Tracing Performance Metrics in Kotlin Multiplatform Projects via Compile-Time Code Instrumentation

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

Abstract

Performance analysis requires to collect metrics such as method execution times. For JVM bytecode, this is typically achieved through *load-time code instrumentation* using *agents*. Kotlin, however, presents unique challenges as it can be compiled to various targets beyond the JVM, including WASM and native code. These diverse targets render traditional agent-based instrumentation unsuitable, requiring a more versatile technique to cover all target platforms conveniently.

In this paper, we leverage Kotlin's flexible compiler architecture and present a work-in-progress Kotlin compiler plugin that employs *compile-time instrumentation*. By adding *target-agnostic* performance metric tracing code during compilation instead of target-specific code at load time, we address a critical need in the growing field of Kotlin Multiplatform projects.

1 Introduction

Application performance analysis requires continuous collection of monitoring data to identify issues or to extract performance models [9]. Various approaches exist for this collection [8]: (1) Modify the *execution environment* (e.g., a custom Java VM) to access internal information, (2) register callbacks on the execution environment to *sample performance events*, or (3) *instrument* the application by adding code, either before compilation (source code), at compile time (IR code, i.e., the compiler's intermediate representation), or at run time (bytecode / machine code). Each approach has its advantages and disadvantages [5], for example how detailed the collected data is or how much overhead is introduced.

In this work, we focus on a problem that arises when using a language that can be compiled to different *target platforms*. Kotlin, for example, can be compiled to JVM bytecode, JavaScript, WebAssembly (WASM) and to machine code. In a Kotlin Multiplatform (KMP) project, code is compiled for multiple targets at once, e.g., an Android application (JVM bytecode) and an iOS application (native). If we now want to collect performance metrics in both targets, it is not possible to use a single modified *execution environment*, nor do we have a unified way to *sample* from the execution environment – we are left with instrumentation. Since there is no load-time instrumentation technique that can instrument JVM bytecode *and* machine code (i.e., inconveniently needing two different target-specific agents/weavers), modifying the source code before compilation or add additional code at compile time remain viable options.

In this work, we explore the latter by leveraging Kotlin's extendable compiler mechanism that provides *custom compiler plugins* [11]. Our contributions are (1) a presentation of Kotlin's compiler and its compiler plugin system in Section 2, (2) a prototype that adds performance metric tracing code at compile-time in Section 3, and (3) a brief discussion of the plugin's memory and performance impact in Section 4.

2 The Kotlin Compiler and its Plugins

This section gives an overview of Kotlin's compilation process and discusses how to implement plugins for it.

2.1 Compilation Process

As shown in Figure 1, Kotlin's new K2 compiler regularly transforms Kotlin source code into FIR (frontend IR), which is then translated to backend IR, based on which the final target-specific code is generated.

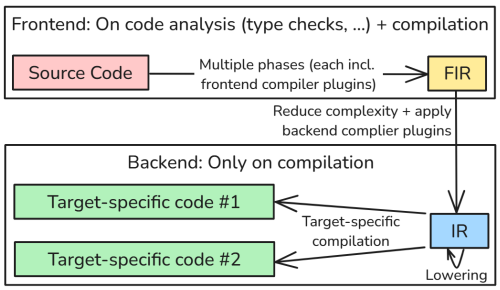

Figure 1: Simplified Kotlin K2 compilation process.

The **frontend** consists of multiple phases and builds FIR primarily for static code analysis and control flow analysis [11]. FIR is not only generated during compilation, but also for analyses by the IDE.

The **backend** translates the FIR into backend IR. This is used for lowering, i.e., for converting high-level semantic constructs into lower-level representations. For example, during lowering, Kotlin's foreach-loops are converted to either `Iterator`-based or counter-based loops depending on the loop variable type.

The **target backends** (JVM, native, ...) then use lowered IR to generate optimized target-specific code.

2.2 Plugins

Kotlin compiler plugins, in essence, perform additional operations, in the frontend and/or the backend.

Frontend For each frontend phase, the Kotlin compiler provides an interface that plugin developers can implement. In the overridden methods, additional operations specific to the respective phase can be performed. The big upside of frontend plugins is that changes performed in them are considered during code analysis and reflected in the IDE [11]. Unfortunately, currently *[...] the only way to communicate between user code and plugins is to mark something in code with some annotation [...]*[1]. Freely modifying unannotated code (for example, adding code to collect performance metrics) is only possible in backend plugins. Another downside is the extreme lack of official documentation for frontend compiler plugin development.

Backend Backend plugins operate on the IR and can fully modify it. By implementing the `IrGenerationExtension` interface developer get access to all Kotlin files currently being compiled. The most straight-forward way to perform changes on the code elements being compiled is to use the visitor system built into the IR [12]. To get started with backend plugin development, one of the best resources we found is the comprehensive tutorial by Norman [14].

Restrictions and Considerations While compiler plugins are powerful tools that facilitate the generation of platform-agnostic code, they do not come without certain restrictions or considerations. They are *restricted to Kotlin code* being compiled, e.g., they cannot modify external libraries. Their additional operations during compilation can lead to *longer compile times*. Further, the instrumented code *increases the size of the compiled output* and by modifying a system we also change its behavior (*Observer Effect* [4]), thus we should keep code changes minimal. Finally, the divergence between the original source and the instrumented code might *complicate debugging*.

3 Instrumentation Prototype

The primary goal of our Kotlin backend compiler plugin is to collect the execution time of every method call [6] using code instrumentation at compile time.

For example, our plugin modifies code such as Listing 1 to become code such as Listing 2 by adding additional IR nodes during compilation that encompass: (1) two logging methods `_enter` and `_leave` (lines 1 and 2), (2) calls to `_enter` at method starts (line 5), and (3) `try` blocks wrapping all method bodies, calling `_leave` in a `finally` block (line 8).

[1] https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md

```
1 fun sayHello(pre: String = "Hello",
2              name: String = "You") = "$pre, $name!"
```

Listing 1: Original `sayHello` method.

```
1 fun _enter(methodId: Int) : TimeMark { ... }
2 fun _leave(methodId: Int, start : TimeMark) { ... }
3 fun sayHello(pre: String = "Hello",
4              name: String = "You"): String {
5   val startTime = _enter(2)
6   try {
7     return "$pre, $name!" // original code
8   } finally { _leave(2, startTime) }
9 }
```

Listing 2: Instrumented `sayHello` method.

`_enter` and `_leave` currently write logs to a file. `_enter` logs that a given method has been entered and returns a `TimeMark`, *a time point that [...] allows querying for the duration of time elapsed from that point.*[2] `_leave` logs the passed time since the call to `_enter`, building traces as shown in Listing 3. In the future, sophisticated tracing or monitoring frameworks such as OpenTelemetry [9] can be implemented.

```
1 >;1       // main entered
2 >;2       // sayHello entered
3 <;230     // sayHello left after 0.23ms
4 >;2       // sayHello entered
5 <;250     // sayHello left after 0.25ms
6 <;7500    // main left after 7.5ms
```

Listing 3: Trace format annotated with comments.

Multiplatform Considerations We must generate IR code whose classes and functions are available on all target platforms. For example, our logging methods must not use `java.io.File` since it is only available on the JVM platform. Instead, we use `kotlinx.io`[3] for multiplatform file I/O. Similarly, we do not use `System.currentTimeMillis` (only available on JVM) but `kotlin.time.TimeMark` (implemented on all platforms) for time measurement.

Performance Considerations We currently do not apply explicit optimizations (batching, etc.) when writing to files. Nevertheless, methods are assigned a `methodId` to reduce log size. To translate the IDs back to method names, our plugin also generates a *symbols* file. For example, for Listing 3, the symbol file contains { `"1": "main", "2": "sayHello"`}.

Trace Processing To demonstrate our plugin's applicability, we generate call graphs from recorded traces: nodes represent methods, edges represent method calls. Nodes contain the method's name and the time spent in it. Edges are labeled with the number of calls to the pointed method. We use color to encode a method's contribution to the overall run time. Figure 2 shows the call graph for a Kotlin Multiplatform CLI *Game of Life* with a "red path" from `main` at the top to `nLiveNeighbors` three call levels deep.

[2] https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.time/-time-mark/

[3] https://github.com/Kotlin/kotlinx-io

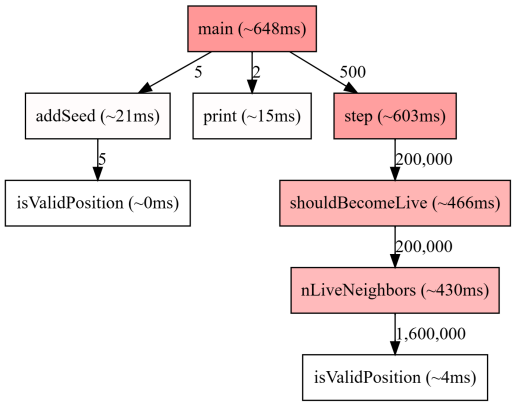

Figure 2: Call graph of 500 steps in *Game of Life*.

Availability The code of our plugin, the call graph generator, as well as the Kotlin Multiplatform *Game of Life* are open source at `https://github.com/NeonMika/k-perf` and archived at `https://doi.org/10.5281/zenodo.13958416`.

4 Discussion

In this section, we briefly explore the effects on build size and execution time of performance metric tracing using our compile-time instrumentation in Kotlin.

KMP Game of Life	**JVM**	**JS**	**native (.exe)**
Build Size w/o plugin	1753.2 kB	106.9 kB	741.9 kB
Build Size w/ plugin	1912.5 kB	167.9 kB	851.5 kB
Run Time w/o plugin	83.0 - 85.3 ms	127.5 - 130.8 ms	242.9 - 248.0 ms
Run Time w/ plugin	411.2 - 429.8 ms	9364.8 - 9771.0 ms	5322.4 - 5497.0 ms

Table 1: Build sizes (incl. Kotlin runtime + dependencies) and run times (95% conf. int. of 100 runs).

Table 1 shows how our plugin, applied to *Game of Life*, influences the resulting build artifact sizes (added instrumented code + newly added dependency on `kotlinx.io`). It also shows the execution times when running the game for 500 steps[4]. Turning on tracing (i.e., tracking and logging the execution time of *every* method call) introduces overhead ranging from 5x (JVM) to 74x (JS). The reasons for this discrepancy (e.g., less efficient compilation or less optimized `kotlinx.io` or `TimeMark` implementations) are research in progress.

5 Related Work

Our previous work on how to perform annotation-based code modification in Kotlin [12] motivated this work on compile-time IR instrumentation to collect performance metrics. While IR instrumentation, specifically in Kotlin, has not yet been extensively studied, load-time bytecode manipulation (with tools such as AspectJ [2] or Javassist [1, 3]) is a well-known technique that can guide future work. Further, performance improvements to our plugin can be inspired by Reichelt et al. [13] who have investigated how to minimize monitoring overhead.

[4]SW: Java 21, Kotlin v2.0.20, Node.js v20.15.1, Gradle 8.5, Win11 Pro 23H2; HW: Intel Core Ultra 7 155H, 32GB RAM

6 Summary and Outlook

Our ongoing work addresses the increasing adoption of Kotlin in cross-platform projects. We contribute a unified approach for performance metric tracing facilitating code instrumentation at compile-time. This helps developers to monitor and optimize their Kotlin applications across various target environments.

In the future, we will investigate the *impact of compiler plugins on compile time*. Further, we aim to incorporate *advanced tracing techniques*, including metrics such as allocation rates and object lifetimes [7]. Furthermore, *adaptive recording* [10] should adjust the kinds and amount of collected data depending on analysis-specific goals. Lastly, we propose *trace-based performance comparisons* across Kotlin platforms to understand platform-specific characteristics.

References

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

[2] G. Kiczales et al. “An Overview of AspectJ”. In: *ECOOP*. 2001.

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

[4] E. Machkasova, K. Arhelger, and F. Trinciante. “The Observer Effect of Profiling on Dynamic Java Optimizations”. In: *OOPSLA*. 2009.

[5] V. Bitto and P. Lengauer. “Building Custom, Efficient, and Accurate Memory Monitoring Tools for Java Applications”. In: *ICPE*. 2016.

[6] S. Kong et al. “Runtime Monitoring of Software Execution Trace: Method and Tools”. In: *IEEE Access* (2020).

[7] M. Weninger, E. Gander, and H. Mössenböck. “Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance”. In: *SSP* (2020).

[8] M. Weninger et al. “Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study”. In: *PACM HCI EICS* (2020).

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

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

[11] M. Moskala. *Advanced Kotlin*. Leanpub, 2023.

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

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

[14] B. Norman. *Writing Your Second Kotlin Compiler Plugin*. `https://blog.bnorm.dev`. 2024.

All web references last accessed 2024-10-20.