

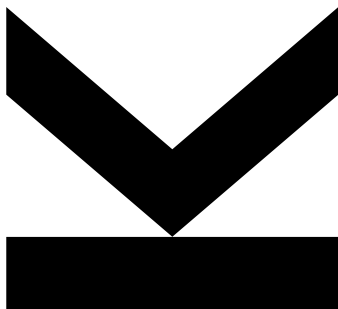
Author
Jannik Heinzlreiter

Submission
**Institute for System
Software**

Thesis Supervisor
**Dipl.-Ing. Dr. Markus
Weninger, BSc**

Month Year
August 2026

MooBench X Kotlin: Porting the MooBench Performance Benchmark Platform to Kotlin



Bachelor's Thesis
to confer the academic degree of
Bachelor of Science
in the Bachelor's Program
Informatik

Bachelor's Thesis

**Moobench X Kotlin: Porting the Moobench Performance
Benchmark Platform to Kotlin**

Dipl.-Ing. Dr. Markus Weninger, BSc

Institute for System Software

T +43-732-2468-4361

markus.weninger@jku.at

Student: Jannik Heinzlreiter

Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc

Start date: March 2026

Background and Motivation

Benchmarking the overhead of monitoring frameworks is crucial to ensure that performance observation does not itself become a bottleneck. **Moobench** is a well-established benchmarking platform specifically designed to measure the overhead introduced by Java-based monitoring agents (such as Kieker).

At the Institute for System Software, we have developed **k-perf**, a Kotlin compiler plugin designed for performance measurement. Unlike traditional Java agents that instrument bytecode at runtime, *k-perf* operates at compile time by modifying the Kotlin Intermediate Representation (IR). Because existing Moobench workloads are designed for Java and often rely on bytecode manipulation or specific Java agent hooks, they are not directly applicable to Kotlin-specific instrumentation techniques.

To bridge this gap and allow for a fair comparison between compile-time IR instrumentation and traditional runtime bytecode instrumentation, a Kotlin-native version of Moobench is required.

Objective

The primary objective of this thesis is to develop **Moobench X Kotlin**, a port of the Moobench benchmarking suite to the Kotlin ecosystem. The project aims to provide a standardized environment to measure the execution overhead of Kotlin-specific monitoring tools.

A key deliverable of this thesis is a Pull Request to the official Moobench repository. This contribution should include a new Kotlin-based benchmark workload, accompanied by all necessary build and benchmark scripts to allow the repository maintainers to automatically measure the overhead of tools like *k-perf* alongside existing Java-based solutions.

Specific Tasks

To achieve these goals, the student will:

- **Analyze Moobench:** Study the architecture and methodology of the existing Java-based Moobench to understand its micro-benchmarking approach.
- **Port Workloads:** Implement the Moobench benchmark logic in Kotlin, ensuring it represents typical Kotlin idioms and structures.
- **Integration with Kotlin Build Tooling:** Develop Gradle-based build scripts to integrate the benchmark with the Kotlin compilation pipeline and *k-perf*.

- **Automation:** Create benchmark scripts to automate the execution, data collection, and result generation process across different configurations.
- **Comparative Evaluation:** Use the newly developed platform to measure and compare the overhead of *k-perf* against a baseline and, if applicable, traditional Java monitoring agents.

Non-Goals

- This thesis does **not** aim to develop new monitoring features for *k-perf* itself, but rather to evaluate its performance.
- The goal is **not** to optimize the Kotlin compiler, but to provide a measurement platform for instrumentation overhead.

Dissemination

In addition to the bachelor's thesis, a short 3-page paper summarizing the design of Moobench X Kotlin and the initial benchmarking results should be prepared for submission to the **Symposium on Software Performance (SSP 2026)**.

Modalities:

The progress of the project should be discussed at least every three to four weeks with the advisor(s). A time schedule and a milestone plan must be set up within the first 2 weeks and discussed with the advisor(s). It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.08.2026.

Abstract

Measuring the overhead of monitoring and observability tools is essential to ensure that adding additional code does not significantly affect the run-time of the observed application.

MooBench is an established benchmarking platform for analyzing the overhead of monitoring tools and agents. However, existing MooBench workloads are designed for Java, Python, and traditional forms of instrumentation, such as bytecode instrumentation. This thesis presents *MooBench X Kotlin*, a port of the central MooBench benchmark logic to the Kotlin ecosystem. The goal is to provide a standardized environment for measuring the overhead of *k-perf*, a Kotlin-specific observability instrumentation tool. *k-perf* is a Kotlin compiler plugin that enables performance measurements through compile-time instrumentation of the Kotlin Intermediate Representation (IR). In contrast to traditional Java agents, the instrumentation is therefore performed during the build process. As part of this thesis, the MooBench workload was ported to Kotlin and implemented as a Kotlin Multiplatform benchmark for JVM, JavaScript, and Native.

The Kotlin/*k-perf* extension was integrated into the existing MooBench infrastructure. This includes Gradle-based build steps, scripts for generating non-instrumented and *k-perf*-instrumented artifacts, and benchmark configurations for the supported target platforms. The generated artifacts can then be compared pairwise to determine the overhead caused by *k-perf*. *MooBench X Kotlin* thus provides a foundation for reproducible overhead measurements.

Kurzfassung

Die Messung des Overheads von Überwachungs- und Observability-Tools ist wichtig, um sicherzustellen, dass das Hinzufügen von Instrumentierungscode die Laufzeit der überwachten Anwendung nicht wesentlich beeinträchtigt.

MooBench ist eine etablierte Benchmark-Plattform zur Analyse des Overheads von Überwachungstools und -agenten. Die bestehenden MooBench-Workloads sind jedoch für Java und Python konzipiert und zielen ausschließlich auf traditionelle Formen der Instrumentierung ab, wie beispielsweise die Bytecode Instrumentierung. Diese Arbeit stellt „*MooBench X Kotlin*“ vor, eine Portierung der zentralen MooBench-Benchmark-Logik in das Kotlin-Ökosystem. Ziel ist es, eine standardisierte Umgebung zur Messung des Overheads von *k-perf*, einem Kotlin-spezifischen Überwachungstool, bereitzustellen. *k-perf* ist ein Kotlin-Compiler-Plugin, das Leistungsmessungen durch Instrumentierung der Kotlin Intermediate Representation (IR) zur Kompilierungszeit ermöglicht. Im Gegensatz zu herkömmlichen Java-Agenten erfolgt die Instrumentierung während des Build-Prozesses der Artefakte. Im Rahmen dieser Arbeit wurde die MooBench-Workload auf Kotlin portiert und als Kotlin-Multiplatform-Benchmark für JVM, JavaScript und Native implementiert.

Die Kotlin-Erweiterung *k-perf* wurde in die bestehende MooBench-Infrastruktur integriert. Die Integration umfasst eine implementation von MooBench in Kotlin, Skripte zur Generierung von nicht -und instrumentierten Artefakten sowie Benchmark-Konfigurationen für die unterstützten Zielplattformen. Die generierten Artefakte können anschließend paarweise verglichen werden, um den durch *k-perf* verursachten Overhead zu ermitteln. *MooBench X Kotlin* bietet somit eine Grundlage für reproduzierbare Overhead-Messungen.

Table of Content

Contents

Abstract	i
Kurzfassung	i
1 Introduction	1
1.1 Problem Statement	1
1.2 Contributions	2
2 Background	3
2.1 MooBench	3
2.2 Kotlin	3
2.3 Kotlin Compiler Plugins	4
2.3.1 k-perf	5
2.4 Gradle	6
3 Overview	7
3.1 Architecture	7
3.2 Workflow	8
4 Translating the MooBench Workload To Kotlin	10
4.1 Recursive Benchmark Workload	10
4.2 Busy Waiting Workload	11
4.3 Implementation of the Simple Workload	12
4.4 The Monitored Class Abstraction	12
5 Implementing the Multiplatform Benchmark Harness	14
5.1 Harness Overview	14
5.1.1 KMP - expected/actual	15
5.2 Tasks of the Harness	15
5.3 Coroutine-Based Execution Model	16
5.4 Shared Harness Code	16
5.4.1 BenchmarkingThreadMilli	17
5.5 Platform-Specific Harness Code	17
5.5.1 BenchmarkingThreadNano	18
5.5.2 BenchmarkMain	20
5.5.3 ClassRegistry	22
5.6 Kotlin Benchmark Entry Points	23
6 Benchmark Execution Integration	25
6.1 Role of the Existing MooBench Infrastructure	25
6.2 Added Infrastructure for MooBench X Kotlin	25
6.3 Setup and Build Benchmark for k-perf	28

6.4	Execution of Platform-Specific Artifacts	30
6.4.1	Starts of the Artifacts	30
6.4.2	Run-time Configuration	31
6.5	Setup, Build and Running Benchmarks	32
6.6	Results of k-perf	33
7	Usage and Evaluation	34
7.1	Prerequisites	34
7.2	How to Interpret the Results	35
7.3	Evaluation of k-perf	35
8	Related Work	37
8.1	MooBench in Previous Work	37
8.2	Observability Overhead Benchmarking	37
9	Limitations and Future Work	39
9.1	Limitations	39
9.2	Future Work	39
10	Conclusion	40
	Literature	41

1 Introduction

Modern software systems often depend on monitoring and observability tools to control their run-time performance and detect problems. However, such tools introduce additional overhead because they collect, process, and store measurement data while the observed application is running. Furthermore, adding instrumentation code to the original codebase introduces additional run-time overhead. If this overhead becomes too large, the monitoring system can influence the performance characteristics of the application. Therefore, it is important to evaluate the overhead of monitoring and instrumentation approaches in a controlled and reproducible manner. MooBench is an established benchmarking platform for measuring the overhead of monitoring frameworks, especially in the context of Java-based applications and Java agents.

Existing MooBench frameworks are based on Java-specific concepts and traditional instrumentation mechanisms, such as run-time bytecode instrumentation. This makes them well suited for evaluating tools such as Kieker [18], OpenTelemetry [20], and other Java-based monitoring agents, but less suitable for instrumentation techniques that are specific to Kotlin.

Kotlin applications can target multiple platforms, including the JVM, JavaScript, and Native. Furthermore, Kotlin compiler plugins can operate at compile time by modifying the Kotlin Intermediate Representation (IR) rather than instrumenting bytecode. As a result, existing Java-based MooBench workloads cannot be used to evaluate the overhead of Kotlin-specific instrumentation in a fair and representative manner. The motivation for this thesis is therefore to bridge the gap between the established MooBench benchmarking methodology and Kotlin-specific compiler-based instrumentation.

1.1 Problem Statement

Existing MooBench workloads are designed for Java-based monitoring frameworks and are therefore closely connected to the Java ecosystem. They assume Java-specific execution models, run-time behavior, and instrumentation mechanisms, such as bytecode instrumentation or Java agents. This limits their applicability to Kotlin-specific monitoring approaches, especially when instrumentation is performed by a Kotlin compiler plugin before the application is executed. The *k-perf* compiler plugin follows a different instrumentation model. It modifies the Kotlin Intermediate Representation (IR) during compilation and therefore produces instrumented artifacts before run time. As a result, the existing MooBench execution model cannot be reused without adapting the workload, build process, benchmark configuration, and execution scripts. A fair comparison requires both non-instrumented and instrumented Kotlin artifacts to be generated, stored, executed, and evaluated consistently. In addition, Kotlin Multiplatform (KMP) introduces further challenges. It allows developers to share common application logic across multiple target platforms while still providing platform-specific implementations where necessary. A single KMP project can therefore target environments such as the JVM, JavaScript, and native binaries, while maintaining a largely shared codebase. A benchmark implementation should share as much logic as possible across the JVM, JavaScript, and Native targets. These differences make a direct one-to-one port of the Java MooBench workload impossible. The problem addressed in this thesis is therefore the lack of a Kotlin-native MooBench workload and execution infrastructure that enables the overhead of *k-perf* to be measured in a reproducible and comparable manner across multiple Kotlin targets. This requires porting the central benchmark workload

to Kotlin, separating platform-independent and platform-specific parts, integrating the resulting artifacts into the existing MooBench infrastructure, and enabling pairwise comparisons between non-instrumented and *k-perf*-instrumented benchmark executions.

1.2 Contributions

This thesis makes the following contributions by benchmarking the observability compiler plugin *k-perf*:

Porting MooBench to Kotlin Multiplatform. The central MooBench workload was ported from Java to Kotlin Multiplatform while preserving its core concepts, including the recursive workload structure, configurable parameters, and the busy-wait workload. This provides a Kotlin-native basis for evaluating instrumentation overhead without relying on the original Java-specific implementation.

Benchmarking *k-perf*. The overhead of the observability compiler plugin *k-perf* was evaluated using the MooBench-based benchmarking approach. This enables a systematic analysis of compile-time IR instrumentation in Kotlin.

Benchmarking three different targets. Since MooBench was ported to Kotlin Multiplatform rather than only to Kotlin/JVM, it is now possible to benchmark *k-perf* on JavaScript, JVM, and Native. This demonstrates how a mostly shared benchmark codebase can be used to execute comparable benchmark scenarios across different Kotlin target platforms.

2 Background

This section provides the theoretical and technical background required for the development of *MooBench X Kotlin*. First, Section 2.1 introduces the existing MooBench micro-benchmark, its supported monitoring frameworks, and its general measurement approach. Afterward, Section 2.2 gives an overview of Kotlin, including Kotlin Multiplatform (KMP) and common application areas of the language. Subsequently, Section 2.3 presents Kotlin compiler plugins, the concept of instrumentation, and *k-perf* as a compile-time instrumentation approach for performance measurement. Finally, Section 2.4 introduces Gradle and explains its role as the build automation tool used in the project.

2.1 MooBench

MooBench is a micro-benchmark designed to measure the performance overhead of common monitoring and observability frameworks. MooBench was developed to determine the impact of instrumentation frameworks on the run-time behavior of applications under controlled conditions. Monitoring provides important information about a system’s run-time behavior, but it also introduces overhead that cannot be ignored from a performance perspective. To determine the overhead of observability frameworks, MooBench introduces an example workload application with a defined configuration. This program contains a monitored method whose execution time and recursion depth can be adjusted. The recursion depth allows control over the number of method calls performed. During a complete benchmark run, various configurations, such as runs with and without instrumentation, as well as additional framework-specific settings, are compared. Finally, the measured data is evaluated using an R script and presented in a clear, organized manner. MooBench can be used to examine various aspects of observability frameworks. These include logs, metrics, and traces. While logs provide chronologically ordered information about events, metrics describe numerical characteristics of system behavior. Traces, on the other hand, represent sequences of requests, transactions, or method calls. Therefore, they are particularly relevant for analyzing control flows and run-time behavior. MooBench makes it possible to measure the additional overhead incurred by collecting and processing such telemetry data.

MooBench supports, among others, Java-based frameworks such as Kieker [18], OpenTelemetry [20], Elastic APM [1], Pinpoint [26], and Scouter [33]. Furthermore, MooBench supports benchmarking the Python implementation of OpenTelemetry. For each framework, MooBench provides its own configurations and scripts that can be used to run automated benchmarks. The results include statistical metrics such as the mean, median, standard deviation, quartiles, and confidence intervals.

2.2 Kotlin

Kotlin is a modern, statically typed programming language that can be used for various purposes, such as backend, mobile, desktop, and web development. This language was designed with the goal of eliminating boilerplate code, such as getter and setter methods, and ensuring interoperability with existing Java code (between Java and Kotlin in both directions) [11]. This interoperability allows Kotlin code to use established Java libraries and frameworks. On the Java

Virtual Machine, Kotlin can be combined with well-known backend frameworks such as Spring or Ktor.

In addition, Kotlin can be compiled to JavaScript or, using Kotlin/Native, to native binaries, allowing Kotlin code to run outside of the Java Virtual Machine as well. Therefore, one of the most interesting features of Kotlin is Kotlin Multiplatform (KMP). KMP allows the same code to be used across multiple platforms. Furthermore, platform-specific implementations can also be added in a separate folder structure. With Kotlin Multiplatform, it is possible to develop applications that run on different platforms without the additional effort of implementing the logic separately for each platform. KMP supports web, server, desktop, iOS, and Android, among other areas. Kotlin is currently the recommended programming language for Android development. In the context of this thesis, KMP is particularly relevant because *k-perf* (see Section 2.3.1) was also developed for KMP. This enables the investigation of performance measurements in projects that use Kotlin-specific language features and tools.

2.3 Kotlin Compiler Plugins

Compiler plugins are extensions for the compiler that allow the code compilation process to be customized or extended without modifying the compiler itself. These plugins make it possible to add code or modify existing code. The K2 Kotlin compiler is a modular compiler that can be roughly divided into two phases: the compiler frontend and the compiler backend. Figure 1 shows a simplified diagram of the compiler.

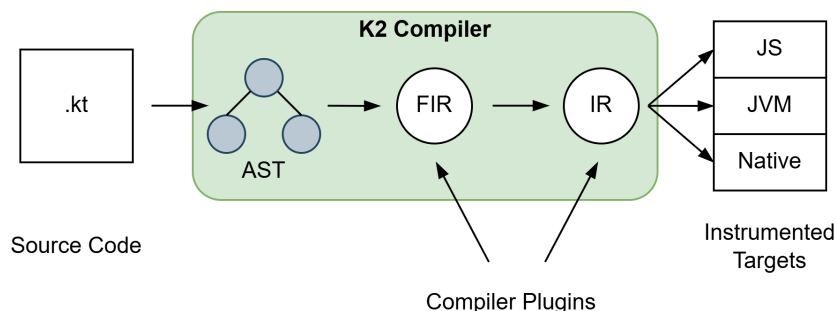


Figure 1: Simplified overview of the K2 compiler.

The **frontend** is primarily responsible for the syntactic and semantic analysis of the Kotlin code. This is where parsing, name and symbol resolution, as well as type checking and type inference take place. At the end of this phase, a semantically enriched representation of the Abstract Syntax Tree (AST) is produced: the Frontend Intermediate Representation (FIR).

The **backend** handles the further transformation of the analyzed code through the generation of the target code. In this process, the FIR is converted into the Kotlin Intermediate Representation (IR). During this phase, the compiler also performs optimizations on the code. Additionally, Kotlin-specific constructs such as inline functions, properties, and lambdas are converted into structures more suited to the target platform. Finally, this phase generates the desired target code [25].

A key concept in this context is instrumentation. This involves inserting additional code (instructions) into the existing code to measure the run-time behavior of, functions for example. First, there is source code instrumentation. Here, logic is added to the original code. It is also possible to modify code that has already been compiled (bytecode), which is used in JVM applications for example. Compile-time instrumentation on the other hand intervenes during the compilation process. Finally run-time instrumentation exists, which modifies the code at run time using agents. The most relevant forms for this thesis are the following [28].

Run-Time instrumentation. Run-time instrumentation modifies a program while it is being loaded or is already running. In JVM-based systems, this is commonly implemented through Java agents. Such agents are packaged as JAR files and can be started together with the application or attached to an already running JVM. The Java instrumentation API allows agents to transform class files at load time and, depending on the JVM capabilities, to redefine or retransform already loaded classes. Probes are inserted by modifying the bytecode of selected classes and methods before or during execution. Such probes consist of measurement code that is added to the program. This approach is flexible because it does not require any changes to the original source code or recompilation of the application. However, run-time instrumentation introduces additional complexity, since the transformation is performed in the execution environment itself and may influence class loading, startup time, JIT compilation behavior, and the overall run-time overhead of the monitored application [24, 28].

Compile-Time instrumentation. Compile-time instrumentation is performed during the compilation process, before the final executable artifact is generated. Instead of modifying already compiled bytecode at load-time or run-time, the compiler or a compiler extension inserts measurement logic into an intermediate representation of the program. In the case of Kotlin, backend compiler plugins can operate on the Kotlin Intermediate Representation (IR). A plugin can therefore inspect declarations and function bodies. Furthermore, it can insert additional statements, for example at function entry and exit points. Compared to run-time instrumentation, compile-time instrumentation produces an already instrumented artifact and does not require a separate run-time agent. This makes the instrumentation more reproducible and tightly integrated into the build process. At the same time, it is less flexible at run time, since changes to the instrumentation usually require recompilation. Since the instrumentation happens at compile time, it is not necessary to instrument each target separately, as the different targets are generated from the instrumented IR [9, 28].

2.3.1 k-perf

k-perf is a backend Kotlin Compiler plugin that instruments a Kotlin program at the IR level to generate detailed execution traces. It is used to add measurement logic to the original code in order to measure, the run-time of a specific method for example. Function calls and returns are traced so that run-time information, such as entry and exit events, and especially execution times, can be recorded. After instrumentation, the modified IR is processed by the regular backend phases and then translated into the respective target code.

The following configuration options can be set for *k-perf* during the build process of the different artifacts:

- `kperfEnabled`: Enables or disables *k-perf*

- `kperfFlushEarly`: Flushes trace file after every entry/exit (slower but safer)
- `kperfInstrumentPropertyAccessors`: Trace property getters/setters (note: JVM back-end may optimize these away)
- `kperfMethods`: Comma-separated regexes matched against fully qualified function names ("`.*`" means all methods)

k-perf creates trace files during execution, which can be further anal if required. These files look like the following:

```

1 # trace file example
2 >;0      enter method 0 (main)
3 >;1      enter method 1 (GameOfLife.step)
4 <;42     exit method 1, elapsed 42 us
5 <;105    exit method 0, elapsed 105 us

```

Listing 1: Example trace file generated by *k-perf*

2.4 Gradle

Gradle is a tool used to automate and configure the build process of a software project. This automation is achieved using scripts. These are `build.gradle` files that use a specific DSL (Domain-Specific Language). For example, Groovy DSL or Kotlin DSL is used. Among other things, the scripts can be used to define plugins, dependencies, compiler options, tasks, and project-specific configurations. Furthermore, it is possible to define properties that can be used to make the build process more dynamic.

A central concept of Gradle is the task model. A task represents a single executable unit of work within a build process, such as creating JAR archives, copying dependencies, or publishing artifacts to repositories.

Gradle also supports KMP projects for Kotlin. This makes it possible to specify configurations for individual artifacts in the Gradle script, such as the artifact name based on a flag, the entry point for the program, and the Java version to be used. For larger projects, Gradle also allows defining subprojects. In this case, the project consists of a main project and several subprojects that are defined in a shared configuration file [6].

In addition, Gradle allows plugins to be integrated into a software project. This enables the addition of project-specific functionality. Plugins can be used to encapsulate reusable build logic and add configurations or extensions to the project. With an extension like this, processes such as running tests or generating artifacts do not need to be defined entirely manually in the build script [5, 7]. In the context of this thesis, Gradle is required to configure the *MooBench X Kotlin* build as well as the *k-perf* plugin integration.

3 Overview

This section provides a high-level overview of the structure and scope of *MooBench X Kotlin*. First, the architectural differences between the original Java-based MooBench and the Kotlin Multiplatform implementation are introduced. Then, the general workflow of the benchmark is presented.

3.1 Architecture

An overview of the architectural differences between the original MooBench implementation and MooBench in Kotlin is presented in this section. The main difference is the point at which instrumentation is applied. The original MooBench uses Java-based run-time instrumentation, whereas the new *MooBench X Kotlin* uses compile-time instrumentation through *k-perf*.

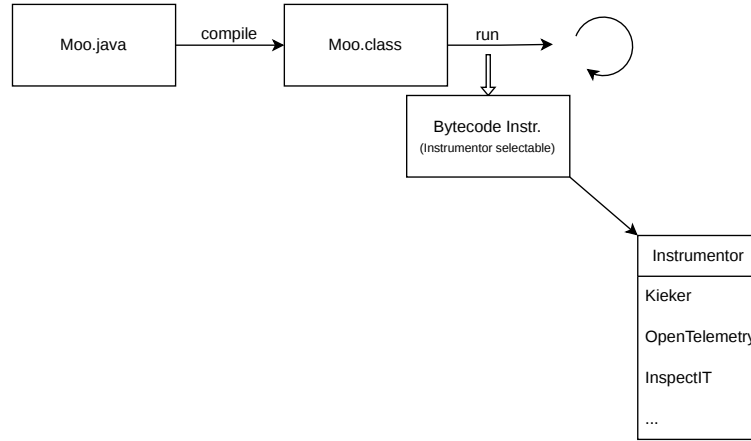


Figure 2: Architecture of the original MooBench with run-time bytecode instrumentation.

Figure 2 presents the original architecture of MooBench. The compiled `.class` files are executed on the JVM. At run time, the application is instrumented by the selected instrumentation or monitoring framework, which inserts probes at method entry and exit.

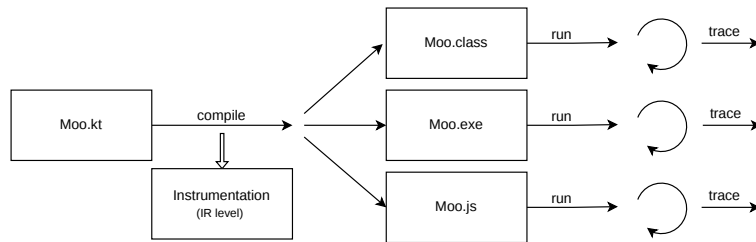


Figure 3: Architecture of *MooBench X Kotlin* with compile-time IR instrumentation for multiple target platforms.

In contrast, Figure 3 shows the architecture of *MooBench X Kotlin*. The monitored application is implemented in Kotlin and processed by the Kotlin compiler. During compilation, *k-perf* instruments the Kotlin Intermediate Representation before the final target-specific code is generated. The result is a plain or instrumented artifact for JS, JVM, or Native. These artifacts generate trace files (see Section 2.3) during execution.

3.2 Workflow

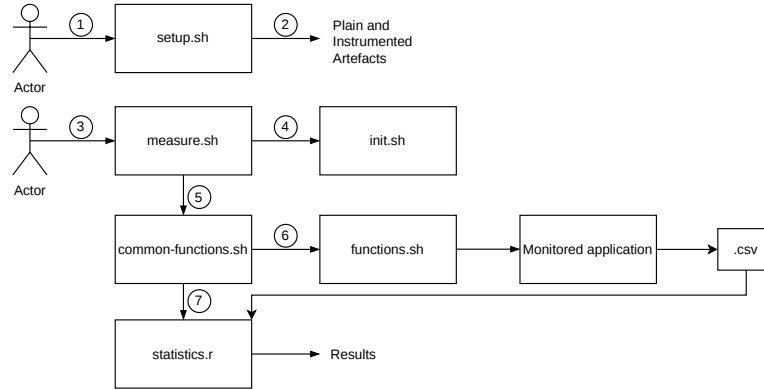


Figure 4: Workflow of *MooBench X Kotlin* from artifact generation to benchmark execution and result analysis.

Figure 4 shows the simplified workflow of *MooBench X Kotlin*. The workflow can be divided into two main phases. First, the required benchmark artifacts are built. Afterwards, a framework-specific benchmark script is used to execute and evaluate an instrumentation framework.

① A user or an automated script starts `setup.sh`. This script builds the configured *MooBench* projects and checks whether all additional scripts required for the build are available. The complete setup and build process is described in more detail in Section 6.5.

② After `setup.sh` has finished, the plain and instrumented artifacts of *MooBench X Kotlin* are located in the `frameworks/k-perf-kotlin/build` directory. At this point, the artifacts required for the benchmark execution have been generated.

③ A benchmark is started by executing the framework-specific `measure.sh` script. For example, the k-perf benchmark is started through `frameworks/k-perf-kotlin/measure.sh`. Therefore, no additional instrumentor parameter is required. The selected `measure.sh` script already determines which framework is benchmarked. The script verifies that the required tools, such as Java and R, are installed, invokes `init.sh` to prepare the benchmark configuration, and uses `common-functions.sh` to run the required benchmark configurations of the selected framework.

④ The `init.sh` script prepares the run-time configuration for the benchmark. It assigns the default values defined by *MooBench* to the variables. These include, for example, `NUM_OF_LOOPS`,

THREADS, RECURSION_DEPTH, TOTAL_NUM_OF_CALLS, METHOD_TIME and SLEEP_TIME. Values explicitly provided before starting the benchmark are retained and can therefore override the defaults. The run-time configuration is described in more detail in Section 6.4.2.

⑤ The `common-functions.sh` script contains the generic benchmark control flow shared by the different MooBench frameworks. Its `executeAllLoops` function coordinates the configured benchmark executions and delegates the framework-specific execution to the corresponding `functions.sh` script. For k-perf, this is `frameworks/k-perf-kotlin/functions.sh`. After all benchmark executions have finished, `common-functions.sh` also starts the statistical evaluation.

⑥ The framework-specific `functions.sh` script contains the commands required to run and measure the monitored application. For the k-perf integration, the plain and instrumented artifacts are executed with the same benchmark parameters for the supported targets. The measured execution data is written to `.csv` files, which are later used for the statistical evaluation. The platform-specific artifact execution is explained in more detail in Section 6.4.

⑦ Finally, the `statistics.r` script analyzes the generated `.csv` files. It calculates statistical values such as the mean, standard deviation, 95% confidence interval, quartiles, maximum, and minimum. The resulting statistics are then printed to the terminal by the R-script. Further information about the evaluation is provided in Section 7.3.

Using this scripting approach, the benchmarking process can also be automated. For example, the benchmark can be executed automatically after a push to the repository. In a continuous integration environment such as GitHub Actions, workflows can be configured to run on specific repository events, including push events [3]. The benchmark scripts can therefore be executed whenever the implementation of k-perf, the workload, or the benchmark harness changes. This allows performance regressions to be detected earlier without requiring a manual benchmark execution after every change. However, benchmark results obtained in CI environments must be interpreted carefully, because shared or virtualized runners can introduce additional noise through varying system load and scheduling effects. Reichelt et al. show that MooBench can be executed continuously and discuss the influence of different execution environments on overhead measurements [29].

4 Translating the MooBench Workload To Kotlin

This section provides an overview of the MooBench workload and how it is implemented in Kotlin to provide an almost entirely Kotlin-based solution. Section 4.1 describes the recursive benchmark workload in general and why a recursive benchmark is a good choice for benchmarking observability frameworks. Furthermore, this section discusses the workload using busy waiting, in particular, why it is important to use `TimeSource.Monotonic` instead of `Sleep`, and how this is implemented in the thesis. Finally, this section shows how the entire workload is implemented, including two different versions of the workload.

4.1 Recursive Benchmark Workload

The MooBench workload is based on a recursive benchmarking approach. In this approach, an instrumented method is called recursively until a configured maximum recursion depth is reached. Figure 5 shows the basic structure of this workload. The structure consists of a call chain in which each method call either executes another recursive call or busy waits (Section 4.2).

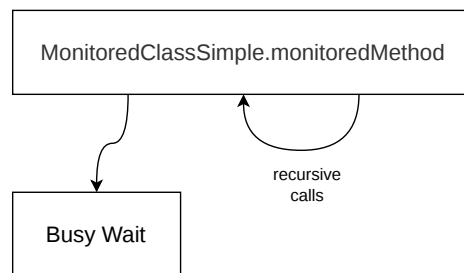


Figure 5: Recursive MooBench workload with a configurable recursion depth and busy-waiting.

The recursive structure serves to increase the number of method calls in a controlled manner. Each recursive step generates an additional method entry and a corresponding method exit. For example, if the benchmark is executed to a depth of `recursionDepth`, a linear chain of nested method calls is created. This allows the number of potential instrumentation points to be varied in a targeted manner without having to change the underlying logic of the benchmark. This structure is particularly relevant for observability and monitoring frameworks, since instrumentation often occurs at method boundaries [32]. Typical probes are executed before entering a method and after exiting it. In the case of recursive calls, this results in additional events for each level of recursion. Instrumenting method entries and exits thus generates at least two measurement points per method call. As `recursionDepth` increases, so does the number of trace events generated.

This relationship is central to overhead measurement. The measured overhead arises not only from the one-time activation of a monitoring mechanism but primarily from the repeated execution of the inserted probes. Each probe incurs additional overhead, for example, by capturing a timestamp, generating a trace entry, writing to a buffer, or serializing measurement data [30, 34]. Recursion allows this overhead to be systematically scaled. A low recursion depth

generates few measurement points, while a high recursion depth places a greater load on the monitoring system. The recursive workload also has the advantage of being easily reproducible. Unlike complex real-world applications, it contains only a few uncontrolled factors. This makes it well-suited for isolating and analyzing the overhead of an instrumentation strategy. While real-world applications contain many different control flows, I/O operations, and dependencies, this workload focuses on the basic case of repeated method calls. This simplification is particularly important for a micro-benchmark, as the impact of instrumentation should be separated as clearly as possible from other run-time effects.

In the context of this thesis, this approach is used to quantify the overhead of Kotlin-specific instrumentation. Since *k-perf* operates at the level of the Kotlin Intermediate Representation (IR), the relevant methods are instrumented at compile time during compilation. The recursive workload then ensures that the instrumented methods are executed frequently enough to make the additional overhead visible. Recursion thus forms the basis for a controlled load on the instrumentation mechanism.

4.2 Busy Waiting Workload

In addition to the recursion depth, `methodTime` is a crucial parameter of the workload. While `recursionDepth` determines the number of method calls and thus the number of potential instrumentation points, `methodTime` specifies how long the busy waiting of the executed method i.e., the recursion’s base case, should take. This allows the ratio between the actual workload and the instrumentation overhead to be controlled.

The method execution time is not generated by real logic but is artificially achieved using busy waiting. This is useful for a benchmark, as real application logic would introduce additional influencing factors. For example, memory accesses, I/O operations, or external libraries could affect run-time behavior [2].

```

1 val start_time = TimeSource.Monotonic.markNow()
2 var currentTime : Long
3 do {
4     currentTime = start_time.elapsedNow().inWholeNanoseconds
5 }while(currentTime < methodTime)

```

Listing 2: Implementation of method run-time using busy waiting in Kotlin.

Listing 2 presents the busy waiting of the implementation. To simulate work, the method performs busy waiting using Kotlin’s `TimeSource.Monotonic`. This time source is intended for measuring time intervals with nanoseconds accuracy and returns a `TimeMark` via `markNow()`, which is bound to the respective time source. The time elapsed since this marker can then be determined using `elapsedNow()`. By default, the Kotlin standard library uses a monotonic time source for such interval measurements; `TimeSource.Monotonic` is described as the most precise time source available on the respective platform [15, 16]. The parameter `methodTime` is specified at the start of the program and is then passed to the instrumented method.

Using an active wait is more appropriate for a benchmark than using `sleep`. A call to `sleep` hands execution over to the scheduler, and thus it cannot be guaranteed that the thread will resume exactly after the specified time. The actual sleep duration depends, among other things, on the operating system, scheduler, timer resolution, and system load [21]. For a micro-benchmark designed to investigate small differences in run-time, this additional timing inaccuracy

would be problematic. An active wait avoids this scheduler-based control flow switch and keeps execution within the current thread. Furthermore, `sleep` is not KMP-compatible and therefore cannot be used for all targets in the benchmark, as not all Kotlin Multiplatform targets, such as JavaScript, provide a threading model that supports blocking thread operations in the same way as the JVM.

4.3 Implementation of the Simple Workload

As already described in Section 4.1, the workload is implemented as a recursive method called `monitoredMethod`. The method takes two parameters:

- The method time
- The recursion depth

As described in Section 4.2, `methodTime` is needed to wait for a specific duration in the innermost function call. `recursionDepth` on the other hand controls the recursion depth of the benchmark. Both of these parameters must be passed as arguments when the program is started. Section 6.4 discusses the program invocation in more detail.

```

1 actual override fun monitoredMethod(methodTime: Long, recursionDepth: Int): Long {
2     if (recursionDepth > 1) {
3         return monitoredMethod(methodTime, recursionDepth - 1)
4     } else {
5         val startTime = TimeSource.Monotonic.markNow()
6         var currentTime: Long
7         do {
8             currentTime = startTime.elapsedNow().inWholeNanoseconds
9         } while (currentTime < methodTime)
10        return currentTime
11    }
12 }

```

Listing 3: Implementation of the recursive `monitoredMethod` method.

Listing 3 shows the implementation of the workload, specifically the `monitoredMethod`. Each time the method is called, the current recursion depth is checked. As long as it is greater than 1, the method goes one level deeper into recursion and calls itself with `recursionDepth - 1`. As soon as the last level, `recursionDepth == 1`, is reached, the method run-time using busy waiting is executed, and the measured time value is returned afterward. The returned time value is essentially meaningless, but it is stored in an array (outside of the `monitoredMethod`) to prevent the JIT compilers from optimizing away the `monitoredMethod` call. A closer look at this problem will be provided in Section 5.4 and 5.5.

The workload itself is deliberately kept simple, since additional code - such as I/O operations or various control flows - would make the measurement results more difficult to interpret. As a result, the method run-time would be influenced not only by the configured `methodTime` and instrumentation, but also by other factors such as garbage collection, JIT compilation, or even external system effects. [2, 34].

4.4 The Monitored Class Abstraction

Since the benchmarks can be run with multiple threads rather than just one, there are also two different implementations of the `MonitoredClass`. The workload implementation in Sec-

tion 4.3 describes only the `MonitoredClassSimple` version, which currently does not support multithreading. The abstraction is represented by a common interface, `MonitoredClass`. This interface only ensures that all implementing classes provide an implementation of `monitoredMethod`.

`MonitoredClassSimple` represents the basic single-threaded variant of the workload. A more detailed description of this variant can be found in Section 4.3. This variant is particularly well-suited for benchmarking the framework itself, as it avoids threading-related effects. Thus, only the overhead caused by using *k-perf* is actually measured here.

`MonitoredClassThreaded` is the second implementation of the workload seen in Listing 4. As the name suggests, this variant uses multiple threads. For this reason, `TimeSource.Monotonic` cannot simply be used in the same way as in `MonitoredClassSimple`, since threads can be interrupted by the scheduler during execution and resumed after an indeterminate amount of time. If `TimeSource.Monotonic` were used in this case, the time during which the thread is suspended would also be included in the measured method time. To ensure that only the wall-clock time (CPU time) by the current thread is measured, the external Java API `ThreadMXBean` is used, which provides the `currentThreadUserTime` value representing only the CPU user time consumed. The remainder of the `monitoredMethod` is identical to the version in `MonitoredClassThreaded`.

```

1 actual class MonitoredClassThreaded : MonitoredClass {
2     val threadMXBean: ThreadMXBean = ManagementFactory.getThreadMXBean()
3     actual override fun monitoredMethod(methodTime: Long,
4                                         recDepth: Int): Long {
5
6         if (recDepth > 1) {
7             return this.monitoredMethod(methodTime, recDepth - 1)
8         } else {
9             val exitTime: Long = this.threadMXBean.currentThreadUserTime
10                + methodTime
11             var currentTime: Long
12             do {
13                 currentTime = this.threadMXBean.currentThreadUserTime
14             } while (currentTime < exitTime)
15             return currentTime
16         }
17     }
18 }

```

Listing 4: JVM-specific implementation of `monitoredMethod` using `ThreadMXBean`.

Since `ThreadMXBean` can only be used within the JVM, this is a JVM-specific implementation. For the Native and JS targets, `MonitoredClassThreaded` uses the same implementation as `MonitoredClassSimple`. This is necessary as there is currently no comparable library available for these two targets.

By dividing the monitored workload into simple and thread-based implementations, MooBench in Kotlin can evaluate an instrumentation overhead under various execution conditions. The simple variant focuses on the basic recursive call structure, while the thread-based variant adds platform-specific threading behavior. In both cases, the instrumented method remains `monitoredMethod`, ensuring that the generated instrumentation events are comparable across all workload variants.

5 Implementing the Multiplatform Benchmark Harness

In this section, we will provide an overview of the harness implementation. In more detail, we will start with an introduction of the tasks of the harness. Afterward, in Section 5.6 we will give a short introduction to the entry points of the program and why each target requires its own. Next, we will take a look at the coroutine-based execution model, where an overview of `Coroutines` will be provided. Then, Section 5.4 and 5.5 introduce the shared components and the platform-specific components of the benchmark harness. Finally, we inspect the `ClassRegistry` which was implemented as a workaround because KMP does not offer reflection at run time for every platform.

In the following, the term *benchmark harness* refers to the components that configure, coordinate, and measure the execution of the workload. A *benchmark worker* repeatedly invokes the monitored workload and records the corresponding measurements. The *workload* itself is represented by the implementations of `MonitoredClass` described in Section 4. A *benchmark run* denotes one execution using a specific benchmark configuration, while an *artifact* refers to a generated executable JVM, JavaScript, or Native benchmark variant.

5.1 Harness Overview

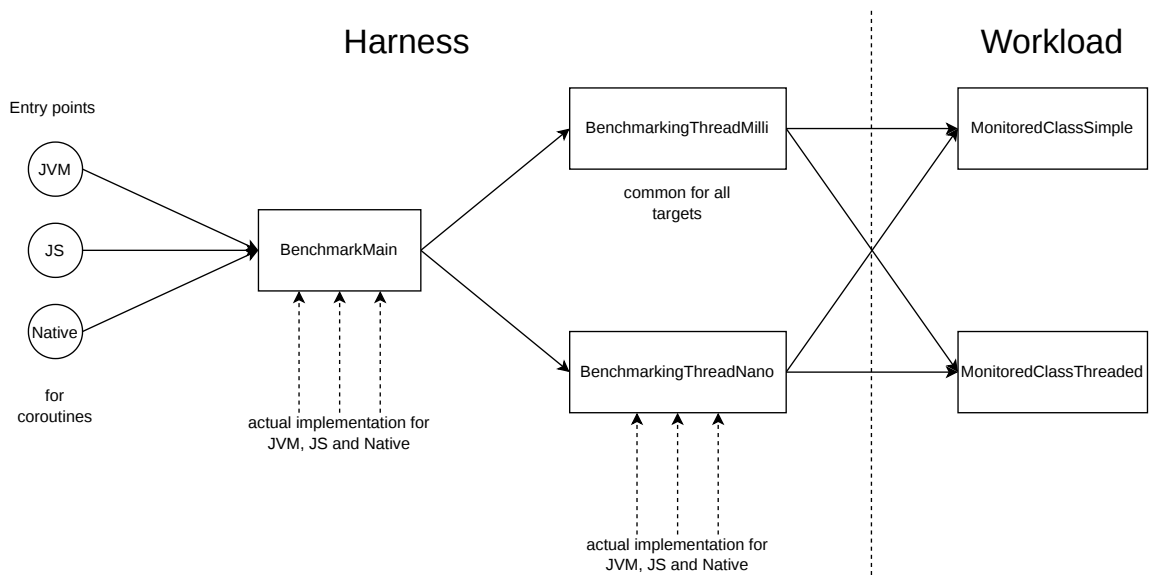


Figure 6: Overview of the benchmark harness and its interaction with the workload.

Figure 6 provides an overview of the main components of the benchmark harness and their interaction with the workload. The harness is separated from the actual workload, which is represented by the two implementations `MonitoredClassSimple` and `MonitoredClassThreaded`. The benchmark execution starts at a platform-specific entry point for JVM, JavaScript, or Native. These entry points are required because the supported targets use different execution models for

starting the coroutine-based benchmark. Independent of the target, control is then delegated to `BenchmarkMain`. This component initializes and coordinates the benchmark execution and creates the selected benchmark worker.

Two benchmark worker implementations are available. `BenchmarkingThreadMilli` contains only platform independent functionality and can therefore be shared across all supported targets. `BenchmarkingThreadNano`, on the other hand, requires platform-specific implementations because some of the collected run-time information depends on target-specific APIs. Both worker implementations execute one of the available workload variants by repeatedly invoking its `monitoredMethod`. This separation keeps the workload independent of the surrounding measurement and coordination logic.

5.1.1 KMP - `expected/actual`

KMP distinguishes between shared and platform-specific implementations using the `expect/actual` mechanism. An `expect` declaration defines an API in the shared code, while an `actual` declaration provides its implementation for a specific target. *MooBench X Kotlin* uses this mechanism for components whose implementation depends on JVM, JavaScript, or Native functionality.

5.2 Tasks of the Harness

The benchmark harness is responsible for controlling the entire benchmark and measuring the workload. The actual execution-time measurements are performed by benchmark workers, such as `BenchmarkingThreadMilli` and `BenchmarkingThreadNano`, while the monitored workload itself only contains the code whose execution is measured. Therefore, the most important tasks of the harness include:

- Reading configuration parameters.
- Initializing the benchmark environment.
- Coordinating execution.
- Storing the results.

At the start of the benchmark, the various parameters passed to the benchmark program at startup (for a more detailed explanation of the parameters see Section 6.4) are read in and cached. These parameters include the recursion depth, the method execution time, the number of benchmark iterations, and the number of threads used. After parsing the values, the harness is initialized with the corresponding components, and the monitored workload instance is created. While the benchmark is running, the harness coordinates the various threads. Depending on the configuration, one or more threads are started, each of which individually invokes the workload with the parsed parameters. As shown in Section 5, no measurement data is processed within the workload; rather, the workload is simply executed.

By centralizing the control logic in the harness, the same workload can be used under different execution configurations. The harness thus defines the basic flow of a benchmark run, although platform-specific details—such as program entry points, system API calls, specifics of parameter

parsing, or system information such as heap usage are implemented using KMP *expect/actual* constructs.

Once the worker execution has been completed, the harness aggregates the recorded metrics and writes them to the configured output file. This includes, in particular, the measured execution times of the individual operations, as well as additional metadata that may be needed for later analysis. Finally, the harness executes the necessary cleanup and termination logic. It thus forms the technical execution layer of the benchmark: it manages configuration, flow control, worker coordination, and result persistence, while the workload provides only the program logic to be measured.

The statistical evaluation of the generated measurement files is not part of the benchmark harness itself. It is performed afterwards by the MooBench analysis scripts, in particular by the `statistics.r` script described in Section 7.

5.3 Coroutine-Based Execution Model

Coroutines were chosen as the implementation model for the benchmark. A traditional thread-based implementation was not suitable for Kotlin Multiplatform, because not all supported target platforms provide the same threading model. The *kotlinx.coroutines* library provides coroutine support for KMP [14].

A coroutine differs from a traditional thread in that a coroutine itself is not an operating system thread. A normal thread is an execution unit managed by the scheduler. A coroutine, on the other hand, is an execution structure managed by the coroutines run-time. It encapsulates a suspendable program unit that can be interrupted at suspension points and later resumed by a dispatcher on the same or a different thread. In this context, “suspendable” means that the execution of a coroutine can be interrupted at specific points. This does not permanently block the thread underlying the coroutine. Coroutines do not replace threads; rather, they abstract the specific execution mechanisms of the respective target platform.

On the JVM, coroutines can therefore be executed on Java threads or thread pools. On the JavaScript target, however, a classic Java thread model is not available. Here, coroutines are embedded in the event-loop-based execution model, and asynchronous processes are typically represented by `Promise` objects. For Kotlin/Native, execution is based on the native runtime environment and its platform-specific threading and scheduling characteristics.

The different underlying execution models of the targets must be considered when interpreting the benchmark results. These differences can affect scheduling, parallelism, timing, and execution order. Therefore, measurement results across targets cannot be used as a basis for direct performance comparisons.

5.4 Shared Harness Code

The shared harness code includes all components that are shared across the KMP targets. In this specific case, it covers the code for parameter extraction, workload abstractions (see Section 4.4), and the coordination logic for benchmark threads. With this implementation, the amount of target specific code can be minimized. Instead, code duplication can be reduced to a minimum.

5.4.1 BenchmarkingThreadMilli

The class `BenchmarkingThreadMilli` represents a shared worker implementation that executes benchmark operations and records their execution time with millisecond granularity. Since this implementation does not use any platform-specific code, it is used across all possible targets of KMP.

```
1 class BenchmarkingThreadMilli (private val mc: MonitoredClass,
2                               private val totalCalls: Int,
3                               private val methodTime: Long,
4                               private val recursionDepth: Int)
5                               : BenchmarkingThread {
6     private val timings: LongArray = LongArray(totalCalls)
7     //to prevent the (JIT) compiler from not calling the monitored method
8     private val mcTimings: LongArray = LongArray(totalCalls)
9
10    override fun run() {
11        val timeSource = TimeSource.Monotonic
12        for (i in 0..
```

Listing 5: Shared implementation of the `BenchmarkingThreadMilli` benchmark worker.

Listing 5, shows the implementation of `BenchmarkingThreadMilli`. First, the class receives the monitored workload instance `mc` and the benchmark configuration through its constructor. The parameter `totalCalls` defines how often the monitored method is executed, while `methodTime` and `recursionDepth` are forwarded to the workload on each invocation. Thus, overall `totalCalls * recursionDepth` executions of the monitored method will be performed. The measured execution times from `mc.monitoredMethod` are stored in a `LongArray` to prevent the JIT compiler from treating the method invocations as unused and potentially optimizing them away. The real measured execution times of the plain and instrumented method are also stored in a fixed-size `LongArray`. At the start of the `run` method, in which the actual benchmark is implemented, `TimeSource.Monotonic` is defined as a time source. For each benchmark operation, a time mark is created before invoking `mc.monitoredMethod`. After the monitored method has finished, the elapsed time is calculated, converted into milliseconds and stored in the corresponding array. This benchmark is executed `totalCalls` times. Furthermore, the current iteration is printed every 100000 calls. Finally, the `print`-method converts the array containing the benchmark results into a string. This method is called from `BenchmarkMain` (see Section 5.5.2).

5.5 Platform-Specific Harness Code

Most of the harness code is equal for all supported KMP targets. However, not everything is reusable. Therefore, it was necessary to implement nearly the same functionality in different

ways. For example, `BenchmarkingThreadNano` uses JVM-specific code such as `MemoryMXBean` or `GarbageCollectorMXBean` in its original implementation. No equivalent Kotlin Multiplatform implementation is available for this code. Therefore, it was necessary to implement the JVM version with this specific code, while the other targets currently do not support this feature.

Kotlin Multiplatform provides the `expect/actual` mechanism for this purpose. A declaration in `commonMain` can define that a certain function, class, or property is expected to exist, while each platform source set provides the corresponding `actual` implementation. This makes it possible to call platform-dependent functionality from shared code without exposing the shared harness to the concrete implementation details of each target.

5.5.1 BenchmarkingThreadNano

`BenchmarkingThreadMilli` was already presented in Section 5.4.1. This section provides an overview of a more advanced benchmark. `BenchmarkingThreadNano` measures time with nanosecond granularity. Furthermore, contrary to the simpler `BenchmarkingThreadMilli`, this benchmark also keeps track of:

- Heap memory
- Garbage collection count

Since the libraries used to collect this information are only available for the JVM, the benchmarks for JS and Native are still implemented in the same way as `BenchmarkingThreadMilli`. In the future, it will be necessary to implement a more advanced benchmark for JS and Native when Kotlin provides libraries to gather this information (see Section 9.2).

```

1 actual class BenchmarkingThreadNano actual constructor(
2     private val mc: MonitoredClass,
3     private val totalCalls: Int,
4     private val methodTime: Long,
5     private val recursionDepth: Int) : BenchmarkingThread {
6
7     actual val executionTimes: LongArray = LongArray(totalCalls)
8     private val memory: MemoryMXBean = ManagementFactory.getMemoryMXBean()
9     private val usedHeapMemory: LongArray = LongArray(totalCalls)
10    private val gcCollectionCountDiffs: LongArray = LongArray(totalCalls)
11    private val collector: List<GarbageCollectorMXBean> =
12        ManagementFactory.getGarbageCollectorMXBeans()
13    //to prevent the (JIT) compiler from not calling the monitored method
14    private val mcTimings: LongArray = LongArray(totalCalls)
15
16    actual override fun print(index: Int, separatorString: String): String =
17        String.format(
18            "%d%s%d%s%d",
19            executionTimes[index], separatorString,
20            usedHeapMemory[index], separatorString,
21            gcCollectionCountDiffs[index]
22        )
23
24    actual override fun run() {
25        var lastGcCount = this.computeGcCollectionCount()
26        var currentGcCount: Long
27        val timeSource = TimeSource.Monotonic
28
29        for (i in 0..

```

```

31         val elapsed = start_ns.elapsedNow().inWholeNanoseconds
32         currentGcCount = this.computeGcCollectionCount()
33         executionTimes[i] = elapsed
34         usedHeapMemory[i] = memory.heapMemoryUsage.used
35         gcCollectionCountDiffs[i] = currentGcCount - lastGcCount
36         lastGcCount = currentGcCount
37         if ((i % 100000) == 0) {
38             println(i)
39         }
40     }
41 }
42
43 private fun computeGcCollectionCount(): Long {
44     var count: Long = 0
45     for (bean in this.collector) {
46         count += bean.collectionCount
47     }
48     return count
49 }
50 }

```

Listing 6: JVM implementation of `BenchmarkingThreadNano` using JVM management APIs for execution time, heap usage, and garbage-collection counts.

Listing 6 presents the complete `BenchmarkingThreadNano`. The real measured time is equal to the benchmark in Section 5.4.1 and is therefore not explained again. Since this benchmark uses nanosecond granularity, this variant is more suitable for short-running workloads.

The main additional part is the gathering of the garbage collection count and the used heap memory. Both values are obtained through JVM management interfaces. The `MemoryMXBean` interface is used to access the current heap memory usage, while the list of `GarbageCollectorMXBean` instances is used to query the number of garbage collection runs performed by the JVM.

Before the measurement loop starts, the current total garbage collection count is stored in `lastGcCount`. During each benchmark iteration, the worker first measures the execution time of the monitored method. After the method returns, the current garbage collection count is computed again. The difference between the current value and `lastGcCount` is stored in the `LongArray gcCollectionCountDiffs`. The values indicate whether one or more garbage collection events occurred between two benchmark iterations. Afterwards, `lastGcCount` is updated for the next iteration. Heap memory usage is recorded by accessing `memory.heapMemoryUsage.used`. This value represents the amount of heap memory currently in use is reported by the JVM at the time of measurement. It is stored for every benchmark operation in the `usedHeapMemory` array. Together with the execution time and the difference in garbage collection count, this allows the benchmark to relate run-time behavior to JVM memory activity. The method `computeGcCollectionCount` sums the collection counts of all available garbage collectors. This is necessary since a JVM may use more than one garbage collector component, for example separate collectors for young and old generations [22]. The resulting value is therefore not tied to a single collector but represents the aggregated number of collection events reported by all available `GarbageCollectorMXBean` instances.

It should be noted that the Java API allows `getCollectionCount()` to return -1 if the collection count is undefined for a collector [23].

Overall, this benchmark variant is more suitable for analyzing whether changes in execution time are accompanied by memory-related run-time effects. However, because these metrics rely on JVM-specific management APIs, equivalent implementations for JavaScript and Native are required in the future.

5.5.2 BenchmarkMain

BenchmarkMain is the central component for controlling the benchmark execution. After the platform-specific entry point has started the coroutine-based execution, control is delegated to BenchmarkMain. This component receives the normalized command-line arguments and uses them to construct the internal benchmark configuration.

```

1 actual object BenchmarkMain {
2     actual var sink: Sink? = null
3     actual var parameter: BenchmarkParameter = BenchmarkParameter(
4         totalCalls = 0, methodTime = 0, totalThreads = 1,
5         recursionDepth = 1, outputFile = "", quickstart = false,
6         forceTerminate = false, runnableClassname = null,
7         applicationClassname = null
8     )
9     actual var monitoredClass: MonitoredClass? = null
10    actual suspend fun run(args: Array<String>) {
11        parseAndInitializeArguments(args)
12        val benchmarkingThreads = arrayOfNulls<BenchmarkingThread>(parameter.totalThreads)
13        /*
14         // Not implemented for Native: this code depends on JVM Runtime memory APIs
15         // (and GC behavior) to allocate and measure free heap, which are not available
16         // or comparable on those targets.
17         if (!parameter.quickstart) {
18             for (l in 0..3) {
19                 val freeMemChunks = Runtime.getRuntime().freeMemory() shr 27
20                 val memSize = 128 * 1024 * 128
21                 for (j in 0..<freeMemChunks) {
22                     val grabMemory = LongArray(memSize)
23                     for (i in 0..<memSize) {
24                         grabMemory[i] = Clock.System.now().toEpochMilliseconds()
25                     }
26                 }
27             }
28             delay(5000)
29         }*/
30        val startTime = TimeSource.Monotonic.markNow()
31        try {
32            coroutineScope {
33                for (i in 0..<parameter.totalThreads) {
34                    val benchmarkingThread = monitoredClass?.let {
35                        BenchmarkingThreadNano(
36                            it,
37                            parameter.totalCalls,
38                            parameter.methodTime.toLong(),
39                            parameter.recursionDepth
40                        )
41                    }
42                    benchmarkingThreads[i] = benchmarkingThread
43                    if (benchmarkingThread != null) {
44                        launch {
45                            benchmarkingThread.run()
46                        }
47                    }
48                }
49            }
50        } catch (e: Exception) {
51            println("Benchmarking interrupted: ${e.message}")

```

```

52         exitProcess(1)
53     }
54     val totalTime = startTime.elapsedNow().inWholeSeconds
55     val out = sink ?: error("Output sink not initialized")
56     out.use { s ->
57         for (h in 0..

```

Listing 7: Platform-specific implementation of `BenchmarkMain` for benchmark initialization, worker execution, and result writing.

At the beginning, `BenchmarkMain` parses the parameters that are interpreted as benchmark configuration values (lines 11 and 96–99). These are required for the workload and other components in the program. After parsing, these values are stored in configuration objects that are used by the remaining harness components.

The second part of `BenchmarkMain` provides a warmup for the benchmark. Since this part uses JVM-specific libraries, it is only implemented in the JVM version (lines 17–29). An equivalent warmup implementation for JavaScript and Native was not implemented as part of this

thesis. This section is executed depending on the parameter `quickstart`. The code estimates the currently available heap memory in chunks of approximately 128 MiB and repeatedly allocates large `LongArray` instances of this size. Each array is fully written using values from `System.nanoTime()`, which forces the allocated memory to be touched rather than only reserved. The warmup is used to create controlled memory pressure before the benchmark measurements begin. This can trigger JVM-internal behavior such as heap expansion, memory page initialization, and garbage collection before the actual workload is measured.

The next part is the initialization (lines 77–94). Based on the parameters `applicationClassname` and `runnableClassname`, the appropriate workload implementation is selected and initialized. Here, the `ClassRegistry` is used (see Section 5.5.3) to determine the correct class because run-time reflection is not available in JS and Native. For the JVM, it is implemented using the standard run-time reflection.

After initialization, `BenchmarkMain` starts the benchmark execution using coroutines (lines 33–55). It coordinates the threads and waits until they have completed their operations. Once the execution of all threads has finished, it triggers the result writing logic (lines 57–68). In this part, the measured data, like times and relevant metadata are written to the configured output. The output is configured during parameter initialization. Finally, `BenchmarkMain` performs the required shutdown logic, such as closing output resources or terminating the process if the target requires explicit termination (lines 69–72).

In the end, command-line argument handling differs for the JavaScript target when the benchmark is executed with Node.js. Node.js automatically stores the path to the Node.js executable as the first element of `process.argv` and the path to the executed JavaScript file as the second element. These values are not benchmark parameters and are therefore removed before the remaining arguments are processed by the harness. For this purpose, the JavaScript-specific helper method shown in Listing 8 uses `process.argv.slice(2)` to return only the arguments explicitly passed to the benchmark. This method is executed right before the parameters are extracted.

```
1 @SuppressWarnings("UNCHECKED_CAST_TO_EXTERNAL_INTERFACE")
2     private fun nodeArgs(): Array<String> =
3         js("process.argv.slice(2)").unsafeCast<Array<String>>()
```

Listing 8: JavaScript-specific extraction of benchmark arguments from Node.js `process.argv`.

5.5.3 ClassRegistry

As shown in Section 5.5.2, the benchmark can be configured using a `runnableClassname` parameter at the start of the program. This parameter specifies which benchmark variation should be used for execution. In the original Java implementation, this problem can be solved using reflection. This means that the class name is resolved at run time, and the corresponding class is instantiated dynamically. However, KMP does not support reflection at run time across all supported targets. To solve this problem, the `ClassRegistry` was implemented. With this approach, classes do not need to be resolved dynamically. Instead, all supported implementations are registered under their fully qualified class names. At run time, the program uses the parameter as a key to retrieve the corresponding factory function from the registry.

Listing 9 presents how such a class or runnable class can be registered. Furthermore, it shows how `MonitoredFactory` and `BenchmarkingThreadFactory` are type aliases for `MonitoredClass` and `BenchmarkingThread`. Those are stored in mutable maps. For a runnable class, it is also necessary to pass all required parameters.

```

1 private val monitored = mutableMapOf<String, MonitoredFactory>()
2 private val runnable = mutableMapOf<String, BenchmarkingThreadFactory>()
3 init {
4     registerMonitored("application.MonitoredClassSimple") {
5         MonitoredClassSimple()
6     }
7     registerRunnable("benchmark.BenchmarkingThreadMilli") { mc, totalCalls, methodTime,
8         recursionDepth ->
9         BenchmarkingThreadMilli(mc, totalCalls, methodTime, recursionDepth)
10 }

```

Listing 9: Registration of workload and benchmark-worker factories in `ClassRegistry`.

Listing 10 presents the two lookup functions used to create an instance of a registered class. `createMonitored` uses the provided class name to retrieve and invoke a factory from the `monitored` map, while `createRunnable` performs the corresponding lookup for benchmark workers, forwards the required run-time parameters, and invokes the runnable.

```

1 fun createMonitored(name: String?): MonitoredClass? =
2     name?.let { monitored[it]?.invoke() }
3
4 fun createRunnable(
5     name: String?,
6     mc: MonitoredClass,
7     totalCalls: Int,
8     methodTime: Long,
9     recursionDepth: Int
10 ): BenchmarkingThread? =
11     name?.let {
12         runnable[it]?.invoke(mc, totalCalls, methodTime, recursionDepth)
13     }

```

Listing 10: Creation of registered workload and benchmark-worker instances in `ClassRegistry`.

Thus, when a class name is provided as a benchmark parameter, the registry performs a lookup in the corresponding map and invokes the stored factory function. If no matching entry exists, the lookup returns `null`, allowing the harness to handle invalid configurations explicitly.

This design makes class selection deterministic and platform-independent. It also keeps the shared harness code compatible with Kotlin Multiplatform, since object creation is expressed through regular function calls instead of run-time class loading.

5.6 Kotlin Benchmark Entry Points

In order to launch the application on the various platforms, platform-specific entry points are required. Two different entry points variants are necessary because coroutines are used in the implementation. Therefore, the JS target must be launched using an asynchronous execution model.

Listing 11 presents the Entry points for **JVM** and **Native**.

```

1 fun main(args: Array<String>) = runBlocking {
2     BenchmarkMain.run(args)
3 }

```

Listing 11: Coroutine-based benchmark entry point for JVM and Native using `runBlocking`.

For the JVM and Native targets, the benchmark can be started using `runBlocking`. This coroutine builder creates a coroutine scope and blocks the current thread until the code executed within this scope, including its child coroutines, has completed. This behavior is suitable for command-line executions because the benchmark process should remain active until the complete benchmark run has finished. Without this blocking behavior, the main function could return before the coroutine-based benchmark execution has completed. Therefore, `runBlocking` acts as a bridge between the synchronous platform entry point and the coroutine-based implementation of the benchmark.

Inside this coroutine scope, the platform-specific main function simply forwards the received command-line arguments to the central harness component `BenchmarkMain` by calling `BenchmarkMain.run(args)`. `BenchmarkMain` is responsible for parsing the parameters, initializing the benchmark configuration, coordinating the benchmark workers, and handling the results for the respective target.

Listing 12 presents the Entry points for **JS**.

```

1 @OptIn(DelicateCoroutinesApi::class)
2 fun main(args: Array<String>) {
3     GlobalScope.promise {
4         BenchmarkMain.run(args)
5     }
6 }

```

Listing 12: Promise-based JavaScript benchmark entry point using `GlobalScope.promise`.

JavaScript is based on an event loop, and asynchronous operations are commonly represented by `Promise` objects. Therefore, the benchmark execution is started using `GlobalScope.promise`. This creates a coroutine and exposes its execution as a JavaScript-compatible `Promise` [13].

The use of `GlobalScope` means that the benchmark coroutine is launched as a top-level coroutine that is not bound to a parent job or a structured coroutine scope. This is why `GlobalScope` is marked as a delicate API: coroutines launched in this scope are not automatically canceled along with a surrounding scope and cannot be managed through a common parent job [10].

6 Benchmark Execution Integration

This section provides an overview of the current MooBench structure and how the new benchmark is integrated into it. First, the role of the existing infrastructure will be explained. Second, the additions required to provide the new benchmark are described. Then, the build and execution of the artifacts are explained. Finally, the environment variables are presented and their use in configuring the benchmark.

6.1 Role of the Existing MooBench Infrastructure

Since one goal of this thesis is to integrate the Kotlin benchmark with *k-perf* into the existing MooBench infrastructure, the implementation does not introduce a completely independent benchmark environment.

The current structure separates the framework-specific benchmark execution logic, e.g., starting Java Moobench with Kieker instrumentation, from generic scripts for initialization, setup, and result analysis. The framework-specific directories, therefore, contain the benchmark start script, as well as configurations, labels, and the actual target-specific execution script. Having this framework-specific separation provides an easy way to add new frameworks to the MooBench repository.

For **MooBench X Kotlin**, a new framework-specific integration has been added for *k-perf*. This integration follows the existing MooBench directory structure and provides the execution logic required for the Kotlin Multiplatform artifacts. The setup script builds the artifacts needed for the benchmark, while the execution scripts start the corresponding artifacts with the configured benchmark parameters. The generated measurement files are written into a newly created results directory within the framework-specific directory.

Reusing the existing infrastructure has several advantages. First, it reduces the number of new scripts that need to be implemented. Furthermore, it allows existing analysis tools to be reused. Finally, it is possible to run many different benchmarks, and new benchmarks can be integrated into the infrastructure very quickly.

6.2 Added Infrastructure for MooBench X Kotlin

To integrate the Kotlin *k-perf* benchmark to the existing MooBench infrastructure, a new framework-specific directory, **frameworks/k-perf-kotlin**, was added. Similar to the existing MooBench integrations, this directory contains the scripts and configuration files required to initialize and execute the framework-specific benchmark. Figure 7 shows the newly added files. The **measure.sh** script serves as the entry point for starting the *k-perf* benchmark and connects the framework-specific integration to the generic MooBench execution infrastructure. The **functions.sh** script contains the commands required to execute the generated Kotlin artifacts for the supported target platforms. The **config.rc** file provides framework-specific configuration values, while **labels.sh** defines the labels used for the benchmark output and subsequent evaluation.

Furthermore, Figure 8 gives an overview of the structure of MooBench and the following paragraphs give an overview of the changes made to the original project.

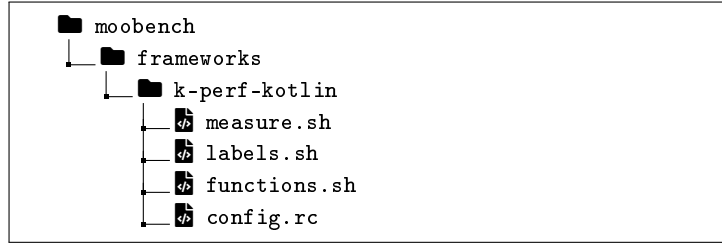


Figure 7: Framework-specific files added for the *k-perf* integration in MooBench.



Figure 8: Root directory structure of the MooBench project.

gradle.properties The file was newly added. This file is used to define Gradle and project-specific configuration properties outside the build scripts. It contains the property `kotlin.native.cacheKind.linuxX64=none`, which disables the Kotlin/Native compiler cache for the `linuxX64` target. This avoids reusing cached Native compilation artifacts and ensures that the Linux Native artifact is rebuilt without relying on previously generated compiler caches. In the context of this benchmark, this is useful to reduce cache-related build effects when generating plain and instrumented Native artifacts [17].

Furthermore, the integration of *k-perf* required not only additional infrastructure but also modifications to existing MooBench files to make the benchmark compatible with the new Kotlin-based implementation. The following modifications were made:

build.gradle: Listing 13 shows the modification of the root `build.gradle` file. The repository configuration is applied to all subprojects in the repository so that dependencies can be resolved

from Maven Central. In the original MooBench structure, all projects built by default are Java-based and can therefore use the Gradle `java` plugin. However, the new `:tools:SuT-kotlin` module is a Kotlin Multiplatform project and requires its own Kotlin-specific Gradle configuration. For this reason, the Java plugin is only applied if the current subproject is not `:tools:SuT-kotlin`.

```

1 subprojects {
2     repositories {
3         mavenCentral()
4         maven { url = "https://oss.sonatype.org/content/repositories/snapshots/" }
5     }
6
7     if (project.path != ':tools:SuT-kotlin') {
8         apply plugin: 'java'
9     }
10 }

```

Listing 13: Modification of the root build.gradle file.

setup.sh: It was also necessary to expand the setup script by adding new code for the Kotlin benchmark. This is explained in Section 6.3.

settings.gradle: This file contains all projects that are built by the setup script. This project should only be built, when the flag `withKotlin` is passed at the start of the build process.

```

1 if (providers.gradleProperty("withKotlin").present) {
2     include ':tools:SuT-kotlin'
3 }

```

Listing 14: Registration of the Kotlin benchmark project in settings.gradle.

.gitignore: Adding a new framework to the benchmark also creates additional files and directories during both the benchmark run as well as the build of that framework. These files are not relevant for the repository because they are newly created during every run or build. Therefore, they should not be committed to the repository. To insure this, several Kotlin/*k-perf* specific lines were added to the `.gitignore` file.

```

1 /.kotlin/
2 # Do not store compile results
3 # Ignore every compile-results directory ...
4 compile-results/
5 # ... except /tools/compile-results/
6 !/tools/compile-results/
7 !/tools/compile-results/**
8 /tools/compile-results/build/
9 # Do not store measurement results
10 frameworks/k-perf-Kotlin/results-k-perf-Kotlin/
11 # Kotlin specific
12 frameworks/k-perf-Kotlin/*.txt
13 kotlin-js-store/
14 DEBUG.txt

```

Listing 15: gitignore rules added for generated Kotlin and *k-perf* benchmark files.

Listing 15 presents the newly added lines. This means a `.kotlin` directory, the results directory, a debug file, a JS-specific directory, and *k-perf* trace files will be excluded from the repository.

6.3 Setup and Build Benchmark for k-perf

Before any benchmark can be started, the entire MooBench application must be compiled and installed. For this purpose, the `setup.sh` file has to be called from the repository root directory. With this Bash script, both the original JVM-based MooBench projects and *MooBench X Kotlin* are built. The script checks whether all required scripts are available and then builds all underlying subprojects.

Listing 16 shows the commands that have been added to the current script.

```
1 ./gradlew build
2
3 # Optionally build Kotlin benchmark artifacts
4
5 if [[ "$1" == "withKotlin" ]]; then
6     mkdir -p frameworks/k-perf-kotlin/build/lib/deps
7     mkdir -p frameworks/k-perf-kotlin/build/jsPlain/
8     mkdir -p frameworks/k-perf-kotlin/build/jsInstrumented/
9     mkdir -p frameworks/k-perf-kotlin/build/lib/
10
11 # plain
12 ./gradlew build :tools:SuT-kotlin:build -PwithKotlin -PkperfEnabled=false
13 cp tools/SuT-kotlin/build/jvmRuntimeClasspath/*.jar
14   "frameworks/k-perf-kotlin/build/lib/deps/"
15 cp "tools/SuT-kotlin/build/libs/kotlin-plain.jar" "frameworks/k-perf-kotlin/build/lib/"
16 cp -r build/js/packages/kotlin-plain/kotlin/. \
17   frameworks/k-perf-kotlin/build/jsPlain/
18
19 # instrumented
20 ./gradlew build :tools:SuT-kotlin:build -PwithKotlin -PkperfEnabled=true
21   -PkPerfMethods="application.MonitoredClassSimple.monitoredMethod"
22 cp "tools/SuT-kotlin/build/libs/kotlin-instrumented.jar"
23   "frameworks/k-perf-kotlin/build/lib/"
24 cp -r build/js/packages/kotlin-instrumented/kotlin/. \
25   frameworks/k-perf-kotlin/build/jsInstrumented/
26
27 cp -r tools/SuT-kotlin/build/bin/. \
28   frameworks/k-perf-kotlin/build/bin/
29 fi
```

Listing 16: Added code for the *MooBench X Kotlin* build in `setup.sh`.

First, the default subprojects of MooBench are built. The script then checks whether it was started with the flag `withKotlin`. If the flag is not set, none of the Kotlin project are built. Otherwise, the directories needed for the different targets are created. Afterwards, *MooBench X Kotlin* is built without *k-perf* instrumentation, which means no method is instrumented at compile-time. The directory `frameworks/k-perf-kotlin/build/lib/deps` is used to store the runtime dependencies of the JVM benchmark. These dependencies are copied from the `jvmRuntimeClasspath` of the Kotlin benchmark project. This is necessary because the generated benchmark JAR does not contain all required dependency classes itself. By storing the dependencies separately, the benchmark can later be started with the correct classpath. Otherwise, the benchmark could not be started. Furthermore, the artifacts must be copied into the right directories to prevent all plain artifacts from being deleted during the second build. For Native, one copy is sufficient.

In the next step, the instrumented version is built by setting `kperfEnabled` to `true`. Additionally, the property `kPerfMethods` specifies which method should be instrumented. In this case, only

`application.MonitoredClassSimple.monitoredMethod` is passed to the compiler plugin. The other instrumentable method is `application.MonitoredClassThreaded.monitoredMethod`. During this build, *k-perf* instruments the specified method at compile time. The instrumented JAR file and the generated JavaScript files are copied then to the *k-perf* framework directory.

Finally, the native binaries generated by the Kotlin benchmark project are copied to `frameworks/k-perf-kotlin/build/bin/`. This makes the generated JVM, JavaScript, and Native artifacts available in a common location under `frameworks/k-perf-kotlin/build/`. The setup script therefore prepares both the plain and the instrumented benchmark variants and organizes them in a structure that can be used by the subsequent benchmark execution scripts.



Figure 9: Generated plain and instrumented benchmark artifacts after the *k-perf* build process.

After both build variants have been generated, Figure 9 shows the resulting artifact structure below `frameworks/k-perf-kotlin/build`. The `lib` directory contains the plain and instrumented JVM JAR files together with their required run-time dependencies. The JavaScript artifacts are stored separately in `jsPlain` and `jsInstrumented`. Native executables are located below `bin` in target-specific directories such as `linuxX64` and `mingwX64`.

This structure provides separate plain and instrumented artifacts for each supported target platform. These artifacts are subsequently used by the framework-specific execution logic described in Section 6.4.

6.4 Execution of Platform-Specific Artifacts

After the plain and instrumented artifacts have been built, the next step is to run the benchmark and examine how the different executions are performed. Post running the setup script described Section 6.5 the `measure.sh` script can be executed.

A standard benchmark execution consists of running the plain and instrumented artifact with the same parameters. This ensures that both executions perform the same number of workload invocations and use the same recursion depth and busy waiting time. The only relevant difference should be the presence or absence of *k-perf* instrumentation.

6.4.1 Starts of the Artifacts

A *k-perf* benchmark run is started by executing the generated benchmark artifact with a specific set of parameters. These parameters define the workload configuration and the execution model of the harness. Those parameters can be configured through environment variables (for further information look Section 6.4.2). The most important parameters are the number of calls, the workload using busy waiting, the recursion depth, the number of threads, and the output file. The execution commands in the following listings are excerpts and do not show the complete start script.

```
1 MOOBENCH_JAR_PLAIN="${BASE_DIR}/build/lib/kotlin-plain.jar :  
2   ${BASE_DIR}/build/lib/deps/*"  
3 #start  
4 "${JAVA_BIN}" -cp "${MOOBENCH_JAR_PLAIN}" benchmark.MainKt \  
5   --output-filename "${RAWFN}-${i}-${RECURSION_DEPTH}-${k}.csv" \  
6   --total-calls "${TOTAL_NUM_OF_CALLS}" \  
7   --method-time "${METHOD_TIME}" \  
8   --total-threads "${THREADS}" \  
9   --recursion-depth "${RECURSION_DEPTH}" \  
10  ${MORE_PARAMS} \  
11  &> "${RESULTS_DIR}/output-${i}-${RECURSION_DEPTH}-${k}.txt"
```

Listing 17: Execution of the plain JVM benchmark artifact with the configured MooBench parameters.

Listing 17 shows how the plain JVM artifact is started. More specifically, the variable `MOOBENCH_JAR_PLAIN` defines the classpath used for the JVM execution. It contains the runnable JAR file as well as the required dependencies. Furthermore, `JAVA_BIN` contains the path to the Java command-line binary. `benchmark.MainKt` represents the JVM entry point as described in Section 5.6. The following arguments are forwarded to the benchmark harness and define the configuration. In addition, optional parameters can be appended via `MORE_PARAMS`. The final redirection writes both standard output and standard error to a separate console output file in `RESULTS_DIR`. This file is useful for debugging and documenting the execution of each individual benchmark run, while the actual measurement values are written to the CSV file configured through `-output-filename`.

```
1 MOOBENCH_NATIVE_PLAIN_BIN="${BASE_DIR}/build/bin/${NATIVE_TARGET}/  
2   releaseExecutable/kotlin-plain.${NATIVE_EXT}"  
3 #start  
4 "${MOOBENCH_NATIVE_PLAIN_BIN}" \  
5   --output-filename "${RAWFN}-${i}-${RECURSION_DEPTH}-${k}.csv" \  
6   --total-calls "${TOTAL_NUM_OF_CALLS}" \  
7   --method-time "${METHOD_TIME}"
```

```
7 ... more parameters ...
```

Listing 18: Execution of the plain Native benchmark artifact using the selected target binary.

Listing 18 presents the start of the Native benchmark artifact. The parameters are identical to those used in Listing 17 and are therefore not explained again. The Native benchmark can be executed on Linux, Windows, and Apple machines. For this reason, the benchmark script uses a dynamic Native target and a platform-dependent executable extension. The variable `MOOBENCH_NATIVE_PLAIN_BIN` contains the path to the generated Native executable. This path depends on the selected `NATIVE_TARGET`, for example a Linux or Windows target directory, and on the corresponding file extension stored in `NATIVE_EXT`. On Linux, the generated executable usually does not require an additional extension, whereas on Windows, the executable uses the `.exe` extension.

```
1 NODE_BIN="${NODE_BIN:-node}"
2 NODE_ARGS="${NODE_ARGS:-}"
3 MOOBENCH_JS_PLAIN_BIN="${BASE_DIR}/build/jsPlain/kotlin-plain.js"
4 #start
5     "${NODE_BIN}" ${NODE_ARGS} "${MOOBENCH_JS_PLAIN_BIN}" \
6         --output-filename "${RAWFN}-${i}-${RECURSION_DEPTH}-${k}.csv" \
7         --total-calls "${TOTAL_NUM_OF_CALLS}" \
8         ... more parameters ...
```

Listing 19: Execution of the plain JavaScript benchmark artifact with Node.js.

Finally, Listing 19 provides an overview of the JS artifact execution. `NODE_BIN` is the command used to start the Node.js runtime. The variable `NODE_ARGS` contains optional arguments that can be passed directly to Node.js before the JavaScript benchmark file is executed. In contrast to the JVM target, the JavaScript artifact is not executed through a classpath. Instead, the generated JS file is passed as an input file to the Node.js runtime. The benchmark parameters are then appended after the JavaScript file.

6.4.2 Run-time Configuration

The parameters required for benchmark execution are initialized by the generic MooBench script `init.sh` at the start of a benchmark run. This script initializes all unset parameters with default values. These values define the workload, the number of benchmark executions, and additional run-time behavior. After `init.sh` has been executed, the benchmark scripts can access these values when starting the artifact.

There are two ways to change the parameter values. First, the default values can be modified in the `init.sh` script. Second, individual values can be overridden when starting the artifact (see Section 6.4). This makes it possible to modify the benchmark run without changing the benchmark itself. The relevant default parameters are shown in Table 1.

The parameter `TOTAL_NUM_OF_CALLS` does not denote the total number of individual recursive method calls. Instead, it defines how often the complete recursive workload is invoked during one benchmark run. Since one workload invocation executes `monitoredMethod` recursively up to the configured `RECURSION_DEPTH`, the number of executed method calls per benchmark run is calculated as follows:

$$N_{\text{calls,run}} = \text{TOTAL_NUM_OF_CALLS} \cdot \text{RECURSION_DEPTH}. \quad (1)$$

Parameter	Default value	Description
SLEEP_TIME	30	Waiting time between benchmark runs [seconds]
NUM_OF_LOOPS	10	Number of repeated benchmark executions
THREADS	1	Number of benchmark workers
RECURSION_DEPTH	10	Maximum depth of the recursive workload
TOTAL_NUM_OF_CALLS	2000000	Number of workload invocations
METHOD_TIME	0	Busy waiting time of the innermost method call
DEBUG	false	Enables or disables debug output

Table 1: Default run-time configuration parameters

With the default values shown in Table 1, this results in:

$$N_{\text{calls,run}} = 2,000,000 \cdot 10 = 20,000,000. \quad (2)$$

Thus, one benchmark run with the default configuration executes 20 million monitored method calls. Since the benchmark is repeated NUM_OF_LOOPS times, the total number of method calls per configuration is:

$$N_{\text{calls,total}} = \text{NUM_OF_LOOPS} \cdot \text{TOTAL_NUM_OF_CALLS} \cdot \text{RECURSION_DEPTH}. \quad (3)$$

For the default configuration, this yields:

$$N_{\text{calls,total}} = 10 \cdot 2,000,000 \cdot 10 = 200,000,000. \quad (4)$$

Consequently, TOTAL_NUM_OF_CALLS controls the number of workload invocations, while the actual number of monitored method calls also depends on the configured recursion depth.

6.5 Setup, Build and Running Benchmarks

Section 6 explains the technical aspects of running a benchmark and the modifications required for the Kotlin *k-perf* benchmark. To start the right benchmark, only a few commands need to be executed on the machine's command line.

```

1 ./setup.sh withKotlin
2 cd frameworks/k-Perf-Kotlin/
3 ./measure.sh

```

Listing 20: Commands for building and starting the *k-perf* benchmark.

Listing 20 provides the required code to execute the Kotlin benchmark. First, the `setup.sh` script needs to be executed in order to provide all necessary artifacts and additional dependencies. Then, the correct directory needs to be accessed with the second command. Finally, the entire Kotlin *k-perf* benchmark can be started with the last command. At the end, the results are displayed as a table in the command line.

6.6 Results of k-perf

During the benchmark run, several output files are created. First, the benchmark harness writes `.csv` files containing the measured execution times and, depending on the selected worker implementation, additional run-time information such as heap usage or garbage collection counts. These files are used to analyze the benchmark measurements. Second, the benchmark start scripts redirect the console output of each run to separate `.txt` files. These files contain all passed parameters, as well as a step counter that writes an entry every 100,000 calls and the used memory for the JVM. Error messages may also be written to these files. Third, when the instrumented artifact is executed, *k-perf* creates trace files as described in Section 2.3. These files contain method entry and exit events, as well as the execution duration of the method.

At the end of the benchmark script, the evaluated results of all runs are displayed in a table (see Section 7.3). These values are calculated using an R script provided by the original `MooBench`. Furthermore, all results are compressed into a `.zip` file at the end.

An overview of the resulting files is shown in Figure 10. The directory contains the measurement and configuration output generated during the benchmark runs.

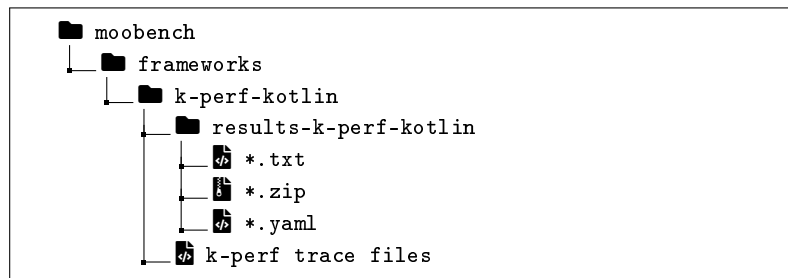


Figure 10: Generated benchmark result files after executing the *k-perf* benchmark.

7 Usage and Evaluation

This section provides an overview of the usage and evaluation of *MooBench X Kotlin*. First, the prerequisites for running the benchmark are explained. Then, the results of k-perf will be evaluated and interpreted.

7.1 Prerequisites

Several tools are required to build and execute *MooBench X Kotlin*. Since the benchmark is implemented as a Kotlin Multiplatform project, the toolchain must support JVM, JavaScript, and Native targets. More specifically, the following prerequisites are required:

- Recent R installation
- Recent Java Development Kit (JDK) installation
- Node.js environment: For running *MooBench X Kotlin* as JS target
- Kotlin/Native toolchain: For running *MooBench X Kotlin* as Native target
- Linux system: For executing the benchmark scripts

KMP projects use a Gradle Wrapper and do not require a globally installed version of Gradle. When the wrapper script is executed for the first time, it downloads the configured Gradle distribution and runs the build with that version [4].

R can be used for benchmark analysis. It provides the **Rscript** frontend for executing R scripts from the command line, making it suitable for automated post-processing of measurement data [27]. In this context, R is used to calculate:

- Mean
- Standard deviation
- 95% confidence interval
- First quartile
- Second quartile
- Third quartile
- Maximum
- Minimum

Additional tools are required to execute other MooBench frameworks. For further information, see the MooBench Github repository.

The **statistics.r** script processes the measurement data generated during the benchmark runs and calculates the statistical values presented above. In the extended version, the script can additionally remove extreme outliers separately for each benchmark run. If enabled, the smallest

and largest 0.5% of the remaining measurements are discarded after the warm-up values have been removed. Furthermore, the script takes the configured recursion depth into account to normalize the measured run-time of a complete workload invocation to the run-time per individual `monitoredMethod` call. The script also reports how many workload invocations and method calls are discarded during warm-up and optional outlier removal, and how many measurements are finally used for the statistical evaluation.

Furthermore, the script reports run-time values in the selected time unit both per complete workload invocation and per individual `monitoredMethod` call. For the latter, the measured duration of a complete recursive workload invocation is normalized by the configured recursion depth.

7.2 How to Interpret the Results

The benchmark results can be interpreted in two different ways. First, the absolute overhead of the instrumented run can be calculated. Second, it is also possible to calculate the relative overhead using the non-instrumented benchmark execution as the baseline.

The formulas are as follows:

$$overhead_{abs,p} = runtime_{instrumented,p} - runtime_{plain,p} \quad (5)$$

$$overhead_{rel,p} = \frac{runtime_{instrumented,p} - runtime_{plain,p}}{runtime_{plain,p}} \cdot 100 \quad (6)$$

where p denotes the target platform, e.g., JVM, JavaScript, or Native.

These formulas can only be used if the same parameters are applied for both executions. Otherwise, the results are not comparable. Furthermore, calculating and directly comparing overheads across different target platforms is not methodologically equivalent to comparing plain and instrumented variants within the same platform. For example, coroutine execution on the JVM can be mapped to Java threads, whereas the JavaScript target is integrated into an event-loop-based execution model. Kotlin/Native again uses its own runtime and memory management model [8, 19, 12].

System load is an important factor regarding the comparability of benchmarks. Background processes, CPU frequency scaling, operating system scheduling, and other concurrent workloads can affect the measured run-time. This is especially relevant for micro-benchmarks, where small timing differences are significant. For this reason, benchmark runs should be performed on an otherwise quiet system, and the environment should be documented as part of the evaluation.

7.3 Evaluation of k-perf

To measure the overhead of *k-perf*, the following experiment was executed on a machine running Debian GNU/Linux 13 (trixie) with Linux kernel 6.12.86+deb13-amd64. The system was equipped with an AMD Ryzen 9 9950X processor, 64 GB of DDR5-6000 memory, and a 1 TB SSD. The software versions used for the experiment were Kotlin 2.3.0, Eclipse Temurin Java 25.0.3+9.0.LTS, and Node.js 24.18.0.

Parameter	Value
NUM_OF_LOOPS	10
TOTAL_NUM_OF_CALLS	200000
METHOD_TIME	0 ns
THREADS	1
RECURSION_DEPTH	100

Table 2: Benchmark configuration used for the evaluation of *k-perf*.

The benchmark was executed using the configuration shown in Table 2.

Since each workload invocation executes `monitoredMethod` recursively according to the configured recursion depth, this configuration results in 20,000,000 monitored method calls per benchmark run. The first 10,000,000 method calls are excluded as warm-up. Additionally, the fastest and slowest 0.5% of the remaining measurements are removed as extreme outliers.

The resulting run times are normalized by the number of monitored method calls in order to obtain the run time and instrumentation overhead per individual method call. The values reported in Table 3 represent the mean run time together with the standard deviation. To provide a direct comparison between the plain and instrumented variants, both the absolute overhead and the relative slowdown are reported.

Platform	Plain run time [ns]	Instrumented run time [ns]	Absolute overhead [ns]	Slowdown [$\times$]
JVM	0.99 ± 0.06	47.46 ± 4.32	46.47	48.1
Native	0.75 ± 0.06	171.54 ± 47.10	170.78	227.8
JavaScript	6.14 ± 1.29	898.65 ± 81.65	892.51	146.4

Table 3: Absolute overhead and slowdown introduced by *k-perf* per monitored method call.

The JVM shows the lowest absolute overhead, with 46.47 ns per monitored method call and a slowdown of $48.1\times$ compared with the plain execution. Native on Linux shows the highest relative slowdown at $227.8\times$, corresponding to an absolute overhead of 170.78 ns per method call. JavaScript exhibits the highest absolute overhead at 892.51 ns per method call, with a slowdown of $146.4\times$.

Overall, the measurements show that *k-perf* introduces run-time overhead on all evaluated targets and that both the absolute overhead and the relative slowdown differ considerably between the target platforms.

8 Related Work

This section discusses research related to the work presented in this thesis. First, two studies regarding MooBench and its extensions are introduced. Subsequently, research on reduction of observability overhead, as well as the influence of execution environments, is presented.

8.1 MooBench in Previous Work

MooBench has previously been used to evaluate and compare the run-time overhead introduced by different monitoring and observability frameworks [30, 35]. Reichelt et al. [30] extended MooBench with support for a new framework, OpenTelemetry, and compared its overhead with Kieker. The evaluation of the work considered several configurations, including deactivated probes, local serialization, and external data processing. Overall, the results showed that Kieker introduced slightly less overhead than OpenTelemetry when processing traces. Furthermore, the overhead increased with the call tree depth and, therefore, with the number of instrumented method executions. The study also shows that the relative performance of the evaluated frameworks can depend on the underlying hardware, such as a Raspberry Pi or conventional PC.

Yang et al. [35] implemented the new framework Cloudprofiler for MooBench, which is a profiler written in C++. This new profiler supports native and JVM-based applications [35]. To reduce the overhead on the monitored application, Cloudprofiler moves parts of the data processing outside the target process and uses memory buffers and separate compression threads. The authors integrated several Cloudprofiler handlers into MooBench and evaluated the effects of buffering and compression. Their results showed that the buffered handler using Zstandard compression achieved an average execution time of 2.28 μ s, making it 6.15 times faster than the version without a buffered and compression handler.

These studies demonstrate that MooBench can be extended by integrating additional monitoring frameworks and evaluate different instrumentation methods and data-processing configurations. However, the evaluated configurations primarily target JVM-based applications as well as native profilers connected to JVM workloads. In contrast, this thesis adapts MooBench for Kotlin Multiplatform and tests the instrumentation performed by the *k-perf* Kotlin compiler plugin across the JVM, Native, and JavaScript targets.

8.2 Observability Overhead Benchmarking

In addition to comparisons between monitoring frameworks, previous work has investigated how observability overhead can be reduced [31, 35]. For example, Reichelt et al. [31] analyzed the sources of overhead in Kieker and proposed four optimizations: replacing AspectJ instrumentation with source-code instrumentation, reducing the amount of recorded data, using a different queue implementation, and aggregating measurement data before writing it. The effects of these optimizations were evaluated using the MooBench observability overhead micro-Benchmark. The results of this work show that the execution duration could be reduced from 4.77 μ s to 0.39 μ s per method invocation on average when all optimizations were combined.

While the study in the previous paragraph focuses on reducing monitoring overhead, Reichelt et al. [29] also examined the influence of the execution environment on the stability of MooBench

measurements. The study compared MooBench executions in a GitHub Actions cloud environment with three bare-metal environments. The results showed that bare-metal systems generally provided lower run-times and lower variability for multithreaded executions. With Github Actions, performance changes of up to 4.41% were detected when using sequential workloads only. For further information on the behavior of different environments, the authors also performed measurements using 2, 4, 8, and 12 threads. These measurements show a more rapid increase in run-time with an increasing number of threads using Github Actions than with a desktop system.

The findings of these studies are relevant to the evaluation conducted in this thesis. First, the comparison of different instrumentation points shows that the point at which probes are inserted can affect the resulting overhead. *k-perf* follows a different approach by inserting probes into Kotlin’s Intermediate Representation during compilation. Second, the observed influence of the execution environment emphasizes the need for repeated executions, warm up phases, statistical analysis, as well as a system specifically designed for benchmarks. Accordingly, the benchmark implementation developed in this thesis executes multiple iterations and reports statistical measures such as the mean, median, standard deviation, quartiles, and confidence intervals.

9 Limitations and Future Work

This section points out some problems that occurred during the implementation of `MooBench X Kotlin`. Furthermore, future adaptations and improvements are discussed, as well as additional benefits that these improvements could provide to `MooBench X Kotlin`.

9.1 Limitations

During the implementation of `MooBench X Kotlin`, several limitations were encountered. These limitations mainly result from differences between the supported target platforms and from functionality that is specific to the Java ecosystem and for which no equivalent KMP library is currently available.

Java-specific libraries: Some libraries used by the original JVM-based `MooBench` implementation are only available for the JVM and do not provide KMP alternatives. For example, the `ThreadMXBean` API can only be used to obtain information about thread execution but is currently only used in the JVM version.

Reflection at run time: The original `MooBench` uses reflection to start a implementation specified at the start of an artifact. Since KMP does not support reflection at run time across all targets, for Native and JavaScript, it was necessary to implement a workaround as explained in Section 5.5.3. For the JVM, however, reflection is still implemented in the original manner.

Comparability across targets: The benchmark results of all targets are only comparable to a limited extent. Since each target uses its own runtime environment, memory management system, and execution model, the time measurements differ between platforms. Therefore, only the platforms themselves can be compared in a controlled and reproducible manner.

9.2 Future Work

Although the main goal of developing a benchmark infrastructure for `k-perf` has been achieved, several areas for future improvement remain.

Memory usage measurement: Future work could extend the multithreaded benchmark for the Native and JavaScript targets by implementing memory monitoring similar to that of the JVM implementation (see Section 5.5.1). This would allow the benchmark to measure the memory consumption of instrumented and non-instrumented applications, which would additionally enable the evaluation of the memory overhead introduced by the `k-perf` compiler plugin.

Automated evaluation and visualization: More advanced automated processing and presentation of the benchmark results could be an interesting improvement for the future. This could include automatically generated tables or diagrams of the measurements. For example, after each push to the `k-perf` repository, an automated benchmark could be executed and the calculated values could be visualized in a diagram covering all commits.

Integration into continuous integration environments: The `k-perf` benchmark could be integrated into a continuous integration pipeline to execute performance measurements automatically after changes to `k-perf`. With the previously explained future improvement performance regressions could be detected and potential code problems could therefore be discovered very quickly.

10 Conclusion

This thesis introduced *MooBench X Kotlin*, a Kotlin Multiplatform adaptation of the MooBench benchmark infrastructure for evaluating the run-time overhead of instrumentation tools such as *k-perf*. The original MooBench infrastructure primarily targets Java-based workloads and run-time or load-time instrumentation. In contrast, *MooBench X Kotlin* extends the benchmark pipeline to generate and execute plain and instrumented artifacts for JVM, JavaScript, and Native, thereby also supporting compile-time instrumentation at the Kotlin Intermediate Representation (IR) level. The *k-perf* compiler plugin was used as a concrete example to evaluate this extended benchmarking approach.

The central MooBench workload and harness were ported to KMP while preserving their main characteristics, such as the recursive workload structure, configurable benchmark parameters, and the busy-wait-based method execution time. Platform-independent functionality was shared across all three platforms, while platform-specific implementations were used where necessary. Using this approach, it was possible to generate instrumented and non-instrumented artifacts for the JVM, JavaScript and Native targets.

The original MooBench infrastructure was reused for the new *k-perf* benchmark and the additional benchmark components were integrated into the existing infrastructure. Furthermore, the new framework *k-perf* was evaluated using the MooBench methodology. For all targets, the absolute and relative overheads were calculated and analyzed.

Overall, this thesis presents an intuitive and practical approach to measure Kotlin-specific instrumentation. The resulting benchmark infrastructure enables a reproducible way to evaluate the performance overhead introduced by the observability tool *k-perf* across multiple Kotlin targets. Future work could extend the benchmark with memory usage measurements, automated result evaluation and visualization, as well as integration into CI/CD pipelines for the continuous detection of performance regressions.

List of Tables

1	Default run-time configuration parameters	32
2	Benchmark configuration used for the evaluation of <i>k-perf</i>	36
3	Absolute overhead and slowdown introduced by <i>k-perf</i> per monitored method call.	36

List of Figures

1	Simplified overview of the K2 compiler.	4
2	Architecture of the original MooBench with run-time bytecode instrumentation.	7
3	Architecture of <i>MooBench X Kotlin</i> with compile-time IR instrumentation for multiple target platforms.	7
4	Workflow of <i>MooBench X Kotlin</i> from artifact generation to benchmark execution and result analysis.	8
5	Recursive MooBench workload with a configurable recursion depth and busy-waiting.	10
6	Overview of the benchmark harness and its interaction with the workload.	14
7	Framework-specific files added for the <i>k-perf</i> integration in MooBench.	26
8	Root directory structure of the MooBench project.	26
9	Generated plain and instrumented benchmark artifacts after the <i>k-perf</i> build process.	29
10	Generated benchmark result files after executing the <i>k-perf</i> benchmark.	33

Listings

1	Example trace file generated by k-perf	6
2	Implementation of method run-time using busy waiting in Kotlin.	11
3	Implementation of the recursive <code>monitoredMethod</code> method.	12
4	JVM-specific implementation of <code>monitoredMethod</code> using <code>ThreadMXBean</code>	13
5	Shared implementation of the <code>BenchmarkingThreadMilli</code> benchmark worker.	17
6	JVM implementation of <code>BenchmarkingThreadNano</code> using JVM management APIs for execution time, heap usage, and garbage-collection counts.	18
7	Platform-specific implementation of <code>BenchmarkMain</code> for benchmark initialization, worker execution, and result writing.	20
8	JavaScript-specific extraction of benchmark arguments from Node.js <code>process.argv</code>	22
9	Registration of workload and benchmark-worker factories in <code>ClassRegistry</code>	23
10	Creation of registered workload and benchmark-worker instances in <code>ClassRegistry</code>	23
11	Coroutine-based benchmark entry point for JVM and Native using <code>runBlocking</code>	24
12	Promise-based JavaScript benchmark entry point using <code>GlobalScope.promise</code>	24
13	Modification of the root <code>build.gradle</code> file.	27
14	Registration of the Kotlin benchmark project in <code>settings.gradle</code>	27
15	gitignore rules added for generated Kotlin and <i>k-perf</i> benchmark files.	27
16	Added code for the <i>MooBench X Kotlin</i> build in <code>setup.sh</code>	28
17	Execution of the plain JVM benchmark artifact with the configured MooBench parameters.	30
18	Execution of the plain Native benchmark artifact using the selected target binary.	30

19	Execution of the plain JavaScript benchmark artifact with Node.js.	31
20	Commands for building and starting the <i>k-perf</i> benchmark.	32

References

- [1] Elastic. Elastic Application Performance Monitoring. <https://www.elastic.co/docs/reference/apm/observability/apm>, 2026. Accessed: 2026-09-10.
- [2] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76. ACM, 2007. DOI: <https://www.doi.org/10.1145/1297027.1297033>.
- [3] GitHub. Events that Trigger Workflows. <https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows>, 2026. Accessed: 2026-07-30.
- [4] Gradle, Inc. Gradle Wrapper Basics. https://docs.gradle.org/current/userguide/gradle_wrapper_basics.html, 2026. Accessed: 2026-07-15.
- [5] Gradle, Inc. Introduction to Plugins. <https://docs.gradle.org/current/userguide/plugins.html>, 2026. Accessed: 2026-08-03.
- [6] Gradle, Inc. Multi-Project Builds. https://docs.gradle.org/current/userguide/multi_project_builds.html, 2026. Accessed: 2026-08-03.
- [7] Gradle, Inc. Understanding Tasks. https://docs.gradle.org/current/userguide/more_about_tasks.html, 2026. Accessed: 2026-08-03.
- [8] JetBrains. Coroutine Context and Dispatchers. <https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html>, 2025. Accessed: 2026-07-15.
- [9] JetBrains. Custom Compiler Plugins. <https://kotlinlang.org/docs/custom-compiler-plugins.html>, 2026. Accessed: 2026-08-03.
- [10] JetBrains. GlobalScope. <https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/>, 2026. Accessed: 2026-07-15.
- [11] JetBrains. Kotlin Programming Language. <https://kotlinlang.org/>, 2026. Accessed: 2026-08-11.
- [12] JetBrains. Kotlin/Native Memory Management. <https://kotlinlang.org/docs/native-memory-manager.html>, 2026. Accessed: 2026-07-15.
- [13] JetBrains. promise. <https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/promise.html>, 2026. Accessed: 2026-07-15.
- [14] JetBrains. The Basics of Kotlin Multiplatform Project Structure. <https://kotlinlang.org/docs/multiplatform/multiplatform-discover-project.html>, 2026. Accessed: 2026-08-09.

- [15] JetBrains. Time Measurement. <https://kotlinlang.org/docs/time-measurement.html>, 2026. Accessed: 2026-07-13.
- [16] JetBrains. TimeSource.Monotonic. <https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.time/-time-source/-monotonic/>, 2026. Accessed: 2026-07-13.
- [17] JetBrains. Tips for Improving Compilation Time. <https://kotlinlang.org/docs/native-improving-compilation-time.html>, 2026. Accessed: 2026-08-09.
- [18] Kieker Project. Kieker Documentation. <https://kieker-monitoring.net/documentation/>, 2026. Accessed: 2026-09-10.
- [19] Node.js. The Node.js Event Loop. <https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick>, 2026. Accessed: 2026-07-15.
- [20] OpenTelemetry. OpenTelemetry Documentation. <https://opentelemetry.io/docs/>, 2026. Accessed: 2026-09-10.
- [21] Oracle. Class Thread. <https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/Thread.html>, 2026. Accessed: 2026-07-14.
- [22] Oracle. Garbage Collector Implementation. <https://docs.oracle.com/en/java/javase/17/gctuning/garbage-collector-implementation.html>, 2026. Accessed: 2026-07-15.
- [23] Oracle. GarbageCollectorMXBean. <https://docs.oracle.com/en/java/javase/26/docs/api/java.management/java/lang/management/GarbageCollectorMXBean.html>, 2026. Accessed: 2026-07-15.
- [24] Oracle. Instrumentation Interface. <https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html>, 2026. Accessed: 2026-08-03.
- [25] D. Pfeffer and M. Weninger. On the applicability of annotation-based source code modification in Kotlin (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR '23, pages 2–11. ACM, 2023. DOI: <https://www.doi.org/10.1145/3617651.3622983>.
- [26] Pinpoint Project. Pinpoint Application Performance Monitoring. <https://pinpoint-apm.github.io/pinpoint/>, 2026. Accessed: 2026-09-10.
- [27] R Core Team. Rscript: Scripting Front-End for R. <https://search.r-project.org/R/refmans/utils/html/Rscript.html>, 2026. Accessed: 2026-07-15.
- [28] D. G. Reichelt, L. Bulej, R. Jung, and A. van Hoorn. Overhead comparison of instrumentation frameworks. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, pages 249–256. ACM, 2024. DOI: <https://www.doi.org/10.1145/3629527.3652269>.
- [29] D. G. Reichelt, R. Jung, and A. van Hoorn. Overhead measurement noise in different runtime environments. *Softwaretechnik-Trends*, 44(4):12–14, 2024. Proceedings of the 15th Symposium on Software Performance (SSP 2024).

- [30] D. G. Reichelt, S. Kühne, and W. Hasselbring. Overhead comparison of OpenTelemetry, inspectIT and Kieker. In *Proceedings of the Symposium on Software Performance (SSP 2021)*, volume 3043 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: <https://ceur-ws.org/Vol-3043/short3.pdf>.
- [31] D. G. Reichelt, S. Kühne, and W. Hasselbring. Towards solving the challenge of minimal overhead monitoring. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE '23 Companion*, pages 381–388. ACM, 2023. DOI: <https://www.doi.org/10.1145/3578245.3584851>.
- [32] D. G. Reichelt, S. Yang, M. Hansson, and W. Hasselbring. MooBench: A micro-benchmark for performance overhead measurement of observability tools. *Journal of Open Source Software*, 11(123):10400, 2026. DOI: <https://www.doi.org/10.21105/joss.10400>.
- [33] Scouter Project. Scouter: Application Performance Monitoring. <https://github.com/scouter-project/scouter>, 2026. Accessed: 2026-09-10.
- [34] J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, volume 1083 of *CEUR Workshop Proceedings*, pages 59–68. CEUR-WS.org, 2013. URL: <https://ceur-ws.org/Vol-1083/paper7.pdf>.
- [35] S. Yang, D. G. Reichelt, and W. Hasselbring. Evaluating the overhead of the performance profiler Cloudprofiler with MooBench. *Softwaretechnik-Trends*, 44(4):15–17, 2024. Proceedings of the 15th Symposium on Software Performance (SSP 2024).