DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Performance Evaluation of Kotlin and Java on Android Runtime

PATRIK SCHWERMER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Performance Evaluation of Kotlin and Java on Android Runtime

PATRIK SCHWERMER

Master in Computer Science
Date: July 6, 2018
Supervisor: Elena Troubitsyna
Examiner: Johan Håstad
Swedish title: Prestandautvärdering av Kotlin och Java för Android Runtime
School of Computer Science and Communication

Abstract

This study evaluates the performance of Kotlin and Java on Android Runtime using four benchmarks from the Computer Language Benchmarks Game suite, for which a total of 12 benchmark implementations are studied. The metrics used to evaluate the performance includes runtime, memory consumption, garbage collection, boxing of primitives as well as bytecode n-grams. To benchmark the languages, a benchmark application has been developed intended to run on an Android phone. The results indicate that Kotlin is slower than Java for all studied benchmarks by a varying factor. Furthermore, the use of idiomatic Kotlin features and constructs results in additional heap pressure and the need of boxed primitives. Other interesting results indicate the existence of an underlying garbage collection overhead when reclaiming Kotlin objects compared to Java. Furthermore, Kotlin produces larger and more varied bytecode than Java for a majority of the benchmarks.

Sammanfattning

Denna studie utvärderar prestandan mellan Kotlin och Java på Android Runtime genom 12 implementationer av fyra benchmarks från The Computer Language Benchmarks Game. De mätvärden som använts för att utvärdera prestandan inkluderar körtid, minnesanvändning, garbage collection, boxing av primitiver samt bytekod n-grams. För att benchmarka språken har en benchmarkapplikation tagits fram för Android. Resultaten visar att Kotlin är långsammare än Java för samtliga benchmarks. Vidare resulterar användandet av idiomatiska Kotlin-funktioner i ökad minnesanvänding samt behovet att representera primitiver som klasser. Andra intressanta resultat inkluderar existensen av en overhead för garbage collectorn för frigörandet av objekt som allokerats av Kotlin jämfört med Java. Vidare producerar Kotlin större bytekodfiler och uppvisar mer varierad bytekod än Java för en majoritet av de benchmarks som studerats.

Contents

Acronyms

AOT Ahead-of-Time.

API Application Programming Interface.

APK Android Application Package.

ART Android Runtime.

CLBG Computer Language Benchmarks Game.

DVM Dalvik Virtual Machine.

ELF Executable and Linkable Format.

GC Garbage Collector.

JIT Just-in-Time.

JVM Java Virtual Machine.

LIFO Last In First Out.

OS Operating System.

PCA Principal Component Analysis.

VM Virtual Machine.

Chapter 1

Introduction

This chapter introduces the motivation behind this study, specifies the problem statement and research question, defines the scope and limitations, puts this study in the context of ethical aspects and sustainability and finally presents the disposition of this paper.

1.1 Introduction

Today, Java is one of the world's most used programming languages and for good reason. After its introduction in 1995, it now features a strong community and rich documentation. A part of Java's success can also be attributed to its *write once run everywhere* philosophy. By compiling source code into intermediate Java bytecode, and then executing it on a virtual machine on the target system, Java provides portability and platform independency which few other languages can offer. Although there are some inherent benefits of using Java, a major drawback is its age showing in its lack of many modern programming concepts and features. As such, a trend has emerged in the last decade where new languages have been announced, introducing a modern alternative to Java while still reaping some of the benefits which contributed to Java's popularity by targeting the Java Virtual Machine (JVM). Some popular alternatives to Java are Scala, Groovy and Clojure.

Another big part of Java's popularity is its connection to the Android platform as the first officially supported language. However, in May 17 2017, Google announced official support for another language, namely Kotlin [19]. Although introduced in 2011, Kotlin has gained most of its popularity since Google's announcement [34] and is today being used to an increasing degree. Kotlin introduces several new concepts and features full interoperability with Java, presenting itself as a tempting alternative to be used on the Android platform.

Although an attractive alternative to Java for Android development, little research has been conducted in regards to Kotlin on Android. This is a problem, both for academia and businesses who are considering making a transition to Kotlin. This thesis aims at bridging this gap in research, focusing on the performance difference of Java and Kotlin on Android.

1.2 Problem statement

Kotlin features full interoperability with Java, providing the possibility of gradually transitioning applications written in Java to Kotlin. With modern features and an outspoken goal of providing more concise syntax and improved null safety over Java, it is a tempting alternative for Android developers to make the transition to Kotlin. However, due to its sudden increase in popularity, there exists a gap in the understanding of Kotlin's behaviour on Android Runtime. This thesis aims at bridging the gap by investigating the following question:

> *Are there any differences in runtime efficiency and memory consumption between functionally equivalent Java and Kotlin implementations on Android Runtime? If so, can the difference be explained by profiling metrics and bytecode analysis?*

1.3 Scope and Limitations

This study evaluates the performance of Java and Kotlin on Android Runtime using synthetic benchmarking programs. The benchmarking programs have been chosen for their variation in characteristics and their usage in previous studies. This study is only concerned with evaluating the languages from a computational perspective and is not involved with other aspects such as rendering or graphics. Further limitations are as follows:

- This study will not compare nor analyse different algorithmic solutions for the benchmark problems but simply implement it in the fashion imposed by the creators of the benchmark suite.

- The benchmark programs will be of a sequential character. Parallel implementations are considered outside the scope of this study due to their intricate behaviour, making it difficult to compare the implementations.
- The benchmarks will only be executed on Android Runtime (ART), since this study is not concerned with comparing different runtime systems.
- Benchmarking will only be carried out on one Android device running version 8.0 or later, since this study is not concerned with comparing the performance of different mobile devices.

1.4 Ethics and Sustainability

There are two inherent ethical aspects related to this paper. Since this study is concerned with evaluating the performance of Java and Kotlin, great care must be taken to ensure that the method used is objective and does not favour a particular language. Furthermore, since the findings in this study may be used by developers when deciding on which language to use for implementation, the results must be reproducible. This is ensured in two ways; firstly through openness by providing the source code of all applications and benchmark implementations as well as the raw data of all results. Secondly, a thorough and systematic description of the benchmark environment, tools and configurations are provided.

A natural connection to sustainability exists due to the relation between performance and power consumption. There is a possibility that developers could reduce the battery consumption of applications running on ART by taking the findings of this study into account when implementing Android applications. Furthermore, compiler designers might find this study useful for optimisation purposes, further contributing to reduced power consumption.

1.5 Disposition

This paper is organised as follows: Chapter 2 establishes the necessary background as well as presents related work. Chapter 3 defines the methodology used, focusing on benchmarking programs, profiling metrics and bytecode analysis. Chapter 4 presents the results following from performing profiling and static analysis on the benchmarks. Chapter 5 contains a detailed discussion of the results, the method and the importance of this study. Chapter 6 concludes this paper.

Chapter 2

Background

In this chapter, the necessary background for this thesis is established. First, a description of compilers, compilation techniques and common compiler optimisations are given. This is followed by an overview of Android, its architecture, runtime systems and the build process for applications. Additionally, a brief overview of Kotlin is given as well as a description of a cross-language benchmarking suite used in this study. Finally, related work with respect to cross-language benchmarking is presented.

2.1 Compilers

Compilation is the process of translating code from a source language to a target language. A compiler is a piece of software capable of performing such translations. A compiler can provide cross-language compilation between high-level source code, so called transpiling. However, the term is more often used to refer to the process of translating high-level source code to a lower-level language such as bytecode, assembly or native code. Native code, unlike source code or intermediate bytecode, are platform specific instruction which can be executed directly on the processor of the target machine. Therefore, the produced native code is tailored to the environment on which the application is intended to run.

The compilation process can take place during different stages of the build process, each posing different benefits and drawbacks. A general description of two different compilation techniques is given in Section 2.1.1 and 2.1.2. Furthermore, in order to increase the performance of the program the compiler sometimes makes optimisations. Some of the most common compiler optimisations are covered in Section 2.1.3.

2.1.1 Ahead-of-Time Compiler (AOT)

An Ahead-of-Time (AOT) compiler provides a static compilation process, meaning that the compilation is done prior to execution of the application [36]. Therefore, all optimisations made by the AOT compiler are based on static analysis. Another approach is dynamic compilation, which is utilised by a Just-in-Time compiler and described in Section 2.1.2.

Benefits of AOT compilation is that it is straightforward, since the compilation process is based entirely on the static source code. Thus, no compilation steps are performed during runtime, which eliminates runtime overhead. Drawbacks of AOT compilation include longer upfront compilation time and potential wasted compile time on the methods or paths rarely used, which instead could be interpreted during runtime. Furthermore, AOT compilation is unable to make as well guided and precise optimisations as JIT compilation since optimisations based on the actual execution history of the application is not possible.

2.1.2 Just-in-Time Compiler (JIT)

A Just-in-Time (JIT) compiler provides a dynamic compilation process, meaning that the compilation takes place during runtime [29]. In order to achieve this, JIT combines a compiler with an interpreter. The interpreter, unlike a compiler, does not translate higher-level source code into native code but instead interprets the higher-level instructions and performs the actions described by the higher-level language by letting the virtual machine do some conversion during runtime. There is an overhead associated with the runtime conversion and therefore, an application being interpreted performs worse than its compiled counterpart. However, the cost of compilation is eliminated.

During runtime, the code starts off being interpreted. The JIT compiler performs profiling on the application during runtime to obtain data of its behaviour. The data acts as a basis for deciding whether or not to compile a particular method or block of code. Two common metrics which the JIT compiler uses is the execution count of methods and loop detection [6]. If a method is frequently invoked, it might be beneficial to spend some time to compile it into native code for efficient execution. If a loop is executed many times it might be worth optimising it by, for instance, loop unrolling or inlining methods invoked in the loop body to reduce the overhead of function calls. These compiler optimisations techniques are further described in Section 2.1.3. Therefore, a JIT compiler wastes no time on compiling the parts of code that do not add to the overall performance of the application.

A consequence of this approach to compilation is that the application will take additional time to execute during the first run due to being interpreted, however, the performance will improve over time as more parts of the application gets compiled to native code [28]. This phenomena is often referred to as the warmup time. Furthermore, the JIT compiler will make more informed optimisations the more execution history it has to consider.

2.1.3 Compiler optimisations

In the compilation process, the compiler sometimes perform optimisations based on the static or dynamic context. The goal of such optimisations is to improve performance or reduce resource consumption of an application, while preserving the program definition. In order to provide an understanding of how these optimisations are characterised, five common techniques are presented in this section. It should be noted, however, that these techniques are by no means exhaustive. Examples are given in the form of Java code as opposed to bytecode or native code for its easy-to-read syntax.

Constant folding and propagation

When dealing with expressions or subexpressions containing constants, the compiler can make optimisations during compile time to save the runtime system from performing costly calculations. Consider the variable assignment to the left in Listing 1.

```java
int x = 5*20/2;
```

```java
int x = 50;
```

Listing 1: Unoptimised assignment (left) and assignment after constant folding (right)

Since the expression only contains constants, the compiler knows that the value of the expression will not change during runtime. Thus, by performing constant folding, the value of the expression is calculated at compile time and assigned to `x`.

Constant propagation is closely related to constant folding and is simply the process of propagating the value of a constant fold in expressions using the pre-computed variable.

Common subexpression elimination

When a subexpression occurs in several expressions, the compiler can perform subexpression elimination to reduce overhead associated with recomputing the same subexpression several times. This is illustrated by the examples in Listing 2, where the compiler in the optimised version only computes the value `a * b` once and reuses the result in the assignments. However, this optimisation can only be performed given that the result of the subexpression will not change between the statements.

```
int x = a * b + c;
int y = a * b - c;
```

```
int tmp = a * b;
int x = tmp + c;
int y = tmp - c;
```

Listing 2: Unoptimised expressions (left) and expressions after performing subexpression elimination (right)

Dead code elimination

Another optimisation which modern compilers often utilise in order to reduce the size of the compiled code is dead code elimination. If the compiler can determine that a branch will never be taken, it can simply omit the body of the branch in the resulting code.

Inline expansion

Since there is some overhead associated with the method calls, inline expansion is an optimisation technique applied when a method is frequently invoked, for instance in a loop such as in Listing 3. If the frequently invoked method instead would be expanded at call site, this would eliminate the overhead associated with the method calls with the effect of increased code size. Although there is a trade off, such an optimisation can often be considered beneficial, in particular for smaller methods.

```
private int addition(int a, int b){
    return a+b;
}

...

int x = 0;
for(int i = 0; i < 1000; i++){
    x = addition(x, 10);
}
```

Listing 3: Frequently invoked method *addition*.

By expanding the method *addition* to the call site in the loop body, the 1000 function calls can be eliminated as illustrated in Listing 4.

```
int x = 0;
for(int i = 0; i < 1000; i++){
    x = x + 10;
}
```

Listing 4: Optimised loop where the method *addition* has been expanded at call site, reducing 1000 method calls.

Loop unrolling

Loop unrolling is a transformation technique which optimises the performance of loops by increasing the number of operations performed at each iteration in the loop body. Consider the unoptimised loop in Listing 5.

```
int[] x = new int[1000];
for(int i = 0; i < 1000; i++){
    x[i] = i;
}
```

Listing 5: Unoptimised version of the loop, performing one operation at each iteration.

By increasing the number of operations performed at each iteration, the number of control instructions and loop condition checks will be reduced resulting in increased performance.

```
int[] x = new int[1000];
for(int i = 0; i < 200; i+=5){
    x[i] = i;
    x[i+1] = i+1;
    x[i+2] = i+2;
    x[i+3] = i+3;
    x[i+4] = i+4;
}
```

Listing 6: Optimised version of the loop, performing five operations at each iteration.

2.2 Android

Android is an operating system developed by Google, mainly targeted towards the smartphone, tablet, Internet-of-Things and wearable markets. Today, it is the world's most popular mobile operating system and as of Q2 2017 it had a market share of 87.7% [11]. In recent years, Android has undergone major changes in its runtime environment. In this section, a detailed description of the Android software architecture, different runtime environments and the Android build process are presented.

2.2.1 Android Architecture

The Android software architecture is presented as in Figure 2.1. The software stack builds on top of the first layer, the Linux Kernel, providing drivers and power management as well as underlying functionalities such as threading, security features and low-level memory management [15].

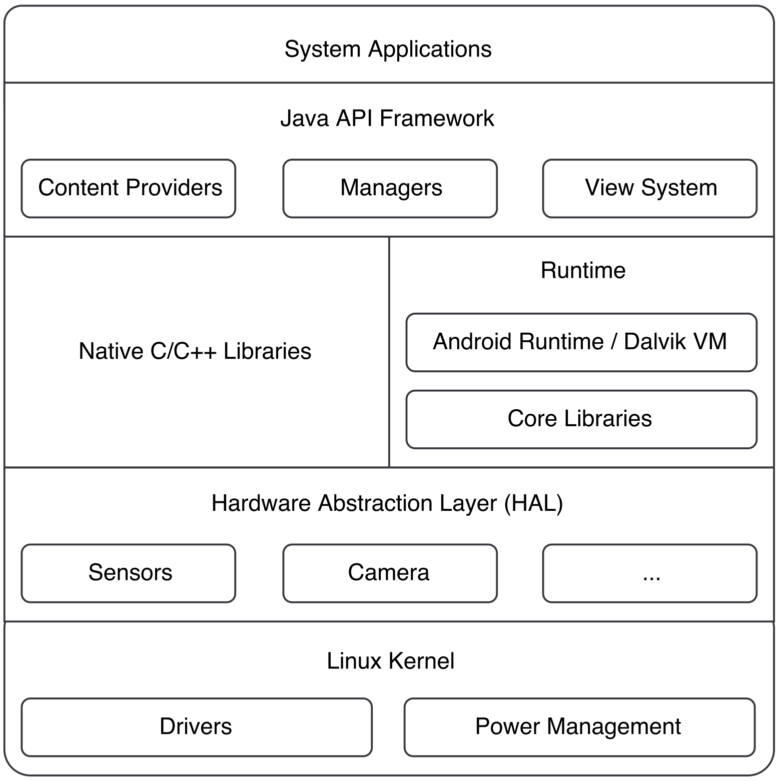


Figure 2.1: Android Architecture

The second layer in the software stack consists of the Hardware Abstraction Layer (HAL), which provides hardware interfaces for each of the hardware components included in mobile devices such as sensors, cameras, Bluetooth and audio. This abstraction enables the Java API Framework to access hardware components, without them being exposed directly [15].

The third layer in the stack consists of native C/C++ libraries [15]. Many underlying system components are built on top of native code, such as the runtime systems, thus the requirement for native libraries exists. The second part of the third layer is the runtime, containing core libraries used by Java and Kotlin as well as the actual runtime systems. The Dalvik Virtual Machine and Android Runtime are covered in more detail in Section 2.2.2 and Section 2.2.3.

The fourth layer contains the Java API Framework which provides reusable components aiding in development of Android applications. Such components includes the View System providing UI features, the Content Providers enabling accessing data from other apps and several Managers providing additional features and interactions with the Android OS and hardware [15].

The top layer in the software stack consists of system applications, including default applications as well as third-party applications available by phone manufacturers or downloadable from Google Play [15].

2.2.2 Dalvik Virtual Machine (DVM)

The Dalvik Virtual Machine (DVM) is the runtime system used on Android devices running versions below 5.0 (API Level 21) and has since been replaced with Android Runtime, covered in Section 2.2.3 [12]. The DVM is a register-based JIT compiled virtual machine, as opposed to the stack-based Java Virtual Machine (JVM) commonly used on desktop and server platforms.

A stack-based architecture utilises a memory structure known as a stack to evaluate expressions in a Last In First Out (LIFO) manner. When evaluating an expression, operands stored on top of the stack are fetched using `pop` and evaluated. The result is then stored back on top of the stack using `push`. A register-based architecture, on the other hand, does not deal with a stack but instead utilises registers in order to store and fetch data. A register-based architecture is considered to be more efficient than a stack-based architecture [33], and is therefore used by DVM. Additionally, a register-based virtual machine requires less bytecode instructions to implement the same functionality for high-level source code [28].

2.2.3 Android Runtime (ART)

With the introduction of Android 5.0, Google made architectural changes to the software stack, replacing the DVM with a new runtime system, namely Android Runtime (ART) [12]. ART initially brought three main improvements over the DVM: ahead-of-time compilation, improved garbage collection and better debugging and development tools [14]. The introduction of AOT compilation reduced the runtime overhead associated with JIT compilation, by compiling the application to native code on installation. Furthermore, the new runtime system provided slightly better battery life compared to DVM [37]. However, although ART reduced runtime overhead, the initial compilation time increased causing longer installation times and increased local storage consumption for applications [37]. Additionally, due to the character of AOT compilation, no runtime optimisations could be performed.

Since its first introduction in Android 5.0, further improvements have been made to ART. Android 7.0 introduced a hybrid, device-configurable compilation process, using both AOT compilation and JIT compilation. The application starts off being AOT compiled where frequently invoked methods are then JIT compiled into native code based on profiling data from the execution of the application [14]. In Android 8.0 several new features have been added such as a concurrent compacting garbage collector, resulting in an average of 32% smaller heap sizes and 70% faster allocations than in Android 7.0 [13]. Further improvements includes more advanced loop optimizations such as dead-code removal inside loop bodies as well as loop unrolling and more advanced inlining [13].

The Garbage Collector (GC) used in ART uses fewer garbage collection steps, performs parallel processing and also consists of a special GC for recently-allocated short-lived objects [14]. The need for such a specialised GC aligns with a phenomena known as the Generational Hypothesis which states that many recently-allocated objects only live for a short duration and therefore, their occupied memory can be reclaimed soon after allocation. The specialised GC exploits this behaviour by only examining recently-allocated objects, thus minimising the work required to reclaim unused objects [27].

2.2.4 Android Build Process and Runtime

The Android build process constitutes several steps involving transforming source code into a runnable application on the Android platform. The process is illustrated in Figure 2.2. It begins with Java or Kotlin source code contained in the .JAVA or .KT files which are compiled into Java bytecode stored in the .CLASS file format. In the case of Java, the javac-compiler is used [36] whereas the kotlinc-compiler is used to compile Kotlin source code into Java bytecode.

The Java version used on Android is not derived from a particular specification. Instead Google has chosen to implement a more limited version of the Java standard libraries [8] to suite the needs of the Android platform.

One such example is the exclusion of the Swing-library which is used for UI and windowing which is considered not inline with the design and behaviour of the Android applications.

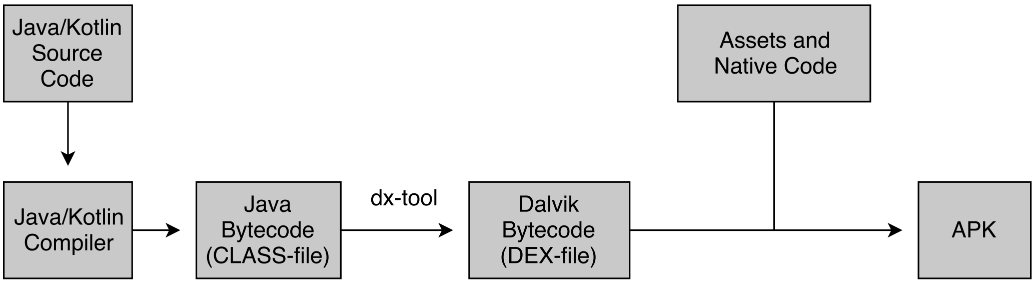


Figure 2.2: Android APK Build Process

Furthermore, the Android runtime systems do not utilise Java bytecode but instead use Google's proprietary Dalvik bytecode introduced with the Dalvik VM. Dalvik bytecode is optimised for a minimal memory footprint [8, 10] which is of high importance to a platform with limited resources. One difference between the `.CLASS` file format and `.DEX` (Dalvik Executable) is that all application classes are stored in a single `.DEX` file, unlike one class per `.CLASS` file. This allows for using a shared constant pool eliminating duplication across several classes and resulting in significant memory savings [8]. In order to convert the `.CLASS` files into a single `.DEX` file, the `dx`-tool is used.

The `.DEX` file is then packaged with other assets as well as native code and zipped into an Android Application Package (APK). The assets includes resources used by the application such as graphics, images and sound whereas the native code corresponds to pre-compiled libraries. The compression allows for reduced space-consumption which allows for faster downloads of the APK when installed on a target device.

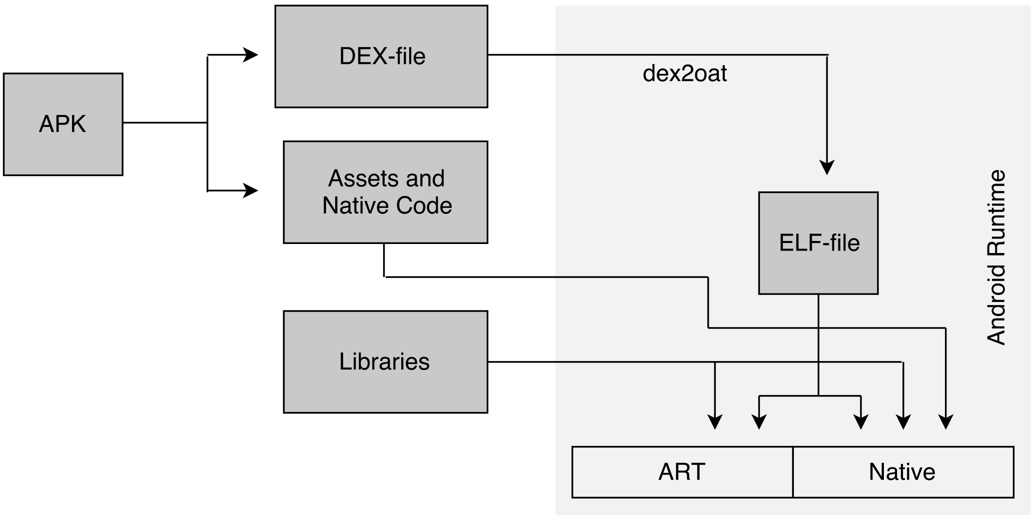


Figure 2.3: APK Installation and Runtime

When the application has been packaged into an APK, it is ready to be installed on a target device. Figure 2.3 describes the installation process for a device using ART. The APK is first unzipped into its constituents, namely the `.DEX` file as well as the assets and native code. Using the `dex2oat`-tool, parts of the DEX file is compiled into executable native code stored in the `OAT`-file format. During compilation, the AOT compiler determines which parts to compile into native code and which parts to leave intact. The native code as well as the bytecode is contained within an Executable and Linkable Format (ELF) file containing the executable files as well as object code. Finally, external libraries are linked in.

The application is now ready to be executed on the platform. Android uses a system known as Zygote in order to provide an environment in which the application can execute. Zygote is a process which is started on boot of the Android system containing pre-initialised core libraries. When the Android system requests that an application should be launched, Zygote forks itself loading the APK to be executed. This eliminates the cold start which otherwise would have occurred and greatly speeds up the startup time of an application. By this design, all Android application are executed in a separate instance of the VM [8].

Once the application has obtained its instance of the VM it can now be executed. As previously mentioned, Android uses a hybrid compilation process as of version 7.0. Therefore, there exists a need for prioritising whether AOT-compiled code or JIT-compiled code should be used once a method is invoked. This decision process can be described as follows [14]:

1. If the method is JIT compiled (method exists in JIT cache) native code from the JIT compilation is used.
2. If the method is AOT compiled (method exists in OAT-file), native code from the AOT compilation is used.
3. If the method is neither JIT compiled nor AOT compiled, the method is interpreted. Profiling of the method determines whether or not to compile it in the future (based on metrics such as method hotness).

2.3 Kotlin Language

Kotlin can be described as a statically-typed language, combining object-oriented principles with functional features. The language is general-purpose, meaning that it is intended to work server-side, client-side and on mobile platforms. The language was initiated by JetBrains, which still is a major contributor to the open-source project. Kotlin was officially released in February 2016 [4], but has been in development since 2010. Although used by the Android community before Google's official endorsement, the popularity of the language increased significantly after the announcement. Today, Android developers constitute a considerable part of Kotlin's user base. Apart from targeting Android and ART, Kotlin also targets the JVM, can be transpiled into JavaScript and compiled into native code through LLVM.

From the start, an outspoken design focus of Kotlin is that it should be *pragmatic* [4]. This influences several aspects of Kotlin. Firstly, Kotlin provides full interoperability and seamless support with Java. This enables a gradual transition from Java to Kotlin, preventing Kotlin adopters from rebuilding projects from scratch. Furthermore, it also provides the opportunity to use Java's rich libraries which can both be extended and referenced. Secondly, Kotlin's syntax focuses on removing verbosity and its semantics is focuses on removing boilerplate code. Additionally, Kotlin features an improved type system focusing on null-safety and type inference. An overview how these concepts are translated into language features is given in Section 2.3.1.

Another aspect of Kotlin's goal of being a pragmatic language is its strong tooling support from release. From the start, Kotlin was supported in several industry-standard IDEs including IntelliJ IDEA, Eclipse and Android Studio. Furthermore, it supports integration with build tools such as Maven and Gradle, as well as platforms such as Github. This provided developers with a sufficient maturity for integrating Kotlin in real-world applications from the release.

2.3.1 Constructs and Concepts

In this section, a brief overview of Kotlin's constructs and concepts is given to provide an insight into the differences of Java and Kotlin and describe how the design principles translate into language features. The features listed here are by no means exhaustive. An exhaustive list is available on the Kotlin website [20].

Basic types

All basic types such as `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double` and `arrays` commonly used in Java are represented as classes in Kotlin. This is in line with Scala's representation of basic types, ensuring consistency and avoiding explicit casting between primitives and their class representations. However, due to performance reasons some of the basic types are represented as primitives during runtime [20].

Data classes

Unlike in Java, Kotlin has introduced a special type of class used when the sole purpose is to represent data, namely data classes. To implement a data class, the keyword `data` is used.

```
data class Animal(val name: String, val age: Int)
```

Listing 7: Data class construct in Kotlin

A data class automatically provides several commonly used member functions such as `hashCode()`, `toString()`, `equals()` and `copy()` as well as getters and setters for each property, without having to implement them explicitly. This supports a pragmatic way of handling classes of such character and eliminates boilerplate code otherwise present in Java.

Type system

One of Kotlin's strongest features is its improved type system over Java. It allows for strong null-safety and removes explicit type declarations through type inference. The strong null-safety is provided in two ways. Firstly, Kotlin distinguishes between nullable and non-nullable types and references. Non-nullable types cannot be assigned null values. When referencing such variables, the developers can be certain not to encounter a `NullPointerException` during runtime. Instead, they would receive an error during compilation. However, using the `?` operator upon declaration can make all types nullable upon request.

Secondly, Kotlin does not allow accessing a nullable variable without first controlling that it does not hold a null reference. Kotlin introduces a new operator `?.` for safe referencing where the statement only is executed if the reference (left-hand side of the operator) is not null.

The examples in Listing 8 demonstrates the capabilities of Kotlin's type system.

```kotlin
val nonNullableString: String = null //Compiler error
val nullableString: String? = null //No compiler error
val inferredString = "Hello World"

val cat: Animal? = null
val dog: Animal = Animal(name = "Jack", age = 2)
dog.age = 3
cat.age = 6 //Compiler error, unchecked reference
cat?.age = 6 //Not executed, cat is null
```

Listing 8: Type system in Kotlin

Functional programming

While being object-oriented, Kotlin has a strong support for functional programming concepts which is seamlessly included in the language. Features supported by Kotlin include higher-order functions, lambda expressions, lazy evaluation and several categories of methods for working with collections including aggregation, filtering, mapping and ordering. An example demonstrating some of Kotlin's powerful functional collection handling is presented in Listing 9.

```
data class Animal(val name: String, val age: Int,
        val isTame: Boolean)

val animals = mapOf(Animal(10, false), ..., Animal(5, true))

animals.filter{ animal -> animal.isTame && animal.age > 10}
    .sortedBy{animal -> animal.age}
    .take(5)
    .map{animal -> animal.name.toUpperCase()}
```

Listing 9: Functional programming example in Kotlin. Gives the name of the five youngest animals over the age of 10 in upper case.

2.4 Benchmarking

Benchmarking is the process of comparing different systems using data and metrics obtained from real or synthetic applications, so called benchmarks [9]. In the synthetic context, the behaviour of the benchmarking program is to simulate intensive resource consumption. Another way of distinguishing between different types of benchmarks is through the concepts of micro- and macrobenchmarking. The purpose of microbenchmarks is to evaluate the performance through small, synthetic, well-defined applications which allow one to reason on a low granularity level. Macrobenchmarks, on the other hand, are larger, often real-world applications, which provide the benefit of a more exhaustive coverage of instructions and scenarios with the drawback that the data cannot be analysed on such a low granularity level as in the case of microbenchmarks.

In order to provide more extensive benchmarking, benchmark suites are often used. Benchmarks suits are collections of multiple benchmarks, often with different characteristics [9]. In the world of Java, there exists several benchmarking suites which have been used extensively in academia, some of the most influential ones being SPEC [5] and DaCapo Benchmark Suite [35, 3]. However, it is an acknowledged problem that there exists few cross-language benchmark suites [32].

2.4.1 Computer Language Benchmarks Game (CLBG)

The Computer Language Benchmarks Game (CLBG) is, to our knowledge, the only cross-language benchmark suite that has been used extensively in academia. To date, the benchmark suite consists of 10 microbenchmarks with implementations in over 25 languages [17]. The benchmarks consists of small, well-defined algorithmic problems, which characterise different types of workloads. In order to offer such a vast variety of languages included in the suite, CLBG allows for user contributed implementations of the benchmarks. To guarantee the comparability and correctness of the benchmarks, CLBG imposes rules on how implementations are supposed to be written so that the solutions are functionally equivalent. In this context, functionally equivalent refers to the fact that the implementations produce the same output given the same input, thereby solving the same problem. Furthermore, it also puts restrictions on how the algorithms are supposed to be implemented. This is done to guarantee that the benchmarks actually test the performance of language constructs and features and not optimisations of the algorithms.

Although CLBG enforces rules on how implementations are supposed to be written, there is some critique pointing out that too few restrictions are imposed precluding many relevant conclusions [25]. For instance, CLBG allows for both concurrent and sequential implementations of the algorithms, which quite significantly can impact the runtime performance. However, although we recognise the validity of this critique, we also realise that it is difficult to provide a benchmark suite which allows for sufficient implementational freedom, to highlighting different language constructs and features and yet, at the same time, ensures cross-language comparison.

2.4.2 Statistical theory

The process of benchmarking is inherently nondeterministic. Factors such as JIT-compilation, scheduling, cache misses and page faults are uncontrollable. As such, several executions are needed to obtain reliable results. Given a sample size n, sample means can be calculated for different metrics. Commonly, two means are used for benchmarking, namely the arithmetic mean as defined in Formula 2.1 and the harmonic mean as defined in Formula 2.2.

$$\bar{x}_A = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.1}$$

Arithmetic mean [24, p.30].

$$\bar{x}_H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \tag{2.2}$$

Harmonic mean [24, p.31].

The arithmetic mean is used when we are interested in measuring the average of the sum of the raw results, such as execution time while the harmonic mean is used when we want to compute average rates [24, p.40]. To calculate the dispersion of the data obtained, the standard deviation is used as given by Formula 2.3

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{2.3}$$

Standard deviation [24, p.39].

Using the standard deviation, a confidence interval can be formed around the mean. This allows reasoning about the results on a confidence level of $1 - \alpha$. A lower value of α forms a larger confidence interval.

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}} \qquad c_2 = \bar{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{2.4}$$

Two-sided confidence interval [24, p.49].

A two-sided confidence interval can be defined as in Formula 2.4, or expressed in a more compact form as in Formula 2.5. The value of $z_{1-\alpha/2}$ is the number z such that the probability that for a normally distributed, random variable x with $\mu = 0$ and $\sigma^2 = 1$ we have $Pr(x > z) = \alpha$.

$$\bar{x} \pm z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{2.5}$$

2.5 Bytecode analysis

Bytecode analysis refers to the process of analysing the bytecode of a program to draw conclusions about its behaviour and performance or relation to other programs. The analysis can either be carried out in a qualitative or a quantitative way, meaning that the bytecode can either be analysed manually by inspection or in a more automatic fashion through bytecode metrics. This section focuses on establishing the theory of quantitative bytecode analysis, as carried out in [23].

2.5.1 N-gram

An n-gram can be defined as a consecutive n-atom sequence, extracted from a longer sequence of atoms. N-grams are often used within the field of Natural Language Processing for classification and prediction, where the atoms represent words. The value of n can differ, and sequences of different lengths are often studied. A 1-atom sequence is commonly referred to as a unigram, whereas 2- and 3-atom sequences are referred to as bigram and trigrams respectively.

In the context of bytecode, n-grams are constructed from n-length sequences of consecutive bytecode instructions. In [23], an additional constraint of that the bytecode instructions are to belong to the same code block is added, so that the n-grams adhere to the bytecode grammar. The construction of unigrams, bigrams and trigrams from a bytecode sequence is illustrated in Figure 2.4.

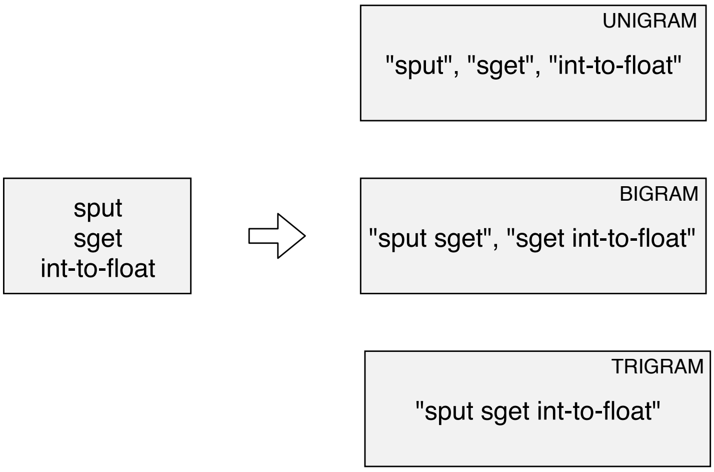


Figure 2.4: Uni-, bi- and trigram construction from a bytecode sequence

2.5.2 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical method commonly used for dimensionality reduction of high-dimensional datasets [23]. Since this thesis is neither concerned with nor dependent upon the mathematical aspects of PCA, a detailed mathematical descriptions is omitted. In brief, the process consists of calculating $min(n, p - 1)$ principal components, where n is the number of samples and p is the number of variables or dimensions. Each principal component represents dimensions containing the greatest variance in the dataset in decreasing order (the first principal component contains dimensions of the highest variance, the second principal component contains dimensions the second highest variance and so on). Thus, the goal of PCA is to reduce the dimensions while containing as much information regarding the data as possible.

Since the theoretical unique bytecode n-grams is b^n, where b represents the unique bytecode instructions, the number of dimensions grows exponentially with n. In [23], PCA is applied to the bytecode by constructing n-grams and calculating their relative frequencies. Thus, the relatedness of the benchmarks can be visualised by plotting the first two principal components of each benchmark.

2.6 Related work

The primary objective of this study is to evaluate the performance of Kotlin and Java on Android Runtime through benchmarking. Given that such a short period has passed since Kotlin's official release, to the best of our knowledge, no previous studies have been conducted related to Kotlin in the context of performance benchmarking. However, other cross-language studies have been carried out.

2.6.1 Dynamic Metrics for Java

In Dufour et al. [7], the authors identified requirements for dynamic metrics to allow for comparison of compiler optimisation techniques and runtime system designs. The authors also suggested a set of dynamic metrics suitable for benchmarking Java. The suggested metrics have influenced the design of several benchmark suites, among others, the DaCapo benchmark suite [3, 25].

2.6.2 JVM-Hosted Languages: They Talk the Talk, but do they Walk the Walk?

In Li, White, and Singer [23], the authors performed a cross-language study to compare the behaviour between Java and other JVM-hosted languages including Scala, Clojure, JRuby and Jython. Although several languages had gained traction within the industry, the authors identified that little academic consideration had been given in regards to the characterisation of the languages apart from Java and Scala. To benchmark the languages, a set of ten CLBG benchmarks along with real-world applications were used.

The languages were benchmarked using several metrics, some of which can be seen as applicable both for micro- and macrobenchmarking whereas others are better suited towards the latter. Metrics applicable for both included dynamic n-gram analysis using PCA for visualisation, stack depth, object lifetimes and amount of allocated boxed primitives. Metrics more suitable for real-world applications included method sizes as well as method and basic block hotness.

Some of the most interesting findings indicated that Java executed more diverse bytecode compared to other JVM-hosted languages and that the non-Java languages produced bytecode sequences not observed for Java. Furthermore, the results indicated that Scala and Clojure exhibited a larger stack depth than the other languages, that objects of non-Java languages were more short lived and that all non-Java languages used more boxed primitives compare to Java.

2.6.3 Workload characterization of JVM languages

Continuing on cross-language evaluation of JVM-hosted languages, Sarimbekov et al. [32] performed workload characterisation and analysis of Java, Scala, Closure, Jython, JRuby and JavaScript using a combination of CLBG benchmarks and real-world application, mainly focusing on the differences between static and dynamic languages.

The study covers a vast amount of metrics, where some can be seen as better suitable for the context of characterising real-world applications whereas others are more applicable for synthetic applications. Some of the more interesting metrics with respect to this study includes the amount of allocated and requested boxed primitives, object lifetimes an object sizes. Although the authors only analyse and discuss metrics which best highlights the differences between static and dynamic languages, the methodology is still relevant and applicable to this study.

The authors also compared the relatedness between the results for the synthetic and real-world applications. For Java and Scala, the behaviour of the CLBG benchmarks differed compared to the real-world applications, but for the other languages the results were more similar. However, the authors stress the importance of microbenchmarks, justifying that they allow one to reason and understand the behaviour of a language on a more detailed level than macrobenchmarks.

2.6.4 Cross-language Compiler Benchmarking - Are We Fast Yet?

In Marr, Daloze, and Mössenböck [25], cross-language compiler benchmarking was conducted for Java, JavaScript, Ruby, Crystal, Newspeak and Smalltalk to compare the performance of a set of the core language abstractions. The authors criticise CLBG for being too permissive in regards to the benchmark implementations, stating that one of the major issues of cross-language benchmarking is the difference in standard libraries.

Therefore, a new benchmark suite is introduced and evaluated in the study, consisting of both macro- and microbenchmarks. The metrics used in the study closely follows those suggested by Dufour et al. [7]. However, the benchmark suite has not gained traction outside the work of the original authors, and is therefore not as well-explored as CLBG. Furthermore, the difference in standard libraries between Java and Kotlin is seemingly small compared to the languages studied in [25]. In fact, Kotlin uses Java's standard library and only extends upon the underlying implementation through extension functions to allow for idiomatic access.

Chapter 3

Methodology

This chapter describes the benchmarks used in this study for comparing the performance of Kotlin and Java on Android Runtime. Furthermore, the performance metrics used to characterise the languages are defined and descriptions of the implementations are provided. Finally, the environment and setup of the experiment is presented.

3.1 Benchmarks

As described in Section 2.4.1, the only widely acknowledge cross-language benchmark suite used in literature is the Computer Language Benchmarks Game (CLBG). Therefore, this study uses CLBG to evaluate the performance of Kotlin and Java on ART. As discussed in Section 1.3, the languages will only be evaluated from a computational perspective. This is reflected by the use of CLBG.

Although CLBG features benchmark implementations in over 25 programming languages, the support of Kotlin is lacking. As of the writing of this paper, there only exists two Kotlin implementations which have been auto-generated from Java. Therefore, in order to evaluate the performance of the languages, original Kotlin implementations are developed. The method used for implementing the benchmarks as well as the aspects considered while doing so are further described in Section 3.3.

Due to the time constraints of this study, providing implementations for all CLBG benchmarks are considered infeasible. As such, this study limits it scope to only evaluate the performance of Kotlin and Java on a subset of the CLBG benchmarks.

Since only a subset of the CLBG benchmarks is studied, it is important to include benchmarks of different character to ensure the validity of the results. In Li, White, and Singer [23], the authors distinguish between some of the CLBG benchmarks by classifying them based on operation characterisation. Operation characterisation represents which main type of operation(s) are executed by a particular benchmark with respect to Integer, Floating-point, Pointer and String operations. The findings in [23] are described in Table 3.1.

Benchmark	**Integer**	**Floating-point**	**Pointer**	**String**
Binary-Trees	—	—	Yes	—
Fannkuch-Redux	Yes	—	—	—
Fasta	—	—	Yes	—
K-Nucleotide	—	—	Yes	Yes
Mandelbrot	—	Yes	—	—
Meteor-Contest	Yes	—	—	—
N-body	—	Yes	—	—
RegexDNA	—	—	—	Yes
Reverse-Complement	—	—	—	Yes
Spectral-Norm	—	Yes	—	—

Table 3.1: Operation characterisation of CLBG benchmarks [23]

The method used to distinguish between CLBG benchmarks in [23] helps to select the subset of benchmarks used in this study. Furthermore, it can be argued that the operation types, to some extent, also represent applications of different character. For instance, Floating-point operations are commonly performed in applications relying on sensor data whereas String operations are performed in I/O intensive applications such as text editors.

This study ensures a good variety of benchmark types by providing minimum coverage of the operation types described in Table 3.1. The Fannkuch-Redux benchmark is included in this study to evaluate the performance on Integer operations. N-body is used to compare the languages on floating-point intensive operations, since a similar benchmark has been used to compare three programming models on Android in Qian, Zhu, and Li [30]. For pointer operations, the Fasta benchmark is included since several other benchmarks are reliant on its output as input. Among these benchmarks is Reverse-Complement which is included to study the performance on String operations.

As such, this study evaluates the performance of Kotlin and Java using a total of four CLBG benchmarks. The benchmark included are further introduced in the subsequent subsections.

3.1.1 Fasta benchmark

The Fasta benchmark is concerned with generating three DNA sequences. The benchmark takes an integer n as input, representing the length of the sequences to be generated. The sequences are to be outputted with a line length of 60 nucleotides per line.

The first DNA sequence is to be generated by repeating a predefined sequence while the other two sequences are generated using pseudo-random selection from two alphabets of nucleotides, a described by the function in Listing 10.

```
IM = 139968
IA = 3877
IC = 29573
Seed = 42

Random (Max)
   Seed = (Seed * IA + IC) modulo IM
   = Max * Seed / IM
```

Listing 10: Pseudo-random generator [17]

As described in Table 3.1, the Fasta benchmark is categorised as Pointer-intensive due to the frequent referencing to data structures representing the DNA sequence.

3.1.2 Fannkuch-Redux benchmark

The Fannkuch-Redux benchmark was first introduced in a paper by Anderson and Rettig [2] where the authors analysed the behaviour of the LISP programming language. Fannkuch is an abbreviation of Pfannkuchen which is German for pancakes, and is an analogy to flipping pancakes, namely flipping integer sequences.

The Fannkuch-Redux benchmark takes an integer n as input, which represents the set $S = \{1, 2, \dots n\}$. Let P represent a permutation of S. The task is to reverse the first k digits of P, where k represents the first digit in the permutation, until $k = 1$. This should be done for all n-length permutations of S, where the output of the benchmark is a checksum to validate the computations as well as the maximum number of reversions needed until $k = 1$ for any permutation of S.

The CLBG does not impose any particular algorithm for generating the the $n!$ permutations of S. However, both the Java and Kotlin implementations used in this study use the algorithm as suggested by Oleg Mazurov [18] to strengthen the validity of the results.

Fannkuch-Redux belongs to the group of Integer intensive benchmarks as shown in Table 3.1 due to the frequent flipping of integer sub sequences.

3.1.3 N-body benchmark

The N-body benchmark is concerned with simulating the movements of the Jovian planets in the solar system, which includes Jupiter, Saturn, Uranus, Neptune and the Sun. A similar benchmark has been used in Qian, Zhu, and Li [30], involving particle mechanics instead of celestial mechanics.

The N-body benchmark takes an integer n which represents the number of simulation steps. The system, that is the mass, velocity, coordinates as well as the movements of the planets are defined by Mark C. Lewis who suggested the benchmark problem. The simulation is then advanced n steps and the output is the energy of Jovian planets before and after the simulation.

As described by Table 3.1, the benchmark deals with intensive floating-point operations due to the simulation of the movement of the planets.

3.1.4 Reverse-complement benchmark

The Reverse-Complement benchmark takes input from `stdin` consisting of output from the Fasta-benchmark. The task is to compute the DNA strands which will bind to each of the three Fasta DNA sequences. As the benchmark name suggests, a strand which will bind to a sequence is the complement of each nucleotide in reverse. The complements are calculated using a pre-defined translation table provided by CLBG. The output consists of the name of the DNA sequences and the reverse-complements in Fasta format. As such, the Reverse-complement benchmark is considered to deal with String-intensive operations as defined in Table 3.1. Furthermore, due to the input and output of large text sequences, it can also be considered I/O intensive.

3.2 Java Implementations

As described in Section 1.3, this study is only concerned with evaluating the sequential performance of Kotlin and Java. Kotlin will be benchmarked against the fastest sequential Java implementations for each benchmark used in this study. However, since the Java implementations have been developed for the JVM, slight modifications have been made to allow for execution on ART. These modifications involves handling of input, as `stdin` is not a reliable method of program input on ART. The correctness of the Java implementations have been verified using the sample input and output provided by CLBG for each benchmark, ensuring that the program given a particular input gives the expected output.

The overall design and architecture of each Java implementation is provided in the subsequent subsections. Furthermore, the Java source code is available in full in Appendix A.

3.2.1 Fasta

The Java implementation for Fasta used in this study has been derived from [1]. No performance affecting modifications have been made to the source code.

The program takes an integer n as input, corresponding with the length of the DNA sequences that are to be produced. The first produced DNA sequence, `ALU`, is based on repeated copying from an existing sequence stored as a `String` of length 287 nucleotides. For the second and third sequence, namely `IUB` and `HOMO_SAPIENS`, the sequences are generated using the pre-defined pseudo-random function in Section 3.1.2.

Both `IUB` and `HOMO_SAPIENS` are represented by a class `FloatProbFreq` containing a byte array of nucleotides and a double array of the corresponding frequencies as well as a method of calculating the cumulative probability of selecting each nucleotide as well as the earlier mentioned pseudo-random function.

Each DNA sequence is stored in a buffer in FASTA-format (60 nucleotides per line) with a capacity of $62,464$ bytes. Once full, the buffer is printed to `stdout` through the means of an `OutputStream`.

3.2.2 Fannkuch-Redux

The Fannkuch-Redux implementation provided by Gouy and Mazurov [18] has been used in this study. No major alterations has been made to the source code.

As described in Section 3.1.2, the program takes an integer n as input with the goal of computing the maximum number of flips needed for any permutation of the set $S = \{1, 2, ..., n\}$. As such, the implementation contains two performance critical problems. The first one generates the permutations of S, whereas the second one flips each permutation until $k = 1$.

The first problem is solved in an iterative way, since generating the permutations recursively would put a heavy pressure on the stack even for small n's. The algorithm implemented can be expressed with the pseudo code in Listing 11.

The second problem relates to flipping a permutation until the pattern $k = 1$ emerges. The approach taken simply involves iterating over all k elements once, swapping places with the elements in an outer-to-inner fashion. This process is repeated until $k = 1$.

As such, the flipping operation can be replaced with the line *print P* in Listing 11, allowing for generating a permutation, calculating its flip count and then repeating the process until all permutations have been flipped and a maximum flip count has been obtained.

3.2.3 N-body

The N-body implementation provided by Lewis and Whipkey [22] is used in this study. The program takes in an integer n, representing the number

```
fun void generate-permutations(int n)
  P := [1,2,...,n]
  r := n //Current rotation depth
  count := int[n] //Control structure used for keeping
                  //track of rotation depths.

  while (true) do
    while (r != 1) do
      count[r-1] := r
      r--

    print P

    while true do
      if (r == n) then //Base case, no more rotations
        return

      rotate first r digits in P to left (wrap-around)
      count[r] := count [r] - 1

      if count[r] > 0 then
        break
```

Listing 11: Pseudo code for permutation generation

of advancements that are to be computed for the planet simulation. The task is to output the total amount of energy in the planet system after n advancements.

The planets are represented by a `Body` class. Each planet is made up by seven fields of type `double`, three position fields for the three dimensions x, y, z, the current velocity in each of the three directions as well as the mass of the planet. These fields are initialised to pre-defined values.

The simulation starts with offsetting the momentum of the centre of gravity, namely the sun. Using a time step of 0.01, the planets are advanced n times where planets with higher mass inflict forces upon those of lesser mass causing a change in velocity of the planets. The actual physic calculations used are not further described by CLBG but involves simple floating point operations. After n advancements, the total amount of energy in the planet system is outputted.

3.2.4 Reverse-complement

The Reverse-complement implementation used in this study is provided by Lei [21]. The source code has been modified slightly to use an InputStream from file instead of `stdin`.

The program gets the Fasta output as an `InputStream`, consisting of three DNA sequences for which the reverse complement should be computed. A byte array is used for translating nucleotides to their complements. The array consists of the first 128 ASCII characters, stored as bytes where the nucleotides with DNA complements have been replaced with their corresponding complement.

The implementation uses a custom buffered reader and writer to reduce time on I/O. After a read-operation has been performed, the bytes read are translated and appended to a byte array for later output. Since a whole sequence is needed to be read and translated before it can be reversed and outputted, the benchmark is memory intensive for large sequence inputs. Once a whole DNA sequence has been read and stored in `data`, it is then copied over in reverse order to `outputBuffer` and written to `stdout`. This process is repeated for all three sequences.

3.3 Kotlin Implementations

As mentioned in Section 3.1, the support of Kotlin in CLBG is lacking. Therefore, original Kotlin implementations have been developed for each benchmark studied. As implementations can differ widely, several factors have been considered as to ensure the validity of the results.

First, Marr, Daloze, and Mössenböck [25] identified that CLBG sometimes impose too few restrictions on the benchmark implementations, causing too great of a variety in the approaches taken. Problems of such character can be observed for Fannkuch-Redux, where the algorithm for permutation generation is not strictly defined.

Second, considering Kotlin has an outspoken design goal of being pragmatic, the implementations should to some degree reflect this. Although Kotlin code can be written as low-level as Java, it is usually not common practice. Instead, higher-level constructs and features are used.

With both of these factors in mind, this study has set on using two groups of Kotlin implementations for each benchmark. The first group, called `kotlin-converted`, consists of auto-converted implementations based on the Java benchmark implementations. Using the auto-converter tool provided by JetBrains, Java code can be converted to Kotlin. This allows us to study the performance overhead of using Kotlin in a performance critical context, and further means that the exact same algorithms are used to avoid the threat identified in [25].

The second group, called `kotlin-idiomatic`, consists of original implementations. Although inspiration has been taken from the Java implementations used in this study, the idiomatic implementations favour pragmatism over performance. This allows us to study the performance and behaviour of Kotlin in a context better reflecting how the language is commonly used.

The correctness of the implementations of `kotlin-converted` and `kotlin-idiomatic` have been verified using the sample input and output provided by CLBG for each benchmark, ensuring that the program given a particular input produces the expected output.

In the subsequent subsection, descriptions of the architecture of both Kotlin implementations for each benchmark are provided. However, considering `kotlin-converted` the descriptions focuses on the potential deviations from the auto-coversions. Furthermore, the Kotlin source code for both `kotlin-converted` and `kotlin-idiomatic` is available in full in Appendix B.

3.3.1 Fasta

The `kotlin-converted` implementation has been auto-converted from the corresponding Java implementation using the tool provided by Android Studio 3.0.1. No additional modifications to the source code has been made. Therefore, the `kotlin-converted` uses the same data structures, classes and buffer size as the corresponding Java implementation.

The `kotlin-idiomatic` implementation uses the same pseudo-random function for selecting nucleotides from the `IUB` and `HOMO_SAPIEN` alphabets. However, to represent the alphabets a data class `AminoAcid` is used for storing the nucleotide and their corresponding probability.

Additional differences includes using a more pragmatic approach when outputting the sequences. Instead of a byte array buffer and an OutputStream, as used by the java and kotlin-converted implementation, kotlin-idiomatic uses a BufferedWriter, which is flushed each 100 lines. As such, instead of storing the nucleotides as bytes, they can be stored as chars and appended to the BufferedWriter directly.

3.3.2 Fannkuch-Redux

No major alterations have been made to the kotlin-converted implementation and therefore, it closely follows the corresponding Java implementation.

For the kotlin-idiomatic implementation, the same algorithm for permutation generation has been used due to its practical implications allowing generations on the fly. Although a recursive implementation could have been used, this would have required storing all permutations in memory.

However, for flipping of the first k elements, a more functional approach has been implemented, relying on the take() and reversed() operations on the permutation array instead of looping through the first k elements in an outer-to-inner fashion as with the Java implementation. This goes hand-in-hand with the pragmatical language design of Kotlin.

3.3.3 N-body

The kotlin-converted implementation has been auto-converted from the corresponding Java implementation. No major alterations has been made to the source code.

For the kotlin-idiomatic implementation, a more pragmatic approach has been taken when it comes to representing the planets, where a data class Body has been used consisting of the seven planet properties (coordinates, velocities and mass) of type Double. Further differences includes using the annotation @JvmField to instruct the compiler to don't generate getter and setter functions for the public properties, as this would lead to a function call overhead when frequently manipulated by other methods. However, due to the computational characteristics of the N-body benchmark, seemingly few differences exists between the implementation and the corresponding `java` and `kotlin-converted` implementations.

3.3.4 Reverse-Complement

The `kotlin-converted` implementation has been auto-converted from the Java implementation provided by [21]. Some alterations have been made to the auto-generated implementation, including changing internal declarations to private, using `toInt()` to allow for lookup of characters in `transMap` as well as replacing an object declaration of ReverseComplement with a class. The architecture and data structures used remain the same as in the original implementation.

The `kotlin-idiomatic` implementation takes on a more pragmatic approach. Instead of using a byte array of ASCII characters for translation, a `HashMap<Char,Char>` is used, storing only the nucleotides and their complements. Furthermore, instead of implementing a custom buffered reader and writer, the `BufferedReader` class is used to read from the `InputStream` and `BufferedWriter` is used to output to `stdout`.

A lambda expression is used to parse each line from the `BufferedReader`, computing the complement of each nucleotide. The complements are stored in a `MutableList<Char>` data structure. Once a complete complement sequence has been calculated, it is traversed in reverse order and appended to a `BufferedWriter` and outputted to `stdout`. The data structure storing the complements is then emptied to make room for the next sequence. This processes is repeated until all the reverse complements of all DNA sequences have been outputted. The default buffer sizes have been used for both the `BufferedReader` and `BufferedWriter`, however, to avoid untimely flushes potentially breaking the 60 nucleotides per line format, the `flush` method is invoked every 100 lines (6100 bytes) to empty the buffer before it is full.

3.4 Metrics

This study has set on using both static and dynamic metrics to study the behaviour and performance of Kotlin and Java on ART. The metrics used are based on previous work, mainly inspired by [23] and [32], as described in Section 2.6. However, this study also utilises the profiling support offered by the Android environment for dynamic metrics.

Since previous studies heavily relied upon instrumented versions of the JVM, some of the metrics used are not easily obtainable on ART. Metrics relying on VM instrumentations include, but is not limited to; object lifetimes (where the virtual machine has been modified to record the allocation and deallocation time of all objects) and dynamic n-gram analysis (where modifications have been made to record all executed bytecode instructions). These metrics are considered outside the scope of this study, since obtaining them would require modifications to the runtime system and thus extensive knowledge of ART. Furthermore, some metrics are not relevant due to architectural differences between JVM and ART.

However, instead of performing n-gram analysis in a dynamic context, this study performs the analysis in a static context by using the AOT compiled bytecode of each implementation to measure the relatedness of Java and Kotlin bytecode on ART. This static bytecode metrics used are further described in Section 3.4.5.

Android supports performance profiling, both through the interactive Android Profiler tool in Android Studio as well as a several debugging methods. Considering the benchmarks are to be executed multiple times there is a need for automatic data collection. As such, performance data is collected through application instrumentation using the `android.os.Debug`, `android.os.MemoryInfo` as well as `android.os.SystemClock`.

Taking previous studies as well as the supported profiling methods of Android API 26 into consideration, the following categories of dynamic metrics are used to evaluate the performance of Kotlin and Java on ART: runtime, garbage collection, memory consumption and boxing of primitives. The dynamicnmetrics and the method for obtaining them are further described in Section 3.4.1-3.4.4.

3.4.1 Runtime metric

Runtime evaluates the execution time of a benchmark. However, care needs to be taken since Android supports multiple functions for time measurements, and not all are monotonic or continuous with respect to the resolution of time. A monotonic property of a clock refers to the fact that the time never decreases, for instance as a result of when a user changes the time. Continuous with respect to the resolution of time refers to the fact that the time only can increase by one time resolution unit every tick. In this study the `elapsedRealtimeNanos()` method from `android.os.SystemClock` is used, which provides the monotonic and continuous time in nanoseconds since system boot. The runtime is then measured by calculating the difference before and after an execution of a benchmark.

3.4.2 Memory consumption metrics

On Android, the memory count is divided into several different categories. In this study, memory consumption refers to the heap memory used by the benchmark corresponding to object allocations. This statistic is obtainable through the `getMemoryStat()` method in `android.os.Debug.MemoryInfo`. The memory consumption metrics used in this study consists of object allocation count, byte allocation count as well as a bytes per object ratio. The metrics are measured as the difference between the start and the end of a benchmark execution.

3.4.3 Garbage collection metrics

Garbage collection is closely related to the performance of an application. Time used by the garbage collector to reclaim unused memory could otherwise have been spent on executing application code. Although unobtainable on ART, object lifetimes have been used in previous studies to identify how well a language makes use of the cheaper generational garbage collector, where a large number of short lived objects means that memory can be reclaimed faster.

On Android, it is possible to obtain more detailed garbage collection data through instrumentation using the `getRuntimeStat()` method in `android.os.Debug`. Four metrics are used in this study related to garbage collections, namely garbage collection count, average time spent on garbage collection, total number of bytes freed as well as number of bytes freed per garbage collection.

3.4.4 Boxing of primitives metric

Boxing of primitives relates to the conversion between primitive data types and their corresponding wrapper classes. For instance, Java provides a primitive data type `int` with a corresponding wrapper class `Integer`.

Conversely, Kotlin only provides wrapped data types. Although the wrapper classes allows developers to write cleaner code, the boxing operation is expensive since it creates additional heap pressure [32].

The boxing of primitives metric has been used both by Sarimbekov et al. [32] and Li, White, and Singer [23]. In [32], the metric used refers both to the boxed primitives allocated and the boxed primitives requested (by calls to the `valueOf()` method for the wrapper classes) whereas in the case of [23] only the former metric has been used. In this study, boxing is measured through the `startMethodTracing()` method in the `Debug` class where the metric represents the amount of requested primitives as a percentage of total method calls.

3.4.5 Bytecode metrics

The bytecode metrics used in this study refers to the relatedness of the benchmark programs. The metrics are inspired by those used in [23], but in a static context due to limitations of ART. The metrics consists of measuring the total and unique bytecode n-grams of each implementation as well as the n-gram coverage. This study uses unigrams, bigrams and trigrams. Furthermore, as in [23], the relative frequencies of each n-gram are calculated to which PCA is applied to representing the implementations visually by plotting the two first principal components.

3.5 Environment and setup

3.5.1 Software and tools

Benchmark application

In order to collect dynamic benchmark data for the Kotlin and Java benchmark implementations, an Android application has been developed using Android Studio. The benchmark application allows the user to choose the benchmark type, implementation and metric as well as the number of benchmark iterations as presented in Figure 3.1.

The application creates a thread used for benchmarking using the `android.os.ThreadHandler` and `android.os.Handler` classes. The benchmark thread has its thread priority set to the highest available for Android processes, ensuring a CPU execution time of roughly 95% [16].

Once all iterations of the benchmark has been executed, the results are reported back to the main thread where an asynchronous task is created on a background thread using `android.os.AsyncTask` in order to log the benchmark results to file for analysis.

The use of a separate benchmark thread and a background thread ensures that other tasks related with the application, such as UI handling does not inflict with the execution of the benchmarks. This is particularly important for the Runtime, Memory and Garbage Collection metrics in order to provide reliable results.

N-gram application

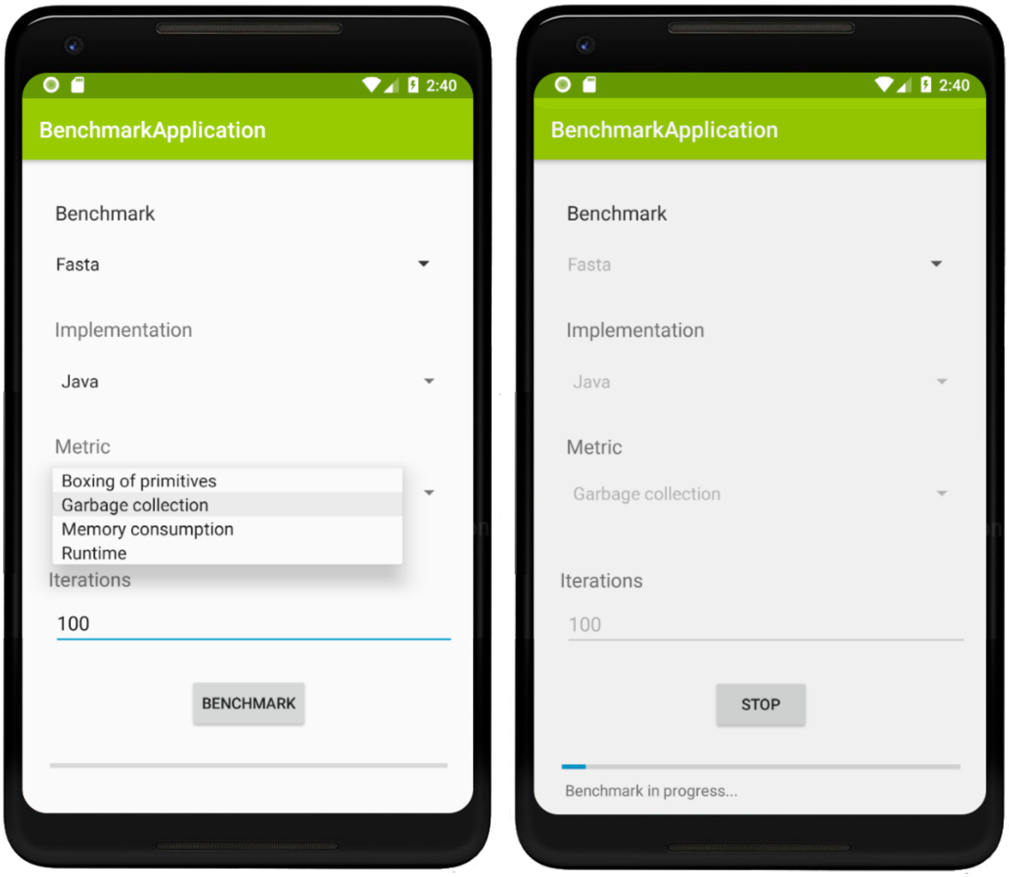
In order to extract the n-grams of the generated `.dex`-files associated with each benchmark implementation, a custom Kotlin application is used. The program accepts 13 arguments, where the first argument is an integer representing n. The remaining 12 arguments represents the `.dex`-files associated with the benchmark implementations. First, the application performs lexical analysis on the bytecode files, forming a sequence of bytecode operation codes. The N-gram applications supports a total of 230 different bytecode instructions. Then, the n-grams are formed by iterating on the bytecode sequence. However, as in [23], n-grams belonging to different blocks are discarded.

For the resulting valid n-grams, the total and unique counts are calculated. Furthermore, to compute the n-gram coverage these figures are put in relation to the other implementations belonging to the same benchmark type.

The application outputs two files, where one is a `.csv`-file consisting of a matrix with the benchmark implementations (samples) as columns and n-grams as rows where each cell contains the relative frequency of that particular n-gram for the implementation. The second file contains the unique and total n-gram count of each implementation as well as the n-gram coverage for each implementation per benchmark type. The source code of the application is available in Appendix C, whereas the generated result files are available in Appendix D.5.

Principal Component Analysis tool

In order to perform principal component analysis on the data set obtained from the N-gram application, the online data analysis tool ClustVis [26] has been used. The data is inputted as a `.csv`-file with implementations as columns and the n-grams as rows, where each cell contains the relative frequency of the specific n-gram for the implementation. The tool calculates each of the 12 principal components for each of the implementations, where the first two components representing the greatest variance are plotted in a 2-dimensional scatter plot.



Benchmark
Fasta
Implementation
Java
Metric
Iterations
100

Benchmark
Fasta
Implementation
Java
Metric
Garbage collection
Iterations
100
Benchmark in progress...

Figure 3.1: Android benchmark application

Software versions

The software and environment versions presented in Table 3.2 have been used throughout this study for development and benchmarking.

Software	Version
Android Studio	3.0.1
Android API	26
Java JRE	1.8.0_111
Kotlin	1.2.21

Table 3.2: Software versions

3.5.2 Hardware

As previously mentioned in Section 1.3, this study is not concerned with comparing the performance of different mobile devices. Hence, the benchmarks are executed on a single Android device. Due to the support of various new profiling features, Android 8.0 has been chosen as the target version of Android and is, as of the writing of this report, the latest stable release. The specifications of the device which the benchmarks are executed on is presented in Table 3.3.

Device	Samsung Galaxy S9
Model number	SM-G960F
OS	Android 8.0.0
CPU	Exynos 9810 (Octa-core)
RAM	4GB
GPU	Mali G72 MP18

Table 3.3: Specifications of the benchmarking device [31]

3.5.3 Data collection and analysis

Preparation

To collect the benchmark data, the Android application described in Section 3.4.1. is used. Before starting the benchmark application, all other running applications are terminated. Furthermore, all unnecessary services such as WiFi, Location, GPS, Bluetooth and others are turned off as to minimise the impact from background processes and services.

Each metric used in this study is measured in isolation, meaning that throughout the benchmark execution only one metric is measured at a time. This ensures that the overhead associated with measuring does not skew the results. Therefore, for four benchmarks, three implementations per benchmark and four metrics studied, there exists 48 instances for which results are obtained.

Workloads

To cater for the time needed to obtain the results for the 48 benchmarking instances, the workloads of each benchmark needs to be adapted to the specific device on which they are measured. Although CLBG has defined a recommended workload for each benchmark, these have been produced specifically for the JVM with a desktop/server platform in mind. Considering that the resources on modern mobile devices, such as the one used in this study, generally are more limited than on desktops or servers, the workloads used in this study deviates from CLBG to suit ART and the hardware limitations of the benchmarking device.

To adapt the workload, a limitation of an execution time of roughly one second is defined for any benchmark implementation. The goal is, however, not to find a workload which perfectly fits into this limitation but rather to use it as a guide in order to ensure that the samples can be obtained in reasonable time. Therefore, a trial-and-error process has been applied to find appropriate workloads, where the ones used in this study are presented in Table 3.4 along with the recommended workloads within parenthesis.

Furthermore, the maximum execution time of one second does not take into account the additional overhead of measuring a particular metric. For instance, obtaining results for boxing of primitives adds substantial pressure on the runtime, sometimes resulting in an execution time of several minutes.

Benchmark	**Workload**
Fasta	350000 (25000000)
Fannkuch-Redux	9 (12)
N-body	350000 (50000000)
Reverse-Complement	225000 (25000000)

Table 3.4: Workloads used for each benchmark (CLBG defaults)

Statistical method

Although precautions have been taken to strengthen the results, some factors are uncontrollable as described in Section 2.4.2. Therefore, each benchmark instance needs to be executed several times and expressed as a sample mean with a confidence interval to ensure statistically sound results. To calculate sample means, two methods are used. The harmonic mean, presented in Formula 2.2, is used for the number of bytes freed per garbage collection as well as bytes allocated per object, since these metrics are rate-based. For all other metrics, the arithmetic mean is used, as presented in Formula 2.1.

The sample size used in this study is $n = 100$ and is considered a suitable trade off between a sufficiently large sample size and the time required to obtain the results. However, since there generally is an additional overhead of executing a program the first few times, known as the warmup time, these samples are unrepresentative of a programs performance. By observing the runtime behaviour of the applications, an overhead could be observed for the first 2-5 iterations. Hence, to allow for some margin, the first ten benchmark iterations are discarded to allow for more representative and stable results. Each benchmark is executed 110 times, where the samples consist of the results for iteration $10 - 110$.

To compute the standard deviation, needed to obtain a confidence interval, Formula 2.3 is used. Furthermore, a two-sided confidence interval is used as expressed in Formula 2.5, as we are not interested in evaluating whether a particular language performs better or worse than the other, but only whether there is a performance difference between Kotlin and Java. The significance level used in this study is $\alpha = 0.05$ providing a confidence level of 95%. This gives us that $z_{1-\alpha/2}$ in Formula 2.5 is $z_{0.975}$ which corresponds with the value 1.96. Hence, the confidence interval can be expressed as:

$$\bar{x} \pm 1.96 \frac{s}{\sqrt{100}}$$

If the confidence intervals of the results of two implementations are not overlapping, we can conclude that there is a performance difference with a confidence level of 95%.

The statistical method described above is used for all benchmark metrics apart from boxing of primitives since it is considered infeasible from three perspectives. First, there is a substantial runtime overhead of method tracing. Second, each produced `.trace` requires roughly `8.5MB` of storage, or `10.2GB` for all benchmarks and implementations. Third, manual analysis is needed in order to obtain the number of invocations of `toValue()` methods. Furthermore, a variation in the number of unboxing invocations should only be observed as a result of JIT optimisations. Therefore, for the boxing of primitives metric, each benchmark is executed 100 times. However, method tracing is only recorded for the first, middle and last execution to observe differences as a result of JIT compiler optimisations.

Chapter 4

Results

This chapter presents the results obtained for the metrics studied on each benchmark implementation. The results are presented in a by-metric-order, beginning with the runtime followed by memory consumption, garbage collection, boxing of primitives and bytecode results. Further analysis of the results is presented in Section 5.

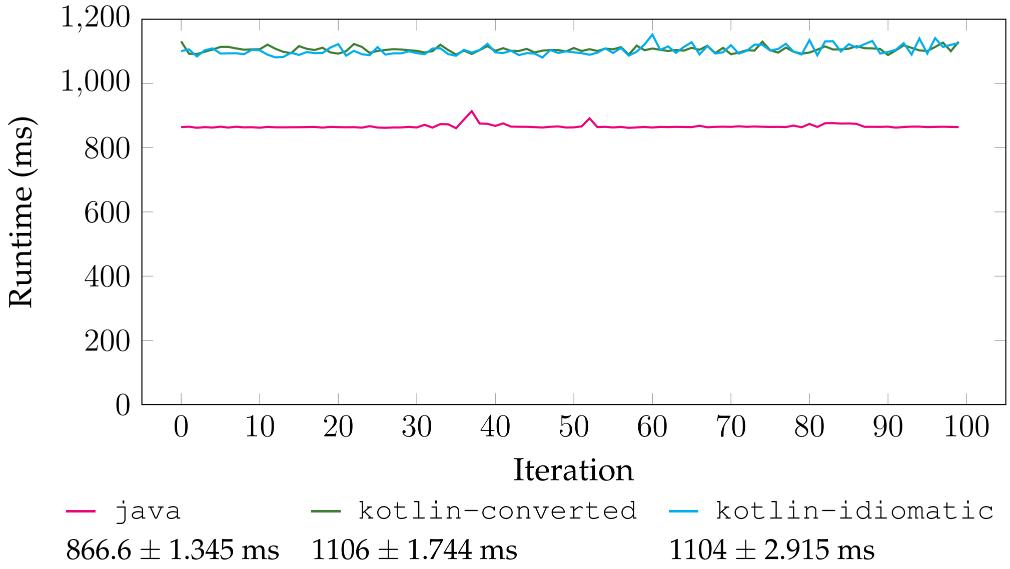
4.1 Runtime

As described in Section 3.5.3, all benchmarks and implementations have been executed 110 times, from which the first ten executions are regarded as warmup and discarded. Hence, the last 100 executions have been used as the samples to represent the runtime of each implementation. The runtime of each implementation is presented in a line chart, along with the sample mean and 95% two-sided confidence interval with a resolution of four significant figures. This allows us to reason whether or not the results are statistically significant. Raw data for the runtime results of all benchmark implementations, along with the warmup executions, sample means, standard deviations and confidence intervals are available in full in Appendix D.1.

Fasta

The runtime results of Fasta are presented in Figure 4.1 where `java` is the fastest implementation by a margin of around 250 milliseconds. The `kotlin-idiomatic` implementation has a smaller sample mean but a larger standard deviation than `kotlin-converted`. However, since the confidence intervals of both Kotlin implementations are overlapping, there is no significant performance difference between the two.

Figure 4.1: Runtime of Fasta

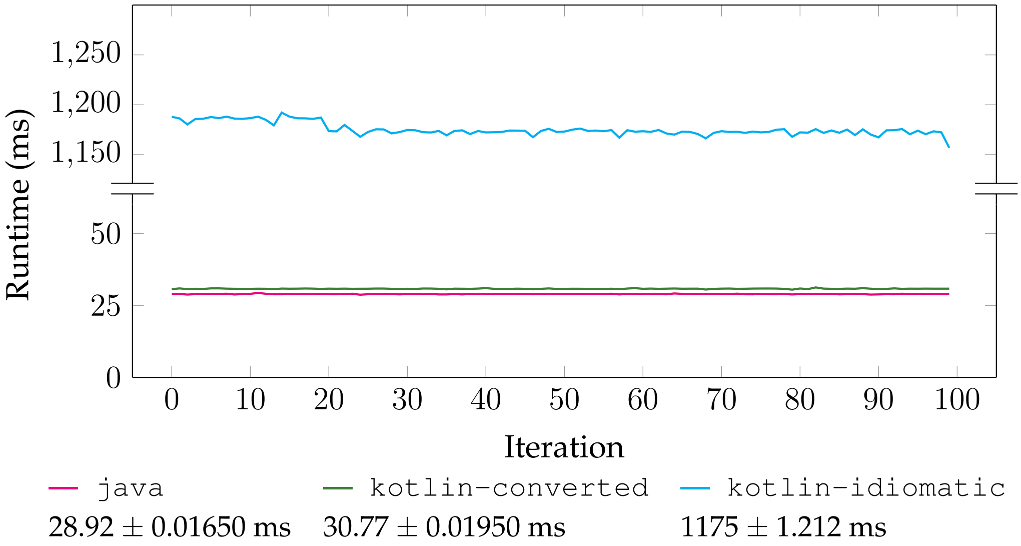


Fannkuch-Redux

For the Fannkuch-Redux benchmark, the runtime of `java` and `kotlin-converted` are similar, with `java` performing slightly better. However, the runtime of `kotlin-idiomatic` is notably slower than the two other implementations, by a factor of around 40. Furthermore, the confidence interval for `kotlin-idiomatic` is larger than for the other implementations due to a larger standard deviation.

Due to the non-overlapping confidence intervals, there is a strong indication that the ranking of the runtime performance for each implementation holds true, with Java being the fastest language by a small margin.

Figure 4.1: Runtime of Fannkuch-Redux



N-body

The runtime results of the N-body benchmark are presented in Figure 4.3. `java` exhibits the fastest runtime of the implementations, but by a small margin compared to `kotlin-idiomatic`. However, due to the non-overlapping CIs, this gives a strong indication that the ranking between the implementations holds true. For the `kotlin-converted` implementation, the runtime is slower by a factor of around 15. This is a result of inefficient conversion, which is further discussed in Section 5.1.

Reverse-complement

For the Reverse-complement benchmark, as depicted in Figure 4.4, `java` performs better than both Kotlin implementations by a factor of two. Furthermore, there is a significant difference between the two Kotlin implementations, with `kotlin-idiomatic` being roughly 250 milliseconds slower than `kotlin-converted`. Furthermore, both Kotlin implementations exhibits a large standard deviation and as a result a large CI. This is a result of a spike of around 500 milliseconds for 20 samples.

Figure 4.3: Runtime of N-body

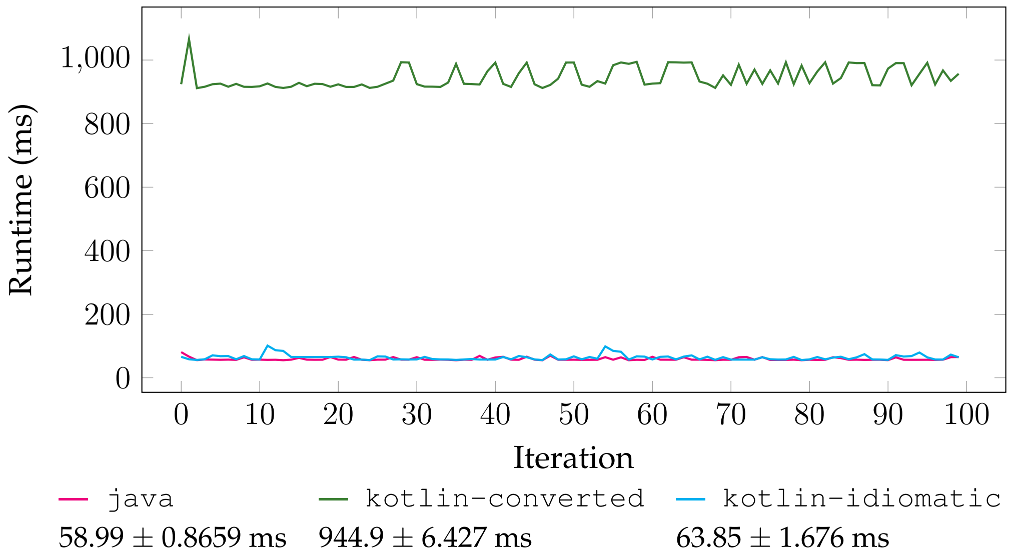
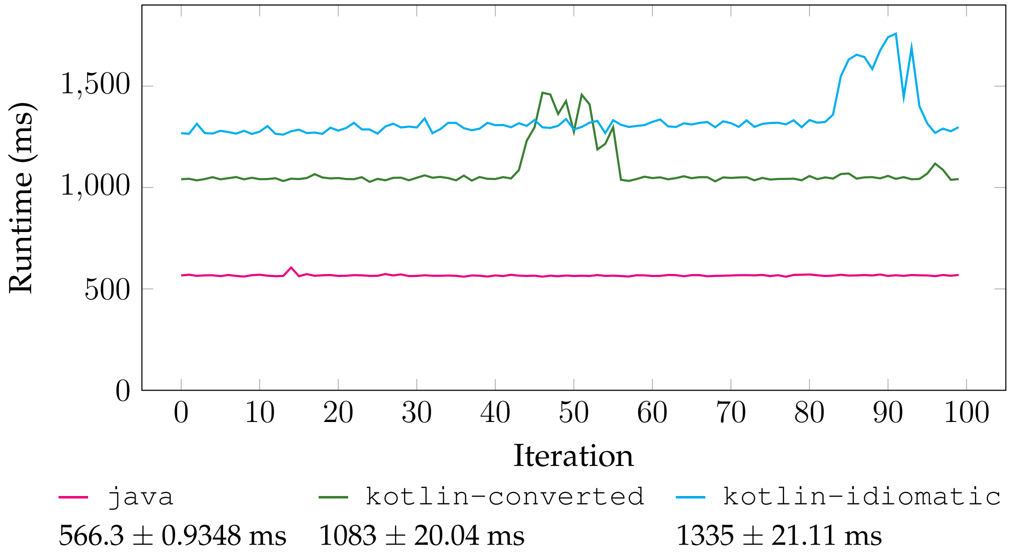


Figure 4.4: Runtime of Reverse-complement



4.2 Memory consumption

The memory consumption results consists of the three metrics described in Section 3.4.2, namely object count, allocation count and bytes allocated per object. All metrics are presented in the way of a sample mean with a confidence interval. Furthermore, all metrics apart from object count are presented in bytes. The standard deviation for all benchmark implementations and metrics remained at 0.0, which is seen as a consequence of precise measuring as well as the fact that no memory affecting JIT compilations occurred during the execution of any of the benchmark programs. All results are presented with a resolution of four significant figures, where raw data for all memory consumption results are available in Appendix D.2.

Implementation	**Object Count**	**Total size**	**Size/Object**
`java`	175030 ± 0.0	6731192 ± 0.0	38.46 ± 0.0
`kotlin-converted`	175030 ± 0.0	6731192 ± 0.0	38.46 ± 0.0
`kotlin-idiomatic`	175686 ± 0.0	6593616 ± 0.0	37.53 ± 0.0

Table 4.1: Memory consumption of Fasta (object count, total size and size/object in bytes)

Fasta

The memory consumption results of the Fasta benchmark are presented in Table 4.1. The result of `java` and `kotlin-converted` are identical with regards to the object count, total size and size per object metrics. The `kotlin-idiomatic` implementation allocated 656 objects more than the other two implementations. However, it consumed roughly 134.35 kB less than both `java` and `kotlin-converted`, resulting in a smaller bytes per object ratio of 0.9267. Thus, it is clear that the memory consumption of all implementations for the Fasta benchmark are in the same order of magnitude.

Implementation	Object Count	Total size	Size/Object
`java`	10 ± 0.0	384 ± 0.0	38.40 ± 0.0
`kotlin-converted`	13 ± 0.0	456 ± 0.0	35.08 ± 0.0
`kotlin-idiomatic`	9557541 ± 0.0	2.867×10^8 ± 0.0	29.99 ± 0.0

Table 4.2: Memory consumption of Fannkuch-Redux (object count, total size and size/object in bytes)

Fannkuch-Redux

The results of memory consumption for Fannkuch-Redux are depicted in Table 4.2. The `java` implementation exhibits the smallest object count and byte allocation. `kotlin-converted` allocates three more objects than `java`, resulting in an increased memory consumption of 72 bytes but a lower bytes per object ratio of 3.32 bytes. As such, the memory consumption results of both `java` and `kotlin-converted` are in the same order of magnitude.

However, the results for `kotlin-idiomatic` stand out, allocating almost one million more objects than both other implementations. Furthermore, there is a drastic increase by a factor of 10^6 of the total number of bytes allocated. However, `kotlin-idiomatic` exhibits a lower bytes per object ratio than the other implementations.

Implementation	Object Count	Total size	Size/Object
`java`	91 ± 0.0	2704 ± 0.0	29.71 ± 0.0
`kotlin-converted`	91 ± 0.0	2704 ± 0.0	29.71 ± 0.0
`kotlin-idiomatic`	97 ± 0.0	2880 ± 0.0	29.69 ± 0.0

Table 4.3: Memory consumption of N-body (object count, total size and size/object in bytes)

N-body

The result of the memory consumption for the N-body benchmark is presented in Table 4.3. As with the Fasta benchmark, the results of `java` and `kotlin-idiomatic` are identical. Furthermore, `kotlin-idiomatic` once again exhibits a larger object count and a larger total size than the other implementations. However, `kotlin-idiomatic` still uses fewer bytes per allocated object than both `java` and `kotlin-converted`. As such, it is clear that the memory consumption results of all implementations are within the same order of magnitude.

Implementation	Object Count	Total size	Size/Object
`java`	113009 ± 0.0	4235304 ± 0.0	37.48 ± 0.0
`kotlin-converted`	113009 ± 0.0	4235304 ± 0.0	37.48 ± 0.0
`kotlin-idiomatic`	196798 ± 0.0	$4.237 \times 10^7 \pm 0.0$	215.3 ± 0.0

Table 4.4: Memory consumption of Reverse-complement (object count, total size and size/object in bytes)

Reverse-complement

For the Reverse-complement benchmark, the memory consumption results are presented in Table 4.4. As with both Fasta and N-body, both `java` and `kotlin-converted` exhibits identical memory consumption results. For `kotlin-idiomatic`, 83789 more objects are allocated than for the other two implementations. Furthermore, roughly ten times more memory is used compared to `java` and `kotlin-converted`.

4.3 Garbage collection

The results of garbage collection are presented using the metrics described in Section 3.4.3. The garbage collection count metric are presented using the method described in 3.5.3. However, all other metrics are dependant on that at least one GC event has been triggered during an iteration. Therefore, the iterations where no GC event has occurred are excluded from the samples of those metrics. Furthermore, the N-body benchmark allocated too few bytes to trigger a GC event for any implementation and iteration and has, as such, been excluded from this section. However, the raw data for the garbage collection results can be found in Appendix D.3. Additionally, all results are presented with a resolution of four significant figures.

Fasta

The garbage collection results of Fasta are presented in Table 4.5. For the GC count metric, all implementations performs roughly one GC event per iteration. Due to the overlapping confidence intervals, one can not conclude that there is a significant performance difference between the languages. The same goes for the total number of bytes freed by the garbage collector, where the confidence intervals are overlapping. This is not surprising, considering that all implementations exhibited similar memory consumption results.

Although there is no significant difference when it comes to the number of GC events triggered as well as the total number of bytes freed, there is a significant performance difference when it comes to the time spent by the garbage collector on reclaiming the objects. The garbage collector spent the least amount of time reclaiming bytes allocated by the `java` implementation, whereas the garbage collector spent roughly 11 ms more on `kotlin-idiomatic` and 14.5 ms more on the `kotlin-converted` implementation. The fact that the garbage collector spent less time on `kotlin-idiomatic` can be seen as a result of that the total number of bytes allocated was less than `kotlin-converted`.

Fannkuch-Redux

The results of Fannkuch-Redux are presented in Table 4.6. The `java` and `kotlin-converted` implementations did not trigger any GC events, which is not surprising considering the low amount of objects and bytes allocated by the implementations. The `kotlin-idiomatic` implementation, on the other hand, triggered over 30 garbage collections per iteration, spending 261.7 ± 15.18 ms (or $20 - 25\%$) of the runtime on garbage collection. This shows that apart from the overhead that comes with allocating objects, there is also a large overhead associated with reclaiming the bytes from the heap, especially if the objects are short-lived as in the case of the Fannkuch-Redux benchmark.

Reverse-complement

For the Reverse-complement benchmark, the garbage collection results are presented in Table 4.7. `kotlin-converted` triggers the fewest amount of GC events, with `java` close behind by a small margin. However, `kotlin-idiomatic` triggers significantly more GC events by a factor of five. The GC reclaims the fewest amount of bytes for the `java` implementation and also spends the least amount of time on garbage collection. The GC for `kotlin-converted` reclaims roughly 12% more bytes in total than `java`, but exhibits a larger runtime overhead by a factor of 2. Unsurprisingly, the GC reclaims around ten times as many bytes for the `kotlin-idiomatic` implementation, and as a result also spends more time on garbage collection. Thus, java spends the least time on reclaiming memory per garbage collection, with kotlin-converted on the second place with an increase by a factor of 2, whereas kotlin-idiomatic uses roughly 40 ms more per garbage collection than java.

Implementation	GC Count	Time (ms)	Bytes freed	Bytes freed/GC	Time/GC
`java`	1.090 ± 0.05640	7.460 ± 0.3993	6583858 ± 342830	6038467 ± 5206	6.816 ± 0.07630
`kotlin-converted`	1.090 ± 0.05640	21.97 ± 1.831	6623904 ± 351443	6070224 ± 10313	17.82 ± 1.292
`kotlin-idiomatic`	1.070 ± 0.05030	18.42 ± 1.436	6659836 ± 317950	6220726 ± 5920	15.33 ± 0.9851

Table 4.5: Garbage collection results for Fasta

Implementation	GC Count	Time (ms)	Bytes freed	Bytes freed/GC	Time/GC
`java`	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
`kotlin-converted`	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
`kotlin-idiomatic`	31.43 ± 1.211	261.6 ± 15.18	2.874 $\times 10^8$ ± 1179972	9148710 ± 902715	7.687 ± 2.0800

Table 4.6: Garbage collection results for Fannkuch-Redux

Implementation	GC Count	Time (ms)	Bytes freed	Bytes freed/GC	Time/GC
`java`	0.6800 ± 0.09190	9.147 ± 0.8185	6240311 ± 6774	6240123 ± 6774	8.0933 ± 0.8446
`kotlin-converted`	0.6100 ± 0.09610	21.43 ± 1.536	6996802 ± 17873	6996803 ± 17874	18.52 ± 1.640
`kotlin-idiomatic`	3.100 ± 0.07100	177.3 ± 11.20	4.246 $\times 10^7$ ± 701705	13656161 ± 298282	47.36 ± 4.466

Table 4.7: Garbage collection results for Reverse-complement

4.4 Boxing of primitives

As described in Section 3.5.3, for the boxing of primitives metric, a `.trace`-file is generated for three of the 100 executions and manually analysed extracting the unboxing method invocations of the wrapper classes. The median of the three unboxing method invocation counts for each primitive type is used to represent the amount of unboxing for each benchmark implementation. Both the Fasta and N-body results were found uninteresting due to the few unboxing calls made for all implementations. The results of those benchmarks, along with the raw data of all method traces are available in Appendix D.4.

Fannkuch-Redux

For the Fannkuch-Redux benchmark, both `java` and `kotlin-converted` did not exhibit any unboxing method invocations for any of the primitives. However, `kotlin-idiomatic` made significantly more unboxing invocations for the integer datatype. Further analysing the `.trace`-file indicates that all unboxing invocations were a result of the `take()` function, used for flipping the permutations. The overhead associated with performing the unboxing was measured to 3.0% of the execution time. However, due to the overhead associated with method tracing, these results should not be compared with the runtime results directly. For the other primitives, no unboxing calls were made by the `kotlin-idiomatic`. This is not surprising considering the characteristics of the benchmark, as described in Table 3.1.

Implementation	Byte	Char	Double	Float	Int	Long	Short
`java`	0	0	0	0	0	0	0
`kotlin-converted`	0	0	0	0	0	0	0
`kotlin-idiomatic`	0	0	0	0	17908	0	0

Table 4.8: Unboxing method invocation medians for Fannkuch-Redux

Reverse-complement

The Reverse-complement benchmark exhibits a similar behaviour as Fannkuch-Redux, where both `java` and `kotlin-converted` did not perform any unboxing invocations for any of the primitives. The `kotlin-idiomatic` implementation, on the other hand, had a median of 25915 invocations to the `toValue()` method for the `Character` datatype. Further inspecting the method traces indicates that these calls occur in the `read()` function when parsing the inputted characters using a lambda expression consisting of `useLines()` and `forEach()`. The observed overhead associated with unboxing is 4.4% of the total runtime.

Implementation	Byte	Char	Double	Float	Int	Long	Short
`java`	0	0	0	0	0	0	0
`kotlin-converted`	0	0	0	0	0	0	0
`kotlin-idiomatic`	0	25915	0	0	0	0	0

Table 4.9: Unboxing method invocation medians for Reverse-complement

4.5 Bytecode analysis

The bytecode is extracted from each implementation by parsing the AOT compiled `.dex` files. The 1-, 2- and trigram sequences are then computed from which the metrics described in Section 3.4.5 are extracted. Furthermore, using ClustVis, the principal components are calculated for each implementation and n-gram and are visualised using scatter plots. The bytecode results for the unigrams, bigrams and trigrams are presented in the subsequent subsections. Furthermore, the raw data of the bytecode results are available in Appendix D.5.

4.5.1 Unigram results

The unigram results represent individual bytecode instructions, where the bytecode sequence results are presented in Table 4.10. For the Fasta implementation, although `kotlin-idiomaic` produces the most bytecode instructions in total, it produces the least amount of varied bytecode with 54 unique instructions. This can be compared with `java` with a similar unique instructions count but fewer total instructions by a factor of 2. For both Fannkuch-Redux and N-body, `java` exhibits the least amount of total and unique instructions, whereas `kotlin-idiomatic` contains the most.

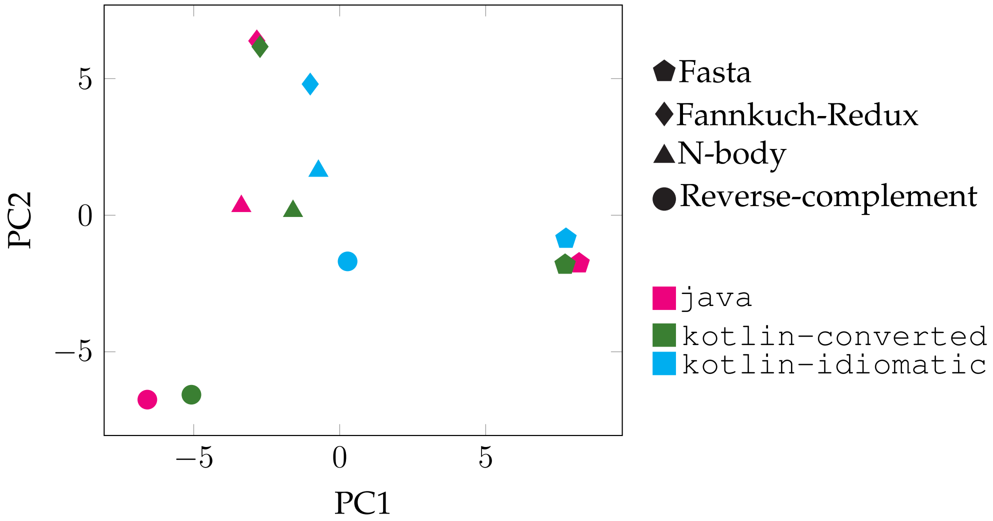
For the Reverse-complement benchmark, `kotlin-idiomatic` exhibits the fewest unique and total bytecode instructions, with `java` close behind. `kotlin-converted`, on the other hand, contains significantly more bytecode instructions in total by a factor of $2.5 - 3.5$ compared to the other implementations.

Benchmark	**Implementation**	**Unique***	**Total**
Fasta	`java`	55/66.27%/52.38%	235
Fasta	`kotlin-converted`	63/75.90%/60.0%	425
Fasta	`kotlin-idiomatic`	54/65.06%/51.43%	516
Fannkuch-Redux	`java`	30/71.43%/28.57%	104
Fannkuch-Redux	`kotlin-converted`	32/76.19%/30.48%	109
Fannkuch-Redux	`kotlin-idiomatic`	38/90.48%/36.19%	164
N-body	`java`	41/68.33%/39.05%	466
N-body	`kotlin-converted`	47/78.33%/44.76%	614
N-body	`kotlin-idiomatic`	56/93.33%/53.33%	734
Reverse-complement	`java`	36/70.59%/34.29%	277
Reverse-complement	`kotlin-converted`	40/68.42%/38.10%	716
Reverse-complement	`kotlin-idiomatic`	31/60.78%/29.52%	203

Table 4.10: Unigram bytecode sequence results *(* = per implementation/percentage of benchmark implementations/percentage of all implementations)*

There exists 105 unique unigrams for all benchmark implementations. Applying PCA to the frequencies of the unigrams allows for dimensionality reduction and visual representation of the implementations by plotting the two major principal components. As depicted in Figure 4.5, implementations belonging to the same benchmark type is closely grouped together, with Fasta being the the most closed group. The `kotlin-idiomatic` implementation of Reverse-complement exhibits fewest similarities for any benchmark group. This can be seen as a result of the fact that the idiomatic implementation produced the fewest amount of total and unique unigrams for the Reverse-complement implementations. It is also evident that for all benchmarks apart from N-body, `java` and `kotlin-converted` produces the most similar unigrams.

Figure 4.5: 1-gram bytecode principal component scatter plot



4.5.2 Bigram results

The bigram results are presented in Table 4.11. Unlike the unigrams which represent individual bytecode instructions, the bigrams takes into account the order of the bytecode. At first glance, the fact that the total bigram counts are smaller than the corresponding unigram counts for all implementations might seem unintuitive. However, as described in Section 3.5.1, n-grams consisting of instructions belonging to different methods or blocks are discarded since such n-grams do not make any sense. Considering bigrams can contain such sequences whereas unigrams cannot results in a lower total bigram than unigram count. However, when it comes to the total count, the trend remains the same for unigrams and bigrams, with `java` producing the fewest total bigrams for all implementations except Reverse-complement.

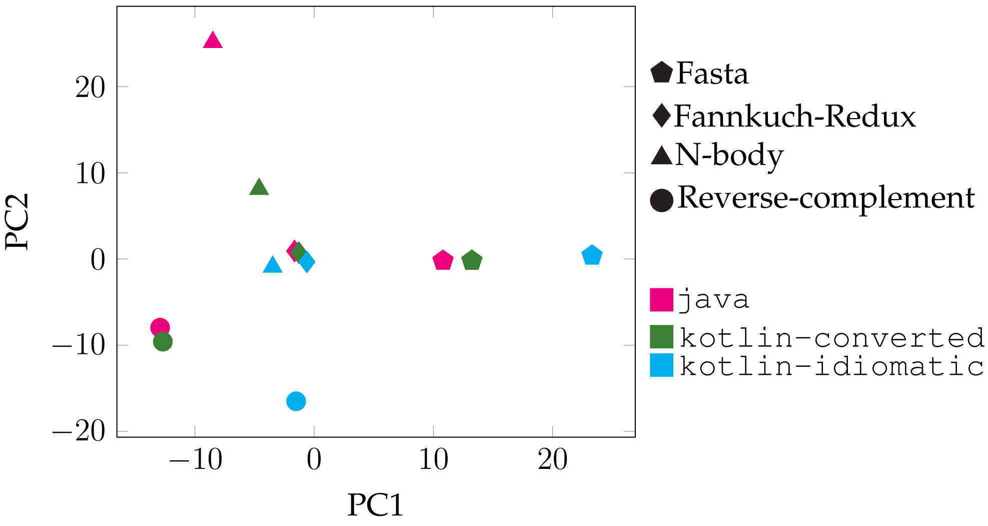
The unique count for bigrams is higher for all implementations than for unigrams, considering that order of the bytecode is taken into account. A clearer trend can be seen in the bytecode difference, where the difference in the unique count is higher for most implementations. Furthermore, the unique bigram coverage of each implementation is between 8-23%, meaning that bigrams more clearly distinguish different benchmark groups from each other.

The implementational similarities and differences in regards to the bytecode is illustrated in Figure 4.6. The implementations contained 769 unique bigrams in total. It is clear that the Fannkuch-Redux benchmark is the most closed group, although it is worth noting that the scale has changed. The general trend is that the benchmarks are less related to each other compared to the unigram results. Furthermore, it is also clear that the `java` and `kotlin-converted` implementations are more closely related than `kotlin-idiomatic` for all implementations.

Benchmark	Implementation	Unique*	Total
Fasta	`java`	122/36.53%/15.86%	207
Fasta	`kotlin-converted`	170/50.90%/22.11%	375
Fasta	`kotlin-idiomatic`	170/50.90%/22.11%	487
Fannkuch-Redux	`java`	62/49.2%/8.06%	96
Fannkuch-Redux	`kotlin-converted`	64/50.79%/8.32%	99
Fannkuch-Redux	`kotlin-idiomatic`	100/79.37%/13.0%	150
N-body	`java`	107/41.47%/13.91%	449
N-body	`kotlin-converted`	129/50.0%/16.77%	567
N-body	`kotlin-idiomatic`	175/67.83%/22.76%	693
Reverse-complement	`java`	127/44.72%/16.51%	258
Reverse-complement	`kotlin-converted`	136/47.89%/17.69%	630
Reverse-complement	`kotlin-idiomatic`	107/37.68%/13.91%	187

Table 4.11: Bigram bytecode sequence results *(* = per implementation/percentage of benchmark implementations/percentage of all implementations)*

Figure 4.6: 2-gram bytecode principal component scatter plot



4.5.3 Trigram results

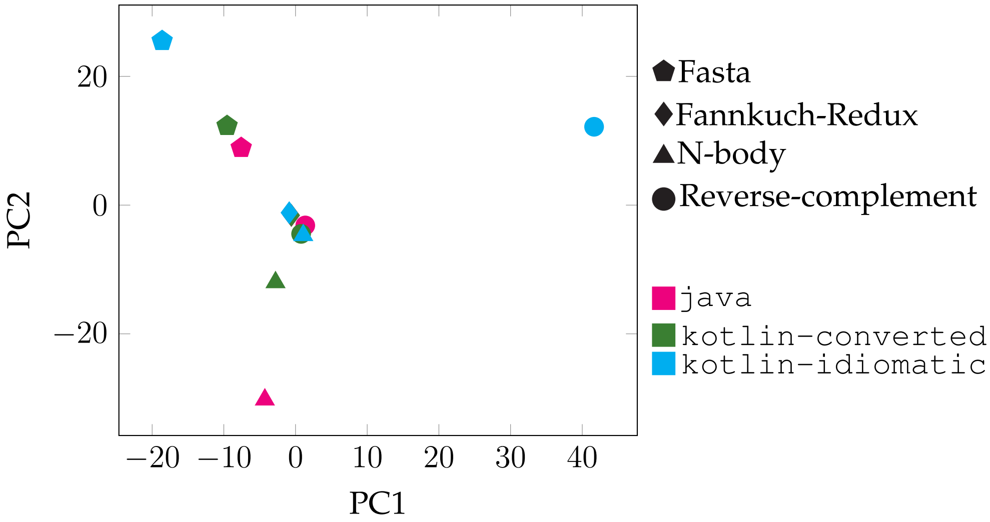
The trigram results are presented in table 4.12. Again, the total trigram count is smaller for all implementations compared to the corresponding bigram and unigram counts. `java` once again exhibits the lowest total trigram count, except for Reverse-complement. Furthermore, the unique trigram coverage of each implementation ranges between $5 - 17\%$, which is smaller compared to the previous results. In regards to the unique count of all implementations, `java` produces the fewest unique trigrams, followed by `kotlin-converted` and `kotlin-idomatic` for all implementations except Reverse-complement.

Benchmark	Implementation	Unique*	Total
Fasta	`java`	129/27.27%/9.28%	185
Fasta	`kotlin-converted`	196/41.44%/14.1%	334
Fasta	`kotlin-idiomatic`	227/47.99%/16.33%	462
Fannkuch-Redux	`java`	74/46.25%/5.32%	88
Fannkuch-Redux	`kotlin-converted`	77/48.13/5.54%	89
Fannkuch-Redux	`kotlin-idiomatic`	116/72.50%/8.35%	138
N-body	`java`	145/36.34%/10.43%	434
N-body	`kotlin-converted`	173/43.36%/12.47%	522
N-body	`kotlin-idiomatic`	231/57.89%/16.62%	653
Reverse-complement	`java`	189/40.47%/13.60%	241
Reverse-complement	`kotlin-converted`	188/40.26%/13.53%	548
Reverse-complement	`kotlin-idiomatic`	140/29.98%/10.07%	172

Table 4.12: Trigram bytecode sequence results *(* = per implementation/percentage of benchmark implementations/percentage of all implementations)*

The bytecode of all implementations contained 1390 unique trigrams in total, from which dimensionality reduction has been applied using PCA and plotted in Figure 4.7. It is clear that Fannkuch-Redux is the most closed group. Furthermore, `java` and `kotlin-converted` exhibits the most similarities for all benchmark types except for N-body, where `kotlin-converted` and `kotlin-idiomatic` are more closely grouped together.

Figure 4.7: 3-gram bytecode principal component scatter plot



Chapter 5

Discussion

This chapter discusses and analyses the results in a by-metric order, identifying trends and putting them in a greater context. Furthermore, a more general discussion regarding the methodology and the limitations of this study is carried out out at the end. This leads up to the conclusion presented in Chapter 6.

5.1 Runtime

For the runtime results, a clear trend can be observed where Java implementations are the fastest for all benchmark types. For Fasta, both Kotlin implementations are slower by roughly 27%. There are substantial implementational differences between the two Kotlin implementations with respect to I/O handling. However, there is no significant runtime difference. This indicates that there is an underlying overhead for both Kotlin implementations. It can be explained by the dependency to the Java API for I/O handling.

For Fannkuch-Redux, the auto-converted Kotlin implementation is only 6% slower than Java. This is not surprising, considering that only core functions are used within respective language. However, the idiomatic Kotlin implementation is substantially slower by a factor of around 40. This can be explained by the major implementational difference between the implementations, depending on how the permutations are flipped. Both `java` and `kotlin-converted` simply manipulate an int array, whereas the `kotlin-idiomatic` implementation uses the functions `take()`, `reversed()` and `forEachIndexed()` to achieve the same result.

For N-body, the `java` and `kotlin-idiomatic` implementations exhibited a similar runtime behaviour. However, the `kotlin-converted` implementation was significantly slower by a factor of roughly 16. Inspecting the method traces shows that the converted implementation has a significant amount of function call overhead. The NBody class contains several public fields which are used frequently for updating the velocity and position of the planets. Kotlin, on the other hand, have replaced fields with properties. Therefore, Kotlin does not expose fields publicly but instead generates getter and setter functions to manipulate them. Since the properties are manipulated frequently in the simulation process, this results in a significant function call overhead for the converted implementation. The idiomatic implementation, however, makes use of the `@JvmField` annotation to instruct the compiler to compile the properties to fields as in the case of Java. This eliminates the function call overhead, and therefore results in significant better runtime performance compared to the converted Kotlin implementation.

For Reverse-complement, the converted Kotlin implementation is roughly 91% slower than Java, whereas the idiomatic implementation is around 135% slower. As with the Fasta benchmark, this can be seen as a result of the dependency to the Java API for I/O. The fact that the runtime of the idiomatic implementation is slower than the converted implementation can be explained by the functional collection handling reversing the nucleotides. Furthermore, two runtime spikes exist for both Kotlin implementations. Runtime measurements are particularly sensible to external factors such as scheduling, and although some precautions have been taken, guaranteeing the exclusive execution of an application is impossible. Such sudden and drastic increases in runtime of the two implementations is thought to be the effect of the execution of a background process. The reasons no such spikes can be observed for the Java implementation can be explained by the fact that the total amount on time spent on benchmarking `java` is lower than the total time spent on benchmarking the two Kotlin implementations, reducing the probability that a background process is executed during that time.

Although Java seems to exhibit a significantly better runtime performance than Kotlin on ART, one also has to consider the computationally intensive characteristics of synthetic benchmarks. Therefore, the runtime impact of real-world applications might not be as severe as the benchmark results suggest.

Furthermore, for both Fasta and Reverse-complement, the auto-converted Kotlin implementations exhibited a runtime behaviour closer to the idiomatic Kotlin implementation than to Java. Both benchmarks depends on the Java API for I/O handling, and this can be seen as a common underlying overhead of Kotlin.

5.2 Memory consumption

The memory consumption results show that both Kotlin and Java can be written in an equally memory efficient way, where the difference between the `java` and `kotlin-converted` implementations were small or even non-existent for all benchmarks. However, there were major differences between the two aforementioned implementations and `kotlin-idiomatic` for Fannkuch-Redux and Reverse-complement.

In the case of Fannkuch-Redux, almost ten million more objects were allocated resulting in almost 300 million more allocated bytes compared to `java` and `kotlin-converted`. The increase in the amount of objects and bytes allocated can be seen as a result of the idiomatic implementation for flipping the permutations, using the core `take()` and `reversed()` collection functions. As such, each permutation flip results in a duplication of the subset of the permutation array on the heap. Since hundreds of thousands of flips are performed, the overhead of doing so is amplified and thus reflected by the results.

For Reverse-complement, the idiomatic Kotlin implementation once again exhibited a larger memory consumption than the two other implementations. However, the results were not as severe as in the case of Reverse-complement, with 60000 more object allocations and an increased total size by a factor of 10. This increase can once again be explained by the use of functional collection handling and lambda expressions for the idiomatic Kotlin implementations, which resulted in the creation of new objects for each processed data item.

This suggests that although Kotlin can be written in a memory efficient way, as indicated by the `kotlin-converted` implementations, idiomatic implementations relying on functional collection handling for data intensive tasks can result in substantially increased pressure on the heap.

5.3 Garbage collection

The garbage collection results are closely connected to the results of the memory consumption, in that benchmark implementation allocating much memory also exhibits a larger garbage collection overhead. As such, it is non-surprising that `kotlin-idiomatic` freed more bytes per garbage collection than `java` and `kotlin-converted` for all benchmark implementations apart from N-body (which did not put enough pressure on the heap to trigger any garbage collection events for any implementation).

More interestingly is that there seems to be a garbage collection overhead of reclaiming objects allocated by Kotlin compared to Java on ART. In Table 4.5, the difference in garbage collection count and total number of bytes freed are non-significant on a confidence level of 95% for all implementations. Furthermore, although there is a significant difference in bytes freed per garbage collection, the difference is small. For `kotlin-converted`, the garbage collector frees roughly 0.5% more bytes per event compared to `java`. The difference between `kotlin-idiomatic` and `java` is around 3%. This would intuitively suggest that the time consumed by the garbage collector reclaiming those bytes would be within the same margins. However, the garbage collector spends roughly 161% (11 ms) and 125% (8.5 ms) more time on reclaiming objects for the Kotlin implementations compared to Java. Similar observations can be made for `java` and `kotlin-converted` in Table 4.7.

Given that idiomatic Kotlin implementations have shown an increased memory consumption as a result of lambda expressions and core functions for dealing with collections, the garbage collection overhead of Kotlin implementations utilising these features has the potential of being further amplified.

5.4 Boxing of primitives

The boxing of primitives results were for many implementations non-existent. For Java, this is a consequence of the language semantics. The keywords for both unboxed and boxed primitives allow developers to request each when deemed appropriate. Kotlin, taking on a more pragmatic approach, relieves the developer of this decision. Instead, the compiler decides whether or not a primitive should be boxed or unboxed. As such, the results suggest that the compiler efficiently can avoid using boxed primitives in many cases.

However, the `kotlin-idiomatic` implementations of Fannkuch-Redux and Reverse-complement resulted in unboxing of integers and characters respectively. As this was not the case for `kotlin-converted` implementations, it suggests that the use of functional collection handling and lambda expressions sometimes have a hidden cost of introducing an unboxing overhead.

These results confirms the findings in [23] and [32], where Java in general exhibited a lower amount of boxing compared with other JVM-hosted languages. It should be noted, however, that [23] measured the amount of boxed primitives by the relative frequency of allocated objects whereas [32] measured it both by the relative frequency as well as the amount of requested boxed primitives (by invocations to the `valueOf()` method for each class). Due to limitations on obtaining the relative frequency on ART, which would have required instrumentation of the runtime system, only the latter approach was used in this study.

5.5 Bytecode

The unigram bytecode results, presented in Table 4.10, shows which bytecode instructions are produced by the AOT compiler from the Java and Kotlin source code. A clear trend can be observed, where the Java implementations in general are compiled to more concise bytecode. The measured difference is in the smallest case 3.125% but goes as high as 119%. Idiomatic Kotlin implementations in general exhibit a larger difference compared to Java. Furthermore, all Java implementations except for Reverse-complement also contain the least amount of unique bytecode instructions with a difference between 3.226%-63.55%.

The bigram and trigram results, presented in Table 4.11 and Table 4.12, indicate which 2-length and 3-length bytecode sequences that are produced, giving an idea of the amount of bytecode variation. As with the unigram, Java code is compiled to the fewest amount of total n-grams but also exhibits the fewest unique sequences for all benchmarks except Reverse-complement.

Considering the runtime system makes optimisations based upon bytecode sequences, it is possible that fewer optimisations can be made for Kotlin code compared to Java, due to the fact that the bytecode exhibits a larger variation for Kotlin. Thus, the runtime system might be unfamiliar with sequences produced by Kotlin, resulting in increased runtime.

The PCA scatter plots, presented in Figure 4.5-4.7, show a clear trend where benchmarks belonging to the same type are generally grouped together. Furthermore, it is clear that the bytecode produced for `java` and `kotlin-converted` are more related within each benchmark type than for `kotlin-idiomatic`. This is unsurprising, given that the source code has been converted and therefore similarities are carried over to the resulting bytecode.

Unlike in [23], no clear trend could be observed where the implementations were closely grouped together by language type. However, it should be noted that the previous study investigated more applications and that as a result language specific trends could be observed.

5.6 General

To the best of our knowledge, no previous studies have been conducted evaluating the performance of Kotlin and Java, in particular not on Android Runtime. Although the results indicate that Java performs better than Kotlin for most of the metrics studied, the Java implementations are a result of a large community effort, representing the fastest sequential implementations from CLBG.

The Kotlin implementations, on the other hand, are a result of the JetBrains auto-converter tool as well as original implementations to highlight idiomatic uses of the language. Therefore, the auto-converted implementations, although manually reviewed, might not be as well optimised as their Java counterparts. Furthermore, idiomatic Kotlin code often invites to the use of functional constructs, generally regarded as less performant but more easy to read and reason about. As such, care needs to be taken when generalising the results, and overall, more benchmarks and implementations are needed before any definite conclusions can be made regarding the performance of Kotlin and Java.

As previously mentioned, one also has to consider that ART initially was optimised for Java, and as a result of this, Java should intuitively perform slightly better than Kotlin. However, as further improvements are made to ART, the performance difference between the languages attributed to the runtime system are expected to decrease.

Furthermore, one limitation of this study is to only use synthetic benchmark applications to evaluate the performance. Real-world applications might not be of such a computationally intensive character, in particular not Android applications apart from games or other applications requiring physics or handling large data sets. Therefore, the performance differences identified in this study might not be as evident or have large impacts on actual Android applications.

Chapter 6

Conclusion

To answer the research question, this study comes to the conclusion that for the benchmark implementations studied, there are differences between functionally equivalent Java and Kotlin implementations on ART, where Java performed better than Kotlin for all benchmarks. Furthermore, garbage collection, boxing of primitives and static bytecode n-gram metrics partly explain the runtime overhead of Kotlin.

For the runtime efficiency, this study concludes that for the implementations studied, Kotlin is slower than Java by a varying extent. In terms of memory consumption, Kotlin can be implemented equally as memory-efficient as Java. However, introducing idiomatic Kotlin constructs and features, might lead to increased memory consumption.

A garbage collection overhead has been identified, where the garbage collector spends significantly more time reclaiming the same amount of objects allocated by Kotlin compared to Java, where the measured overhead lies around 125-161% (or around 8.5-10 ms) for the studied benchmarks. It is hard to explain why the overhead exists without deeply studying the behaviour of the garbage collector. This study is suggested as a future work.

This study also concludes that both Kotlin and Java exhibit a low amount of boxing of primitives on ART. However, given that Kotlin does not allow developers to declare boxed and unboxed primitives but leaves this decision to the compiler, the risk of unintentionally introducing the need of boxing and unboxing can be seen as higher for Kotlin compared to Java.

In terms of bytecode, this study has found that the AOT compiler produces more compact DEX bytecode for Java compared to Kotlin, where Kotlin bytecode contains 3.125%-119% more instructions than functionally equivalent Java implementations for three of four benchmarks studied. Furthermore, Java also uses fewer unique instructions compared to Kotlin for three of the four benchmarks studied, with a difference of between 3.226%-63.55%. This trend remains the same when looking at bytecode sequences, indicating that Java exhibits less variation in the produced bytecode. Since ART has supported Java longer than Kotlin, it is fair to assume that the runtime system is better optimised for Java. Furthermore, as Kotlin exhibits a larger amount of bytecode variation than Java, this may result in that the runtime system is poorly optimised for the bytecode sequences produced by Kotlin, leading to a negative impact on the runtime.

However, although Kotlin in this study showed to be less performant than Java, the Java implementations were a result of a large community effort whereas the Kotlin implementations were either auto-converted or developed as part of this study. Furthermore, Kotlin introduces several appreciated benefits over Java, including modern constructs and features and improved null-safety. Therefore, it might be worth using Kotlin as the main language of development, and either refrain from idiomatic constructs in performance critical sections or make use of the interoperability with Java to combine the best of the two languages.

Future work

Given that this study, to the best of our knowledge, is the first evaluating the performance of Kotlin and Java on ART, we acknowledge that more work is needed before the results can be generalised. Due to the limitations of this study, only four benchmarks were included. One obvious aspect could therefore be to extend this study by investigating if the findings conforms with additional CLBG benchmarks, non-synthetic applications or different workloads. Furthermore, due to the unexplained garbage collection overhead, another aspect to investigate is whether such an overhead can be observed for other benchmarks, and if so, why such an overhead exists.

Future studies could also look into developing an instrumented version of ART to measure more dynamic metrics as in [23] and [32].

Bibliography

[1] Mehmet D. Akin and Rikard Mustajärvi. *CLBG Fasta Java #4*. https://benchmarksgame.alioth.debian.org/u64q/program.php?test=fasta&lang=java&id=4. [Online; accessed 9-March-2018].

[2] Kenneth R Anderson and Duane Rettig. "Performing Lisp analysis of the FANNKUCH benchmark". In: *ACM SIGPLAN Lisp Pointers* 7.4 (1994), pp. 2–12.

[3] Stephen M Blackburn et al. "The DaCapo benchmarks: Java benchmarking development and analysis". In: *ACM Sigplan Notices*. Vol. 41. 10. ACM. 2006, pp. 169–190.

[4] Andrey Breslav. *Kotlin 1.0 Released: Pragmatic Language for JVM and Android*. https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/. [Online; accessed 20-February-2018].

[5] Standard Performance Evaluation Corporation. *SPEC Benchmarks*. https://www.spec.org/. [Online; accessed 5-February-2018].

[6] Louis Croce. *Lecture notes in Just in Time Compilation*. [Online; accessed 5-February-2018]. Department of Computer Science Columbia University, 2014.

[7] Bruno Dufour et al. "Dynamic metrics for Java". In: *ACM SIGPLAN Notices*. Vol. 38. 11. ACM. 2003, pp. 149–168.

[8] David Ehringer. "The dalvik virtual machine architecture". In: *Techn. report (March 2010)* 4.8 (2010).

[9] Thiago Soares Fernandes, Érika Cota, and Álvaro Freitas Moreira. "Performance Evaluation of Android Applications: A Case Study". In: *Computing Systems Engineering (SBESC), 2014 Brazilian Symposium on*. IEEE. 2014, pp. 79–84.

[10] Nisarg Gandhewar and Rahila Sheikh. “Google Android: An emerging software platform for mobile devices”. In: *International Journal on Computer Science and Engineering* 1.1 (2010), pp. 12–17.

[11] Statista GmbH. *Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017*. `https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/`. [Online; accessed 6-February-2018].

[12] Google. *Android 5.0 Behavior Changes – Android Runtime (ART)*. `https://developer.android.com/about/versions/android-5.0-changes.html`. [Online; accessed 24-January-2018].

[13] Google. *Android 8.0 ART improvements*. `https://source.android.com/devices/tech/dalvik/improvements`. [Online; accessed 8-February-2018].

[14] Google. *ART and Dalvik*. `https://source.android.com/devices/tech/dalvik/`. [Online; accessed 8-February-2018].

[15] Google. *Platform Architecture*. `https://developer.android.com/guide/platform/index.html`. [Online; accessed 6-February-2018].

[16] Google. *Threading performance*. `https://developer.android.com/topic/performance/threads.html`. [Online; accessed 27-March-2018].

[17] Isaac Gouy. *The Computer Language Benchmarks Game*. `http://benchmarksgame.alioth.debian.org/`. [Online; accessed 24-January-2018].

[18] Isaac Gouy and Oleg Mazurov. *CLBG Fannkuch-Redux Java #2*. `https://benchmarksgame.alioth.debian.org/u64q/program.php?test=fannkuchredux&lang=java&id=2`. [Online; accessed 4-March-2018].

[19] JetBrains. *Implementing ART Just-In-Time (JIT) Compiler*. `https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/`. [Online; accessed 19-February-2018].

[20] JetBrains. *Kotlin*. `https://kotlinlang.org`. [Online; accessed 25-January-2018].

[21] Tao Lei. *CLBG Reverse-Complement Java #5*. `https://benchmarksgame.alioth.debian.org/u64q/program.php?test=revcomp&lang=java&id=5`. [Online; accessed 5-March-2018].

[22] Mark C. Lewis and Chad Whipkey. *CLBG N-body Java #2*. https://benchmarksgame.alioth.debian.org/u64q/program.php?test=nbody&lang=java&id=2. [Online; accessed 5-March-2018].

[23] Wing Hang Li, David R White, and Jeremy Singer. "JVM-hosted languages: they talk the talk, but do they walk the walk?" In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM. 2013, pp. 101–112.

[24] David J Lilja. *Measuring Computer Performance: A practitioner's guide*. Cambridge University Press, 2005.

[25] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. "Cross-language compiler benchmarking: are we fast yet?" In: *ACM SIGPLAN Notices*. Vol. 52. 2. ACM. 2016, pp. 120–131.

[26] Tauno Metsalu and Jaak Vilo. "ClustVis: a web tool for visualizing clustering of multivariate data using Principal Component Analysis and heatmap". In: *Nucleic acids research* 43.W1 (2015), W566–W570.

[27] Oracle. *Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html. [Online; accessed 11-February-2018].

[28] Kolin Paul and Tapas Kumar Kundu. "Android on mobile devices: An energy perspective". In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE. 2010, pp. 2421–2426.

[29] Guillermo A Perez et al. "A hybrid just-in-time compiler for android: comparing JIT types and the result of cooperation". In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2012, pp. 41–50.

[30] Xi Qian, Guangyu Zhu, and Xiao-Feng Li. "Comparison and analysis of the three programming models in google android". In: *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC)*. Vol. 14. 2012.

[31] SamMobile. *Galaxy S9 SM-G960F specifications*. https://www.sammobile.com/devices/galaxy-s9/specs/SM-G960F/. [Online; accessed 3-May-2018].

[32] Aibek Sarimbekov et al. "Workload characterization of JVM languages". In: *Software: Practice and Experience* 46.8 (2016), pp. 1053–1089.

[33] Yunhe Shi et al. "Virtual machine showdown: Stack versus registers". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 4.4 (2008), p. 2.

[34] *Stack Overflow Trends*. `https://insights.stackoverflow.com/trends?tags=kotlin`. [Online; accessed 25-January-2018].

[35] *The DaCapo Benchmarking Suite*. `http://www.dacapobench.org/`. [Online; accessed 25-January-2018].

[36] Chih-Sheng Wang et al. "A method-based ahead-of-time compiler for android applications". In: *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*. IEEE. 2011, pp. 15–24.

[37] Radhakishan Yadav and Robin Singh Bhadoria. "Performance analysis for Android runtime environment". In: *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*. IEEE. 2015, pp. 1076–1079.

Appendix A

Benchmark Implementations

A.1 Java implementations

The original Java implementations by [1, 18, 22, 21] can be found at the CLBG website: `http://benchmarksgame.alioth.debian.org/`.

The ART adapted implementations are available at `https://github.com/pschwermer/kth-thesis/tree/master/Code/BenchmarkApp/src/main/java/com/benchmarks/patrik/benchmarks/java`.

A.2 Kotlin implementations

The auto-converted Kotlin implementations can be found at: `https://github.com/pschwermer/kth-thesis/tree/master/Code/BenchmarkApp/src/main/java/com/benchmarks/patrik/benchmarks/kotlinConverted`.

The idiomatic Kotlin implementations are available at: `https://github.com/pschwermer/kth-thesis/tree/master/Code/BenchmarkApp/src/main/java/com/benchmarks/patrik/benchmarks/kotlinIdiomatic`.

Appendix B

Benchmark Application

The benchmark application used for benchmarking the implementations and generating the metric data is available at: `https://github.com/pschwermer/kth-thesis/tree/master/Code/BenchmarkApp/src`.

Appendix C

N-gram Application

The source code of the bytecode analyser, along with the bytecode files for each of the studied benchmark implementations are available at: `https://github.com/pschwermer/kth-thesis/tree/master/Code/BytecodeAnalyser`.

Appendix D

Results

D.1 Runtime

The raw data of the runtime results are available at: https://github.com/pschwermer/kth-thesis/tree/master/Results/runtime.

D.2 Memory consumption

The raw data of the memory consumption results are available at: https://github.com/pschwermer/kth-thesis/tree/master/Results/mc.

D.3 Garbage collection

The raw data of the garbage collection results are available at: https://github.com/pschwermer/kth-thesis/tree/master/Results/gc.

D.4 Boxing of primitives

The raw data of the boxing of primtives results are available at: https://github.com/pschwermer/kth-thesis/tree/master/Results/bop.

D.5 Bytecode results

The raw data of the N-gram results are available at: `https://github.com/pschwermer/kth-thesis/tree/master/Results/bytecode`.