	1.04	0.01	3.30	0.11
(full)	2.32	0.15	7.20	0.55

Table 2: Monitoring Overhead for Compile Time Technologies (in μs)

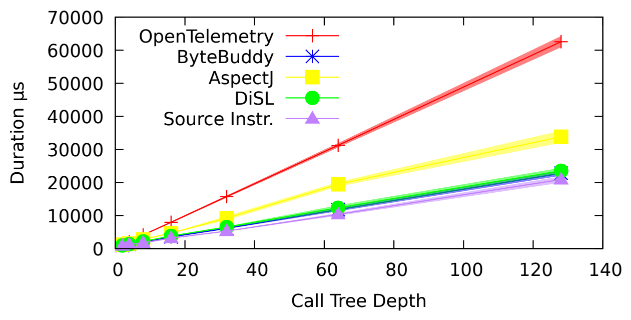

Figure 6: Steady State Performance on i7-4770

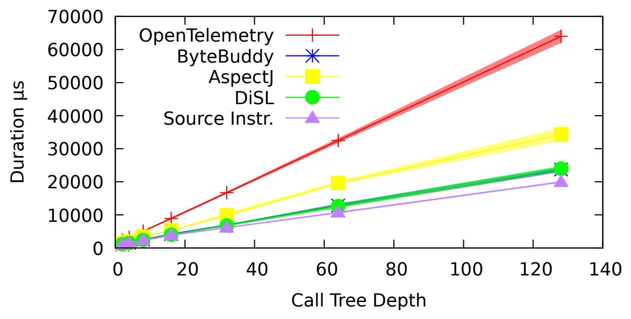

Figure 7: Warmup Performance on i7-4770

mainly repeated, with even lower differences between ByteBuddy, DiSL, and Javassist.

4 RELATED WORK

Related work to the overhead analysis of instrumentation technologies for APM exists in three fields: Analysis of the monitoring overhead of APM tools, analysis of the overhead instrumentation itself, and analysis of overhead of instrumentation for different infrastructures.

4.1 Overhead of APM Tools

Performance and regression benchmarking are a widespread practice for monitoring tools. OpenTelemetry[16] and the monitoring tool GlowRoot[17] provide own benchmarks for their tools. However, comparisons of monitoring tools or of their instrumentation technologies rarely exist. ByteBuddy itself provides a comparison of ByteBuddy, cglib, Javassist, and Java proxies in their basic tutorial.[18] They find that based on different use cases, different instrumentation technologies are faster. Therefore, it is necessary to compare the instrumentation technologies in the context of their usage in monitoring, as we did in this work.

Okanovic et al. [12] already examined the use of AspectJ and DiSL on Kieker. While they pursued the same goal, they did not persistently implement their changes into the Kieker code. Additionally, they used a self-implemented benchmark instead of MooBench and did not examine how manual instrumentation performs in comparison to AspectJ and DiSL.

Banda analyzed OpenTelemetry's performance overhead,[19] which resulted in a commit measuring the span processing overhead.[20] In his work, he compared the performance of different queue types (e.g., `ArrayBlockingQueue` and `ConcurrentLinkedQueue`) with different configurations. He did this only for the exporting part, i.e., he created spans inside of a JMH benchmark. Therefore, in contrast to this work, he focused on the factor of the data processing, whereas we use a benchmark that includes the data creation, data processing, and writing, and focus on the instrumentation part.

Shatnavi et al. [17] examine the monitoring overhead for three human resources and financial management systems, including their web UIs. The systems are built on GWT-Spring, and the instrumentation is done using OpenTelemetry. They evaluate the overhead of a baseline and two usage scenarios of their systems. In their setup, they do not find an overhead of the frontend agent as the frontend is executed at the users' site. Furthermore, they find an overhead of about 3 %-4 % at one backend component and an unacceptable overhead in another component; therefore, they decide to exclude instrumentation for serialization parts of the application.

4.2 Overhead of Instrumentation

Chukri et al. [18] present the BISM (Bytecode-Level Instrumentation for Software Monitoring) tool that allows, like DiSL, to instrument source code. They compare the monitoring overhead with AspectJ and DiSL using the DaCapo benchmark suite and find that BISM creates a lower overhead than both of them. Since the tool is not publicly available, a comparison to BISM is not possible.

Horký et al. [7] use DiSL to obtain performance measurements of performance unit tests. While they also apply DiSL for performance measurement, they focus on the evaluation of performance unit test measurements and do not evaluate the overhead caused by the instrumentation.

Horký et al. [6] examine the monitoring overhead by dynamic instrumentation, with dynamic instrumentation implemented using DiSL. They assume that the monitoring overhead emerges from probe presence, the (byte)code manipulation, the optimization, and the data storage. This is different from our approach since they inject probes dynamically, while Kieker's probes are in the code all the time and are enabled or disabled through variable values. Furthermore, they use the SPEC-jbb2015 benchmark instead of MooBench. Additionally, for their measurement, they consider the recursive call as one method call and only send one method call record, whereas Kieker considers each invocation of recursive calls as a single method call record and sends these as single `OperationExecutionRecord`. Finally, they find that dynamic instrumentation using dynamic injection of probes is a promising and overhead-reducing approach for dynamic monitoring. Since they only used DiSL as instrumentation technology, one further work would be to use the instrumentation technologies we used in

[16]For example https://opentelemetry.io/blog/2023/perf-testing/ — OpenTelemetry has a huge suite of benchmarks for different languages.
[17]https://glowroot.org/overhead.html
[18]https://bytebuddy.net/#/tutorial

[19]https://doordash.engineering/2021/04/07/optimizing-opentelemetrys-span-processor/
[20]https://github.com/open-telemetry/opentelemetry-java/commit/23ce8fe3929d98aaaa63ab8d5d7ab2b99dcea85b

this work. Thereby, it could be checked whether instrumentation overhead can be reduced further, e.g., by using Javassist.

4.3 Overhead of Instrumentation for Different Infrastructures

Bruening et al. [1] build DynamoRIO, which enables instrumentation at the processor operation level. They compare it to Intel's Pin tool that also allows to instrument applications at the processor operation level. Using SPEC CPU2006, they find that DynamoRIO has an overhead of 21 %, whereas Pin has an overhead of 11 %.

Wasabi is an instrumentation technology for Web Assembly (wasm) [10]. Lehmann and Pradel define it and examine its overhead using the PolyBench benchmark suite that was originally developed for benchmarking wasm itself. Based on the instrumented call, they find that overhead might vary from 2 % up to factor 163 when instrumenting all assembler commands.

For Python, Eghbali and Pradel define DynaPyt for program analysis [3]. They find that the overhead varies between 20 % and factor 16.

All experiments on other languages were done using predefined benchmarks, whereas we used a benchmark specifically suited for monitoring overhead. Therefore, the results are not directly comparable. One possible future work would be to apply the probes we created to JVM benchmarks in order to examine the overhead.

5 SUMMARY

In this work, we extended the MooBench benchmark to compare different instrumentation technologies. We compared five instrumentation technologies for the instrumentation framework Kieker: AspectJ, ByteBuddy, DiSL, Javassist, and source instrumentation. We found that AspectJ creates the highest overhead for all configurations, and source instrumentation creates lower overhead in most configurations. For ByteBuddy, DiSL, and Javassist, one or another is faster, depending on the configuration.

In our experiments, we focused on instrumentation that can be enabled and disabled at runtime, thus making adaptive monitoring possible. A static instrumentation that decides on program startup which methods should be instrumented typically has lower overhead. In use cases where this is possible, static probes should be used to obtain minimal overhead.

Reducing the overhead of instrumentation remains a challenge since available technologies change, and, therefore, the most efficient instrumentation technology might change. In future work, we plan to execute our measurements on more heterogeneous hardware and repeat the measurements with future versions of underlying technologies, e.g., newer JVMs or JVMs from different vendors.

Furthermore, MooBench's current implementation focuses on the overhead in terms of CPU time consumption to measure the method execution duration. An extension for other aspects of the overhead (e.g., throughput, CPU, and/or memory usage) and the overhead of measuring different things (e.g., the overhead of jersey for HTTP request processing) would be an important extension in order to minimize performance measurement overhead overall.

Besides monitoring, there are other use cases for instrumentation technologies, including data serialization, logging, mocking, and testing. In this work, we focused on performance for monitoring overhead and, therefore, measured the overhead for changing monitored methods. For other use cases, other aspects of instrumentation, like class creation, might be more important. Therefore, promising future work is also the creation of an instrumentation technology benchmark covering other use cases of instrumentation.

Acknowledgement This work is supported by the German Research Foundation (DFG): Project "SustainKieker" (HO 5721/4-1 and HA 2038/11-1).

REFERENCES

[1] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proc. 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments*. 133–144.

[2] Shigeru Chiba and Muga Nishizawa. 2003. An easy-to-use toolkit for efficient Java bytecode translators. In *Proc. Int. Conf. on Generative Programming and Component Engineering*. Springer, 364–376.

[3] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A dynamic analysis framework for Python. In *Proc. 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 760–771.

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

[5] Wilhelm Hasselbring and André van Hoorn. 2020. Kieker: A monitoring framework for software engineering research. *Software Impacts* 5 (2020), 100019.

[6] Vojtěch Horký, Jaroslav Kotrč, Peter Libič, and Petr Tůma. 2016. Analysis of Overhead in Dynamic Java Performance Monitoring. In *Proc. 7th ACM/SPEC Int. Conf. on Performance Engineering*. ACM, 275–286.

[7] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr Tůma. 2015. Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proc. 6th ACM/SPEC Int. Conf. on Performance Engineering*. ACM, 289–300.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *Proc. 15th European Conf. on Object-Oriented Programming*. Springer, 327–354.

[9] Holger Knoche and Holger Eichelberger. 2018. Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper. In *Proc. 8th ACM/SPEC Int. Conf. on Performance Engineering*. 305–316.

[10] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A framework for dynamically analyzing webassembly. In *Proc. 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 1045–1058.

[11] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A domain-specific language for bytecode instrumentation. In *Proc. 11th Int. Conf. on Aspect-oriented Software Development*. 239–250.

[12] Dušan Okanović, Milan Vidaković, and Zora Konjović. 2013. Towards performance monitoring overhead reduction. In *Proc. 11th IEEE Int. Symposium on Intelligent Systems and Informatics*. 135–140.

[13] Adam Przybyłek. 2018. An empirical study on the impact of AspectJ on software evolvability. *Empirical Software Engineering* 23, 4 (2018), 2018–2050.

[14] David Georg Reichelt, Reiner Jung, and André van Hoorn. 2023. More is Less in Kieker? The Paradox of No Logging Being Slower Than Logging. In *Proc. 14th Symposium on Software Performance*.

[15] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2021. Overhead Comparison of OpenTelemetry, inspectIT and Kieker. In *Proc. 12th Symposium on Software Performance*.

[16] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. 2023. Towards Solving the Challenge of Minimal Overhead Monitoring. In *Comp. 14th ACM/SPEC Int. Conf. on Performance Engineering*. 381–388.

[17] Anas Shatnawi, Bachar Rima, Zakarea Alshara, Gabriel Darbord, Abdelhak-Djamel Seriai, and Christophe Bortolaso. 2023. Telemetry of Legacy Web Applications: An Industrial Case Study. (2023). https://hal.science/hal-04344518/document

[18] Chukri Soueidi, Marius Monnier, and Yliès Falcone. 2023. Efficient and expressive bytecode-level instrumentation for Java programs. *International Journal on Software Tools for Technology Transfer* 25, 4 (2023), 453–479.

[19] Jan Waller. 2015. *Performance Benchmarking of Application Monitoring Frameworks*. Ph. D. Dissertation. Kiel University, Germany.

[20] 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.

[21] J. Waller and W. Hasselbring. 2012. A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring. In *Proc. Int. Conf. on Multicore Software Engineering, Performance, and Tools*. Springer, 42–53.