More is Less in Kieker?
The Paradox of No Logging Being Slower Than Logging

David Georg Reichelt
Lancaster University Leipzig /
Universität Leipzig

Reiner Jung
Christian-Albrechts-
Universität zu Kiel

André van Hoorn
Universität Hamburg

Abstract

Understanding the sources of monitoring overhead is crucial for understanding the performance of a monitored application. The MooBench benchmark measures the monitoring overhead and its sources. MooBench asumes that benchmarking overhead emerges from the instrumentation, the data collection, and the writing of data. These three parts are measured through individual factorial experiments.

We made the counter-intuitive observation that MooBench consistently and reproducibly reported higher overhead for Kieker and other monitoring frameworks when **not** writing data. Intuitively, writing should consume resources and therefore slow down (or, since is parallelized, at least not speed up) the monitoring. In this paper, we present an investigation of this problem in Kieker. We find that lock contention at Kieker's writing queue causes the problem. Therefore, we propose to add a new queue that dumps all elements. Thereby, a realistic measurement of data collection without writing can be provided.

1 Introduction

Monitoring overhead increases the execution time of a program and decreases the accuracy of monitoring data. MooBench measures the monitoring overhead of the application monitoring frameworks Kieker [4], inspectIT[1], and OpenTelemetry[2] [5].

The basic assumption of MooBench is that the overhead emerges from the instrumentation itself, the data collection, and the writing of data [2]. For Kieker, MooBench contains five configurations: (1) activated instrumentation, but no monitoring, (2) activated monitoring, but *no logging*, (3) activated data writing to a text file, (4) activated data writing to a *binary file*, and (5) activated data writing to a TCP receiver. Since MooBench assumes that the three monitoring sources are creating additive overhead, it is assumed that the individual parts of the overhead can be calculated by these factorial experiments — and additionally the overhead of writing to a text file.

The MooBench community maintains a website with regular executions of the MooBench benchmark on the same server.[3] The practical results of these measurements question MooBenchs basic assumption: the measurement with deactivated writing is — and has been for the last years — slower than the measurement with activated writing using the binary writer. Hence, configuration (2) is slower than configuration (4). This effect is present on different execution platforms, for different configurations and even for different frameworks (Kieker and inspectIT).

We examine this effect by factorial experiments. We find that (i) it is reproducible for every practical reasonable configuration, (ii) it is caused by `MonitoringWriterThread.run`, which reads the data from the queue and passes them to the writer, (iii) the overhead of monitoring for *no logging* can be *decreased* by introducing artificial overhead into the writer. We conclude that writing overhead on current hardware is mainly caused by the queue, and not the writing itself, which is run in parallel. Therefore, we introduce a queue that directly discards all records, instead of redirecting them to a writer. This allows realistic measurement of the individual overhead of data collection.

The remainder of this paper is organized as follows. First, we introduce the MooBench benchmark. Afterwards, we describe our experiments. Then, we discuss the implications from our experiments. Subsequently, we discuss related work. Finally, we give a summary and an outlook.

2 MooBench

MooBench measures the performance overhead of monitoring frameworks. It is currently able to measure the overhead of Kieker, OpenTelemetry, and inspectIT in Java [5] as well as the overhead of Kieker for Python [8].

The basic process of MooBench is visualized in Figure 1: MooBench executes each configuration for a given `$NUM_OF_LOOPS` times, where each loop executes each monitoring configuration once. To execute a monitoring configuration, a monitored

[1] `https://www.inspectit.rocks/`
[2] `https://opentelemetry.io/`
[3] `https://kieker-monitoring.net/performance-benchmarks/`

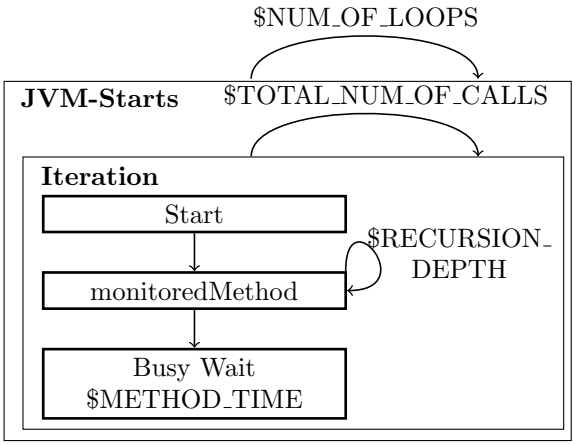

Figure 1: MooBench process

JVM is started and inside the JVM, for a given `$TOTAL_NUM_OF_CALLS`, iterations are executed. In each iteration, the start time is recorded and the `monitoredMethod` is called. This method calls itself for `$RECURSION_DEPTH`. Finally, inside of the method, a busy wait is executed with a given `$METHOD_TIME`.

3 Factorial Experiments

To examine the *no logging* and *binary file* behavior, we first checked the reproducibility with different configurations. Afterwards, we analyzed the benchmark behavior with `perf`.[4] Finally, we examined how the behavior of the activated monitoring with no data writing changes if we introduce artificial overhead. These steps are described in the following subsections.

All experiments have been repeated on OpenJDK 11 and 17 with an i7-4770, i7-6700 und Raspberry Pi 4.[5] The example graphs show the behavior on i7-6700 and JDK 17, but all relations are equal on all examined configurations.

3.1 Reproducibility

To analyze the stability of the effect, we measured the overhead that occured with increase of the calls, the call tree depth, and the method duration. As an example, Figure 2 shows the growth of the execution time depending on the call count. It shows that for small call counts, i.e., 1,000 and smaller, the execution time of *no logging* is lower than the execution time using *binary file*. For higher call counts, *binary file* becomes faster than *no logging*, i.e., the results of the continuous benchmarking is reproducible.

Similar behavior can be observed when looking at the call tree depth and the method time. Overall, we conclude from these experiments that the effect of *no logging* being slower than *binary file* takes place for different configuration options. Hence, it is not only caused by JVM optimizations or special effects.

[4] `https://perf.wiki.kernel.org/`
[5] `https://doi.org/10.5281/zenodo.8197462`

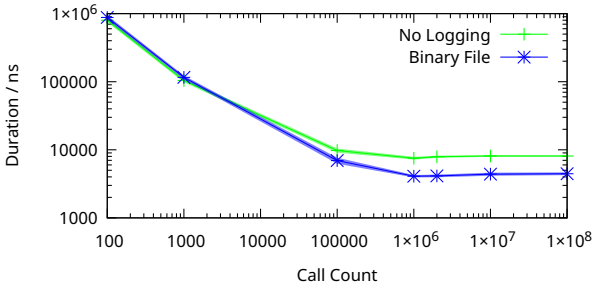

Figure 2: Overhead depending on call count

3.2 Analysis With `perf`

To further analyze the effect, we analyzed the execution using the Linux `perf` tool and the FlameGraph[6] tool. The `perf` tool samples stack traces with a given frequency. By analyis of the stack traces, FlameGraph estimates the durations of method executions. The differential FlameGraph is visualized in Figure 3. By generating a differential FlameGraph, we identified the root cause of the performance differences between *no logging* and *binary file*: The method `MonitoringWriterThread.run` is **faster** for *binary file*. The other methods, including the monitored methods, are not displayed in the differential flame graph, i.e., they did not contain significant performance changes.

The method `MonitoringWriterThread.run` is depicted in Listing 1. It reads the monitoring records from Kieker's central queue (in parallel to the applications threads) and starts their writing.

Listing 1: Main loop of MonitoringWriterThread.run

```
while (record != END_RECORD) {
  writer.writeMonitoringRecord(record);
  record = this.writerQueue.take();
}
```

The call to `writeMonitoringRecord` takes more time than the empty call in the *no logging* variant. We assume that this slows down the insertion into the queue in the monitored thread, i.e., in `MonitoredClassSimple.monitoredMethod`. To check this theory, we introduced artificial overhead in the `DumpWriter`, i.e., the class consuming monitoring data and dumping them without writing.

[6] `https://github.com/brendangregg/FlameGraph`

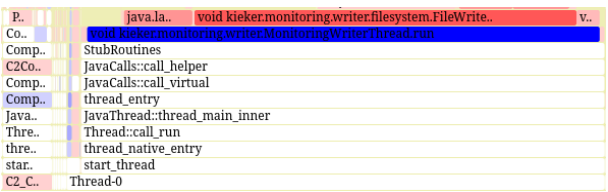

Figure 3: Differential FlameGraph of the execution with call tree depth 100

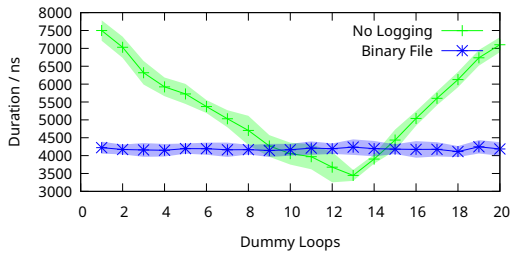

Figure 4: Overhead depending on dummy loops

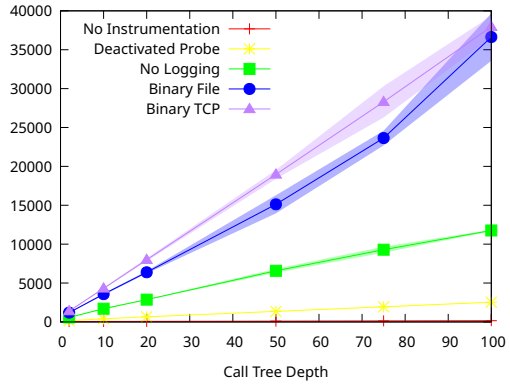

Figure 5: Overhead with `DumpQueue`

3.3 Artificial Overhead in `DumpWriter`

To check how an increase of the writing time in the `DumpWriter` influences the overhead, we added a loop to the `DumpWriter`, that adds random numbers. The loop is configured by the environment parameter `DUMMY_LOOPS`. Figure 4 shows the growth of the execution duration depending on the count of dummy loops. It shows that *binary writer* has nearly constant response time, since it is not affected by the dummy loops. The *no logging* configuration has *shrinking* execution durations, even if the overhead is increased due to the additions of random numbers. This implies that the lock contention in `MonitoringWriterThread.run` is really causing the additional overhead of the *no logging* configuration.

In particular, this is caused by the writer thread being parked and activated after a monitoring record arrives. Writing the data will always take more or at least equally much time as not doing anything; therefore, this behavior cannot be changed on modern execution environments.[7] Instead, we change the monitoring configuration and add a `DumpQueue`, which discards the measured records directly. Thereby, we can realistically measure the monitoring overhead of data collection without logging (see Figure 5).

[7] Slowing down the writing by particularly bad hard disks, e.g., very old hard disks, would change this behavior. Since this would not reflect the typical behavior on modern execution environments, it is not considered further.

4 Related Work

Related work covers the parameterization of benchmarks and the examination of MooBenchs behavior.

Benchmark Calibration Georges et al. [1] define the widely accepted process of performance benchmarking. MooBench, and therefore also our experiments, follow this process. Abdullah et al. [6] examine how to reduce the cost of performance change detection by reducing the number of experiments. Reichelt et al. [7] examine how to spot performance changes of certain size. These works focus on calibration of benchmarks in terms of VMs, iterations, warmup etc. We focus on calibration and implementation of MooBench, which has additional parameters.

MooBench behavior Knoche and Eichelberger also examined the reproducibility of MooBench experiments using the Raspberry Pi [3]. They find that experiments on the Raspberry Pi are replicable on different instances of the Raspberry Pi, and that it therefore can be used for benchmarking.

5 Summary and Outlook

In this paper, we examined why monitoring overhead is lower with deactivated logging than with logging to a binary file in Kieker. We found that the reason is lock contention when using the queue. In future work, we plan to evaluate how multi-threaded workload and different queue implementations affect the overhead. Furthermore, we plan to generalize our analysis to different monitoring frameworks.

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

References

[1] A. Georges, D. Buytaert, and L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *ACM SIGPLAN Notices* 42.10 (2007).

[2] J. Waller and W. Hasselbring. "A Comparison of the Influence of Different Multi-Core Processors on the Runtime Overhead for Application-Level Monitoring". In: *ICMSEPT*. Springer. 2012.

[3] H. Knoche and H. Eichelberger. "Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper". In: *ICPE*. 2018.

[4] W. Hasselbring and A. van Hoorn. "Kieker: A monitoring framework for software engineering research". In: *Software Impacts* 5 (2020).

[5] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Overhead Comparison of OpenTelemetry, inspectIT and Kieker". In: *SSP*. 2021.

[6] M. Abdullah et al. "Reducing Experiment Costs in Automated Software Performance Regression Detection". In: *2022 SEAA*. IEEE. 2022.

[7] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Automated Identification of Performance Changes at Code Level". In: *2022 IEEE 22nd QRS*. IEEE. 2022.

[8] S. Simonov et al. "Instrumenting Python with Kieker". In: *SSP*. 2022.