

MooBench X Kotlin: Kotlin Multiplatform Benchmarking

Jannik Heinzlreiter Markus Weninger
Institute for System Software
Johannes Kepler University Linz, Austria

David Georg Reichelt
Lancaster University Leipzig/
URZ Leipzig

Abstract

The overhead incurred by observability tools in the Kotlin Multiplatform (KMP) ecosystem can vary considerably across target platforms. Given Kotlin’s widespread adoption, measuring and minimizing this overhead is particularly critical. However, MooBench, a benchmark for measuring the performance overhead of observability tools and agents, is primarily designed for Java and bytecode instrumentation. To also support Kotlin, we present *MooBench X Kotlin*, a foundation for reproducibly measuring the overhead of instrumentation tools targeting KMP projects that enables comparisons across various target platforms (JVM, JavaScript, and Native). To demonstrate the applicability of our benchmark, we use it to measure the overhead of *k-perf*, a KMP-specific compile-time performance instrumentation tool. We show that *k-perf* introduces substantially higher overhead on Native and JavaScript than on the JVM, highlighting the importance of cross-platform performance evaluation for KMP instrumentation tools.

1 Introduction

Monitoring and instrumentation can affect the execution characteristics of the systems they observe. Consequently, their overhead should be evaluated under controlled and reproducible conditions. The MooBench micro-benchmark [13] addresses this problem through a well-defined workload and execution methodology, but its implementation and tooling only support Java and Python, focusing on load-time instrumentation (e.g., via bytecode manipulation [1, 2]).

Compiler-based instrumentation for Kotlin differs fundamentally from this model [7]. In particular, *k-perf* modifies Kotlin Intermediate Representation (IR) during compilation, meaning that instrumentation is applied before the benchmark is executed. Supporting this approach therefore requires adaptations beyond translating the benchmark workload itself. Instrumented and non-instrumented artifacts must be built separately and executed under equivalent benchmark configurations.

This work addresses these challenges by reimplementing the MooBench workload in Kotlin while preserving its recursive structure, configurable workload parameters, and busy-wait-based execution mechanism. The benchmark infrastructure is extended to

integrate compile-time instrumentation into the build and execution process to support pairwise evaluation of instrumented and non-instrumented artifacts.

2 Background and Related Work

This section provides an overview of MooBench and discusses related work on observability benchmarking and Kotlin Multiplatform instrumentation.

2.1 MooBench in a Nutshell

MooBench is a micro-benchmark for reproducibly measuring the run-time overhead introduced by observability tools. It consists of a measurement harness and a minimal recursive workload with configurable recursion depth and method execution time. This design enables a controlled number of monitored method calls while reducing the influence of unrelated application behavior. By executing different monitoring configurations, MooBench can distinguish overhead originating from instrumentation, data collection, and data serialization [3].

In addition to the workload itself, MooBench provides automation for configuring and repeatedly executing different observability frameworks. Experiments consist of multiple runs, each executed on a separate runtime (e.g., a new JVM instance), performing warm-up iterations followed by measurement iterations to account for effects such as JIT compilation. The results are stored for subsequent analysis and can also be integrated into continuous benchmarking environments. Originally developed for Kieker [4], MooBench has since been used to evaluate and compare several observability and instrumentation approaches [12].

2.2 Related Work

MooBench has been used to compare the run-time overhead of observability frameworks and instrumentation approaches. For example, Reichelt et al. used MooBench to compare the overhead introduced by OpenTelemetry with the overhead introduced by Kieker [6]. Further studies used MooBench to investigate techniques for reducing monitoring overhead [8] and showed that execution environments can influence both run time and measurement variability [9].

Recent work on KMP has explored compile-time instrumentation using Kotlin compiler plugins and In-

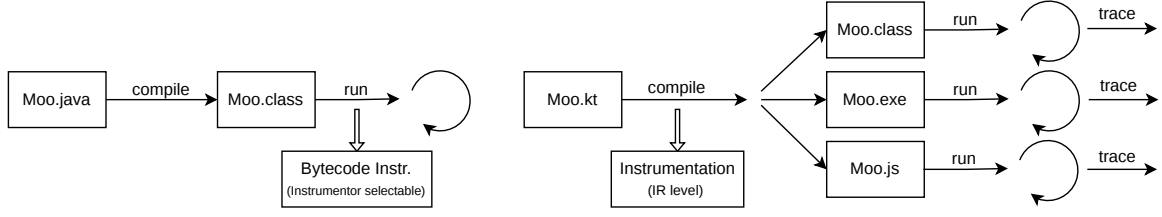


Figure 1: Java bytecode instrumentation workflow compared to Kotlin compile-time instrumentation workflow.

intermediate Representation (IR), including *k-perf* [10] and *k-perf-otel* [11]. While previous MooBench studies mainly focus on JVM workloads, *MooBench X Kotlin* adapts the benchmark to Kotlin Multiplatform and enables the evaluation of compile-time instrumentation across JVM, JavaScript, and Native targets.

3 MooBench X Kotlin

Our goal was to preserve the structure of the original MooBench implementation as closely as possible. At the same time, we aimed to maximize code sharing across the different KMP targets and to introduce target-specific code only where required. In this section, we describe the main architectural changes that were necessary to port MooBench to Kotlin and use it as a harness to measure the performance overhead introduced by compile-time code instrumentation.

3.1 Multiplatform Time Measurement

Java MooBench uses `System.nanoTime()`, which is JVM-specific and thus not suitable for KMP projects. To achieve a platform-independent implementation, it was replaced with KMP-compatible time measurement utilities based on `TimeSource.Monotonic`¹. In particular, the implementation uses `markNow()` to capture the starting point and `elapsedNow()` to determine the elapsed execution time. These changes are highlighted in Listing 1.

```
1 val start = timeSource.markNow()
2 mc.monitoredMethod(this.methodTime, this.recursionDepth)
3 val elapsed = start.elapsedNow().inWholeNanoseconds
```

Listing 1: Kotlin multiplatform time measurement using `TimeSource.Monotonic`.

3.2 Coroutines Replace Threads

KMP does not provide a shared thread abstraction that can be used across all supported targets. Therefore, the thread-based execution model of the original MooBench implementation could not be transferred directly. Instead, we use coroutines from the `kotlinx.coroutines` library, which are the standard concurrency abstraction in KMP². Unlike operating system threads, coroutines are managed by the coroutine runtime and can suspend and resume execution without permanently blocking the underlying thread. On the JVM, coroutines may run on Java threads or

thread pools, while JavaScript relies on its event-loop-based execution mode and Kotlin/Native uses native concurrency mechanisms.

Listing 2 shows how we adapted the original thread-based benchmark execution using coroutines. The `coroutineScope` defines the scope in which the benchmark executions are launched. For each configured benchmark thread, a `BenchmarkingThreadNano` instance is created. Instead of starting a platform-specific thread directly, `launch` creates a coroutine that invokes the corresponding `run()` method.

```
1 coroutineScope {
2   for (i in 0..

```

Listing 2: Benchmark execution using coroutines.

3.3 No Common Run-Time Reflection Across KMP Targets

The original MooBench implementation uses reflection to select a specific benchmark implementation dynamically based on the configured `runnableClassName`. Since KMP does not (yet) provide a common run-time reflection mechanism across all targets, this approach cannot be reused directly.

To solve this problem, we introduced a `ClassRegistry`. Instead of resolving class names to classes dynamically at run time, all supported benchmark implementations are registered statically under their fully qualified class names. The configured `runnableClassName` is then used as a key to retrieve the corresponding factory function from the registry. Although the class names must be registered explicitly, the mechanism preserves the configuration-based selection of benchmark implementations.

3.4 Compile-Time Instrumentation

Since one of our goals for *MooBench X Kotlin* was to measure Kotlin *Multiplatform* instrumentation overhead, MooBench’s existing build infrastructure had to be extended. While the original setup produced a single Java artifact, the Kotlin Multiplatform implementation generates separate artifacts for the JVM, JavaScript, and Native targets, conceptually denoted as `Moo.jar/js/exe`. These artifacts remain suitable for run-time instrumentation using an instrumentation framework supported by the respective target (e.g., Javassist on JVM [1, 2]).

¹<https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.time/>

²<https://github.com/Kotlin/kotlinx.coroutines>

MooBench X Kotlin additionally introduces a compile-time instrumentation pipeline to evaluate compiler-based instrumentation performed by tools such as *k-perf*, see Figure 1. The build process thus further produces one *instrumented* artifact for each Kotlin target, conceptually denoted as `Moo.instrumented.jar/js/exe`. Generating instrumented and non-instrumented artifacts enables pairwise overhead measurements under equivalent benchmark configurations.

4 Results

To demonstrate the applicability of *MooBench X Kotlin*, experiments were performed on a machine running Debian GNU/Linux 13 with Linux kernel 6.12.86+deb13-amd64, powered by an AMD Ryzen 9 9950X processor, 64 GB of DDR5-6000 memory, and a 1 TB SSD. Software versions were fixed to Kotlin 2.3.0, Java Eclipse Temurin 25.0.3+9.0.LTS, and Node.js 24.18.0. The reported experiment used the following MooBench configuration: (1) 10 runs (i.e., restarts of the benchmarking workload), (2) one benchmark thread, (3) a recursion depth of 100, (4) 200,000 benchmark invocations per run, and (5) a method execution time of 0 ns (i.e., the busy-wait time of the innermost recursive function call). This results in 20,000,000 monitored method calls per run, of which the first 10,000,000 are ignored as warm-up, as well as the fastest and slowest 0.5% ones as extreme outliers. Although warm-up behavior differs between the targets, we ensured that the measurements had stabilized within this window for all three targets.

Each benchmark invocation measures the execution time of the complete recursive call chain. With a recursion depth of 100, one measurement therefore covers 100 monitored method calls. The values in Table 1 are normalized by the recursion depth and do not represent individual sub-nanosecond timer measurements. Run times are reported as mean $\pm$ standard deviation. To provide a clear comparison, both absolute overhead and relative slowdown of the instrumented execution are reported.

Platform	Plain [ns]	Instrumented [ns]	Overhead [ns]	Slowdown [x]
JVM	0.99 $\pm$ 0.06	47.46 $\pm$ 4.32	46.47	48.1
Native	0.75 $\pm$ 0.06	171.54 $\pm$ 47.10	170.78	227.8
JS	6.14 $\pm$ 1.29	898.65 $\pm$ 81.65	892.51	146.4

Table 1: Absolute overhead and slowdown introduced by *k-perf* per method call.

The absolute overhead is lowest on the JVM at 46.47 ns per method call, followed by Native (Linux) at 170.78 ns and JavaScript at 892.51 ns. The corresponding slowdown factors are 48.1 $\times$, 227.8 $\times$, and 146.4 $\times$, respectively. Overall, the measurements show that *k-perf* introduces substantial run-time overhead on all targets and that both absolute and relative overhead differ considerably between the targets. Future work should decompose the observed overhead

to quantify the contributions of time measurement, data serialization, and other instrumentation-related operations.

5 Summary and Outlook

This paper introduces *MooBench X Kotlin*, a KMP adaptation of MooBench for evaluating the run-time overhead of Kotlin instrumentation tools such as *k-perf*. The original MooBench was ported while preserving the recursive workload structure, configurable parameters, and busy-wait-based method execution time. The MooBench infrastructure was extended to generate and execute both non-instrumented and instrumented artifacts for multiple KMP targets and to support compile-time instrumentation. This enables reproducible overhead measurements across different instrumentation approaches.

Future work will extend the benchmark with memory utilization measurements for all targets, as well as garbage collection statistics for managed platforms. Advanced automated result processing and visualizations [5] could further support result analysts.

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. Waller and W. Hasselbring. “A comparison of the influence of different multi-core processors on the runtime overhead for application-level monitoring”. In: *MSEPT*. 2012.
- [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] M. Weninger, L. Makor, and H. Mössenböck. “Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations”. In: *STAG*. 2020.
- [6] D. G. Reichelt, S. Kühne, and W. Hasselbring. “Overhead Comparison of OpenTelemetry, inspectIT and Kieker”. In: *SSP*. 2021.
- [7] D. Pfeffer and M. Weninger. “On the Applicability of Annotation-Based Source Code Modification in Kotlin”. In: *MPLR*. 2023.
- [8] D. G. Reichelt, S. Kühne, and W. Hasselbring. “Towards Solving the Challenge of Minimal Overhead Monitoring”. In: *ICPE*. 2023.
- [9] D. G. Reichelt, R. Jung, and A. van Hoorn. “Overhead Measurement Noise in Different Runtime Environments”. In: *SSP*. 2024.
- [10] M. Weninger. “Tracing Performance Metrics in Kotlin Multiplatform Projects via Compile-Time Code Instrumentation”. In: *SSP*. 2024.
- [11] F. Schoenberger and M. Weninger. “OpenTelemetry Instrumentation using Kotlin Multiplatform Compiler Plugins”. In: *SSP*. 2025.
- [12] D. G. Reichelt et al. “Benchmarking the overhead of distributed tracing agents”. In: *ICPE*. 2026.
- [13] D. G. Reichelt et al. “MooBench: A micro-benchmark for performance overhead measurement of observability tools”. In: *JOSS* 11.123 (2026).