Understanding and Reducing OpenTelemetry Tracing Overhead in Kotlin Multiplatform

Lorenz Bader
Institute for System Software
Johannes Kepler University Linz, Austria

Markus Weninger
markus.weninger@jku.at
Institute for System Software
Johannes Kepler University Linz, Austria

Abstract

OpenTelemetry (OTel) provides a standardized framework for generating and exporting traces, but fine-grained tracing in Kotlin Multiplatform (KMP) applications can introduce substantial and platform-dependent run-time overhead. We investigate the sources of this overhead and evaluate approaches to reduce it across JVM, JavaScript, and Native targets. Specifically, we compare alternative timestamping strategies, OTLP transport mechanisms, and span processing and batching configurations in a controlled cross-platform benchmark. Our results show that several plausible optimizations provide little or no benefit, while substantial overhead originates from SDK-internal batch management. Eliminating a quadratic-cost batch-removal in the SDK's span processor reduces run time by 29–52% depending on the target platform. These findings provide insight into the key performance factors of OTel tracing in KMP and guidance for more efficient multiplatform tracing solutions.

1 Introduction

Tracing is one of the major telemetry signals used to understand the dynamic behavior of software systems [5], with OpenTelemetry (OTel) as its de facto standard framework. However, when recording dense traces, e.g., one span per function call, the fixed costs of creating and recording spans quickly dominate the traced program's execution time. Previous work on *k-perf-otel* [10] demonstrated OTel for Kotlin Multiplatform (KMP) using a compiler plugin that automatically injects span tracing code into every function for JVM, JavaScript, and Native targets. Its evaluation, however, exposed severe scalability limitations: fine-grained tracing of about two million function calls caused numerous spans to be dropped and slowdowns from 255× (JVM) up to 19,300× (Native), i.e., 4.3–35.5 µs overhead per function call.

In this work, we revisit implementation decisions regarding span processing and compare alternative variants against *k-perf-otel*. For example, replacing OTLP/HTTP with gRPC/Protobuf reduces run time by about 45% on the JVM and 17% on Native, but provides almost no improvement on JavaScript. Alternative timestamping approaches yield no consistent advantage, while the gRPC + FastBatch variant reduces k-perf-otel's run time by about 61% on the JVM, 34% on JavaScript, and 60% on Native. These results indicate that the dominant costs of OTel traces are not necessarily where initially expected and differ considerably between KMP targets.

2 Background and Related Work

OTel uses *spans* to record timed units of work, for example method calls. Spans are hierarchically related (callee to caller) and form a trace [9]. After construction, a span passes through a span processor, which collects spans in a queue and exports them in batches through an exporter via the OTLP protocol (JSON over HTTP or Protobuf over gRPC) to backends such as Jaeger [13]. Each span therefore pays fixed costs for creation, context handling, timestamping, batching, serialization, and export. Our study targets a community KMP port of the OTel API and SDK [12], which lets us instrument Kotlin code *once* and compile it (including telemetry collection) to JVM, JS, and Native at the same time. Our instrumentation happens with the help of the Kotlin compiler plugin *k-perf-otel* [10]. Every instrumented function starts a span on entry and ends it on exit, resulting in complete function-level traces.

Existing overhead studies of OTel focus on single platforms such as the JVM [3, 4] or cloud environments [8]. Recent work on Kotlin Multiplatform has explored compile-time instrumentation using Kotlin compiler plugins and Intermediate Representation (IR), including *k-perf* [6] and *k-perf-otel* [10]. This paper focuses on cross-platform overhead reduction of OTel instrumentation in KMP.

3 Improvements

We implemented each potential improvement as a separate compiler-plugin variant. The `gRPC` variant replaces the transport from OTLP/HTTP with gRPC and serves as the common base for all further variants, each of which changes exactly one additional aspect.

Overall, we have investigated the following variants, which we discuss in more detail throughout this

k-perf is open-source: `https://github.com/NeonMika/k-perf/`.
The source code, the benchmarking pipeline, as well as the experiment results are archived at `https://doi.org/10.5281/zenodo.22087594`..

section: (1) an *Uninstrumented* version (i.e., no compiler plugin applied) as a first baseline, (2) the original *k-perf-otel* as a second baseline, (3) *gRPC*, the transport switch described above, (4) *gRPC + Anchored* and (5) *gRPC + Monotonic*, two alternative timestamping strategies, (6) *gRPC + Sampler*, which uses a simplified sampling configuration, (7) *gRPC + FastBatch*, which fixes a performance bug in the SDK's batch processing, and (8) *Combined*, which combines the changes of (3), (6), and (7).

3.1 Transport: OTLP/HTTP to gRPC with Protobuf

Variant *gRPC* This variant replaces the original JSON-over-HTTP exporter with Protobuf over gRPC [14]. We also fixed a bug in the original work's `BatchSpanProcessor` configuration (now queue size 2048, batch size 512), but applied this fix to the *k-perf-otel* baseline as well, so the *gRPC* results reflect the transport switch alone.

3.2 Timestamping Strategies

Variants *gRPC + Anchored* and *gRPC + Monotonic* The original OTel plugin reads the system clock twice per function call to record a span's start and end time. With about 44k clock reads per benchmark step, we expected these accesses to be a notable cost. To improve this, *gRPC + Anchored* uses the SDK internal `AnchoredClock`, which reads a wall clock once per trace (not per span) and uses this as an offset to subsequent monotonic clock measurements. *gRPC + Monotonic* avoids the wall clock entirely. This variant takes a monotonic time mark once at startup and subsequently determines each span's timing relative to that mark. Although this leaves us with only relative timing information (i.e., how long a span took, but not when it happened), we still evaluated whether it could further reduce the overhead.

3.3 Sampler Configuration

Variant *gRPC + Sampler* A *sampler* decides for every new span whether it is recorded or not. For example, one could implement a sampler that only records every second span. When creating a new span, the default sampler (which was also used in the original *k-perf-otel*) inspects the new span's parent: if the sampler decides to not record the parent span, the new span will also not be recorded. Since we want to record every span, we don't need this check and instead use the `AlwaysOn` sampler in this variant, which entirely skips the parent check.

3.4 Bug Fix in SDK: Batch Removal

Variant *gRPC + FastBatch* The *gRPC + FastBatch* variant fixes a performance bug that we found in the KMP port's `BatchSpanProcessor` (BSP). The BSP buffers pending spans in a queue and collects spans in a `PersistentList` for batch export (see Listing 1). After each export, the original implementation removed the exported spans from this list using `removeAll`, which repeatedly searches the list once per removed span, resulting in quadratic cost proportional to the batch size. Since the entire current batch is exported at once anyway, our fix atomically takes and clears the batch list.

```
1 val batch = batchInternal.value
2 export(batch)
3 batchInternal.update{it.removeAll(batch)}
4 // fix: take and clear the batch in O(1)
5 // val batch =
      batchInternal.getAndSet(persistentListOf())
6 export(batch)
```

Listing 1: Batch removal fix in the KMP OTel SDK.

3.5 Combining the Most Effective Improvements

Variant *Combined* Our final variant combines the gRPC, the `AlwaysOn` sampler, and the FastBatch fix in a single plugin. We do not apply one of our tested timestamping strategies, as they did not provide consistent major improvements across all targets. Since the sampler change and the batch fix touch different code paths, we expect their effects to add up.

4 Overhead Analysis and Discussion

We benchmarked each variant separately and compared it to the original *k-perf-otel* and an uninstrumented baseline. The following section presents our benchmark procedure and discusses the results.

Benchmark Procedure As the workload for our benchmark we chose `fibonacci(20)`, since it produces lots of very short function calls and is therefore similar to other side-effect free, recursion-based benchmarks such as `MooBench` [1, 2, 11]. We ran each variant 10 times per platform (each time in a fresh process, e.g., a new JVM instance) with 100 steps per run, discarded the first 20 steps of every run as warm-up based on observed warm-up behavior, and discarded the fastest and slowest 1% of the remaining pooled measurements to avoid extreme outliers. One benchmark step, i.e., one `fibonacci(20)` call, yields 21,892 instrumented function calls.

Discussion Table 1 summarizes the per-call run time of all variants. The run time is obtained by dividing reported mean step times by the 21,892 function calls per step. Delta values denote the difference to the original *k-perf-otel* run time on the same platform.

The *gRPC* switch reduces per-call run time by about 44.9% on the JVM and 17.4% on Native, but only 1.0% on JavaScript. We hypothesize that this is caused by JavaScript's lack of native 64-bit integers, making timestamping and span-ID arithmetic sufficiently expensive that the transport optimization has comparatively little effect. Across the three targets, the two timestamping variants change the run time

	JVM		JS		Native (Linux)	
	Run time	**Delta**	**Run time**	**Delta**	**Run time**	**Delta**
Uninstrumented	0.59 ± 0.10		4.83 ± 0.60		0.49 ± 0.03	
k-perf-otel	$1{,}182.1 \pm 591.7$		$5{,}959.3 \pm 390.6$		$4{,}722.6 \pm 489.4$	
gRPC	651.2 ± 114.7	-530.9 ($\times 0.55$)	$5{,}899.7 \pm 180.9$	-59.6 ($\times 0.99$)	$3{,}900.3 \pm 263.2$	-822.3 ($\times 0.83$)
gRPC + Anchored	655.8 ± 117.0	-526.3 ($\times 0.55$)	$5{,}804.8 \pm 193.1$	-154.5 ($\times 0.97$)	$3{,}891.8 \pm 277.3$	-830.9 ($\times 0.82$)
gRPC + Monotonic	666.7 ± 121.9	-515.4 ($\times 0.56$)	$6{,}638.2 \pm 394.7$	$+678.9$ ($\times 1.11$)	$3{,}914.4 \pm 274.6$	-808.3 ($\times 0.83$)
gRPC + Sampler	632.0 ± 113.3	-550.1 ($\times 0.53$)	$5{,}794.9 \pm 123.3$	-164.4 ($\times 0.97$)	$3{,}848.8 \pm 248.3$	-873.8 ($\times 0.81$)
gRPC + FastBatch	465.5 ± 215.0	-716.6 ($\times 0.39$)	$3{,}928.3 \pm 136.8$	$-2{,}031.0$ ($\times 0.66$)	$1{,}879.8 \pm 267.3$	$-2{,}842.8$ ($\times 0.40$)
Combined	437.8 ± 272.1	-744.3 ($\times 0.37$)	$3{,}853.0 \pm 139.1$	$-2{,}106.3$ ($\times 0.65$)	$1{,}783.5 \pm 295.9$	$-2{,}939.1$ ($\times 0.38$)

Table 1: Average function execution time in *ns*, reported as run time ± SD per variant and target platform. We first obtain 800 post-warm-up measurements (10 runs x 80 warmed-up measurement steps) and then discard the fastest and slowest 1% of the pooled measurements to avoid extreme outliers. Delta values and multiplicative factors are relative to the original *k-perf-otel*.
Benchmark machine: Debian GNU/Linux 13 (Linux kernel 6.12.86+deb13-amd64), AMD Ryzen 9 9950X, 64 GB DDR5-6000, 1 TB SSD. Software versions were fixed to Kotlin 2.3.0, Java 21.0.5, and JavaScript Node.js 24.18.0.

relative to gRPC by -1.6% to +12.5%. This falls short of our expectations, as we assumed `System.Clock` queries to be much more expensive. *gRPC + Sampler* reduces the run time relative to *gRPC* by 2.9% on the JVM, 1.8% on JavaScript, and 1.3% on Native, indicating that the skipped parent checks are not a dominant cost on any target, but still incur overhead. *gRPC + FastBatch* has by far the largest effect, reducing the run time relative to the *gRPC* base by 28.5% (JVM), 33.4% (JS), and 51.8% (Native). The *Combined* variant showed the most improvement on all three platforms. It is the fastest configuration on all three KMP target platforms and reduces the run time by 63.0% (JVM), 35.3% (JS), and 62.2% (Native) compared to the original *k-perf-otel*.

Three observations stand out: (1) The costs are strongly platform-dependent. The original *k-perf-otel* instrumentation requires about 1.18 µs per call on the JVM, 5.96 µs on JavaScript, and 4.72 µs on Native Linux, and each optimization affects the platforms differently. Thus, making decisions based on, for example, JVM measurements alone would be misleading. (2) The largest gains were achieved by changing the SDK, rather than our own code. Timestamping and sampler configuration make comparatively little difference, while the *FastBatch* fix accounts for by far the largest improvement. This suggests that further optimization requires a more detailed analysis of the SDK's internal processing path. (3) Room for improvement remains. Even *Combined* requires about 0.438 µs per traced call on the JVM, approximately 746 times the 0.59 ns run time of an uninstrumented call in our workload. Closing this gap requires an even more in-depth analysis.

5 Summary and Outlook

This paper analyzed the run-time overhead of OpenTelemetry tracing in KMP across JVM, JavaScript, and Native. Our evaluation showed strongly platform-dependent costs and identified a quadratic batch-removal step in the SDK's span processor as a major source of overhead. Overall, our *Combined* variant reduces the tracing overhead of the original *k-perf-otel* by 35–63%, depending on the platform.

Nevertheless, our results stem from a single benchmark machine, a call-dense workload, and a community SDK port. Therefore, future work will focus on repeating our measurements using the recently introduced official OTel Kotlin SDK, broadening the evaluated hardware and workloads, and porting the variants' hand-written IR helper code to our KIRHelperKit library [7], which simplifies the development of such compiler plugins.

References

[1] J. Waller. "Performance Benchmarking of Application Monitoring Frameworks". In: *Softwaretechnik-Trends* 35.1 (2015).

[2] J. Waller, N. C. Ehmke, and W. Hasselbring. "Including performance benchmarks into continuous integration to enable DevOps". In: *ACM SIGSOFT SEN* 40.2 (2015).

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

[4] D. G. Reichelt, S. Kühne, and W. Hasselbring. "Towards Solving the Challenge of Minimal Overhead Monitoring". In: *ICPE*. 2023.

[5] X. Wang and C. Tseng. "Building an Observable System Based on OpenTelemetry". In: *ICS*. 2024.

[6] M. Weninger. "Tracing Performance Metrics in Kotlin Multiplatform Projects via Compile-Time Code Instrumentation". In: *SSP*. 2024.

[7] L. Bader and M. Weninger. "Simplifying Kotlin Compile-Time Code Instrumentation". In: *SSP*. 2025.

[8] R. Pröll and M. Weninger. "SaaS Observability on the Microsoft Power Platform and its Performance Impacts". In: *IC2E*. 2025.

[9] D. G. Reichelt et al. "Interoperability From Kieker to OpenTelemetry: Demonstrated as Export to ExplorViz". In: *SSP*. 2025.

[10] F. Schoenberger and M. Weninger. "OpenTelemetry Instrumentation using Kotlin Multiplatform Compiler Plugins". In: *SSP*. 2025.

[11] D. G. Reichelt et al. "MooBench: A micro-benchmark for performance overhead measurement of observability tools". In: *JOSS* 11.123 (2026).

[12] dcxp. *opentelemetry-kotlin: a Kotlin Multiplatform port of the OpenTelemetry API and SDK*. URL: `https://github.com/dcxp/opentelemetry-kotlin`.

[13] Jaeger Authors. *Jaeger: open source, distributed tracing platform*. URL: `https://www.jaegertracing.io/`.

[14] T. Ortel. *gRPC Kotlin Multiplatform*. URL: `https://github.com/TimOrtel/GRPC-Kotlin-Multiplatform`.