Kotlin Multiplatform IR Instrumentation: A Cross-Platform Runtime Overhead Analysis

Lian Hörschläger
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

Compile-time instrumentation of Kotlin's intermediate representation (IR) enables consistent instrumentation across Kotlin's compilation targets, including JVM, JavaScript (JS), and Native. However, the platform-specific runtime costs of instrumentation operations have not been systematically quantified. This paper evaluates the overhead introduced by common instrumentation operations, including `try-finally` blocks, timestamping, counter increments, file I/O, and data structure operations. Using a cross-platform benchmark, we isolate each operation and quantify its per-function overhead across JVM, JS, and Native on macOS, Linux, and Windows hosts. Our results show that `try-finally` and counter increments introduce little overhead, whereas timestamping and file I/O incur substantially higher, target-dependent costs. For these costly operations, JVM consistently exhibits the lowest overhead, followed by Native and JS.

1 Introduction

Collecting information about the behavior of applications at runtime is essential for identifying opportunities to improve performance [12]. In previous work [6] we found that using Kotlin compiler plugins to modify Kotlin's intermediate representation (IR) is a valuable method to collect such data in Kotlin Multiplatform (KMP) applications. To this end, we developed two compile-time instrumentation plugins that collect and serialize data differently: `k-perf` [11] uses a monotonic time source and writes its results to a file in a custom trace format, while `k-perf-otel` [13] utilizes Kotlin's `System.Clock` and the OpenTelemetry platform [10]. Both plugins introduce noticeable overhead, yet this overhead differs substantially due to the different operations instrumented into the observed application. In this work, we systematically investigated the overhead of different atomic instrumentation operations on different target platforms (JVM, JS, and Native) and operating systems (macOS, Linux, and Windows) running on different hardware to gain a better understanding of the influence of those factors on the introduced overhead.

2 Background and Related Work

This section outlines Kotlin's compilation process, its compiler plugins, and related work.

Kotlin Compilation Figure 1 outlines Kotlin's compilation process. The compiler frontend parses source code into a Frontend Intermediate Representation (FIR), which is then converted to a backend IR. The backend repeatedly applies *lowering* to replace high-level constructs with low-level representations. As a final step, the IR is compiled to target-specific code (Java bytecode, JavaScript, or a native binary).

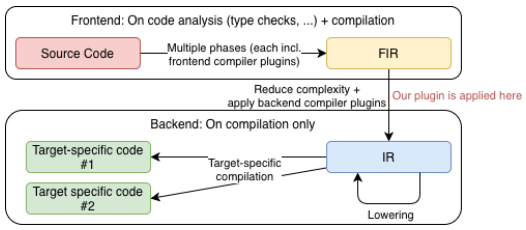

Figure 1: Simplified Kotlin compilation process.

Plugins Kotlin provides two ways to modify the generated code: (1) frontend plugins can modify the FIR, and (2) backend plugins can modify the IR. As previous work [11] has shown, backend plugins have multiple advantages over frontend plugins for collecting performance metrics. Specifically, backend plugins can freely add, remove, or replace entire code structures, whereas frontend plugins are very limited in what they can modify and require developers to annotate the code that should be monitored [8].

Related Work A central application domain of instrumentation is *tracing*. Janes et al. [5] compared 30 tracing tools in terms of features, popularity, benefits, and limitations. A number of tool analyses rely on MooBench [3, 4, 15], a common tool for quantifying the overhead introduced by different tracing agents. For example, Reichelt et al. [14] used MooBench to critically compare Java tracing agents and attributed their overhead to different implementation details. Given the long history of Java-focused tracing analysis and the widespread adoption of Kotlin Multiplatform,

Source code, the benchmarking pipeline, raw experiment results, as well as post-processing scripts are available at `https://github.com/NeonMika/k-perf/tree/ssp26kmp` and archived at `https://doi.org/10.5281/zenodo.22081983`.

measuring, understanding, and minimizing the cross-platform instrumentation overhead of KMP monitoring tools is particularly important.

3 Study Design

To determine the source of the overhead differences, we developed a new plugin that implements all instrumentation operations evaluated in this study. The plugin can introduce top-level mutable variables called system-under-test (*SUT*) variables that hold instrumentation state. For example, a counter operation uses a SUT of type `Int`. The plugin can also modify each function by either replacing the entire function (e.g., wrapping it in a `try-finally`) or adding statements at the *start* or *end* of the function.

We implemented 16 instrumentation operations and report seven representative ones that cover the main categories of operations considered in our study: (1) `TryFinally`, wrapping each function in `try-finally`; (2a) `Time - Clock` and (2b) `Time Monotonic`, measuring function duration via `Clock.System` or a monotonic time source; (3a) `IncrementIntCounter` and (3b) `IncrementAtomicIntCounter`, incrementing an (atomic) integer counter on function entry; (4) `FileLazyFlush`, writing each function name to a file and flushing once after `main`; and (5) `AppendToList`, appending a value (one fixed number per function) on function entry.

We found that either the Kotlin IR compiler or a target-specific compiler might remove instrumentation constructs as part of optimizations. For example, a `try-finally` construct with an empty `finally` block could yield the same target-specific output as if no instrumentation was applied at all. To prevent such optimizations, we generate and access SUT fields (private static fields with public static accessors) when necessary. For example, setting an SUT field in the `finally` block to a fixed value prevents the compiler from removing the `try-finally` construct.

To measure the introduced performance overhead, we used the same *Game of Life* application as in our previous work [11, 13]. The game consists of a 20×20 grid of Boolean cells, each of which can be either alive (`true`) or dead (`false`). In every step of the game, all cells are updated based on the number of their live neighbors. For every combination of (a) instrumentation operation, (b) target platform, and (c) operating system, we executed the application 250 times, each in a separate runtime instance (e.g., a new JVM instance). Each run consisted of 500 game steps, of which the timings of the first 150 steps were ignored as warm-up to ensure steady-state performance. This leaves 350 measured steps per run (i.e., 87,500 step timings per configuration overall). For the statistical analysis, we first average the measured step timings within each run, resulting in 250 independent run-level measurements per configuration. Using these average step run times and the number of instrumented function invocations per step, we report our results in *nanoseconds per instrumented function invocation*, following the reporting scheme established by Reichelt et al. [7, 9]. We statically determined the number of function invocations per step and subsequently confirmed these numbers using the `IncrementIntCounter` operation.

4 Results

Table 1 reports the non-instrumented baseline (*Plain*) and the mean overhead of each operation, computed as mean instrumented minus mean baseline run time. The corresponding 95% confidence interval for the mean overhead is calculated using Welch's t-method. Overheads ≥ 7.5 ns and ≥ 30 ns are highlighted yellow and red, respectively; 30 ns is approximately the average plain runtime and 7.5 ns one quarter thereof.

We classify `try-finally` (-4.39 to $+6.81$ ns), counter increments (-4.32 to $+3.72$ ns), and atomic counter increments (-0.76 to $+4.81$ ns) as *low-cost*. In this single-threaded application, atomic increments add little overhead over non-atomic ones. Negative values (i.e., speed-ups) most probably reflect measurement variability or target-specific execution changes, such as different JIT optimizations. We performed byte-wise comparisons between the non-instrumented and instrumented artifacts to confirm that the instrumentation changed the generated target-specific artifacts (e.g., that every function was wrapped in a try-finally block).

`Clock.System` incurs more overhead ($+33.9$–$+3520$ ns) than monotonic time ($+17.5$–$+540$ ns), with both being particularly costly on JavaScript. This difference is rooted in their respective implementations: wall time must track actual time, including user adjustments and synchronization [2], whereas monotonic time only preserves time relative to a fixed arbitrary origin [1] and is therefore cheaper.

Writing to a file (with lazy flushing) introduces between $+20$ ns and $+288$ ns overhead. However, these overheads exclude the final flush because they capture only the work performed during a step, not the work performed after the final step. Adding values to a list adds between $+2.92$ ns and $+46.5$ ns.

Threats to Validity *Operation Selection*: While real-world profiling might involve more complex operations, they are typically composed of simpler, fundamental building blocks such as counter increments. Thus, we focused our analysis on single-change instrumentation operations that are, in our experience, commonly used across various tracing agents. Aggregating our overheads might then serve as an estimate for more complex instrumentation operations. *Benchmark Selection*: Our single Game of Life microbenchmark is not representative of a full real-world application, but its small scale helps isolate instru-

Instrumentation Operation	macOS			Linux			Windows		
	JVM	JS	Native	JVM	JS	Native	JVM	JS	Native
Plain (reference)	12.1 ± 0.03	23.4 ± 0.09	38.6 ± 0.12	14.0 ± 0.09	48.8 ± 0.75	33.1 ± 0.17	20.5 ± 0.28	40.0 ± 0.62	74.5 ± 0.38
(1) Try Finally	-0.07 ± 0.04	2.89 ± 0.12	-0.01 ± 0.23	-0.07 ± 0.13	-4.39 ± 0.98	-0.38 ± 0.23	-3.94 ± 0.28	6.81 ± 1.22	-3.23 ± 0.45
(2a) Time – Clock	43.0 ± 0.06	1965 ± 1.80	123 ± 0.19	42.7 ± 0.34	3518 ± 14.4	93.9 ± 0.77	33.9 ± 0.44	3094 ± 6.31	162 ± 0.71
(2b) Time – Monotonic	17.5 ± 0.06	380 ± 0.83	63.6 ± 0.17	24.2 ± 0.22	540 ± 3.90	41.9 ± 0.44	41.7 ± 0.37	508 ± 1.34	86.2 ± 0.56
(3a) Increment Int Counter	0.06 ± 0.04	3.72 ± 0.22	0.80 ± 0.15	0.21 ± 0.17	-2.56 ± 1.06	0.28 ± 0.30	-4.32 ± 0.28	0.27 ± 1.03	-2.46 ± 0.46
(3b) Increment Atomic Int	-0.43 ± 0.04	4.81 ± 0.18	1.36 ± 0.17	2.05 ± 0.15	-0.76 ± 1.12	4.08 ± 0.30	-0.34 ± 0.29	4.74 ± 1.21	2.63 ± 0.50
(4) File (Lazy Flush)	20.0 ± 0.10	171 ± 0.28	58.1 ± 0.26	29.5 ± 0.29	265 ± 2.74	73.3 ± 0.72	63.7 ± 0.67	288 ± 1.15	133 ± 1.01
(5) Append to List	2.92 ± 0.04	9.58 ± 0.11	31.2 ± 0.18	7.74 ± 0.22	21.9 ± 1.03	30.6 ± 0.40	15.7 ± 0.33	17.4 ± 0.93	46.5 ± 0.53

Table 1: Average overhead per instrumented function invocation in *ns* relative to the non-instrumented baseline. The baseline is reported as mean ± 95% confidence interval half-width; overhead values are reported as mean difference ± 95% confidence interval half-width.
Machine Specs: *macOS:* Mac Studio, macOS 26.5, 36 GB RAM, Apple M4 Max. *Linux:* Dell Precision 5690, Ubuntu 24.04.3 LTS (Kernel Version 6.17), 64 GB RAM, Intel Core Ultra 7. *Windows:* Dell Precision 5570, Windows 11 (25H2), 32 GB RAM, 12th-gen Intel Core i7. All machines used Java `21` and Kotlin `2.2.0`. All executables were built using Gradle `9.0.0`.

mentation overhead from unrelated behavior such as I/O or network latency and enables easier comparison with previous work [11, 13]. *Confounded Variables*: Because each operating system was evaluated on different hardware, cross-host differences cannot be attributed to the operating system alone; our target-platform comparisons are therefore primarily interpreted within each host system.

5 Summary and Outlook

In this paper we systematically quantified the runtime overhead introduced by Kotlin IR compile-time instrumentation across different target platforms (JVM, JS, and Native) and operating systems. By isolating fundamental instrumentation operations within a cross-platform microbenchmark, we identified significant performance disparities depending on the compilation target and the nature of the operation. Our results demonstrate that simple value modifications (such as counter increments) and control flow wrappers such as `try-finally` introduce negligible overhead across all platforms.

Conversely, operations requiring system interaction, such as timestamping and file I/O, incur substantial and highly target-dependent costs. The JVM consistently executes these heavier operations with the lowest overhead, while JavaScript exhibits severe performance penalties, particularly when retrieving wall-clock time. In line with existing work, our evaluation highlights the critical performance advantage of using monotonic time sources over `Clock.System` for duration measurements, confirming that the choice of time source drastically impacts profiling intrusion.

Future work includes systematic mining of the literature and software repositories to determine which operations are most frequently instrumented in real-world applications. This should include a clear decomposition of complex instrumentation operations into their constituent parts. Furthermore, Kotlin recently introduced WebAssembly as a new compilation target (currently in beta). Rerunning our study with WebAssembly as an additional target would be a natural next step to evaluate its performance.

Acknowledgments

We gratefully acknowledge the support from Dynatrace[1], especially that of Oleg Zinin.

References

[1] M. Rivas and M. Harbour. “Evaluation of new POSIX real-time operating systems services for small embedded platforms”. In: *EECRTS*. 2003.

[2] F. Terraneo et al. “FLOPSYNC-2: Efficient Monotonic Clock Synchronisation”. In: *RTSS*. 2014.

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

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

[5] A. Janes, X. Li, and V. Lenarduzzi. “Open tracing tools: Overview and critical comparison”. In: *JSS* (2023).

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

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

[8] JetBrains. *FIR Plugin API*. Accessed: 2026-08-24. 2024. URL: `https://github.com/JetBrains/kotlin/blob/master/docs/fir/fir-plugins.md`.

[9] D. G. Reichelt et al. “Overhead Comparison of Instrumentation Frameworks”. In: *ICPE*. 2024.

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

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

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

[13] F. Schoenberger and M. Weninger. “OpenTelemetry Instrumentation using Kotlin Multiplatform Compiler Plugins”. In: SSP, 2025.

[14] D. G. Reichelt et al. “Benchmarking the Overhead of Distributed Tracing Agents”. In: *ICPE*. 2026.

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

[1] https://www.dynatrace.com