Benchmarking the Overhead of Distributed Tracing Agents

David Georg Reichelt
Lancaster University Leipzig / URZ Leipzig
Leipzig, Saxony, Germany

Shinhyung Yang
Kiel University
Kiel, Schleswig-Holstein, Germany

Marcel Hansson
University of Hamburg
Hamburg, Hamburg, Germany

Wilhelm Hasselbring
Kiel University
Kiel, Schleswig-Holstein, Germany

Abstract

Tracing is a fundamental technique for analyzing the runtime behavior of software systems. By recording the start and end times of method executions together with contextual metadata, tracing enables detailed performance analysis, architecture reconstruction, and program comprehension. However, such instrumentation inevitably introduces runtime overhead that can distort performance measurements and increase variability. Quantifying and comparing this overhead across tracing frameworks and configurations is therefore essential for selecting suitable tools and ensuring reliable performance evaluations.

The overhead of different tracing frameworks and their configuration can be measured by the MooBench microbenchmark. In this work, we extend the MooBench microbenchmark to support the established Java tracing frameworks Elastic APM Agent, inspectIT, Kieker, OpenTelemetry, Pinpoint, Scouter, and SkyWalking. By executing MooBench with these agents, we find (1) significant differences in performance overhead, whereby the industry standard implementation of OpenTelemetry is comparably slow, while the Kieker agent has the lowest overhead among the functionally correct frameworks, (2) the agents of Pinpoint and Scouter do not store all records, making their behavior not fulfill the functional requirements, and (3) that avoidable overhead of some of the frameworks is created by extensive metadata gathering and needless copying of data.

ACM Reference Format:
David Georg Reichelt, Shinhyung Yang, Marcel Hansson, and Wilhelm Hasselbring. 2026. Benchmarking the Overhead of Distributed Tracing Agents. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3777884.3797004

1 Introduction

Achieving observability is essential for understanding the runtime behavior of a system, including its resource utilization and the emergent runtime architecture. Due to the widespread use of microservice architectures, distributed tracing is becoming a primary tool for achieving observability [2]. While distributed tracing is necessary for understanding resource usage, it introduces additional resource consumption–referred to as *tracing overhead*–for the traced application, and for processing the collected traces. A growing number of tracing agents and frameworks have emerged [22], each offering different trade-offs in terms of features and tracing overhead.

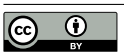
To capture traces, it is necessary to attach *agents* to the systems under observation. Beside functional features, the tracing overhead introduced by these agents is a major criterion when selecting a tracing framework. Existing benchmarks examine the overhead of individual tracing agents in either non-distributed [1, 7, 14, 35, 45, 50] or distributed scenarios [3, 6, 17, 25, 27]. While evaluating individual agents allows for comparing different configurations, scenarios, or versions, it does not facilitate a direct comparison of the overhead and its root causes across different agents.

Despite the widespread adoption of tracing agents, there is limited systematic understanding of the performance differences between agents and the root causes thereof. Such an understanding is vital to select agents based on their overhead and to identify opportunities for overhead reduction in these agents. Prior works comparing different tracing agents [1, 9, 33] are limited to a small selection of tools and lack an in-depth root cause analysis.

To enable a comprehensive comparison of the overhead across different tracing agents, we provide the following contributions:

Analysis of Tracing Agent Popularity: Benchmarking tracing agents requires an initial selection of relevant tools. Since popularity is a key indicator for the long-term viability of open-source projects, we analyze the GitHub star ratings of distributed tracing tools capable of tracing method invocations. We find that Elastic APM Agent, inspectIT, Kieker, OneAgent, OpenTelemetry, Pinpoint, Scouter, and SkyWalking are the most popular tracing agents. Since the overhead for non-distributed workload is a lower limit for tracing overhead in general, we extend the MooBench microbenchmark [45]–a non-distributed benchmark for tracing overhead–to support benchmarking all of these tools. The extension automates the installation, configuration, and tracing process of the agents.

Measurement of Tracing Agent Overhead: Using our extended version of MooBench, we evaluate the overhead of all selected agents. We find that the overhead ranges between 92 *ns* to 657 *ns* per method invocation, with the **industry standard implementation of OpenTelemetry being comparably slow** (315 *ns*). By increasing the count of traced method invocations, we examine the scalability of the tracing agents. Generally, the overhead grows linearly with the number of created records. For Scouter and Pinpoint, the overhead grows linearly with extremely low slope. This low slope is caused by **functional bugs in both agents**, leading to lost tracing records.



ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797004

Root Cause Analysis of Tracing Overhead: Using async-profiler's flame graphs, we analyze the root causes of the overhead, and map the time consumption into five categories. Through in-depth code analysis, we find that agents with **high overhead**, such as OpenTelemetry, conduct **extensive metadata management** and **inefficient implementations**. Consequently, it is possible to reduce the overhead of OpenTelemetry by re-writing these implementations.

The remainder of this paper is organized as follows: First, we describe how tracing works in general and how MooBench measures the tracing overhead. Section 3 describes our process for agent selection and the selected agents, alongside the extensions that are necessary in MooBench to support them. Subsequently, we describe and discuss our measurement results for tracing overhead in Section 4. Based on the identified differences in overhead, we identify root causes for these differences in Section 5. We discuss the results in Section 6 and possible threats to validity of the results in Section 7. In Section 8, we compare our study with related work. Finally, we conclude with a summary and a discussion of potential future work in Section 9.

2 Tracing Overhead Benchmarking

This section first describes how tracing tools work in general. Afterwards, we elaborate on the MooBench benchmark.

2.1 The Tracing Process

The collection of runtime data requires obtaining the data (sometimes called logging), collecting them, processing them, storing them and finally analyzing them, e.g., by creating alerts or visualizing trends. Figure 1 illustrates this process. In this process, the logging is done in the traced system, i.e., in the same process and therefore on the same server. The collection afterwards can be done in a different system, i.e., in a different process potentially on a different server, but requires the traced system to send the data to this separate system. Therefore, we focus in the present paper on the logging and collection part. With Java, these parts are usually done inside an agent, i.e., a separate `jar` that specifies the modification of the executed bytecode, which is required for obtaining the runtime data. Since this overhead introduced into the traced application is crucial for the performance of the traced application itself, MooBench focuses on the agent overhead measurement.

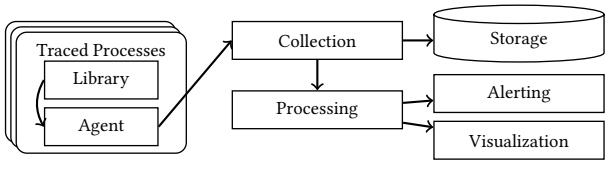


Figure 1: Tracing and analysis pipeline. Adapted from Janes et al. [22], licensed under CC-BY 4.0.

The agent usually contains four steps, that cause the tracing overhead: (1) the instrumentation itself, that weaves the tracing code (*probe*) into the application, (2) the probe execution, that usually obtains call tree information and method start and stop times, and creates a tracing record from them, (3) the insertion of this record into a queue, to allow asynchronous processing of the record, and (4) the writing of the records into a data sink, which could be sending them to another process or serializing them on the hard disk. Figure 2 visualizes these steps.

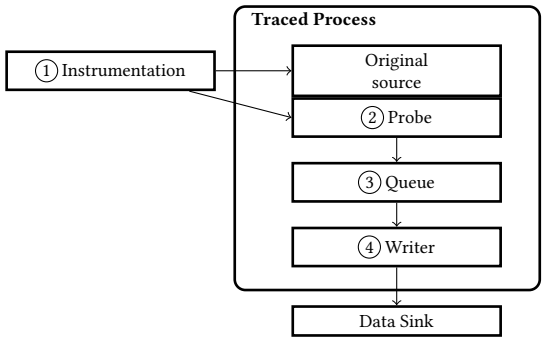


Figure 2: Steps of Distributed Tracing

2.2 MooBench

MooBench "quantif[ies] the performance overhead caused by observability framework components and different observability frameworks".[1] MooBench assumes that tracing overhead is caused by the instrumentation itself (Step ①), the collection of data (Steps ② and ③), and serializing these data (step ④) [47]. The collection of data could be split into the collection itself and the insertion into a queue, which is used for asynchronous processing of collected data. However, a measurement of these different portions of overhead is not possible [32], therefore, MooBench considers these overhead causes as one. To measure this overhead, the core of MooBench is the definition of a benchmarking application containing a `monitoredMethod` , that calls itself recursively for a given `$RECURSION_DEPTH` and waits in the last call for a given `$SLEEP_TIME`. The depth and the sleep time are specified in environment variables. To wait for JVM warmup to finish and allow statistically rigorous performance evaluation, MooBench repeats the measurements `$NUM_OF_LOOPS$` times and repeats the `monitoredMethod` `$TOTAL_NUM_OF_CALLS` times.

This benchmarking application is compiled and used in different tracing frameworks and their configurations. Every framework contains the measurement of a baseline, which is the uninstrumented execution of the application. Additionally, for every framework, MooBench configures the tracing tool in different ways to measure the different sources of overhead, as far as this is possible in the framework implementation. For example, for OpenTelemetry, MooBench contains the following configurations: (1) measurement with *deactivated probe*, to measure the overhead of instrumentation (Step ①), (2) measurement with *no collection*, to measure the overhead of data gathering (Steps ② and ③), and (3) measurement of writing to a *binary file*,[2] to measure the overhead of writing (Step ④). Figure 3 shows how the different frameworks are organized: This basic benchmark definition is used by the frameworks Kieker, OpenTelemetry, and inspectIT with their default agents. In this work, one main goal is to support additional tracing agents.

[1] From: https://github.com/kieker-monitoring/moobench

[2] MooBench also contains a configuration that writes the data into a TCP writer, however, for the Kieker execution, this is not faster than writing the data into a binary file.

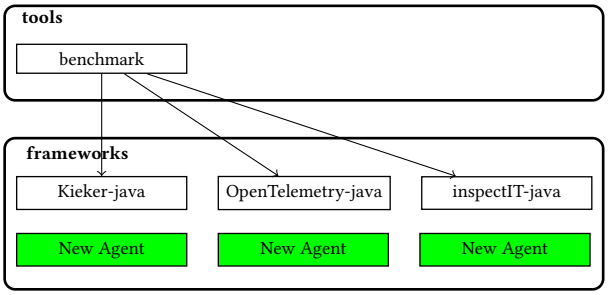


Figure 3: Architecture of MooBench

Tool	URL (github.com/**org/repo-name**)	Stars
Pinpoint	pinpoint-apm/pinpoint	13.4k
MyPerf4J	LinShunKang/MyPerf4J	3.4k
DataDog	DataDog/datadog-agent/	2.9k
Scouter	scouter-project/scouter	2.1k
OpenTelemetry	open-telemetry/ opentelemetry-java-instrumentation	2k
SkyWalking	apache/skywalking-java	800
EaseAgent	megaease/easeagent	580
Elastic APM	elastic/apm-agent-java	568
inspectIT	inspectIT/inspectIT	541
Kieker	kieker-monitoring/kieker	108

Table 1: Benchmarked Tracing Agents

3 Popular Tracing Agents

In this section, we first describe how we selected the agents. Subsequently, we describe how each of them works and how we added them to MooBench.

3.1 Selection of Agents

For our study, we aim to benchmark popular Java tracing agents that are available publicly. We consider agents popular if they have a sufficiently big user base and they are actively maintained. Therefore, our agent selection followed the following systematic process:

- We searched the GitHub API for the keywords "monitoring agent language:Java" "apm agent language:Java", "observability agent language:Java" and "apm javaagent language:Java".
- We manually checked the tools for suitability, especially
  (1) if they had more than 100 stars,
  (2) an English documentation,[3] and
  (3) whether the last commit was newer than six months.
- We manually checked whether the agent supports Java, is executable and aims at generating spans for each method execution (since for some tools, the purpose was obviously different, for example joularjx[4] aims at measuring power consumption and not time consumption).

Sometimes, both outdated and up-to-date versions are online (for example the original inspectIT repository and inspectIT Ocelot[5]). In this case, we reported the Java repository with the most stars, but used the newest version supported by their community.

Since OpenTelemetry does not provide GitHub topics on its Java instrumentation repository, it does not appear as a search result. Due to its widespread usage and importance for standardization of tracing, we manually added it.

Table 1 summarizes the results. Widespread observability tools like Jaeger[6] rely on the OpenTelemetry agent. Since we benchmark the agents in this work, the examination of these tools is out of scope of our paper. MyPerf4j only records aggregated metrics[7] like the average, minimum and maximum response time of a method, but does not allow tracing of individual calls. Therefore, it is omitted from further consideration. Since there are no research licenses available, we also omitted DataDog. We omitted EaseAgent, because it does not support method-level tracing.

3.2 EaseAgent

EaseAgent is an agent-based APM framework, developed and maintained by MegaEase.[8] Following Google Dapper's approach [39], EaseAgent mainly focuses on RPC-based tracing, and its major interests are in tracing Spring Boot-based applications. It provides a Plugin framework for users to develop tracing of any Java-based system. Internally, EaseAgent uses Byte Buddy, and its own Matcher DSL, which is inspired by Byte Buddy. EaseAgent, by default, provides application metrics through Prometheus, and supports several trace backends including Zipkin and Grafana Tempo. We decided to omit EaseAgent from this study because it focuses on service-level tracing, but does not support method-level tracing yet.[9]

3.3 Elastic APM

The Elastic stack, with the core products of the Elasticsearch database and the Kibana visualization frontend, provides a variety of tools for data analysis. One of these tools is the Elastic observability tool, consisting of a server for handling incoming data and the elastic APM agent for instrumentation and gathering trace data. Since the primary goal of the agent is distributed tracing, the Elastic APM agent by default only monitors HTTP requests and method calls inside of these requests. As an alternative, the Elastic APM server supports collecting data from the OpenTelemetry agent.

MooBench Support. To make the Elastic APM agent usable with MooBench, we set the `elastic.apm.trace_methods` property to `moobench.application.*` that lets Elastic APM also monitor the method calls without an HTTP context. Like the OpenTelemetry agent, the Elastic APM agent has a variety of features for handling data, including sampling of traces, compression of spans, and sanitizing fields to avoid passwords being sent to the APM server.[10]

Thanks to the easy starting option, we start the complete Elastic stack using 'docker compose' every time we benchmark a new configuration. While starting using `docker compose` limits the runtime environments, since it requires an appropriate docker

[3] Excluding for example https://github.com/hao117/bee-apm?tab=readme-ov-file
[4] https://github.com/joular/joularjx
[5] https://github.com/inspectIT/inspectit-ocelot
[6] https://www.jaegertracing.io/docs/1.63/client-libraries/
[7] https://github.com/LinShunKang/MyPerf4J/wiki/Metrics
[8] https://megaease.cn/
[9] https://github.com/megaease/easeagent/issues/352
[10] https://www.elastic.co/guide/en/apm/agent/java/current/configuration.html

installation, it is the default installation method for Elastic and thereby makes its use easier. For the Elastic APM agent, we measure the following variants: (1) Deactivated monitoring (`elastic.apm.recording=false`), (2) regular monitoring and (3) monitoring with deactivated sanitizing (`elastic.apm.sanitize_field_names=`).

3.4 inspectIT

inspectIT[11] is an open-source application performance monitoring and tracing framework designed to analyze the runtime behavior of Java applications. It provides automatic instrumentation for collecting timing information, contextual metadata, and request flow data with low manual configuration. inspectIT was first built to support OpenTracing and later on adapted to built on the OpenTelemetry reference implementation. inspectIT relies on YAML configuration files, which enables a wide range of configuration options.

3.5 Kieker

Kieker [18, 42, 52] is an extensible open-source framework for tracing and analyzing the performance of software systems at runtime. It is built on the instrumentation record language (IRL), which allows defining multiple record types that are created by different instrumentation technologies [31] and later on analysed by Kieker's analysis tools. The MooBench microbenchmark has initially been developed to benchmark Kieker's overhead [45].

3.6 OpenTelemetry

Starting with the OpenTracing initiative in 2016 [37], libraries provided standard extensions and auto-instrumentations for various languages. Google open-sourced a similar tool in 2018 [36]. From 2019 to 2023, both tools were merged into OpenTelemetry [38]. The original repositories are archived.

The OpenTelemetry reference implementation[12] supports observability for a variety of languages and technologies via different instrumentation libraries and extension points. Thereby, tracings and gathering metrics becomes possible across different technologies, e.g., it is possible to connect a click of a user within a TypeScript-based application, that creates a REST request, which itself creates a database query. The interaction happens via a public API, which then calls the OpenTelemetry SDK, which is an implementation of the API.

3.7 Pinpoint

Pinpoint is a full APM solution, supporting tracing and metrics collection, instrumentation of a variety of frameworks, sampling from tracing data, and visualization and analysis of the collected data. Its main components are the agent, the collector, and the web view of the data. Distributed tracing often creates streams of data with high velocity, volume and variety, i.e., big data [28]. To enable handling these data, Pinpoint uses a λ architecture with HBase, Kafka, and Apache Pinpoint.

MooBench Support. To make Pinpoint usable with MooBench, we needed to start the required databases, the collector, and the web frontend. Since we do not require live data, we do not start Redis and instead specify `-Dpinpoint.modules.realtime.enabled=false` on collector and web interface start. Afterwards, we start the MooBench application using the Pinpoint Java agent. This can be done using three configurations: With deactivated Pinpoint, with full instrumentation of the MooBench application, and with sampling from the collected data.

3.8 Scouter

Scouter is an open-source APM framework that provides real-time observability for servers and Java applications. It collects and visualizes key metrics such as resource utilization and response times. It consists of agents (Java agent application instrumentation and Host agent for operating system metric collection), a server for saving the metric data, and a client to view the data.

MooBench Support. First, the Scouter server needs to be started and (optionally) the Scouter client to provide the frontend. Afterwards MooBench can be started with the Scouter Java agent. To enable non-HTTP based method profiling, we need to set two configuration options for application start: `hook_service_patterns` to define the service entry points and `hook_method_patterns` to define the methods to be instrumented. As an additional variant to the default, we set `profile_method_enabled=false` to disable the method profiling.

3.9 SkyWalking

Apache SkyWalking is an open-source observability platform that offers application performance monitoring, distributed tracing, and service dependency analysis. It focuses on cloud-native and microservice architectures. It supports many frameworks and languages. Its main components are a server, UI, storage, and various agents for different languages. SkyWalking was first built to be interoperable with OpenCensus and is now compatible with OpenTelemetry, but the tracing code is still independent of the OpenTelemetry reference implementation.

MooBench Support. To monitor MooBench, we need to set up a database and a server. We use BanyanDB, developed by the SkyWalking community, for the database in a Docker container. The server acts as a collector and includes a native UI. Then, MooBench is started with the SkyWalking Java agent. To enable method-level tracing without modifying the source code or developing our own plugin, we use an optional plugin.[13] This plugin enables us to specify which methods should be instrumented. We created three configurations: default (full) instrumentation, sampling of traces, and no data collection. No data collection is done by giving the aforementioned plugin an empty configuration, which means the agent is active, but no actual instrumentation is done.

4 Overhead Benchmarking Results

For benchmarking the overhead of tracing agents, we first measure the single user execution time with the default parameters as the *baseline overhead*. Afterwards, we measure the *scalability* of the overhead regarding call tree *depth*. Based on the results, we examine

[11] https://github.com/inspectIT/inspectit-ocelot
[12] https://opentelemetry.io/docs/specs/otel/overview/
[13] https://skywalking.apache.org/docs/skywalking-java/v9.5.0/en/setup/service-agent/java-agent/customize-enhance-trace/

the low slope of Pinpoint and Scouter. All measurement results are available in our dataset.[14]

4.1 Baseline Overhead

The measured baseline execution times are depicted in Table 2. They ran on two different architectures, x86_64 and ARMv8. They have been executed with MooBench's default parameters, i.e., 2,000,000 executions to overcome warmup and get stable results, 10 starts of the JVM per configuration, and a call tree depth of 10.

For the default tracing configuration on x86_64, the agents follow a distinct performance hierarchy: Kieker < OpenTelemetry < Elastic < Pinpoint < SkyWalking < inspectIT < Scouter. From a practitioner's perspective, this ranking defines the selection order of agents solely based on the performance criterion, though other factors like feature set or usability may also influence the final choice. The slowest framework is Scouter, which creates an overhead of over 14,000 *ns*; this means, an individual call takes (calculated) 1.4 μs. On the other extreme, Kieker is the fastest tool with 0.138 μs per call (calculated).

The x86_64 execution environment is an AMD Ryzen 7 5700G running Linux Kernel 5.14.0 using OpenJDK 21.0.8 and the ARMv8 execution environment is a Raspberry Pi 5, the BCM2712 System-on-Chip (4 Cortex-A76 cores, 8 GiB RAM), running Linux Kernel 6.12.57 using OpenJDK 21.0.9. For the agents and frameworks, we used Elastic APM 9.2.1, inspectIT Ocelot 2.7.0, Kieker 2.0.3-SNAPSHOT, OpenTelemetry Java Agent 1.56.0, Pinpoint 3.0.3 (using hbase 2.6.3, Kafka 7.6.0 and Pinot 1.4.0), Scouter 2.20.0 and SkyWalking 10.2.0 (Agent 9.5.0).

The comparison between the x86 and ARM (Raspberry Pi 5) execution times shows that all frameworks exhibit noticeably higher absolute runtimes on the ARM platform, which reflects its lower single-core performance compared to the x86 system. Nevertheless, the relative ranking of the frameworks remains consistent across both architectures. Scouter continues to incur the highest overhead, with its instrumentation and profiling configuration exceeding 37,000 *ns* on the RPi5 compared to roughly 14,000 *ns* on x86. Similarly, frameworks such as Pinpoint and Elastic APM show substantial slowdowns–around a factor of three to four–when executed on the Raspberry Pi 5. Kieker maintains the lowest per-call overhead on both platforms: while its "Binary file" configuration rises from about 1,900 *ns* on x86 to roughly 4,600 *ns* on ARM.

Overall, the results indicate that although execution times scale up on ARM hardware, the relative performance characteristics of the tracing frameworks are preserved, suggesting that the observed differences stem primarily from CPU performance rather than architectural inefficiencies of the frameworks.

4.2 Depth Scalability

To examine how the frameworks adapt to increasing the method call count, we increased the call tree depth. Therefore, we used 2, 4, ..., 128 as call tree depth and measured the frameworks default configuration that sends data to a data sink via network. The results are displayed in Figure 4, where the area around the line marks the mean $\pm\sigma$. The average relative standard deviation ranges from 0.68 % (Scouter) to 3.66 % (Pinpoint). Since these deviations remain consistently low and do not scale with the call tree depth, we consider the measurements to be reliable.

Configuration	x86_64		ARMv8	
	μ	σ	μ	σ
elasticapm-java				
No instrumentation	64.43	0.11	121.09	3.20
Deactivated probe	150.80	2.72	390.12	6.63
Regular Writing	4,285.96	83.86	14,339.90	1,343.16
Without sanitizing fields	4,290.95	98.59	15,197.10	2,112.33
inspectIT-java				
Deactivated probe	1,133.11	47.21	3,407.49	62.67
No logging	6,846.36	64.27	16,361.40	209.64
Zipkin	7,029.56	124.83	16,781.80	336.91
Prometheus	2,779.65	91.44	7,797.94	189.45
Kieker-java				
Deactivated probe	69.58	0.07	158.46	1.56
No collection	668.18	3.75	2,094.53	13.11
Binary file	1,383.33	36.63	4,631.03	188.63
Binary TCP	1,552.64	28.55	5,151.33	275.91
OpenTelemetry-java				
No logging	2,204.76	84.33	5,704.48	32.27
Zipkin	3,620.98	93.08	8,885.89	183.71
Prometheus	3,209.96	71.94	8313.79	86.30
pinpoint-java				
Deactivated	66.20	1.24	119.81	2.65
No Measurement	67.31	2.05	122.04	4.26
Basic	4,328.52	90.10	11,298.40	1,103.32
Sampling	2,364.14	100.44	6,661.06	490.71
Scouter-java				
Instrumentation and Profiling	14,203.80	129.44	37,776.60	2,474.01
Disabled Profiling	13,236.80	138.47	31,330.70	1,914.03
Skywalking-java				
Instrumentation	4,601.62	101.53	10,670.50	270.36
20Hz Sampling	3,363.73	88.10	7,202.52	195.43
No data collection	64.98	0.10	118.67	0.76

Table 2: Sequential Execution Times (*ns*)

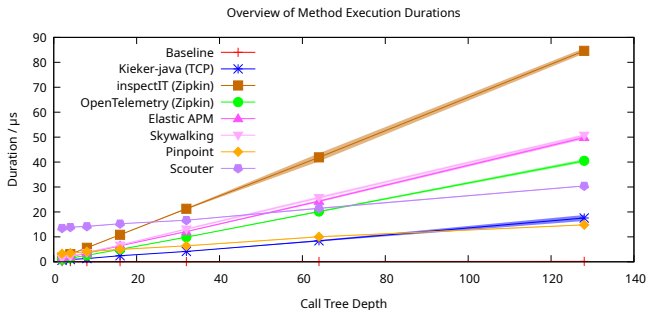


Figure 4: Scalability Results

[14] https://doi.org/10.5281/zenodo.17639772

To increase the understanding of the scalability, we conducted a regression analysis, which is described in Table 3. We see that all of them behave nearly linearly ($R^2 \geq 0.99$). This indicated that our experiments reaches a steady state for every configuration, and no caches are overflowing.

Agent	Slope (a)	Base Overhead (b)	R^2
Elastic APM	384.66	247.92	0.9996
inspectIT	656.79	308.50	0.9999
Kieker	133.92	197.35	0.9985
OpenTelemetry	315.28	47.67	0.9999
Pinpoint	92.79	3348.40	0.9918
Scouter	133.84	13063.42	0.9971
SkyWalking	392.13	566.39	0.9999

Table 3: Linear regression analysis ($y = ax + b$) of tracing overhead across call tree depths (all values in ns).

The regression analysis splits the agents in two groups: For the majority of the agents, we see that the order from the sequential execution time persists (Kieker < OpenTelemetry < Elastic < SkyWalking < inspectIT). For these agents, we were able to measure the overhead, but not identify its root causes. Therefore, we subsequently conducted an in-depth analysis of these sources of overhead in Section 5. In contrast, the two other agents Pinpoint and Scouter have a disproportionately high base overhead ($b = 3348.40$ and $b = 13063.40$) and a low slope, which indicates functional problems. For these agents, the order from the sequential execution time measurement cannot be reproduced: Based on the call tree depth, they produce more or less overhead than the other frameworks. In the following, we examine the root causes for this behavior of Pinpoint and Scouter.

4.3 Low Slope Analysis

To analyze the low slope of Pinpoint and Scouter, we conducted in-depth factorial experiments. To do so, we executed the previously described benchmarks using enhanced debug output.

Pinpoint. Pinpoints architecture is depicted in Figure 5. The span data is obtained in the agent, which is passed as a `-javaagent` to the SuT. Afterwards, the collector receives the span data, and sends them to Kafka, HBase and Pinot for further processing. All of them are used by Pinpoint-web to give the user observability.

For a bigger amount of spans, the data are not received correctly: While the spans are sent by the `SpanGrpcDataSender`, only a portion of spans is received in the `GrpcSpanChunkHandler`. In our experiments, we were not able to detect the reason for this behavior; since no networking issues appeared for the other frameworks, and the issue also appears with local execution on one server, we can exclude network issues. Also a discussion with the Pinpoint team was not able to solve the issue.[15] Therefore, we consider Pinpoint to not fulfill the functional requirement for a tracing system, which is to be able to transport all traces. Consequently, we will not consider Pinpoint for the further analysis.

[15] https://github.com/pinpoint-apm/pinpoint/issues/12970

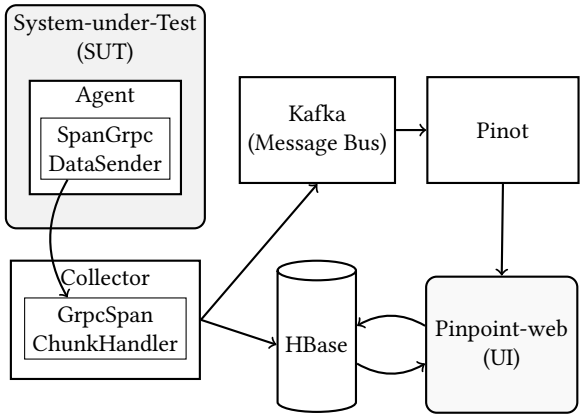


Figure 5: Pinpoint Architecture

Scouter. As mentioned, Scouter consists of an agent, which, similar to the other agents, is passed to MooBench via `-javaagent` and produces profiling records. The agent sends the data to the Scouter Collector server, which processes the data. Users can connect to the server with an additional application, a client, to view the data.

With the default configuration settings, the client only reported a very low number of received records. In test runs with 100,000 method calls in MooBench, only 30%-40% of the records were displayed in the client. Although there is a configuration option of the agent, `profile_step_max_keep_in_memory_count`, which increases the number of records received by the client, it comes at the cost of much longer execution times. This configuration option increases the capacity of an array used for storing the profile data, before sending it to the collector. Increasing the default value of 2,048 to 1,000,000 would result in 99% of the records being visible in the client, but increases the execution time by about 3,500%. This puts Scouter far behind all the other considered frameworks and even with the loss of records, it already was the slowest. Considering, that even with the long execution time, there was still data lost, makes Scouter incomparable to the other frameworks. Consequently, we will not consider Scouter for the further analysis.

5 Root Cause Analysis

After benchmarking the overhead of the frameworks, we investigate the root causes of the tracing behavior. To do so, we first examine why the overhead of some of the frameworks grows slower than linearly. Afterwards, we examine for the remaining frameworks what the root causes for the overhead are. Finally, we investigate the heap usage of the frameworks.

5.1 Approach

Since the tracing frameworks provide partially similar functionality, we examined why the performance differences between the frameworks arise. To do so, different options exist: We could execute factorial experiments, like the original MooBench did it with Kieker. This has the advantage that it does not require additional technologies involved, but it has the disadvantage that the performance of parts of the tracing tools is not necessarily additive, e.g., enabling instrumentation but disabling tracing might not cause any

overhead, but the instrumentation with enabled tracing might still be causing a part of the overhead. Another variant would be tracing the tracing tools with themselves. While this approach has been pursued in the past, it requires choosing one of the frameworks and yields fine-grained tracing values, while we only need an execution profile of the methods to get the overhead sources. Therefore, we decided to use a profiling tool that interacts with the JVM directly.

This would be possible using various sampling tools, including perf and visualvm. To avoid a safepoint bias, we decided to use async-profiler.[16] Among the profiling sampling tools, it is one of the tools with the lowest differences between VM runs [4]. To get profiles of each run, we attached async profiler 10 seconds after the VM start to the process, and afterwards used async-profiler's tooling to generate flame graphs. Figure 7 shows an exemplary flame graph for an OpenTelemetry run. To remove MooBench's data processing and Java's compilation threads, the root `MooBench.monitoredMethod` is selected. In this flamegraph, we see the typical behavior of a traced MooBench execution: `MooBench.monitoredMethod` calls itself, and in every level, additional calls from OpenTelemetry are added. These calls usually follow the same structure, but due to the sampling frequency, some of the calls are missing, especially in the lower levels of the call tree. Besides each `monitoredMethod` call, we see the frameworks behavior. Here, we see that there are three methods from OpenTelemetry called, that themselves create some calls. Some of them finally result in native calls (beige and red). The flame graphs for the other tracing agents are provided in the appendix.

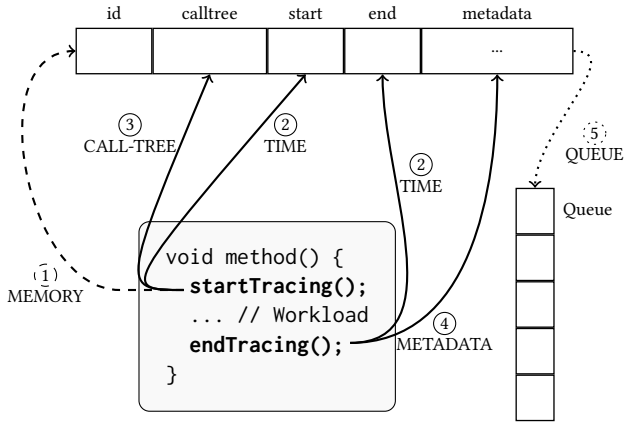


Figure 6: Tasks of Tracing Frameworks

To determine the time-consumption causes of the calls, we parsed the `.collapsed` files of async profiler, which define the call tree and how often each trace of the call tree was detected. Afterward, we manually inspected the source code of the tracing tools. To do so, we distinguished between five source of time consumption, that are summarized in Figure 6. In particular, these are individual tasks that tracing frameworks need to do: (1) MEMORY: Tracing records need to be created or reused, resulting in either allocation or reuse time consumption. This needs to be done at the beginning of the traced method. (2) TIME: The tracing record's start and end time, and potentially other time calls, need to be done. This needs to be done at start and end of the method call. (3) CALL-TREE: The call tree needs to be stored, putting the methods into an order. This typically happens in the beginning of the method. (4) METADATA: Metadata need to be obtained, which at least contain the called methods name, but often also a tracing record id, transaction information, or session information. This can be done at the start of the end of the method. (5) QUEUE: The monitoring records need to be put into a queue for further processing. Afterwards, the processing is done in different threads, so the production thread can go on. We manually analyzed the source code and tagged methods from the frameworks if they serve exactly one of the tasks; if they serve multiple purposes, we tagged child methods.

5.2 Task Overhead Analysis

Figure 8 shows the distribution of samples for each task. The count of samples depends on the length of a benchmark execution, and on the JVM state during an execution; e.g., if a garbage collection takes place, there are no samples showing this, but still, the time consumption is increased. Therefore, mapping the samples to tasks shows the *relative* time consumption of one task; one sample does not correspond to a certain time unit. To estimate the time spend in each task, we additionally multiplied the tasks by the overall execution duration per method invocation.

Additionally, one source of tracing overhead is the instrumentation overhead. With ByteBuddy-instrumented software, we do not see this overhead and it is considered to be low. However, inspectIT is built on javassist.[17] Thereby, it supports custom instrumentation approaches specified as `.yaml` configuration files. To do so, it dynamically replaces classes and injects intermediate classes with the `.yaml`-defined actions, that call the original classes. This causes additional overhead that cannot be attributed to the above-mentioned tasks. Additionally, by changing the classes, code is added that cannot be attributed to one of the tasks, e.g., `java.lang.Long#valueOf` is called directly from the `monitoredMethod`. Therefore, mapping is less exact for inspectIT.

In the following, we describe our analysis of why the frameworks behave differently for the five tasks.

Memory Management. Most of the frameworks create records on the fly, which uses some heap, but does not slow down the process significantly. Therefore, the time share for memory management is close to zero. A notable exception is ElasticAPM, which uses object pools for their spans.[18] This results in increased time for recycling objects, but less memory usage.

Time. Calls to the time function are essential to get start and end times of a span. There are two things that are handled differently here: The implementation of the time function and the call frequency to the time function.

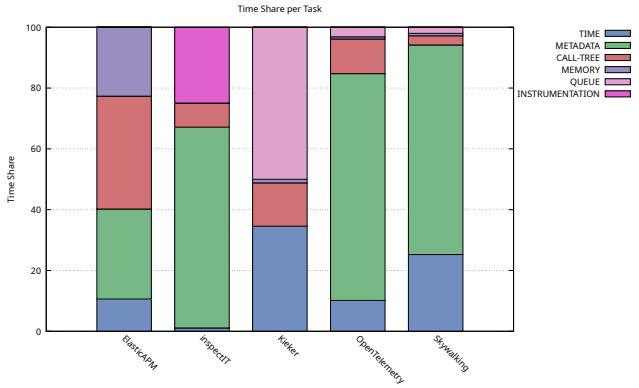
The implementation of the time function usually relies on a Java call to `System.nanoTime`. This call returns a time value in `long`

[16] https://github.com/async-profiler/async-profiler

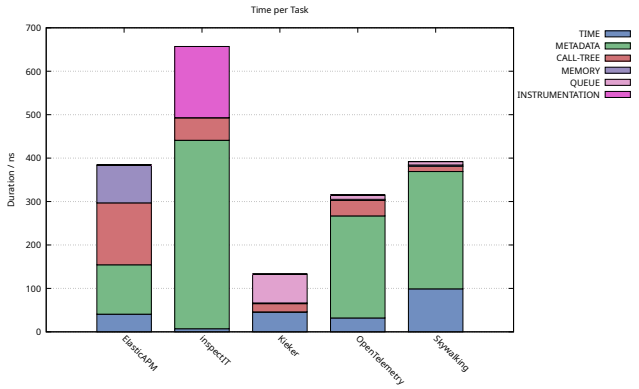
[17] https://github.com/inspectIT/inspectit-ocelot/blob/master/inspectit-ocelot-documentation/docs/instrumentation/actions.md

[18] https://github.com/elastic/apm-agent-java/blob/6e71b29f45c8b26ceef28e0ac46bb6d0120090fd/apm-agent-core/src/main/java/co/elastic/apm/agent/impl/ElasticApmTracer.java#L125

Figure 7: Example Flame Graph: OpenTelemetry



(a) Relative duration root causes



(b) Absolute duration root causes

Figure 8: Duration Root Causes: Relative (Top) and Absolute (Bottom) Values.

that is usable to calculate durations by calculating differences, relying internally on linux `clock_gettime(CLOCK_MONOTONIC)`. It is not usable for absolute timestamps. When tracing distributed systems, absolute timestamps are necessary. Therefore, the frameworks provide own implementations of getting the start nanoTime and calculate it as an offset using `System.currentTimeMillis`, which itself relies on `clock_gettime(CLOCK_REALTIME)`, a function that provides an absolute timestamp.[19] In particular, OpenTelemetrys contains `AnchoredClock`,[20] Kieker contains `SystemNanoTime`,[21] ElasticAPM contains `EpochTickClock`,[22] and inspectIT uses OpenTelemetry's implementation via reflections, since the OpenTelemetry version is not fixed at runtime.[23]

The call frequency to the time function depends on the functional requirements of the framework. Two calls per method invocation are the minimum, since start and end time need to be measured. Kieker and ElasticAPM call the time function twice, at the beginning and the end of a method invocation. OpenTelemetry additionally calls the time function every time a batch of spans is exported. SkyWalking is another exception: It calls the time function six times, before and after the SkyWalking before method interception code, and before and after the SkyWalking after method interception code, `System.nanoTime` is called to get the duration of the interceptions.[24] Additionally, for every span, start and end time are gathered using `System.currentTimeMillis`.[25] These calls, that are used to calculate metrics, cause additional overhead compared to other frameworks.

Call Tree. Reconstructing the call tree requires creating a reference from the parent to each child or vice versa. In distributed traces where parallel executions of methods can happen, these information cannot be reconstructed from span timings.

[19] See JDKs `os_posix.cpp`: https://github.com/openjdk/jdk/blob/b349f661ea5f14b258191134714a7e712c90ef3e/src/hotspot/os/posix/os_posix.cpp#L1557 and linux `clock_gettime`: https://linux.die.net/man/3/clock_gettime

[20] https://github.com/open-telemetry/opentelemetry-java/blob/main/sdk/trace/src/main/java/io/opentelemetry/sdk/trace/AnchoredClock.java

[21] https://github.com/kieker-monitoring/kieker/blob/main/monitoring/core/src/kieker/monitoring/timer/SystemNanoTimer.java

[22] https://github.com/elastic/apm-agent-java/blob/main/apm-agent-core/src/main/java/co/elastic/apm/agent/impl/transaction/EpochTickClock.java

[23] https://github.com/inspectIT/inspectit-ocelot/blob/a9bd839e7bf8a2a789da386b75cc4af91aee831b/inspectit-ocelot-core/src/main/java/io/opentelemetry/sdk/trace/OcelotAnchoredClockUtils.java#L52

[24] https://github.com/apache/skywalking-java/blob/main/apm-sniffer/apm-agent-core/src/main/java/org/apache/skywalking/apm/agent/core/plugin/interceptor/enhance/v2/InstMethodsInterV2.java#L61

[25] https://github.com/apache/skywalking-java/blob/53a00a69372f79af1ab2e57e488ce57ddb0c610a/apm-sniffer/apm-agent-core/src/main/java/org/apache/skywalking/apm/agent/core/context/trace/AbstractTracingSpan.java#L160

OpenTelemetry requires a `Context`, managed by a `ContextStorage`. On every creation of a span, the current `ArrayBasedContext` is copied into a new instance, leaving space for two new entries, a key and a value.[26] The context stores the current span, with the key `SpanContextKey.KEY`.

Like OpenTelemetry, ElasticAPM maintains a `Context`, which is by default a `TraceContext`. This context maintains information like the header of requests. The call tree is stored using a `ThreadLocal` instance of `ActiveStack`. For every call in the current stack, the count of references in the current context is increased and the current span is pushed to an `ArrayDeque`. After the call is finished, the span is removed again. The same mechanism is applied for transactions. While this is more efficient as OpenTelemetry's approach for copying arrays, it still requires maintaining a stack with contexts, which is time-consuming.

Similarly, SkyWalking stores the current stack in a `TracingContext`.[27] Unlike OpenTelemetry, the stack instance is reused.

The Kieker framework serializes the call tree structure by execution order index (`eoi`) and execution stack size (`ess`) [41, p.8]. The `eoi` stores how many methods have been executed before the current method call; the `ess` stores how many methods are above the current method in the call stack. While the `eoi` is incremented with every new call, the `ess` is reset to its prior value at the end of a method invocation; however, during the invocation, it is incremented. Both values are stored inside of `ThreadLocal` variables, so the same thread uses the same values.

Metadata. Metadata at least contain the class and method name, and might contain more data.

Kieker has the lowest metadata management overhead, since it only obtains two types of metadata: From the instrumentation technology, it extracts the method signature, e.g., the `Advice.Origin` from Byte Buddy. Therefore, no parsing of classes is necessary. Furthermore, the trace id is obtained once a new thread is started to be monitored; since this is only done once, it is not visible in the flame graphs.

In contrast to the other frameworks, it takes a significant amount of time for OpenTelemetry to get the class and method name. OpenTelemetry extracts the class and method name using `CodeSpanName-Extractor`. This class does an `indexOf` and a string concatenation. For the class name itself, a cache inside `ClassNames` is used; using the cache also takes a significant amount of time. Furthermore, OpenTelemetry allows for the storage of `Attributes`, that are key-value pairs which are converted into a hash map on every method call. These `Attributes` instances are created using the `CodeAttributesExtractor`, which stores the class name and the method name inside of another `HashMap`. Furthermore, for every span, the thread id and name are set using `AddThreadDetailsSpan-Processor`, and a random id is generated using `RandomIdGenera-tor`. Due to this extensive metadata management, OpenTelemetry is significantly slower than Kieker.

Likewise, inspectIT derives the name in `ContinueOrStartSpan-Action.getSpanName`. This requires regular reflection calls and string concatenations.

Similarly, SkyWalking derives the name of the currently called method and its parameters using reflections. Afterwards, it stores all metadata in a map with constant fields, like `CONFIGURATION_ATTRIBUTE_METHOD` for the method name.

Like Kieker, ElasticAPM gets the signature using Byte Buddy's `Advice.Origin`. Additionally, ElasticAPM tracks various data of each transaction and regularly calculates them in the production thread using `trackMetrics`. This creates a significant overhead for metadata management.

Queue. To further process the monitoring data, each framework needs to pass the data into a queue. To avoid allocating memory, all frameworks use fixed size queues. Therefore, a problem with queue overflow can potentially occur if more records are inserted than removed.

Kieker uses a `MpscArrayQueue`[28] to pass monitoring records from the monitored thread to the writer thread. The writer thread continuously sends single spans, as soon as a span is available. Therefore, the inserting thread potentially has to wait until a lock is cleaned.

OpenTelemetry uses the `MpscAtomicArrayQueue`, which is a variation of the `MpscArrayQueue`. Around the queue, OpenTelemetry uses by default the `BatchSpanProcessor`, which waits until the queue is reached a certain size. Afterwards, a batch of spans are sent with the current exporter, e.g., via gRPC to Zipkin. Due to the extensive overhead in the other parts, writing into the queue is a minor part of OpenTelemetry's overhead.

ElasticAPM uses the disruptor library's ringbuffer.[29] Thereby, it keeps queue overhead at a minimum.

SkyWalking uses an `ArrayBlockingQueue`.[30] It only sends trace segments after they have been completed, i.e., after the stack is empty again.[31] Thereby, it assures low overhead even with a standard queue.

6 Discussion and Lessons Learned

Through our root cause analysis, we learn that overhead differences are caused by functional differences and inefficient implementations.

Functional differences are differences in metadata handling. Properties were stored in different formats in the thread's context or in the spans. Based on the use case, there might be situations where one or the other is necessary, therefore we cannot infer any recommendations from these functional differences. For the frameworks that are not recording all traces, we are unsure whether this is due to undocumented functional requirements or functional bugs. Therefore, we can also not imply any recommendation here.

[26] https://github.com/open-telemetry/opentelemetry-java/blob/3f934f607064999990b86a812b47c91b1715095e/context/src/main/java/io/opentelemetry/context/ArrayBasedContext.java#L68

[27] https://github.com/apache/skywalking-java/blob/53a00a69372f79af1ab2e57e488ce57ddb0c610a/apm-sniffer/apm-agent-core/src/main/java/org/apache/skywalking/apm/agent/core/context/TracingContext.java#L81

[28] https://javadoc.io/doc/org.jctools/jctools-core/3.3.0/org/jctools/queues/MpscArrayQueue.html

[29] https://lmax-exchange.github.io/disruptor/

[30] https://github.com/apache/skywalking-java/blob/53a00a69372f79af1ab2e57e488ce57ddb0c610a/apm-commons/apm-datacarrier/src/main/java/org/apache/skywalking/apm/commons/datacarrier/buffer/ArrayBlockingQueueBuffer.java#L32

[31] https://github.com/apache/skywalking-java/blob/53a00a69372f79af1ab2e57e488ce57ddb0c610a/apm-sniffer/apm-agent-core/src/main/java/org/apache/skywalking/apm/agent/core/remote/TraceSegmentServiceClient.java#L173

However, regarding the implementations, we have seen decisions that are inefficient. Therefore, looking at the implementations, we recommend evaluating whether the following improvements can reduce the tracing overhead: (1) Time: The calls to the time function in SkyWalking could be reduced by optionally obtaining durations of its own operation. Furthermore, SkyWalking could use a time function comparable to OpenTelemetry's `AnchoredClock` to reduce the overhead. (2) Metadata: OpenTelemetry, Elastic APM, SkyWalking, and inspectIT should try to avoid copying the `HashMaps` containing the metadata. Furthermore, OpenTelemetry, inspectIT, and SkyWalking could get the metadata directly from the instrumentation tool instead of using reflections. (3) Call tree: OpenTelemetry could try to remove the copying of stacks. All tools could try to recover the call tree using execution-order-index and execution-stack-size information, instead of storing references within objects. (4) Memory: All frameworks, except the Elastic APM agent, could evaluate whether memory management could be done more efficiently using a ring buffer, instead of re-creating objects. (5) Queue: All frameworks could check whether SkyWalking' approach to send full traces instead of sending every span directly can reduce the overhead.

7 Threats to Validity

Threats to validity concern internal, external, and construct validity.

Internal validity. The internal validity of this study depends on the extent to which the observed performance overhead is truly caused by the tracing frameworks rather than uncontrolled factors. All experiments were executed under consistent conditions, including fixed JVM settings, controlled warm-up phases, and repeated runs to reduce noise. Nevertheless, internal validity may be affected by differences in the implementations of the tracing agents, such as hidden background threads or batching mechanisms that may influence the timing. Although care was taken to ensure consistent measurement procedures, residual nondeterminism in the JVM and garbage collection may still influence measurements.

External validity. External validity concerns whether the results generalize beyond the specific benchmarking environment. MooBench focuses on microbenchmarking isolated method calls, which is representative for measuring instrumentation overhead but may not reflect the behavior of long-running or highly concurrent production systems. Additionally, the evaluations were conducted on a limited selection of hardware platforms, so performance characteristics may differ on other architectures, operating systems, or JVM versions. Therefore, although the results provide strong comparative insights, practitioners should be cautious when extrapolating the exact overhead values to real-world workloads.

Construct Validity. Construct validity addresses whether the study accurately measures what it intends to measure–in this case, overhead introduced by the tracing frameworks. MooBench's design aligns closely with this construct, as it quantifies additional latency introduced by instrumentation at method boundaries. However, tracing frameworks vary in their intended use cases, and some include richer metadata collection or asynchronous pipelines that are not fully captured by micro-level measurements.

8 Related Work

The most closely related work examines the overhead of *tracing* itself. Besides this, other works examine tracing overhead for *microservice interaction* and *other observability technologies*. These works are discussed in the following.

8.1 Tracing Overhead

Tracing tools sometimes examine their overhead themselves, for example, Pinpoint maintains a benchmark online,[32] OpenTelemetry maintains its own microbenchmarks,[33] and Cloudprofiler reports 2.2 % overhead measured by its authors [50]. Since these measurements are vendor-specific, they give an indication of the overhead that can be expected from each tool, but are not usable as a starting point for comparison of tracing tools.

Hence, the generic MooBench microbenchmark [47] has been developed, which is the base of this study. It has been used in different studies to examine how tracing overhead can be reduced [34, 40, 46], how benchmarking can be included into CI [45], and to compare the overhead of tracing frameworks and their configurations [11, 24, 31, 33, 51]. We build upon these works by extending MooBench for multiple tracing frameworks.

Three recent works in the context of NASDAQ evaluated OpenTelemetry's performance overhead [35]. Elias [12] evaluated how the overhead changes with different deployments of the data collector. Karkan [23] investigated the overhead reduction by sampling strategies. While they provide valuable information for their context, they do not aim for generalizability.

Adaptive instrumentation aims to only trace some methods or to activate and deactivate tracing on-the-fly. This can potentially reduce performance overhead and still make it possible to detect performance regressions and to recover the software architecture. Adaptive instrumentation has been applied in the context of Java [10, 19, 48], C [16], and in the HPC context for Score-P [26]. The authors find that in general, adaptive instrumentation helps to reduce the tracing overhead. Nevertheless, they also observe counter-intuitive effects, e.g., that insertion of probes on-the-fly might lead to speed-ups in software execution [19].

8.2 Microservice Interaction

Ahmed et al. [1] compared the functionality of three commercial tracing tools (AppDynamics, New Relic, and Dynatrace) and one open source tracing tool (Pinpoint). By injecting performance regressions, execution of load tests and analysis of the resulting data, they find that all tools are able to detect the injected regressions. However, they do not examine performance overhead.

Eder et al. [9] compare distributed tracing tools in serverless applications by defining their own benchmark. They use the tools Zipkin, OpenTelemetry, and SkyWalking. By comparing the runtime of Node.js microbenchmarks on AWS t2 EC2, they find that Zipkin introduces the lowest runtime overhead (10.73 %).

Instead of tracing inside of applications, various approaches consider applications as black boxes and only trace their interaction

[32] https://pinpoint-apm.gitbook.io/pinpoint/performance
[33] https://open-telemetry.github.io/opentelemetry-java/benchmarks/

[3, 6, 25, 27]. For example, eBPF can be used to measure microservices as a black box [3]. They evaluate their implementations using the DeathStarBench benchmark [13], which is a microservice benchmark application, and find that the overhead is at maximum 4,49 %. Due to the different nature of their system under test (macro benchmark) and the reporting of overall values, instead of per-call values, it is not possible to compare their overhead results to ours.

Hammad et al. [17] study the performance overhead of automated code instrumentation in containerized microservices by comparing instrumented systems against a baseline without instrumentation running on the public AWS and Azure clouds. Their goal is to highlight the need for developing less impactful instrumentation techniques. Conversely, we compare the overhead of different instrumentation techniques, which are implemented as tracing agents. Our analysis may help to achieve the goals of Hammad et al. [17].

8.3 Other Observability Technologies

Tracing is also implemented outside of microservice-targeted applications, mainly in the Linux Kernel, using sampling, and in the context of HPC.

In the Linux kernel, a variety of tracing tools [7, 49] and overhead benchmarks [7, 14] for these tools exist. Desnoyers and Dagenais [7] measured the overhead of LTTng directly after its definition, both using a micro- and a macro-benchmark. They find that the overhead of LTTng is on average 288.5 cycles (96.15 ns) on a Pentium 4 in 2006. Even with this nowadays outdated hardware, their kernel-level tracing creates much lower overhead than Java tracing tools create today. With the increasing adoption of eBPF-based tracing, two recent studies examined their overhead [8, 43]. In both studies, they find that the overhead depends on the configuration and the type of used probes, and that the overhead is low enough in certain scenarios to use the eBPF tracing in production.

Sampling is periodically acquiring the current stack trace.[34] This makes it possible to reconstruct call trees and execution durations of methods. Since the trace is acquired periodically, the overhead is much lower, but at the same time, the accuracy declines. Therefore, various studies [4, 30] examined the accuracy of sampling. They find that sampling tools tend to collect stack traces in safepoints and that they often do not agree on which method should be considered a hot method. Due to the different data acquisition approaches, the related works in the field of sampling are different from the examination of tracing.

In the context of HPC, the VampirTrace tool makes it possible to trace distributed applications [29]. Hunold et al. [20] examine the overhead of tracing tools using real-world HPC applications from the ECP Proxy Applications. They report overhead of up to 20 %, depending on the application. Since they reported relative values, these are not comparable to MooBench's absolute per-method overhead values. Ilsche et al. [21] examine how to combine sampling and instrumentation in the context of HPC. Since they do not report overhead numbers, it is not possible to compare their approach to the described tracing tools.

[34] Not to confuse with sampling in the context of OpenTelemetry, which means selecting a subset of all traces to reduce processing time.

Giamattei et al. [15] did a literature review on monitoring tools. Because for many microservice-based tools no scientific publications exist, they followed the "grey literature review" research method. The authors covered 71 tools, filtered by keyword searches on Google and GitHub. The selected tools include Apache SkyWalking, Elastic APM, OpenTelemetry, Pinpoint, MyPerf4J, DataDog, and Dynatrace. They analyze technological features of the tools, such as the monitoring granularity, but not the monitoring overhead.

9 Summary

In this work, we presented a micro-benchmark for tracing agents that covers all major open-source tracing solutions for the JVM. To achieve this, we identified popular tracing repositories on GitHub, evaluated their overhead using MooBench, and analyzed the root causes of overhead differences. This was done using async-profilers sampling and manual tagging of the methods from the call tree. We found that out of seven examined frameworks, two (Scouter and Pinpoint) do discard spans even if not configured this way. Additionally, we we found that many frameworks create avoidable overhead, especially when doing metadata mangement. This includes the industry standard tool OpenTelemetry copying its stack data between similar data structures. Consequently, we conclude that there are options to reduce the overheads created by these tracing agents.

For future work, we identify three promising directions: extending the execution environment, supporting additional workloads, and evaluating the improvement options derived from our analysis.

First, our measurements were conducted solely on desktop and server hardware. With **different execution platforms**, including alternative JVM implementations such as GraalVM or Eclipse OpenJ9 and diverse hardware architectures such as ARM or RISC-V, the overhead may vary substantially. A comparative study on these platforms would complement our current results and broaden the applicability of our findings.

Second, our current workload focuses on non-distributed, CPU-intensive application behavior. With **other workloads**, such as asynchronous request handling in microservice architectures, disk-intensive operations, or highly parallel execution, tracing overhead may change considerably. Moreover, session-ID handling and relationships to other request types (e.g., database operations) can introduce additional resource usage. Extending MooBench to capture such scenarios would therefore be a valuable next step.

Finally, our root-cause analysis showed that different frameworks require varying amounts of time to perform conceptually similar tasks. To reduce overhead, future work should implement and **benchmark the improvement options** identified in this study and assess their effectiveness in practice.

Acknowledgment

This research is funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. 528713834, and UK Research and Innovation (UKRI) through the UKRI Metascience Research Grants program (Reference S26368).

References

[1] Tarek M Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E Hassan, and Weiyi Shang. 2016. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: an experience report. In *Proceedings of the 13th International Workshop on Mining Software Repositories*. ACM, 1–12. https://doi.org/10.1145/2901739.2901774

[2] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. 2021. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing* 19, 1 (2021), 9.

[3] Rolando Brondolin and Marco D Santambrogio. 2020. A black-box monitoring approach to measure microservices runtime performance. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–26.

[4] Humphrey Burchell, Octave Larose, Sophie Kaleba, and Stefan Marr. 2023. Don't Trust Your Profiler: An Empirical Study on the Precision and Accuracy of Java Profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 100–113.

[5] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[6] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. 2019. Microservices monitoring with event logs and black box execution tracing. *IEEE transactions on services computing* 15, 1 (2019), 294–307.

[7] Mathieu Desnoyers and Michel R Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, Vol. 2006. Citeseer, 209–224.

[8] Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, and Daniel Seybold. 2023. Using eBPF for Database Workload Tracing: An Explorative Study. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. 311–317.

[9] Christina Eder, Stefan Winzinger, and Robin Lichtenthäler. 2023. A Comparison of Distributed Tracing Tools in Serverless Applications. In *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 98–105. https://doi.org/10.1109/SOSE58276.2023.00018

[10] Jens Ehlers, André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2011. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th ACM international conference on Autonomic computing*. 197–200.

[11] Holger Eichelberger and Klaus Schmid. 2014. Flexible resource monitoring of Java programs. *Journal of Systems and Software* 93 (2014), 163–186.

[12] Norgren Elias. 2024. Optimizing Distributed Tracing Overhead in a Cloud Environment with OpenTelemetry.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 3–18.

[14] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead. *ACM Comput. Surv.* 51, 2, Article 26 (mar 2018), 33 pages. https://doi.org/10.1145/3158644

[15] L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolek, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin, and F. Simon Panojo. 2024. Monitoring tools for DevOps and microservices: A systematic grey literature review. *Journal of Systems and Software* 208 (2024), 111906. https://doi.org/10.1016/j.jss.2023.111906

[16] Andreas Gocht-Zech, Alexander Grund, and Robert Schöne. 2021. Controlling the runtime overhead of python monitoring with selective instrumentation. In *2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, 17–25.

[17] Yasmeen Hammad, Amro Al-Said Ahmad, and Peter Andras. 2025. An empirical study on the performance overhead of code instrumentation in containerised microservices. *Journal of Systems and Software* 230 (Dec. 2025), 112573. https://doi.org/10.1016/j.jss.2025.112573

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

[19] 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.

[20] Sascha Hunold, Jordy I Ajanohoun, Ioannis Vardas, and Jesper Larsson Träff. 2022. An overhead analysis of mpi profiling and tracing tools. In *Proceedings of the 2nd Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy*. 5–13.

[21] Thomas Ilsche, Joseph Schuchart, Robert Schöne, and Daniel Hackenberg. 2015. Combining instrumentation and sampling for trace-based application performance analysis. In *Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing, October 2014, HLRS, Stuttgart, Germany*. Springer, 123–136.

[22] Andrea Janes, Xiaozhou Li, and Valentina Lenarduzzi. 2023. Open tracing tools: Overview and critical comparison. *Journal of Systems and Software* 204 (2023), 111793.

[23] Tahir Mert Karkan. 2024. Performance Overhead Of OpenTelemetry Sampling Methods In A Cloud Infrastructure. https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-225869

[24] 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.

[25] Chien-An Lai, Josh Kimball, Tao Zhu, Qingyang Wang, and Calton Pu. 2017. milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 92–102.

[26] Jan-Patrick Lehr, Alexander Hück, and Christian Bischof. 2018. PIRA: Performance instrumentation refinement automation. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*. 1–10.

[27] Joshua Levin and Theophilus A. Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 1–8. https://doi.org/10.1109/CloudNet51028.2020.9335808

[28] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empirical Software Engineering* 27 (2022), 1–28.

[29] Matthias S Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E Nagel. 2007. Developing scalable applications with Vampir, VampirServer and VampirTrace.. In *PARCO*, Vol. 15. 637–644.

[30] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2010. Evaluating the accuracy of Java profilers. *ACM Sigplan Notices* 45, 6 (2010), 187–197.

[31] 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*. 249–256.

[32] 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*.

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

[34] 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.

[35] Frode Sandberg. 2024. Evaluating OpenTelemetry's Impact on Performance in Microservice Architectures.

[36] Pritam Shah. 2018. *OpenCensus: A Stats Collection and Distributed Tracing Framework*. https://opensource.googleblog.com/2018/01/opencensus.html?utm_source=chatgpt.com Google Open Source Blog.

[37] Ben Sigelman. 2016. *Towards Turnkey Distributed Tracing*. https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736 Medium.

[38] Ben Sigelman, Bogdan Drutu, Sarah Novotny, Sergey Kanzhelev, and Yuri Shkuro. 2019. *Merging OpenTracing and OpenCensus: Goals and Non-Goals*. https://medium.com/opentracing/merging-opentracing-and-opencensus-f0fe9c7ca6f0 Medium.

[39] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[40] 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)*.

[41] André van Hoorn, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. 2009. *Continuous monitoring of software services: Design and application of the Kieker framework*. Technical Report 0921. Christian-Albrechts-Universität zu Kiel, Institut für Informatik, Kiel, Germany. https://macau.uni-kiel.de/servlets/MCRFileNodeServlet/macau_derivate_00002997/tr-0921-bericht.pdf

[42] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, 247–248. http://eprints.uni-kiel.de/14418/

[43] Simon Volpert, Sascha Winkelhofer, Jörg Domaschka, and Stefan Wesner. 2025. Towards eBPF Overhead Quantification: An Exemplary Comparison of eBPF and SystemTap. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering* (Toronto ON, Canada) *(ICPE '25)*. Association for

Computing Machinery, New York, NY, USA, 122–129. https://doi.org/10.1145/3680256.3721311

[44] 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.

[45] Jan Waller, Nils C Ehmke, and Wilhelm Hasselbring. 2015. Including performance benchmarks into continuous integration to enable DevOps. *ACM SIGSOFT Software Engineering Notes* 40, 2 (2015), 1–4.

[46] 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).

[47] 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.

[48] Alexander Wert, Henning Schulz, and Christoph Heger. 2015. AIM: Adaptable Instrumentation and Monitoring for automated software performance analysis. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. IEEE, 38–42.

[49] Karim Yaghmour and Michel R Dagenais. 2000. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*. https://www.usenix.org/conference/2000-usenix-annual-technical-conference/measuring-and-characterizing-system-behavior

[50] Shinhyung Yang, Jiun Jeong, Bernhard Scholz, and Bernd Burgstaller. 2022. Cloudprofiler: TSC-based inter-node profiling and high-throughput data ingestion for cloud streaming workloads. *arXiv preprint arXiv:2205.09325* (2022). https://doi.org/10.48550/arXiv.2205.09325

[51] Shinhyung Yang, David Georg Reichelt, and Wilhelm Hasselbring. 2024. Evaluating the Overhead of the Performance Profiler Cloudprofiler With MooBench. In *SSP 2024*. https://doi.org/10.48550/arXiv.2411.17413

[52] Shinhyung Yang, David Georg Reichelt, Reiner Jung, Marcel Hansson, and Wilhelm Hasselbring. 2025. The Kieker Observability Framework Version 2. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 11–15.

Appendix A: Flamegraphs

To illustrate the results from Section 5, we present the flame graphs of all frameworks below. These graphs reveal the heterogeneous structure of the call trees across the different frameworks:

- **Elastic APM** has a large setup during the initial `monitoredMethod` call, with less instrumentation overhead in the subsequent calls. Furthermore, the native method calls (that are time function calls) are direct children of the `monitoredMethod` call, indicating that they are woven directly into the traced code, instead of being invoked by a function of Elastic APM.
- **inspectIT** shows a relatively similar behavior as Elastic APM. In contrast to Elastic APM, we see that most of the native method calls occur deep in inspectITs own stack. This is caused by the indirection of inspectIT, that reads yaml configuration files and thereby creates the instrumentation code.
- **Kieker** shows a flamegraph with relatively high shares of time function calls. Additionally, we see that waiting for the queue to write the monitoring records consumes a large portion of the execution time.
- **Pinpoint** and **Scouter** both exhibit substantial execution times during the first `monitoredMethod` call, further underlining the heavy setup overhead identified in the regression analysis.

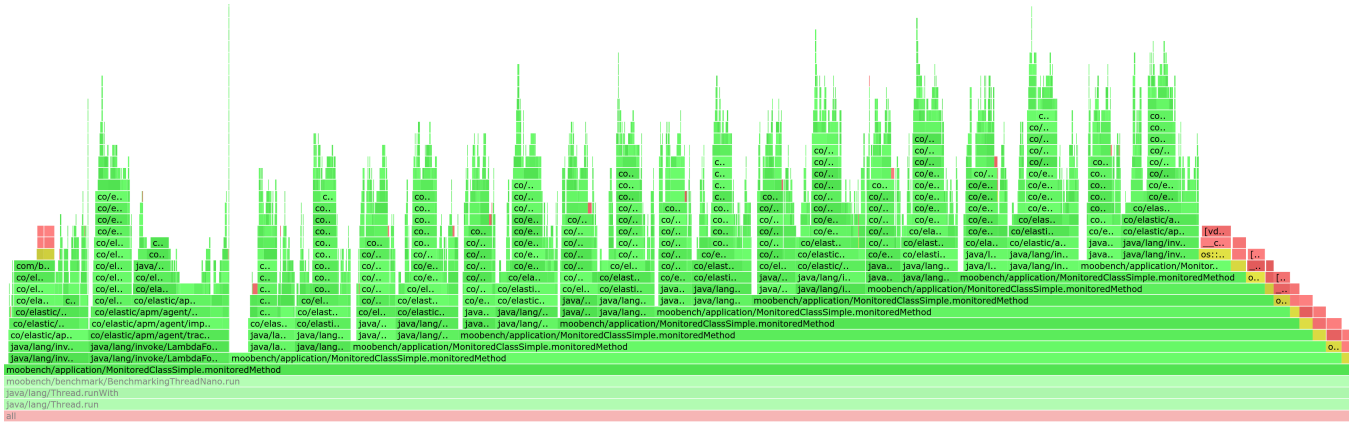


Figure 9: Elastic APM Flame Graph

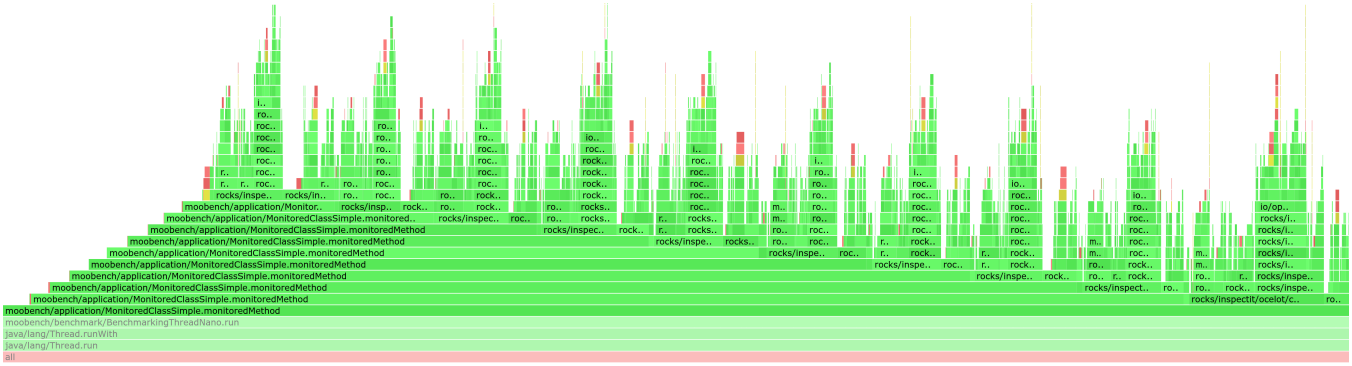


Figure 10: inspectIT Flame Graph

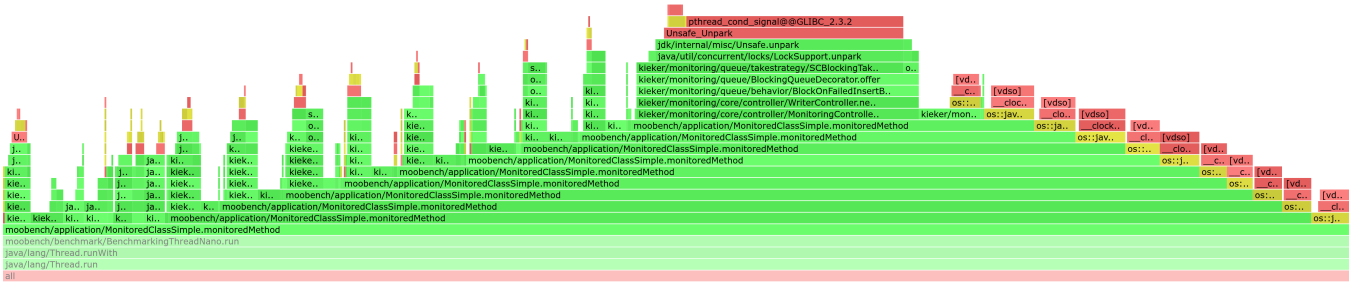


Figure 11: Kieker Flame Graph

Figure 12: Pinpoint Flame Graph

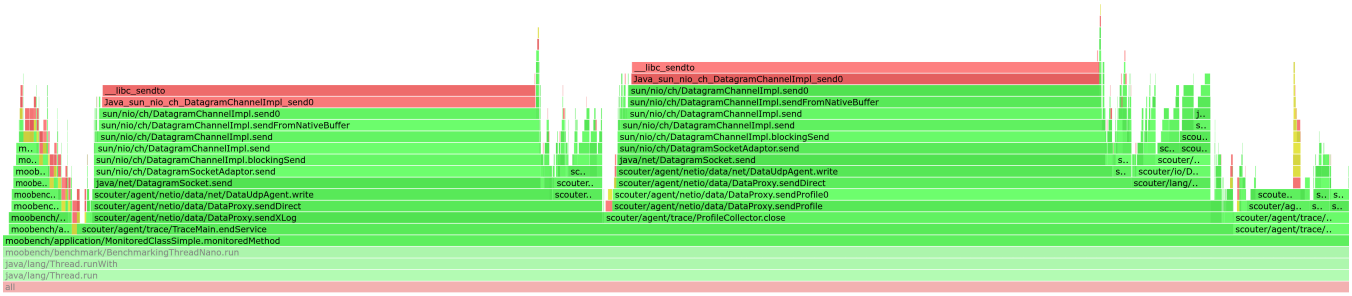


Figure 13: Scouter Flame Graph

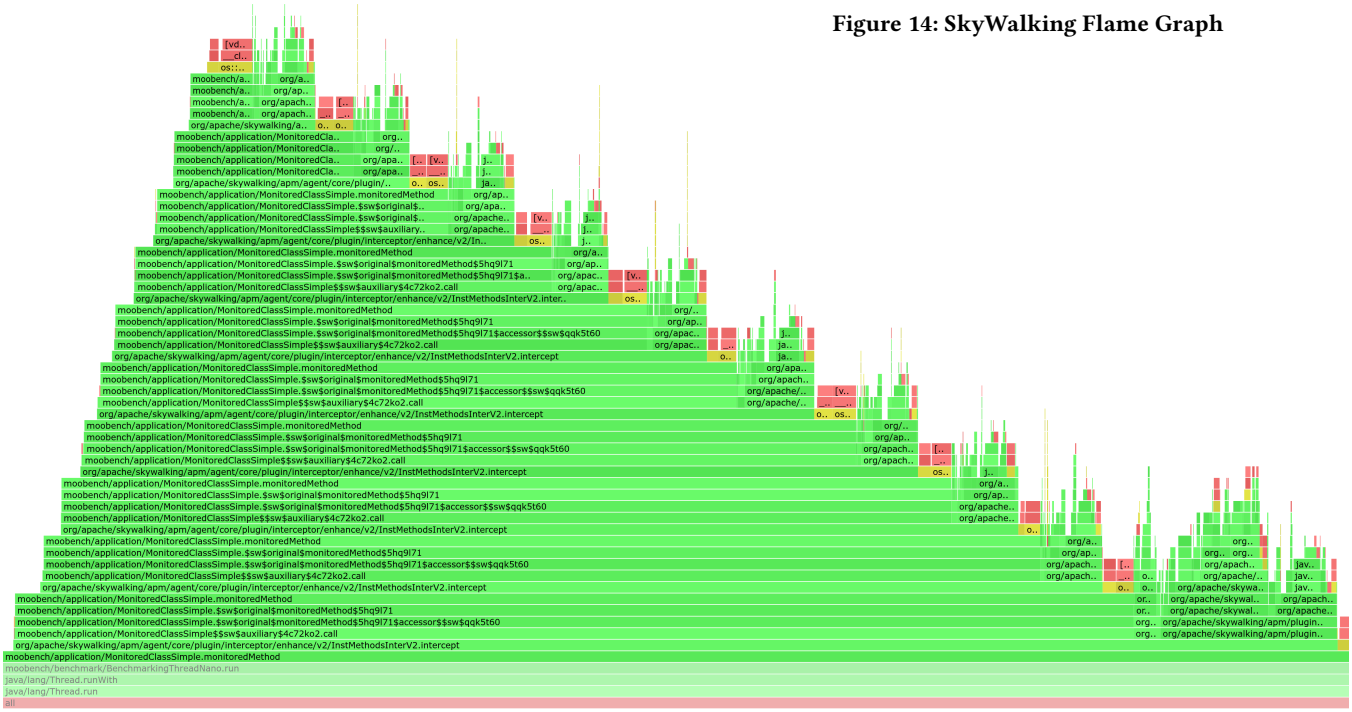


Figure 14: SkyWalking Flame Graph