Evaluating the Overhead of the Performance Profiler Cloudprofiler With MooBench

Shinhyung Yang[1], David Georg Reichelt[2], and Wilhelm Hasselbring[1]

[1]Kiel University, Kiel, Germany
[2]Lancaster University Leipzig & Universität Leipzig, Leipzig, Germany
{shinhyung.yang,hasselbring}@email.uni-kiel.de, d.g.reichelt@lancaster.ac.uk

Abstract

Performance engineering has become crucial for the cloud-native architecture. This architecture deploys multiple services, with each service representing an orchestration of containerized processes. OpenTelemetry is growing popular in the cloud-native industry for observing the software's behaviour, and Kieker provides the necessary tools to monitor and analyze the performance of target architectures. Observability overhead is an important aspect of performance engineering and MooBench is designed to compare different observability frameworks, including OpenTelemetry and Kieker.

In this work, we measure the overhead of Cloudprofiler, a performance profiler implemented in C++ to measure native and JVM processes. It minimizes the profiling overhead by locating the profiler process outside the target process and moving the disk writing overhead off the critical path with buffer blocks and compression threads. Using MooBench, Cloudprofiler's buffered ID handler with the Zstandard lossless data compression ZSTD showed an average execution time of 2.28 microseconds. It is 6.15 times faster than the non-buffered and non-compression handler.

1 Introduction

Cloud-native applications are best characterized by distributed services provided to concurrent users, and deployment of microservices that scale horizontally and vertically through container orchestration, often with Kubernetes [2, 3, 5].

Performance measurement of a cloud-native application is instrumental in preventing failures and alleviating regressions such as prolonged response times [6]. Ongoing efforts include the observability frameworks Kieker [8] and OpenTelemetry [11]. OpenTelemetry defines its data model with the Protocol Buffers serialization framework[1], generating portable API and data exports. Kieker is an observability framework that instruments an application for performance and behavioral analysis.

MooBench addresses another important dimension to the performance engineering of cloud-native applications: performance overhead yielded by performance observability frameworks [1, 10, 16]. It continuously evaluates the performance regression of its targets. In empirical software engineering, benchmarks can be used for comparing different methods, techniques and tools [9]. MooBench is designed for regression benchmarking within continuous integration pipelines [1] of individual monitoring frameworks, not for comparing such frameworks against each other.

Cloudprofiler [15] can instrument both C/C++ and JVM-based applications. Its use case includes profiling distributed matrix computations [13]. Its logging interface is called *handler*, and the identity (ID) handler performs the disk I/O per each log entry. The buffered ID handler reduces the overhead by writing log entries to memory (buffer blocks) instead of disk. The buffered and compressed ID handler further exploits the performance by (1) redirecting buffers from the I/O thread to parallel compression threads, which (2) increases the logs-per-second I/O bandwidth.

In this paper, we use MooBench to evaluate the overhead of Cloudprofiler instrumentation. We added Cloudprofiler to MooBench. MooBench evaluates five Cloudprofiler handlers: the null handler, the non-buffered ID handler, the buffered and binary-encoded ID handler, and two buffered and compressed ID handlers. A buffered and compressed ID handler can be configured with one compression codec, the Zstandard lossless data compression ZSTD codec, or the real-time data compression LZO1X codec. We present the results in Section 4, and share the code[2] and datasets[3].

In the remainder of the paper, we (1) describe how Cloudprofiler is incorporated into MooBench, and go over (2) the experimental setup and (3) results, (4) related work, and (5) finalize in the conclusion section.

2 Instrumenting with MooBench

Cloudprofiler comprises two modules: the target interface via JNI and the C++ shared library. Fig. 1 de-

[1]https://protobuf.dev/

[2]https://github.com/shinhyungyang/cloud_profiler
[3]https://doi.org/10.5281/zenodo.13940072

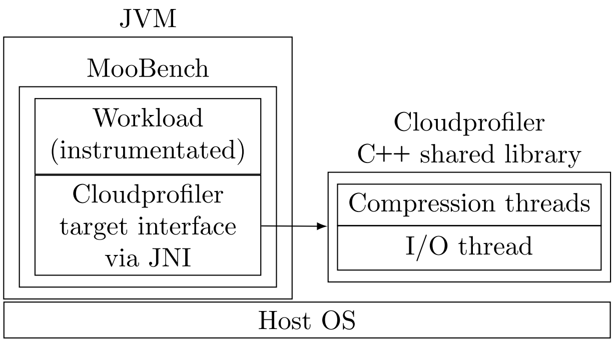

Figure 1: Cloudprofiler Deployment in MooBench

picts the Cloudprofiler modules within the MooBench architecture running as a JVM process on the Host OS. The MooBench workload is instrumented at the source code level with the Cloudprofiler interface. We incorporated the instrumentation structure from [16], which evaluated different instrumentation technologies for JVM applications, including source code-level, and bytecode-level instrumentations.

```
long monitoredMethod(long time, int depth) {
  cloud_profiler.logTS(ch_start, depth);
  try {
    return extractedMethod(time, depth);
  } finally {
    cloud_profiler.logTS(ch_end, depth);
  }
}

long extractedMethod(long time, int depth) {
  if (depth > 1) {
    return monitoredMethod(time, depth - 1);
  } else {
    long exitTime = System.nanoTime() + time;
    long curTime;
    do {
      curTime = System.nanoTime();
    } while (curTime < exitTime);
    return curTime;
  }
}
```

Listing 1: `monitoredMethod` is instrumented to measure the execution time of `extractedMethod`.

The `extractedMethod` function in Listing 1 is MooBench's synthetic workload that represents a trace: the `depth` of the trace and the `time` spent are required. The method is invoked recursively for the number of `depth` and the last invocation returns after a duration of `time` ns. MooBench targets without aspect weaving, e.g., Cloudprofiler, utilize `monitoredMethod` to manually instrument `extractedMethod`. During execution, it deploys two Cloudprofiler channels, `ch_start` and `ch_end`, which measures the span of of `extractedMethod`.

3 Experimental Setup

We extended the main branch of MooBench[4] to incorporate the Cloudprofiler framework. We measured the overhead of three frameworks: Cloudprofiler, Kieker for Java, and OpenTelemetry.

[4] `https://github.com/kieker-monitoring/moobench`

The benchmark is deployed on a bare metal server, operated by Debian 12.6, running OpenJDK 17.0.2, and GCC 12.2.0. It has two Intel Xeon E5-2650 CPUs with eight physical cores on each, and 64 GiB RAM on each NUMA domain. It uses a 480 GB SSD.

We used MooBenchs default configurations, that includes 2 M iterations, where each iteration starts by calling `monitoredMethod` with 10 for `depth`, and 0 for `time`. After an iteration, MooBench collects the elapsed time, the garbage collection counts, and the current used heap memory size. We report the results in Section 4. The measurement repeats 10 times and 20 M results are collected in total. Measuring an observability framework involves more than one configuration, including non-instrumentation, deactivated instrumentation, and other configurations specific to the target framework. MooBench selects 10 M execution results to create statistics for each configuration.

Cloudprofiler configurations We selected five configurations: (1) the null handler, (2) the non-buffered ID handler, (3) the buffered and binary-encoded ID handler, (4) the buffered and ZSTD-compressed ID handler, and (5) the buffered and LZO1X-compressed ID handler. The null handler is an empty JNI function without logging. A buffered handler has 32 buffer blocks, where one block can store up to 1 M log entries. Four compression threads dequeue a non-compressed block from a non-blocking multi-producer/multi-consumer (MPMC) queue, and enqueue a compressed block to another MPMC queue for the I/O thread.

4 Evaluation

We report the execution time results in Fig. 2: the non-instrumentation configuration (1) is the baseline for other configurations. Cloudprofiler's null handler (2) performed 0.461 µs, close to (7) at 0.717 µs. The buffered and binary-encoded ID handler performed 2.294 µs (4), and the buffered and compressed ID handlers performed 2.28 µs (5–6). The Kieker configurations (9, 10) exhibited 5.485 µs and 7.127 µs, respectively. Kieker and OpenTelemetry configurations rarely exhibited GC, less than 10 times during 2 M iterations. We used coefficient of variation (CV) to observe the changes in the used heap memory size. Cloudprofiler, Kieker, and OpenTelemetry exhibited maximum CVs of 11.78 %, 50.39 %, and 54.35 %, respectively.

5 Related Work

Our work aims for measuring the observability overhead, which is the base for overhead reduction. Eder et al. [12] compare the overhead of distributed tracing in cloud environments. They use a microbenchmark that is comparable to MooBench and find that Zipkins agent's overhead is lower than OpenTelemetry's. Reichelt et al. [14] aim for overhead reduction by re-

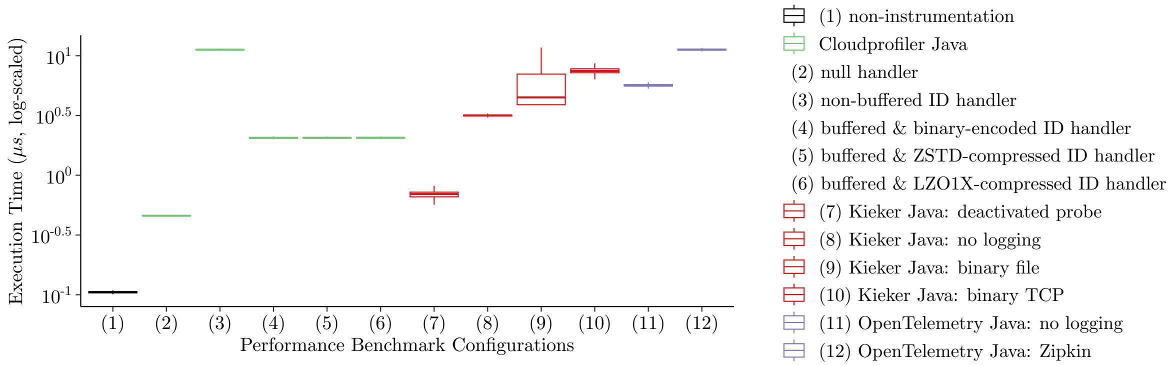

Figure 2: MooBench's evaluation comparison of Cloudprofiler, Kieker, and OpenTelemetry

ducing the values created by Kieker. They find that the overhead can be reduced from 4.77 μs to 0.39 μs (compared to a baseline of 0.05 μs). Lengauer et al. [4] proposed a Java memory tracing techniques. Each tracer thread may compress its buffer before enqueing, which will be read by separate I/O threads. Gebai and Dagenais [7] compared the overhead of system-level tracers with a tight loop microbenchmark that invokes a probe function hooked to a tracepoint instrumentation in application code.

6 Conclusion

In this research, we incorporated Cloudprofiler into MooBench for regression benchmarking, and evaluted its instrumentation overhead differentiated by the use of memory buffers and parallel compression threads. We intend to utilize MooBench for continuously benchmarking Cloudprofiler as well as other frameworks.

Acknowledgment This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834.

References

[1] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps". In: *SIGSOFT Softw. Eng. Notes* 40.2 (Mar. 2015).

[2] B. Burns et al. "Borg, Omega, and Kubernetes". In: *Commun. ACM* 59.5 (Apr. 2016), pp. 50–57.

[3] W. Hasselbring. "Microservices for Scalability". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 2016, pp. 133–134.

[4] P. Lengauer, V. Bitto, and H. Mössenböck. "Efficient and Viable Handling of Large Object Traces". In: *ICPE*. ICPE '16. Delft, The Netherlands: Association for Computing Machinery, 2016, pp. 249–260.

[5] D. Gannon, R. Barga, and N. Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21.

[6] C. Heger et al. "Application Performance Management: State of the Art and Challenges for the Future". In: *ICPE*. 2017, pp. 429–432.

[7] M. Gebai and M. R. Dagenais. "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead". In: *ACM Comput. Surv.* 51.2 (Mar. 2018).

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

[9] W. Hasselbring. "Benchmarking as Empirical Standard in Software Engineering Research". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*. ACM, June 2021, pp. 365–372.

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

[11] D. G. Blanco. *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. APress, 2023.

[12] C. Eder, S. Winzinger, and R. Lichtenthäler. "A Comparison of Distributed Tracing Tools in Serverless Applications". In: *2023 IEEE SOSE*. IEEE. 2023, pp. 98–105.

[13] J. H. Lee et al. "Julia Cloud Matrix Machine: Dynamic Matrix Language Acceleration on Multicore Clusters in the Cloud". In: *PMAM'23*. ACM, 2023.

[14] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Towards Solving the Challenge of Minimal Overhead Monitoring". In: *Companion of the 2023 ACM/SPEC ICPE*. 2023, pp. 381–388.

[15] S. Yang et al. "Cloudprofiler: TSC-based inter-node profiling and high-throughput data ingestion for cloud streaming workloads". In: *arXiv:2205.09325* (2023).

[16] D. G. Reichelt et al. "Overhead Comparison of Instrumentation Frameworks". In: *Companion of the 15th ACM/SPEC ICPE*. 2024.