OpenTelemetry Instrumentation using Kotlin Multiplatform Compiler Plugins

Fabian Schoenberger
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

Achieving consistent performance tracing for Kotlin Multiplatform (KMP) applications is challenging, as conventional instrumentation techniques often rely on platform-specific instrumentation agents that are not universally applicable across Kotlin's compilation target platforms: the JVM, JavaScript, and Native targets. This paper introduces a unified, compile-time alternative: a Kotlin compiler plugin that automatically injects OpenTelemetry-compliant tracing *at compile time*. Our approach guarantees consistent instrumentation that produces traces compatible with existing observability tools across all Kotlin target platforms from a single compilation pass.

1 Introduction

Tracing method execution times is a critical practice in software development. Manual instrumentation often involves a considerable amount of boilerplate code and/or non-standardized formats, while automated approaches (such as load-time instrumentation) are typically limited to a single platform (e.g., using Java agents on the JVM). This reliance on platform-specific tooling is fundamentally limiting for monitoring applications written in languages such as Kotlin, which compile to various target platforms.

To address these challenges, Weninger [12] developed k-perf, a compiler plugin to instrument Kotlin code at compile time. However, this plugin has a key limitation: it uses a proprietary tracing format, making it unable to integrate with standardized monitoring tools. This paper builds upon this foundation by extending the plugin to leverage the open-source observability standard, OpenTelemetry. The goal is to generate OpenTelemetry-compliant traces for Kotlin Multiplatform (KMP) applications, supporting targets including JVM, JavaScript, and Native. This work enables seamless integration with established observability tools such as Jaeger or Zipkin.

2 Background

This section gives an overview of Kotlin's compilation process, compiler plugins, and OpenTelemetry.

Kotlin Compiler and Plugins Kotlin's K2 compiler processes source code in multiple stages, as shown in Figure 1. The source is first reduced to an Abstract Syntax Tree (AST), which is then transformed to Frontend Intermediate Representation (FIR) primarily used for code analysis. FIR is transformed to Backend Intermediate Representation (IR) for code optimization and finally translated to the respective target (e.g., JVM, JavaScript, Native) [3]. K2 supports user-provided extensions, i.e., compiler plugins, for advanced code analyses (FIR) as well as code for instrumentation (IR) at compile-time.

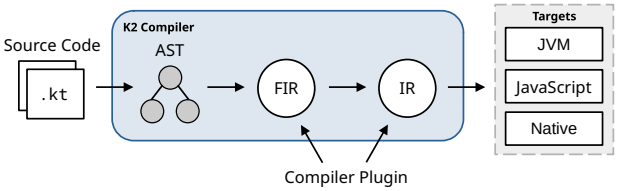

Figure 1: Overview of Kotlin's compilation process.

IR plugins register an `IrGenerationExtension` instance with the compiler, which provides the logic for transforming the code. Transforming IR is achieved by extending and traversing the IR tree, whereas declarations (e.g., functions) can be modified using the visitor pattern. This way, any changes to the code are performed before translation to the specific targets.

OpenTelemetry OpenTelemetry (OTel) is a widely used open-source observability standard that enables tracing of applications. A trace represents the path taken through an application and consists of spans, where each span represent some time-measured unit of work [10] (e.g., a database query or a function call). These traces are then exported using the OpenTelemetry Protocol (OTLP), which ensures compatibility with a wide variety of monitoring tools. Typically, these traces are sent to a central OpenTelemetry Collector, which can forward them to different backends. This approach decouples the application from the monitoring backend, preventing vendor lock-in and allowing monitoring tools to be interchanged through simple configuration changes.

Our Kotlin Multiplatform OpenTelemetry compiler plugin can be found at
`https://github.com/FabianSchoenberger/otel-plugin`

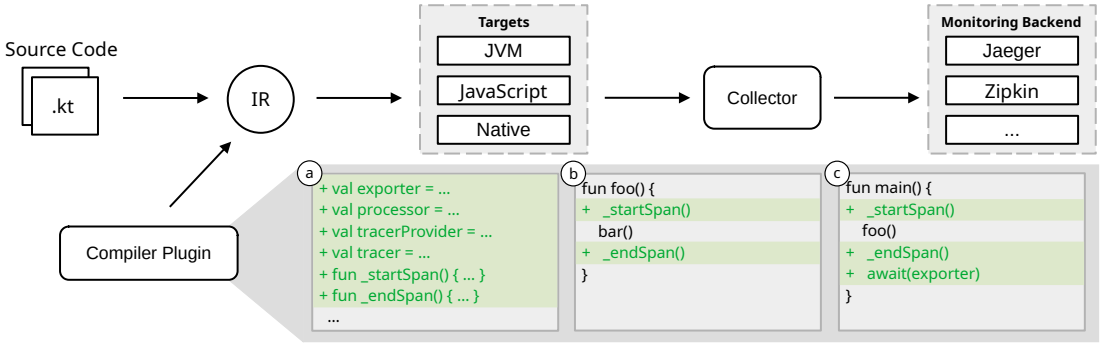

Figure 2: Overview of our compiler plugin's functionality.

3 Compiler Plugin

Our plugin records every function call as an OpenTelemetry span by automatically instrumenting source code at compile-time. The plugin is also responsible for exporting these spans to an OTLP endpoint. Most fundamentally, the plugin may only add code that can be translated to all Kotlin targets, i.e., our compiler plugin may only introduce dependencies on libraries that are developed for KMP themselves. Thus, we rely on a Kotlin Multiplatform port[1] of the Java OpenTelemetry Tracing API/SDK.

Figure 2 illustrates the plugin's overall workflow and its three key modifications to the IR: (a) the required OpenTelemetry components and helper functions are created, (b) every function is instrumented to open and close a span, and (c) the main function additionally waits for all exports to finish before program termination. An example function instrumentation is shown in Listing 1 (the actual instrumentation happens on IR level, not source code level). To maintain the function call hierarchy within the trace, the `Context` is retrieved and used to define the new span's parent-child relationship. The original function body is wrapped in a `try/finally` block, which guarantees that the span is closed correctly even if the function terminates with an exception.

```
// before instrumentation
fun add(a: Int, b: Int) = a + b
// after instrumentation
fun add(a: Int, b: Int): Int {
  val context = Context.current()
  val span = _startSpan("add(Int, Int)", context)
  try { return a + b }
  finally { _endSpan(span, context) }
}
```

Listing 1: An example of an instrumented function.

The modified IR is then translated into the desired target-specific code. Our plugin currently supports the JVM, JavaScript, and Native targets.

At run time, generated spans are buffered by a `BatchSpanProcessor` and sent in batches by a custom OTLP exporter, an optimization that avoids the high network overhead of sending each span individually. This exporter[2] had to be implemented for this project, as no official KMP-compatible OTLP/HTTP exporter was available at the time of implementation. It sends the spans asynchronously to a configured Collector endpoint according to the OTLP specification using the KMP HTTP client library Ktor. Finally, the Collector processes and forwards all received spans to one or more monitoring backends (e.g., Jaeger, Zipkin) based on its configuration. These backends then store and visualize the generated traces.

[1] https://github.com/dcxp/opentelemetry-kotlin
[2] https://github.com/FabianSchoenberger/otlp-exporter

4 Evaluation and Discussion

To verify our plugin's output and measure its performance impact, we conducted a similar benchmark as used for k-perf [12]: tracing *every* method call of a Game of Life implementation. We wanted to investigate whether this is also possible with our OpenTelemetry plugin and how much overhead this introduces. The benchmark runs for 500 steps, resulting in approximately two million function calls (illustrated in Figure 3). For each target platform, the application was executed 100 times to increase statistical significance. During the benchmark, all generated traces were sent to a locally running OpenTelemetry Collector, forwarding them to a Jaeger backend.

Figure 4 shows an example trace with five simulated steps visualized in Jaeger, where the hierarchy of the spans mirrors the application's call graph.

Performance Implications The benchmark reveals an enormous performance overhead, with run-

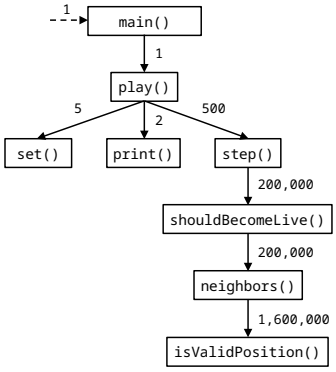

Figure 3: Call graph of Game of Life with 500 steps.

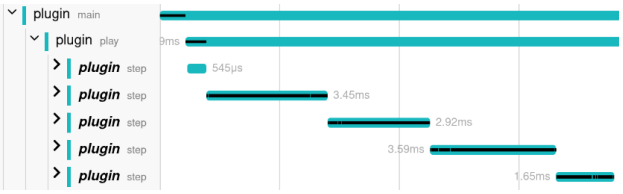

Figure 4: Example of a Game of Life trace in Jaeger.

time slowdowns ranging from 255x on the JVM to 19,300x on Native (Linux) targets. This corresponds to an overhead per method call of 4.3 µs (JVM) to 35.5 µs (Native), indicating room for improvement in future work [9]. Another observation is that the high volume of spans overwhelms the queues of the Collector and the backend, leading to significant data loss. These findings demonstrate that, opposed to a custom trace format [12], recording *every* function call using OpenTelemetry is fundamentally impractical.

One way to circumvent this limitation is using a more performant intermediate trace format and storage that allows for translation to spans [10] during periods of low activity. A more sophisticated solution would be to implement adaptive sampling [7], i.e., selectively and dynamically enabling and disabling span recording for individual frequently-executed functions. This could significantly reduce the volume of spans while still recording deep call stacks regularly for more detailed analyses.

Asynchronous Operations One current key limitation is the handling of asynchronous operations. The plugin relies on a singleton object to track the active span's context. This approach breaks when instrumenting multi-threaded programs, including `suspend` functions (built around Kotlin coroutines). Coroutines can resume on different threads, which can lead to a broken parent-child relationship in the span hierarchy.

The solution to this limitation is to modify our approach of context propagation. Instead of a singleton object, we could additionally instrument each function's parameters to include the parent span's context. This approach, while more complex, would prevent any race conditions from occurring.

5 Related Work

Besides OpenTelemetry, the de-facto observability standard, numerous other tracing tools exist. Janes et al. [6] analyzed 30 different tools, highlighting their features and trade-offs. This includes, for example, Kieker [4, 14], an observability framework often used for research on tracing overhead [5, 9, 11, 13].

Regarding instrumentation strategy, this work's *capture every method call*-approach contrasts with several alternatives. For example, Pfeffer and Weninger [8] perform annotation-based modifications in Kotlin, i.e., instrumenting only functions explicitly marked by a developer. In their work, they present various ways on how to instrument Kotlin programs. While they clearly position compiler plugins as the most flexible (but also most complex approach), non-multiplatform Kotlin programs purely targeted at the JVM can be instrumented at load-time similarly to Java programs using Java agents and bytecode modification libraries such as Javassist [1, 2].

6 Summary and Outlook

This paper demonstrated the feasibility of using a compiler plugin to automatically generate OpenTelemetry-compliant traces for Kotlin Multiplatform applications. While functionally successful, our proof-of-concept revealed that an exhaustive, per-function tracing strategy introduces a significant performance overhead. Future work must therefore prioritize two key areas: (1) implementing performance optimizations, such as adaptive sampling to reduce the volume of generated spans, and (2) adding support for multi-threading and asynchronous operations by correctly propagating span context.

References

[1] S. Chiba. "Load-Time Structural Reflection in Java". In: *ECOOP*. 2000.

[2] S. Chiba and M. Nishizawa. "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators". In: *GPCE*. 2003.

[3] J. Stanier and D. Watson. "Intermediate representations in imperative compilers: A survey". In: *ACM Comput. Surv.* 45.3 (2013).

[4] W. Hasselbring and A. van Hoorn. "Kieker: A Monitoring Framework for Software Engineering Research". In: *Softw. Impacts* 5 (2020).

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

[6] A. Janes, X. Li, and V. Lenarduzzi. "Open Tracing Tools: Overview and Critical Comparison". In: *J. Syst. Softw.* 204 (2023).

[7] J. Mertz and I. Nunes. "Software Runtime Monitoring with Adaptive Sampling Rate to Collect Representative Samples of Execution Traces". In: *J. Syst. Softw.* 202 (2023).

[8] D. Pfeffer and M. Weninger. "On the Applicability of Annotation-Based Source Code Modification in Kotlin". In: *MPLR*. 2023.

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

[10] D. G. Reichelt et al. "Interoperability From Kieker to OpenTelemetry: Demonstrated as Export to ExplorViz". In: *SSP* (2024).

[11] D. G. Reichelt et al. "Overhead Comparison of Instrumentation Frameworks". In: *ICPE*. 2024.

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

[13] S. Yang, D. G. Reichelt, and W. Hasselbring. "Evaluating the Overhead of the Performance Profiler Cloudprofiler With MooBench". In: *SSP* (2024).

[14] S. Yang et al. "The Kieker Observability Framework Version 2". In: *ICPE*. 2025.