DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Hidden Costs and Opportunities of Kotlin versus Java on Android Runtime

JOHAN LUTTU

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

Hidden Costs and Opportunities of Kotlin versus Java on Android Runtime

JOHAN LUTTU

Master in Computer Science
Date: March 16, 2020
Supervisor: He Ye
Examiner: Cyrille Artho
School of Computer Science and Communication
Host company: Spotify
Swedish title: Dolda kostnader och möjligheter hos Kotlin kontra Java för Android Runtime

Abstract

In mid-2019, Kotlin became Google's preferred language for Android application development instead of Java. Therefore, it is in entities' interest to migrate from Java to Kotlin for their Android projects to get the most support from Google. This study has evaluated what a Java (version 8) to Kotlin (version 1.3.50) migration would implicate for an Android application when it comes to run time performance. A set of six features was analyzed, first by a qualitative bytecode analysis, then by running microbenchmarks.

The purpose of the qualitative bytecode analysis was to detect hidden costs or opportunities of Kotlin features that were not visible in the source code. The results for the analysis showed that the bytecode produced by Kotlin, compared to Java, can lead to extra workloads for the garbage collector and an increased method count for the DEX files of an Android application. It was also observed that Kotlin performs systematic boxing and unboxing for short-lived objects if primitive values are passed as arguments to, or returned from, a lambda expression for an example that included a lambda expression that was passed to a higher-order function.

The purpose of the microbenchmarks was to measure the hidden costs' or opportunities' potential impact, and also to see if bytecode optimization tools could be relied on to mitigate the potential costs. The results showed that, in most cases, the manual optimization did not result in better performance than in the case when the code had been optimized by bytecode optimization tools. Overall, the optimized microbenchmarks showed no large differences in execution speed between the Java and Kotlin implementations.

Sammanfattning

I mitten av år 2019 blev Kotlin Googles föredragna programmeringspråk för Android-utveckling istället för Java. Därför är det i entiteters intresse att migrera från Java till Kotlin för sina Android-projekt för att få så mycket stöd som möjligt av Google. Denna studie har evaluerat vad en Java (version 8) till Kotlin (version 1.3.50)-migration skulle innebära för en Android-applikation avseende körtidsprestanda. En mängd med sex stycken språkfunktioner analyserades, först genom en kvalitativ bytekodanalys, och sedan genom att genomföra mikro-benchmarks.

Syftet med den kvalitativa bytekodanalysen var att upptäcka Kotlins dolda kostnader eller möjligheter som inte var synliga i källkoden. Resultaten för analysen visade att bytekoden som Kotlin producerade, jämfört med Java, kan leda till extra arbetsbördor för skräpsamlaren och ett ökat metodantal för Android-applikationers DEX-filer. Det observerades också att Kotlin sytematiskt utför så kallad boxing och unboxing för kortlivade objekt ifall primitiva värden skickas som argument till, eller returneras som, ett lambda-uttryck för ett exempel som involverade att ett lambda-uttryck skickades som argument till en funktion av högre ordning.

Syftet med mikro-benchmarks var att att mäta de dolda kostnadernas eller möjligheternas potentiella påverkan, men också för att se ifall det går att lita på att bytekodoptimeringsverktyg mildrar de potentiella kostnaderna. Resultaten visade, för de flesta fallen, att även ifall det existerade ett sätt att manuellt optimera bort en dold kostnad som Kotlin kunde ha, resulterade inte den manuella optimeringen i bättre prestanda än i fallet när icke-optimerad kod hade blivit optimerad av bytekodoptimeringsverktyg. Övergripande, för de optimerade fallen, var det knappt någon skillnad i exekveringshastigheten för de implementerade språkfunktionerna i båda språken.

Contents

Chapter 1

Introduction

In 2016, version 1.0 of the programming language Kotlin was released [1]. In May 2019, Google, who develops Android, made Kotlin their preferred language for Android application development instead of Java [2]. Therefore, as Android application development is becoming increasingly Kotlin-first, it is in entities' interest to migrate from Java to Kotlin for their Android projects to get the most support from Google.

However, for entities that have large-scale Android projects and where run-time performance is a critical factor for the success of the application, the decision about when to migrate to a new programming language is not trivial. An extra delay of a couple hundred milliseconds in any loading phase of an application can severely damage the perception of an entire brand, even for aspects that are not related to performance, such as content and design [3].

Kotlin brings new convenient language features, but it is still in an early stage compared to languages like Java. Therefore, the purpose of this study is to investigate what a Java to Kotlin migration could implicate when it comes to run time performance.

1.1 Objective

While evaluating the run time performance of a whole programming language is a large task, this study aims to assess what Kotlin does better or worse than Java for a small set of considered features. The desired outcome of this study is that it will grant awareness, to owners of Java Android projects, about some of the hidden costs and opportunities regarding run time performance

for Kotlin applications. More specifically, the thesis aims to identify which features Kotlin implements better or worse than Java from a performance perspective, how much of a performance impact that they will potentially have, and what can be done about them from a pragmatic developer perspective.

The objective of the study is **not** to provide a simple answer to whether Kotlin is better or worse than Java. It solely shines light on implementation details of both languages that will be useful to take into consideration when one is planning to migrate from Java to Kotlin for an Android project.

1.2 Connection to Research

The study is heavily focused on the area of compilers, programming languages, runtime environments, and bytecode optimization. More specifically, implementations of Kotlin 1.3.50, Java 8, JVM, Android runtime (ART), ProGuard, and R8 are assessed. Theory about creating test environments for reliable microbenchmarks is also covered.

Keywords: Kotlin, Java, performance, compilers, bytecode optimization, ART, JVM, ProGuard, R8, microbenchmarks, runtime environments.

1.3 Research Question

The research question for the thesis is the following:

> *What could a Java to Kotlin migration implicate for an Android application when it comes to run time performance?*

The question can be broken down into two more concrete questions:

- What are some of the hidden costs and opportunities when it comes to performance in Kotlin compared to Java?
- How big is their performance impact on the Android runtime?

1.4 Ethics and Sustainability

When it comes to ethical perspectives for this study, there are two aspects that are worth to take into consideration. First, given that this thesis assesses the

runtime performance of two languages and that that the results might influence decisions about whether to use one language over the other, it is important that the methodology is not biased and consists of looking for objective metrics that have an equal impact for both languages. Secondly, since the methodology of this study partly consists of redoing experiments from 2017, it is important from a deontological perspective to be transparent and clear with what has already been discovered from previous experiments in order to not steal credit.

From a sustainability perspective, there is a natural connection between runtime performance and energy consumption, which was mentioned in a similar study regarding Kotlin's runtime performance on Android [4]. If one of the languages has better run time performance than the other, it usually means that fewer computations are required to achieve the same effects, which ultimately leads to less energy consumption. Therefore, the results of this study can provide insights to Android application developers about how to achieve better sustainability.

1.5 Outline

In Chapter 2, background information, regarding what is required to be understood for evaluating the runtime performance of Kotlin compared to Java on Android Runtime, is covered. In Chapter 3, work that this study is related to and based on is presented. In Chapter 4, the methodology of how the research question was answered is presented. In chapter five, the results are presented and discussed. In Chapter 6, threats to the validity of the achieved results are discussed. In Chapter 7, the findings of this study are concluded. Finally, in chapter eight, it is discussed what future work could complement or extend the work of this study.

Chapter 2

Background

This section provides the background information needed to understand the underlying layers for a performance evaluation of Kotlin compared to Java for Android applications.

2.1 Android

This section provides information about the Android operating system and devices it usually run on.

2.1.1 Overview

Android is an open-source operating system, based on the Linux kernel, that is used by multiple devices such as smartphones and tablets [5]. Details about all the layers of the Android architecture are not covered as most of them are not relevant to this study. This study focuses on the *Android Runtime* section which can be seen in Figure 2.1.

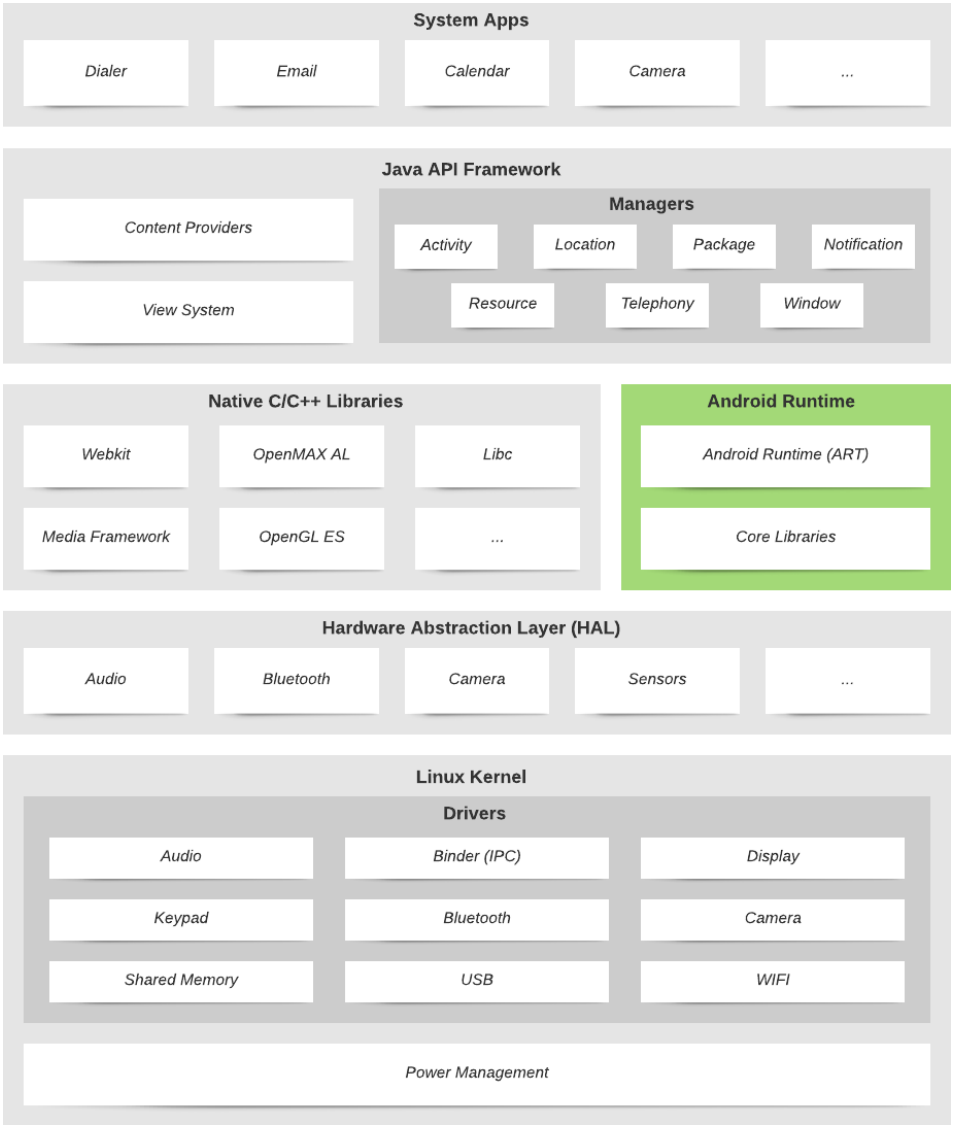

Figure 2.1: Android architecture, illustrated as in Google's Android documentation [5].

2.1.2 Memory Management

In order to ensure that multi-tasking works fluently, Android sets a limit for all running apps when it comes to the heap size. The heap sizes may vary

Figures of this page are modifications based on work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 3.0 Attribution License.

depending on the device. To prevent apps from getting OutOfMemoryError errors, Android makes use of garbage collectors inside its managed runtime environment to free the heap of unreachable data. Modern garbage collectors can be quite fast, however, they can still impair the performance of applications [6].

2.1.3 Exclusive Cores

Since Android 7.0, Android devices can reserve a CPU core exclusively for the top foreground application, with the purpose of improving performance for foreground apps [7].

2.1.4 Dynamic Frequency Scaling

Most Android devices provide the feature of dynamic frequency scaling [8], which is the process of ramping up or decreasing a CPU's clock frequency with the purpose of reducing power consumption or preventing high temperatures but yet have the flexibility to have high performance for processing-heavy workloads. If a CPU is idling, it is common that its clock frequency gets tuned down — especially on mobile devices, where battery life is of great significance for the user experience. On the other hand, if a CPU runs computationally heavy processes, it is common that the clock frequency gets ramped up to ensure higher performance [9].

2.1.5 Android Version Distribution

In order to be able to reason about the capabilities of the most used Android runtime implementations as of today, it is useful to be aware of the Android version usage distribution as of recently as possible.

At the beginning of May 2019, Google recorded data about the usage of Android versions in active devices during a one-week time frame. The statistics showed that a majority of the Android devices at the time, more precisely 57.9 %, ran an Android version that was equal to or newer than version 7 [10].

Android version	Codename	Distribution
7	Nougat	11.4 %
7.1	Nougat	7.8 %
8.0	Oreo	12.9 %
8.1	Oreo	15.4 %
9	Pie	10.4 %

Table 2.1: Android version distribution for active devices during a one-week window in May 2019.

2.2 Kotlin

Kotlin is a general-purpose and statically typed programming language. It can be compiled to multiple target languages, but it mainly targets the JVM and gets compiled to Java bytecode.

Kotlin is fully interoperable with Java. This means that you can call Java code from Kotlin applications and vice versa. Both languages' compilers can produce output that can be hosted on the JVM.

Regarding primitives, unlike Java, Kotlin does not have any primitive types like int, long, boolean, byte, char, etc. All types are objects in Kotlin [11].

2.2.1 Kotlin Coroutines

Support for coroutines in Kotlin exists as part of a first-party library. They were created with the intention of reducing the complexity of asynchronous programming. Essentially, they can be called lightweight threads as they can execute code asynchronously like threads. However, they are not threads. They simply make use of existing thread pools and have the ability to let the execution of themselves to be moved to different threads, allowing non-blocking thread utilization. Thus, it can be argued that they are more resource-efficient than threads in some cases. The reuse of existing threads also lowers the number of context switches that the CPU has to perform. Instead of being managed by the operating system, they are managed by a runtime environment [12].

According to the Kotlin Census Report of 2018, the coroutine library was the most used Kotlin library, and the usage of it had almost been doubled since 2017 [13].

2.2.2 Companion Objects

Kotlin does not have the *static* keyword that Java does have. If one wants a property or function to be tied to a class instead of instances of it, one would have to make use of a companion object. A companion object is a singleton class that is initialized when the class is loaded [14].

2.2.3 Function Types

Kotlin has the ability to express the type of functions with function types.

A function type expresses a function's type in the shape of a parenthesized list of the parameters' types and the type of the return value [15]. For instance, if a method takes two parameters of type Int and returns a value of type Int, the function type would be: (Int, Int) -> Int

One of the goals with function types was to get rid of over twenty hardwired physical function classes. The problem with the function classes was that they would take too much space in the Kotlat runtime library [16].

2.2.4 Null Safety

By default, Kotlin does not allow a declared variable to be nullable. To make variables explicitly nullable, it is possible to use the safe call operator "?" like in line 1 in Figure 2.2. If one wants to force-cast a nullable variable to a non-nullable variable, it is possible to use the not-null assertion operator "!!" like in line 3 in Figure 2.2. The not-null assertion operator throws a NullPointerException if the variable is null [17]

```
val nullableString: String? = null
val notNullableString: String = "Hello"
val stringLength = nullableString!!.length // Throws NPE
```

Figure 2.2: Examples of null safety operators in Kotlin

2.2.5 Extension Functions

In Kotlin, it is possible to extend the functionality of class, outside the class, by declaring extension functions for it. An example of this can be seen in Figure

2.3. To achieve the same functionality in Java, the decorator pattern can be utilized [18].

```
class A(){}
fun A.extensionFunction() = println("Hello")
fun callExtensionFunction(){
  val aInstance = A()
  aInstance.extensionFunction()
}
```

Figure 2.3: An example of an extension function in Kotlin.

2.2.6 Kotlin Android Extensions

There exists an extension of Kotlin, made by JetBrains, that is made for Android development. Its main feature is that it provides a way of binding user interface resources to the application logic in a manner that involves less boilerplate code. The extension also contains experimental features that were not covered in this study [19].

2.3 Compilation techniques

This section covers fundamental aspects of compiling that are essential for understanding this study. First, the general concept of compiling and how it can be distinguished from interpreting are explained. Then, different approaches to compiling, both ahead of time and at run time, are covered. Finally, optimization techniques, that compilers can be capable of, are covered.

2.3.1 Compiling Versus Interpreting

When it comes to the more complex forms of compilation, such as JIT-compilation, it can be difficult to have a clear image of what the difference is compared to interpretation. This subsection will, therefore, serve as an explanation of the differences between interpreting and compiling.

Compiling is the process of **translating** a program p written in programming language X to a semantically equivalent program p' in language Y. The output of the compiler, p', usually consists of code in a lower-level programming language in terms of abstraction. If p' consists of machine code, it can usually be executed directly in the CPU, depending on the CPU's architecture. In other

cases, p' can be a program that consists of an intermediary language such as Java bytecode that can be compiled further or interpreted [20].

On the other side, the process of interpreting can be seen as only **executing** p, one expression at the time, such that its effects are performed and the results are evaluated according to the specification of X [20]. It can even be possible to stretch it and say that CPUs are interpreters for their respective, externally visible instruction sets, given that they have no underlying private instruction sets that the instructions get compiled into.

2.3.2 Ahead-of-Time Compilation (AOT)

As the name suggests, ahead of time compilation is compilation done in advance, before the program runs. Since the compilation is not happening at run time, there is room for complex and advanced code optimizations which would be considered too costly to compute at run time. For the same reason, AOT compilation brings less overhead to run time performance. Consequently, AOT compilation has a positive effect on the battery life of the device that the AOT compiled application runs on, as there are fewer things to compute at run time compared to approaches that involve run time compilation or interpretation [21].

However, on the downside, since AOT compilation does not happen at run time, it cannot optimize an application to the fullest extent. It does not have all the information that is only available at run time and that would allow further optimizations [21].

2.3.3 Just-in-Time Compilation (JIT)

Just-in-time compilation, also called dynamic compilation, is a way of compiling code at run time. It is useful when parts of the code that are to be executed can be optimized or compiled further such that the speedup from the optimizations outweighs the overhead of compiling at run time [22].

Unlike AOT compilers, JIT compilers have access to information that is only available at run time, information such as what type of machine the application is running on, etc. Consequently, a JIT compiler can make optimizations that an AOT compiler cannot. Some of these optimizations can be platform-specific native optimizations, better inlining, and on-stack replacement [23].

Examples of compiler optimizations are covered in the following section about compiler optimizations.

2.3.4 Compiler Optimizations

Understanding compiler optimizations and dynamic compilation is a critical component to creating an accurate microbenchmark [24]. This section will, therefore, serve as an introduction to some common optimizations that can either happen at compile time or run time. The covered optimizations in this section have been selected based on if they can potentially influence the type of microbenchmarks that are relevant to this study.

Inline Expansion

Inline expansion, also called inlining, is an optimization technique with the purpose of reducing the number of function calls in a program. If a function call site gets inlined, the call site gets substituted with the body of the called function. This usually comes with a performance improvement at the cost of space. However, excessive inlining might impair performance if it will occupy too much space in CPU instruction cache [25].

Call to foo() is not inlined

```
foo() {
    x = 2;
}
foo();
```

Call to foo() is inlined

```
foo() {
    x = 2;
}
x = 2; // inline foo();
```

Figure 2.4: An example of inline expansion.

More Compiler Optimizations

Common compiler optimizations that are not covered in this study, but that one should be aware of when constructing microbenchmarks, are the following: Loop unrolling, dead code elimination, constant folding and propagation, common subexpression elimination, code hoisting, and range check elimination.

2.4 Process Virtual Machines

In this study, different process virtual machines are discussed. A process virtual machine, that also can be called a **managed** runtime environment or an application virtual machine, is responsible for supporting a single process and runs inside a host operating system. It is created for one process and then is destroyed when the process exits. The primary reason for its existence is to provide a platform-independent solution for a programming environment such that the choice of the underlying hardware or operating system does not affect, semantically, the way the programming environment executes programs [26].

2.4.1 Java Virtual Machine (JVM)

The JVM is a process virtual machine. As input, it can only process data in a binary format, namely the *Java class file* format. Class files contain bytecode for methods and other information such as a pool of constants [27].

There exist multiple JVM implementations, which may differ depending on the platform they are supposed to run on. However, they all run programs the same way, semantically. Java code can be executed on any platform as long as there is a runtime implemented for it, given that the runtime implementation can execute Java bytecode according to a Java specification [27].

A JVM can usually run a program in three ways. It is always able to interpret the bytecode instruction by instruction, but when it comes to utilizing AOT or JIT-compilation techniques, the capabilities may differ depending on the implementation. Modern JVM implementations such as the HotSpot JVM by Oracle run programs by using a mix of the three methods [28].

When a JVM interprets Java bytecode, it makes use of an operand stack for many of the operations. Whenever a Java method is invoked, a new stack frame is generated with an own set of local variables, a LIFO operand stack, and a reference to a pool of constants of the current method's class. The way it uses the operand stack is explained in the subsection below about Java bytecode [27].

Java Bytecode

Java bytecode is an abstract machine language that can be executed by a JVM. The format of a Java bytecode is illustrated in Figure 2.5.

```
<index> <opcode> [ <operand1> [ <operand2>... ]] [<comment>]
```

Figure 2.5: Java bytecode instruction format.

- **index** is the index for the instruction in an array of bytes that represent Java Virtual Machine code for a method.
- **opcode** is the mnemonic that determines which operation is to be performed.
- **operandN** are the operands of the operation.
- **comment** is usually used for debugging purposes.

When viewing Java bytecode, it is common to encounter method descriptors that describe a method according to the grammar in Figure 2.6.

```
MethodDescriptor     ::= '(' ParameterDescriptor* ')' ReturnDescriptor
ParameterDescriptor  ::= FieldType
ReturnDescriptor     ::= FieldType | VoidDescriptor
VoidDescriptor       ::= V
```

Figure 2.6: Grammar for a MethodDescriptor.

The possible terms of a *FieldType* can be seen in Figure 2.2. An example of a method descriptor can be that if a method has an Object and an int as parameters and returns a String, then its method descriptor would be: (Ljava/lang/Object;I)Ljava/lang/String;

FieldType term	Type	Interpretation
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded in UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L ClassName ;	reference	an instance of class ClassName
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

Table 2.2: Interpretation of characters describing FieldType. The table is from Oracle's Java 8 specification [29].

The following two figures contain code snippets that show an example of what Java source code can be compiled into. In Figure 2.7, a snippet of simple Java code is shown, and its corresponding bytecode is shown in Figure 2.8.

```
1 public static void main(String[] args) {
2   int i = 3;
3   i = i + 4;
4   System.out.println(i);
5 }
```

Figure 2.7: Java source code

```
1  public static void main(java.lang.String[]);
2   Code:
3     0: iconst_3 // Stack: 3
4     1: istore_1 // i[1]: 3, Stack: ϵ
5     2: iload_1  // Stack: 3
6     3: iconst_4 // Stack: 4, 3
7     4: iadd     // Stack: 7
8     5: istore_1 // i[1]: 7, Stack: ϵ
9     6: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
10    9: iload_1  // Stack: 7, Ljava/io/PrintStream;
11    10: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
12    13: return  // Stack: ϵ
```

Figure 2.8: Content from a class file containing Java bytecode. Comments about the state of the stack and the register have been added manually. The epsilon means empty.

The bytecode in Figure 2.8 has been transformed into a more human-readable format by the program *javap*. For instance, the opcodes such as iconst_3 consist of a 32 bit number in real Java bytecode.

In Appendix A, in Table A.1 a set of common Java bytecode instructions is presented.

Invokedynamic

Invokedynamic is a Java bytecode instruction that was introduced in Java 7. It allows programmers to define how call sites are dynamically linked to target methods. Like a railroad branch switch operator can change the direction that a train is going after the tracks have been laid out, with the use of invokedynamic it is possible to change the target method of a call site at run time if the related CallSite object is mutable [30].

Invokedynamic instructions are initially in an unlinked state, which means that from the start there is no specific method for the call site to invoke. The instruction is linked at run time when the call site is reached by the interpreter. The linking is done by supplying the class name and method name to a bootstrap method that returns a CallSite object, which is a holder of a MethodHandle object that points to a method. After the linking is done, the returned CallSite is invoked [30].

2.4.2 Android Runtime (ART)

Android Runtime is the managed runtime environment that Android applications run in. It is similar to the JVM, but the input that it accepts is not class files. Instead, it can only process DEX, OAT, or VDEX files, depending on the Android version. More information about the file formats is described in the subsections of this section.

ART was originally designed to depend solely on AOT compilation for programs to run, but since Android 7.0 "Nougat", ART also has a complementing JIT compiler with code profiling which allows the performance of the Android application to be improved continually as the application runs [31].

Depending on the version of the ART implementation, the way it runs programs varies. It is possible to configure the combination of AOT, JIT, and profile-guided compilation. Since Android 7.0, when JIT-compiling was introduced, the execution process of a program generally looks like the following:

- If a DEX file is supplied to ART, ART starts by interpreting the bytecode. If it eventually detects that some bytecode is hot, i.e., that it is frequently executed, ART will let the JIT compiler compile the code, execute it, and store the native representation in ART's JIT cache. This allows ART to directly execute the instruction natively when the same code path is processed. The process can be illustrated as in Figure 2.9.
- If an OAT file with native code is supplied to ART, ART can just natively execute the program.

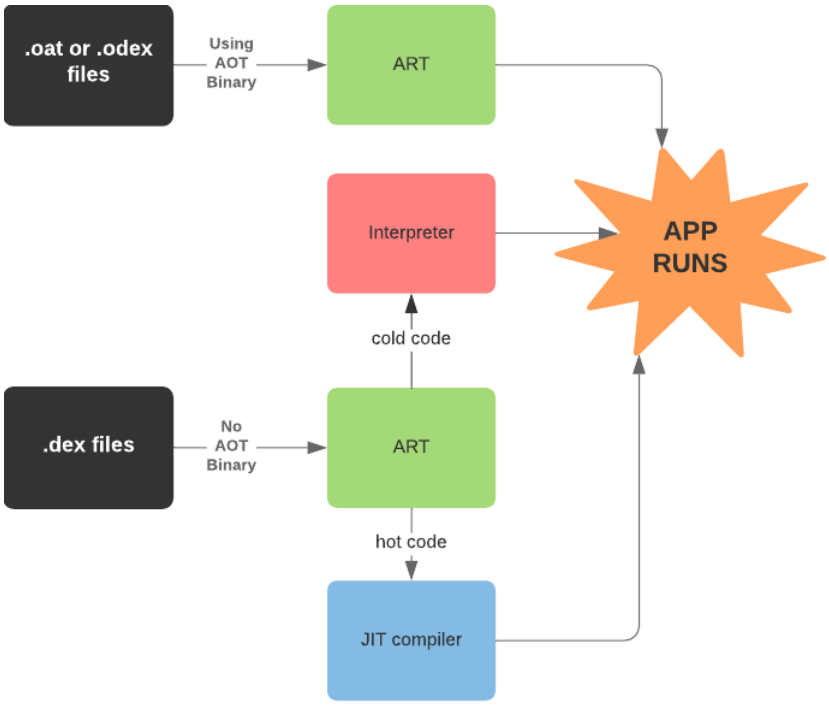

Figure 2.9: ART application execution architecture, illustrated as in Google's Android documentation [23]

The JIT compiler also collects data and creates a profile for the application. This profile can later be used by the AOT daemon that further optimizes the application while the host device is idle and charging, which is illustrated in Figure 2.10.

Figures of this page are modifications based on work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 3.0 Attribution License.

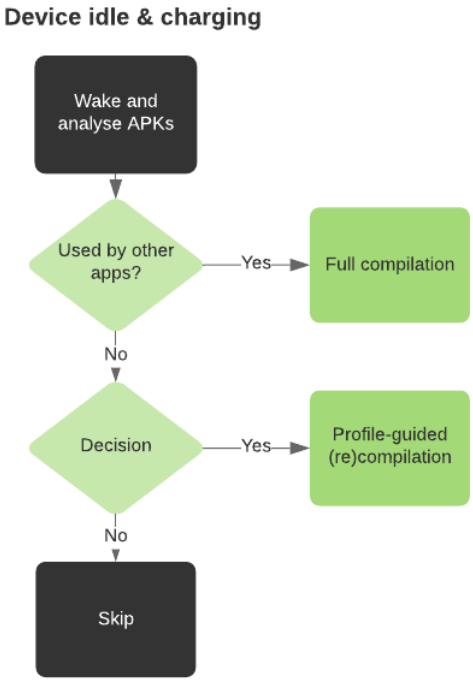

Figure 2.10: AOT daemon compilation process, illustrated as in Google's Android documentation [23]

Dalvik Executables (DEX)

DEX files contain Dalvik bytecode, which is similar to Java bytecode, even isomorphic, but with incompatible opcodes. The DEX format was designed with minimal memory usage in mind. DEX files were the only acceptable input for ART's predecessor, the process virtual machine Dalvik [32].

OAT files

Since ART has an AOT compiler that translates bytecode into native code before the application runs, a new format, the OAT format, was invented to store the compiled Dalvik bytecode — the native code that can be executed directly by ART. OAT files have the extensions .oat or .odex, and are wrapped by the ELF format [33]. The ELF format is the standard format used by Linux to package assembly code [34].

Dalvik bytecode

Dalvik bytecode is the format of instructions that ART can interpret and compile. As this study is only interested in the generated Java bytecode of the Kotlin compiler compared to the Java compiler, details about Dalvik bytecode are not covered. For the remainder of the thesis, when bytecode is mentioned, it will refer to Java bytecode.

2.5 Android build process

Since Android applications usually are written in Java or Kotlin, whose compilers produce class files containing Java bytecode, and the Android runtime only accepts DEX or OAT files, there are intermediary steps in the build process that transform the class files such that they can be processed by ART.

The Android build chain typically looks like the following. First, either the Java or Kotlin code is compiled by the languages' respective compiler, javac or kotlinc. The output of the Java or Kotin compilers, the class files, get desugared and dexed by a DEX compiler. The desugaring and dexing are what transform the Java bytecode into Dalvik bytecode such that Java 8 language features can be recognized by ART. D8 is a tool that handles the desugaring and dexing for modern Android applications [35].

It is common that the Android build chain contains another intermediary link which involves shrinking, optimizing, and obfuscating the Java bytecode. Some tools that are capable of the mentioned bytecode transformations are covered in the subsections below.

At the final stage of the Android build chain, there is the *APK Packager*, which combines the generated DEX files and compiled resources into an APK, which is an acronym for "Android Application Package". After the APK has been created, it can be used to install your application on Android devices [36].

2.5.1 ProGuard

ProGuard is a tool that can shrink, optimize, obfuscate and pre-verify Java class files [37]. However, for this literature study, only the optimization is of interest.

ProGuard can optimize the bytecode inside a class file in numerous ways. For instance, it can inline short methods or methods that are only called once. It can also remove dead code or simplify code based on control flow analysis and data flow analysis. These mentioned examples are only a few of the optimizations that ProGuard is capable of [38]. In Figure 2.11 it is illustrated where ProGuard can operate in the build chain.

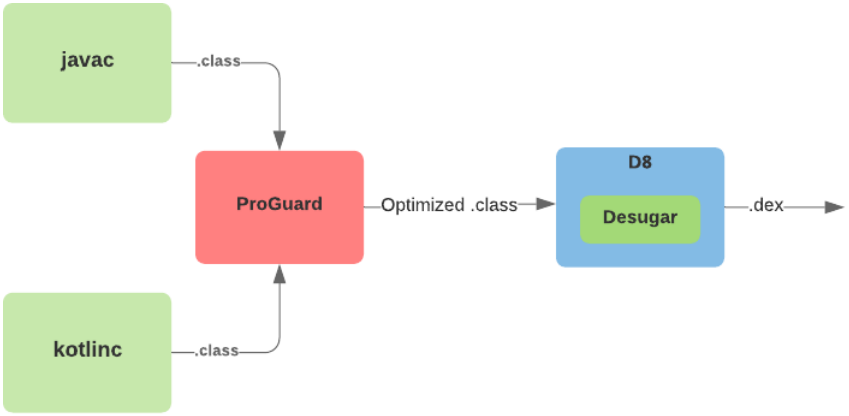

Figure 2.11: Where ProGuard operates in the build chain.

2.5.2 R8

R8 is a tool that not only can shrink, optimize, obfuscate and pre-verify Java class files like ProGuard, but it can also perform desugaring and dexing. It can handle a large part of the Android build chain by accepting class files as the input and producing optimized DEX files.

Like with ProGuard, the only capability that is interesting for this literature study is its bytecode optimization. It is not able to perform as many optimizations as ProGuard is, but it is capable of performing some optimizations for specific Kotlin constructs that ProGuard does not have support for at the time of writing this study [39]. In Figure 2.12 it is illustrated where R8 can operate in the build chain.

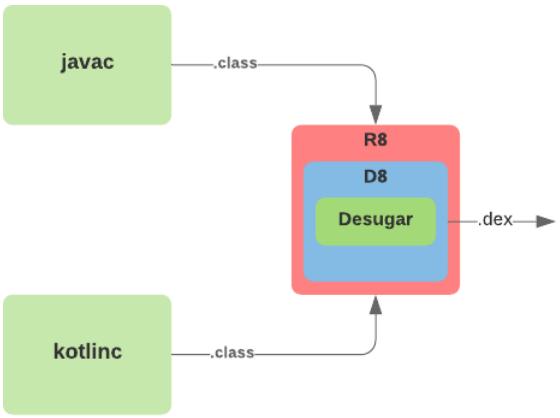

Figure 2.12: Where R8 operates in the build chain.

2.5.3 Multidex

By default, Android applications have a limit when it comes to the number of method references in a single DEX file. The limit is 65536 method references [40]. However, it is possible to bypass this limitation by using an application configuration tool called Multidex. The tool splits your Android application into multiple DEX files.

Unfortunately, the use of Multidex comes with a performance cost according to an unrefereed post on Medium [41]. The conclusion from the post, that the performance cost exists, is going to be assumed to be true for this study. The assumption is based on the fact that, with Multidex, runtime environments are required to perform extra, costly I/O operations to extract the data from the multiple DEX files [40].

2.6 Language Implementation Details

This section covers a small set of implementation details for Java and Kotlin that are referenced in this study.

2.6.1 Boxing and Unboxing

Boxing and unboxing are actions that are common in both Java and Kotlin. Boxing is the process of converting a primitive to an object representing the same type. Boxing is usually done automatically by the compiler, which is called autoboxing. As an example, autoboxing occurs when a developer of

modern Java version tries to add primitives to a collection that only can hold object references like in Figure 2.13.

```
List<Integer> list = new ArrayList<>();
list.add(5); // 5 is autoboxed to an Integer object
```

Figure 2.13: Autoboxing since Java 5

If compilers did not perform autoboxing, developers would have to manually convert primitives to objects more often like in Figure 2.14.

```
List<Integer> list = new ArrayList<>();
list.add(Integer.valueOf(5));
```

Figure 2.14: Necessary manual boxing until Java 5

Unboxing is simply the reverse process of boxing, it converts an object to a primitive.

2.6.2 Java Primitive Object Caches

In order to minimize the number of object allocations that autoboxing might become responsible for, the Java compiler makes use of caches for certain values. The following quote from the Java 8 Language Specification explains which values get their respective, autoboxed object fetched from a cache, containing preallocated objects, by default:

> If the value p being boxed is true, false, a byte, or a char in the range $\backslash u0000$ to $\backslash u007f$, or an int or short number between -128 and 127 (inclusive), then let r1 and r2 be the results of any two boxing conversions of p. It is always the case that r1 == r2 [42].

2.7 Microbenchmark Tools

This section provides information about a benchmarking library for Android.

2.7.1 Jetpack Benchmark Library

For Android development, there is a library available for benchmarking Java-based or Kotlin-based applications on ART. It helps developers to increase the reliability of their benchmarks in multiple ways, for instance:

- It takes care of the runtime warm-up such that the possible impact of the JIT-compilation is factored in.
- It has the ability to lock or stabilize the clock frequency of the Android device's CPU, such that the clock frequency will not get ramped up or tuned down during benchmarks.
- It warns you if you have configured the resulting binary to be instrumented for debugging, which may result in performance penalties.
- It warns you if you try to run the benchmarks in an emulator, which, if ignored, would not reflect a real-world scenario well.

It also provides convenient solutions for measuring the performance and reporting the results back to the Android Studio console [43].

2.8 Background Summary

Both the Kotlin and the Java compiler can produce Java class files containing Java bytecode. The class files can be optimized by bytecode optimization tools and are eventually required to be transformed into DEX files if the code is supposed to be executed in Android runtime. Android runtime is the managed runtime environment in which Android applications run in. Its implementations, on Android devices that run an Android OS version that is equal or newer than version 7, are capable of interpreting, AOT-compiling, and JIT-compiling bytecode. The majority of active Android devices in 2019 ran Android version 7 or a newer version.

There also exist static optimization techniques such as loop unrolling, inlining, common subexpression elimination, code hoisting, range check elimination, dead-code elimination, and the usage of primitive object caches. JIT compilers are capable of other types of optimizations that can occur at runtime. In order to prevent JIT-compiling from skewing the results of microbenchmarks, a microbenchmark library, the Jetpack benchmark library for Android, can be utilized to factor in the effects of JIT-compilation into the results.

Chapter 3

Related Work

This section covers previous research and work that is related to the work of this study.

3.1 Evaluation of Kotlin and Java on ART

In 2018, there was a study made at the Royal Institute of Technology, that evaluated the performance of Java and Kotlin on ART [4]. The performance evaluation methodology was based on a quantitative bytecode analysis approach and running benchmarks on ART from the Computer Language Benchmarks Game (CLBG) suite. It was claimed that the CLBG suite was, to the writers' knowledge, the only cross-language benchmark suite that has been used extensively in academia.

The metrics used for the quantitative bytecode analysis to evaluate performance included memory consumption, garbage collection, boxing of primitives and bytecode n-grams.

The results showed that Kotlin was slower than Java for all the performed benchmarks and that Kotlin produced more varied and larger bytecode than Java for a majority of the benchmarks. The results also showed that the use of idiomatic Kotlin features and constructs resulted in increased heap pressure and the need for boxing of primitives.

The study overlaps in many aspects with this study, as both have the same end goal to evaluate Kotlat run time performance on ART, but with different approaches. Therefore, similar background sections can be seen in both stud-

ies.

3.2 Christophe Beyls' Qualitative Bytecode Analysis

In 2017, a software developer called Christophe Beyls posted experiments online about Kotlin 1.1. He had performed a qualitative bytecode analysis for some common Kotlin features [44].

Beyls' methodology consisted of recognizing the following actions in the bytecode that was generated by the Kotlin compiler:

- Autoboxing of primitive types.
- Instantiating objects that were not in the source code.
- Generating synthetic methods.

Since he did not publish his experiments as part of a peer-reviewed paper, it was required that his results were verified. That was accomplished for this study by compiling the same source code that he compiled, with the Kotlin 1.1 compiler, and then noticing that the generated bytecode was identical to his.

In the following subsections, his result for the qualitative bytecode analysis for a set of Kotlin features are summarized.

3.2.1 Lambdas and Higher-Order Functions

Beyls observed that lambda expressions get compiled into objects that implement a generic functional interface. He noted that if the lambda was non-capturing, i.e., that it does not capture state, then a singleton object would be created for the lambda, which would be reused for the next calls. If the lambda was capturing, then a new instance of a generic functional interface would be created every time a lambda is passed as an argument.

He also observed that autoboxing and immediate unboxing occurred when primitive types were involved [44].

3.2.2 Accessing Constant Class Variables

When companion objects were compiled, Beyls observed that they were implemented as a singleton class. He noted that reading a constant class variable requires two or three extra method calls compared to Java. Three extra methods were generated if the class variable was private, two if it was public [44].

3.2.3 Local Functions

Beyls observed that local functions are compiled into objects that implement a functional interface, like lambdas. However, he noted that since the caller of a local function knew of the instance of the functional interface, it could call the non-boxing method directly, preventing boxing and immediate unboxing if primitives were involved [45].

3.2.4 Null Safety

One thing that Beyls observed was that public functions with non-nullable reference parameters get their null safety measure implemented by calling a function Intrinsics.checkParameterIsNotNull(). He argued that this method call was not needed for private methods because the compiler can guarantee that code within a Kotlin class usually is null safe [45].

He also argued that this could be an unnecessary operation for release builds and stated that it is possible to disable the null checks that occur at run time by using the compiler option -Xno-param-assertions [45].

3.3 Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite

In a scientific paper from 2003, three computer scientists from the National University of Ireland and the Trinity College presented an analysis of dynamic profiles of Java programs when they were executed on the JVM [46].

They ran benchmarks from the Java Grande Forum benchmark suite and used five different compilers that could compile Java source code to Java bytecode. They were partly interested in the impact that the choice of compiler had on

the benchmarks.

The results included that useful information could be gained from a study of bytecode-level data. They argued that by analyzing the bytecode, through looking at method execution frequencies and bytecode counts, it was possible to get different perspectives of where the interpreter is spending its time, depending on the choice of a compiler [46].

They argued that a dynamic approach is superior to static analysis in shining light on compiler differences. The argument was based on the assumption that static analysis is not sufficient since it will not have information about what paths the program would take at run time. As an example, they mentioned that a goto instruction could add millions of extra bytecodes to the executed bytecode count or not be executed at all [46].

3.4 JDK8: Lambda Performance Study

In 2013, Oracle published a performance analysis that compared two lambda implementation approaches for the JVM [47]. One approach was to generate bytecode for an anonymous inner class to represent the lambda expression. The other approach was to use a LambdaFactory object which implemented lambdas at run time with the help of the invokedynamic instruction.

The methodology consisted of running microbenchmarks on a single desktop computer, using the Java Microbenchmark Harness (JMH). JMH helped them to get reliable results from the microbenchmarks.

When non-capturing lambdas were microbenchmarked, the source code in Figure 3.1 was used. The results showed that the invokedynamic approach was slightly faster than generating bytecode for an anonymous class, which can be seen in Table 3.1.

Approach	**single thread**
baseline	5.29 ± 0.02
anonymous	6.02 ± 0.02
cached anonymous	5.36 ± 0.01
lambda	5.31 ± 0.02

Table 3.1: Average time for the microbenchmarks, nanosecs/operation.

```
1  public static Supplier<String> lambda() {
2    return () -> "42";
3  }
4  public static Supplier<String> anonymous() {
5    return new Supplier<String>() {
6      @Override
7      public String get() {
8        return "42";
9      }
10   }
11 }
12 public static Supplier<String> baseline() {
13   return null;
14 }
```

Figure 3.1: Source code implementation of the lambda approaches for the microbenchmarks.

3.5 Related Work Summary

Schwermer evaluated Kotlin's performance on ART compared to Java's, like this study, but with another approach that involved running benchmarks from the CLBG suite [4]. Beyls' study, which was not academic, presented work from a qualitative bytecode analysis of Kotlin, which consisted of recognizing hidden costs in the form of autoboxing of primitive types, instantiation of objects that were not in the source code, and generation of synthetic methods [44] [45] [48]. Gregg et. al's paper was about the impact that the choice of compiler had on benchmarks from the Java Grande Forum benchmark suite. The paper from Oracle compared two approaches to implementing lambda expressions for the JVM. One approach was to generate bytecode for an anonymous inner class to represent the lambda expression. The other approach was to use the invokedynamic instruction and let the translation of the lambda occur at runtime.

Chapter 4

Method

This chapter presents the method that was used to answer the research question of this study. The method consists of two parts: a qualitative bytecode analysis and a quantitative bytecode analysis. First, the methodology of the qualitative bytecode analysis is presented. Then, the methodology of the quantitative approach is presented.

4.1 Language Versions

The version of Kotlin that this study evaluated was version 1.3.50, which was released on August 22, 2019 [49]. The Java version was 8, JDK 8u211, which was released on April 16, 2019 [50]. Also, since the Kotlin compiler produces Java 6 compatible bytecode by default [51], all projects for this thesis were configured such that they generated Java 8 compatible bytecode.

4.2 Qualitative Bytecode Analysis

The qualitative bytecode analysis consists of redoing Beyls' experiments of Kotlin 1.1 from 2017 [44] but for Kotlin 1.3.50 from 2019, for a subset of the features that he looked at, but also for two other features that he did not cover. The same methodology was used, which consisted of trying to recognize the following actions in the Java bytecode that was generated for each feature by both the Java and the Kotlin compiler:

- Autoboxing and immediate unboxing of primitives — it can be considered an unnecessary, negative performance impact if the object is short-lived.

- Instantiation of objects that were not in the source code — extra objects allocated on the heap leads to larger workloads for the garbage collector and can thus impact performance negatively.
- Generation of synthetic methods — more levels of indirection can impact performance negatively. Also, in Android, there is a limit for the number of method references in a single DEX file. The workaround for referencing more methods, which involves using Multidex, comes with performance penalties, as described in the Multidex section of this study's background chapter.

4.3 Analyzed Constructs

The following list contains the features that this study is interested in. They were primarily chosen based on the assumption that their implementations will differ between the two languages and that they are widely used. The features, that had already been assessed by Beyls in 2017 and got picked for this study, were picked based on if there existed room for improvement in the implementations from 2017. It would not be interesting to re-assess the features that had no hidden costs in 2017.

1. Higher-order functions and lambdas
2. Companion objects
3. Local functions
4. View binding with the Android Kotlin extension
5. Null Safety
6. Extension functions

The source code implementation of the features that were used in the bytecode analysis and the microbenchmarks were programmed with the following things in mind:

- the Kotlin and Java implementations should be idiomatic
- the features should be implemented in a minimal manner to increase the accuracy of the microbenchmarks. To the greatest extent, it is desired to microbenchmark the features and nothing more

- an extra operation should be added for both implementations, if necessary, to ensure that the implementation will not be a no-op that might get completely optimized away

4.4 Quantitative Bytecode Analysis

In order to measure the impact of the assessed feature costs or opportunities, microbenchmarks were performed. Every listed Kotlin feature and its corresponding idiomatic Java feature were microbenchmarked with the help of the Jetpack Benchmark library.

It is important to be careful when one creates microbenchmarks. Microbenchmarks can be flawed in many ways, which can ultimately lead to that the microbenchmarks measure something else than what was desired to be measured. Therefore, precautions have been taken in this study to prevent the microbenchmarks from being flawed, which are presented in the following subsections along with other information about how the microbenchmarks were set up.

4.4.1 The Benchmark Application

One Android application was created to run the microbenchmarks. The interoperability of Kotlin and Java allowed benchmarks for both languages to run in one application, in a sequential manner. The logic, that set up the application structure and the Kotlin feature benchmarks, was written in Kotlin, and the logic, that set up the Java benchmarks, was written in Java.

The project settings for the application were like in Figure 4.1.

compileSdkVersion 28
minSdkVersion 26
targetSdkVersion 28

Figure 4.1: Project settings for Android SDK versions.

A link to a repository containing the source code for the benchmark application can be found in the appendix of this study.

4.4.2 Android Device for the Microbenchmarks

The Android device that was used for the microbenchmarks was a Huawei P20 Pro smartphone from 2018. It runs Android 9, it has a CPU with 4 cores @2.36 GHz, and it has 6 GB of RAM.

4.4.3 Static Compiler Optimizations

In order to make sure that microbenchmarks test what has been written in the source code, it is important to understand static compiler optimizations. Some optimizations that have been deemed to have a potential impact on microbenchmarks for this study are the following: loop unrolling, inlining, common subexpression elimination, code hoisting, range check elimination, dead-code elimination, and primitive object caches. Therefore, these optimizations were taken into consideration when the benchmarks were written. For instance, measuring the performance of the features inside loops was avoided due to the static loop optimizations.

4.4.4 Dynamic Compiler Optimizations

Since runtimes usually start executing a bytecode path by interpreting it instead of directly JIT-compiling, it was taken into consideration that JIT-compilation might not happen until the runtime notices that certain code paths are "hot" and thus decides that it will be worth to JIT-compile it. Consequently, it can be said that a runtime has a warm-up time. The warm-up time for the benchmarks was considered as it might affect the results significantly. For instance, if JIT-compilation occurs in the middle of a benchmark, the results will be a sum of interpreting the first lines of code, the delay of the JIT-compilation, and the execution of the JIT optimized code [52]. The Jetpack benchmark library helped to mitigate the impact of the warm-up time by recognizing when the JIT-compilation had occurred.

4.4.5 Hardware Configuration

It is important that the hardware capabilities remain stable while the benchmarks run. If a device changes its CPU's clock frequency in the middle of a benchmark, the results might get skewed. Therefore, the Jetpack benchmark library was used for the benchmarks to help with maintaining a stable clock frequency of the Android device's CPU [43].

4.4.6 Device Configuration

To enable the test device to utilize its resources to the full potential, all background applications were killed. Flight mode was also enabled to minimize the impact of I/O processes that could not be killed in any other way.

4.4.7 Android Project Configuration

It is important to configure the Android project such that the benchmarks run in a release build of the application. The Jetpack benchmark library helped to ensure that the build did not get instrumented for debugging or code coverage that might have impaired the performance. It also made sure that the application ran in the foreground, which allowed the application to utilize the exclusive foreground core.

4.4.8 Benchmark Result Sampling

For each microbenchmark, the Jetpack benchmark library performed a warm-up phase. The number of iterations for the warm-up phase depended on when the results started to stabilize according to the built-in logic of the library. The results of the warm-up were then discarded.

The number of iterations for the microbenchmarks in the warmed-up runtime was also calculated with the built-in logic of the benchmark library. The results of the microbenchmarks were reported in terms of the mean of the execution times, the fastest execution time, and the standard deviation of the execution time distribution.

4.4.9 Bytecode Optimization Tools

As bytecode optimization tools such as ProGuard and R8 are widely used as of the time during this study is written, it is reasonable to assume that they will be used in a real scenario of a Java to Kotlin migration for an Android application. If it turns out that Kotlin has hidden costs, it is interesting to see if bytecode optimization tools can optimize the costs away such that developers do not have to think of them while writing the source code. Therefore, the bytecode optimization tools ProGuard and R8 were used in additional benchmarks.

4.4.10 Manual Source Code Optimization

In order to get an indication if it is worth a programmer's time to manually optimize source code, another microbenchmark was run to see how manual optimizations match up with optimizations done by bytecode optimization tools. If there existed a way to optimize the source code, such as using the inline keyword, it was done for the last microbenchmark. The desired metric to optimize was the execution speed. The manual optimizations that were made for the features can be seen in the source code for the microbenchmarks that are linked to in the appendix of this study.

4.5 Omission of Kotlin Coroutines

One may expect that coroutines would be assessed for a run time performance evaluation of Kotlin, given their popularity and their ability to leverage resource-efficient asynchronous programming. However, this study is not interested in coroutines because it would be meaningless to measure their performance with the metrics of the bytecode analysis that was performed in this study. The results would not reflect coroutines' main ability: how they handle resources when it comes to concurrency. It would also be difficult to try to microbenchmark coroutines and expect to get any meaningful results, as benchmarking concurrency would introduce more challenges when it comes to the reliability of the results.

4.6 Method Summary

The method for getting an answer to the two sub-questions of the research question consisted of two parts.

First, a qualitative bytecode analysis was performed on the compiled implementations of a set of six features, for both languages, with the purpose of identifying hidden costs and opportunities that were not visible in the source code. Three different costs were looked for: autoboxing and immediate unboxing of primitives, instantiation of objects that were not visible in the source code, and generation of methods that were not visible in the source code. What were considered opportunities for Kotlin were the lack of these three costs compared to the Java solution.

Secondly, as a quantitative bytecode analysis, three different microbenchmarks were performed for the implementations of the set of six features. First, non-optimized microbenchmarks were run to measure the impact of the features' hidden costs or opportunities. Then, the microbenchmarks were run again but after having been optimized by the industry-standard [37] bytecode optimization tools ProGuard and R8. The purpose of the second set of microbenchmarks was to determine if the potential hidden costs could be mitigated by bytecode optimization tools. Finally, microbenchmarks were run a third time, but this time after having been manually optimized in the source code and not been processed by bytecode optimization tools. The purpose of the last set of microbenchmarks was to get an indication of if it is fruitful for a developer to perform manual optimizations in the source code or if bytecode optimization tools could be relied on.

Chapter 5

Results and Discussion

This chapter presents and discusses the results of the qualitative bytecode analysis and the microbenchmarks.

5.1 Qualitative Bytecode Analysis

In the following subsections, examples of idiomatic Kotlin and Java feature implementations and their respective, corresponding Java bytecode are shown.

Some bytecodes have been omitted in the figures of this chapter since they are not relevant to the metrics of interest for the analysis and will just decrease the readability. It can be realized where bytecodes have been omitted if a whole code part of a method has been omitted or if there are three dots in the bytecode sequences.

5.1.1 Lambdas and Higher-Order Functions

For each declared lambda expression like in Figure 5.1, Kotlin will generate bytecode for an anonymous inner class that implements a functional interface for the given lambda. This generated class, which can be seen in line 16–33 in Figure 5.2 will reside in a new class file, adding three new methods to the total method count in this case.

Moreover, since Kotlin still does not have specialized functional interfaces that handle different primitive types — and that lambda expressions get compiled into instances of classes which implement fully generic functional interfaces — Beyls' observation still holds, regarding that Kotlin systematically will box

and unbox arguments or return values if they have primitive types. The boxing, in this case to an Integer object, can be seen in line 22 in Figure 5.2. The Integer object, cast to a Number object, will then immediately be unboxed in line 7 in Figure 5.2.

Lastly, since the lambda, in this case, does not capture any state from outside its scope, we notice that Kotlin has created a singleton instance of the anonymous inner class by recognizing the singleton pattern in line 17 and 28-31 in Figure 5.2.

```
1 fun higherOrderFunction(lambda: (KotlinIntSupplier) -> Int) {
2     val num = lambda(KotlinIntSupplier())
3   }
4 }
5 fun foo() {
6   higherOrderFunction({ x -> x.getInt() })
7 }
```

Figure 5.1: Kotlin lambda (non-capturing)

```
1  public final class KotlinNonCapturingLambdaKt {
2    public static final void higherOrderFunction(kotlin.jvm.functions.Function1
         ↪ <? super KotlinIntSupplier, java.lang.Integer>);
3      Code:
4        ...
5        12: invokeinterface 19, 2 // InterfaceMethod
             kotlin/jvm/functions/Function1.invoke:(Ljava/lang/Object;)Ljava/lang/Object;
6        17: checkcast #21 // class java/lang/Number
7       20: invokevirtual 25 // Method java/lang/Number.intValue:()I
8        ...
9    public static final void foo();
10     Code:
11       0: getstatic #48 // Field KotlinNonCapturingLambdaKt$foo$1.INSTANCE:
             ↪ LKotlinNonCapturingLambdaKt$foo$1;
12       3: checkcast #15 // class kotlin/jvm/functions/Function1
13       6: invokestatic #50 // Method higherOrderFunction:(Lkotlin/jvm/functions/
             ↪ Function1;)V
14       9: return
15 }
16 final class KotlinNonCapturingLambdaKt$foo$1 extends kotlin.jvm.internal.
       ↪ Lambda implements kotlin.jvm.functions.Function1<KotlinIntSupplier,
       ↪ java.lang.Integer> {
17   public static final KotlinNonCapturingLambdaKt$foo$1 INSTANCE;
18   public java.lang.Object invoke(java.lang.Object);
19     Code:
20       ...
21       5: invokevirtual #14 // Method invoke:(LKotlinIntSupplier;)I
22       8: invokestatic 20 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
23       11: areturn
24   public final int invoke(KotlinIntSupplier);
25   KotlinNonCapturingLambdaKt$foo$1(); // Constructor
26   static {};
27     Code:
28       0: new #2 // class KotlinNonCapturingLambdaKt$foo$1
29       3: dup
30       4: invokespecial #62 // Method "<init>":()V
31       7: putstatic      #64 // Field INSTANCE:LKotlinNonCapturingLambdaKt$foo$1;
32       10: return
33 }
```

Figure 5.2: Bytecode for the Kotlin code in Figure 5.1

There is a way to eliminate these costs. As Beyls observed, if we declare the higher-order function with the inline keyword, the costs disappear. In Figure 5.3 we can see that the body of the higher-order function has been put inside the foo() function instead of a call to the higher-order function, which means that no boxing and unboxing will take place, and that no anonymous inner class that contains extra methods will be generated.

```
1  public final class KotlinInlineNonCapturingLambdaKt {
2    public static final void higherOrderFunctionInline(kotlin.jvm.functions.
         ↪ Function1<? super KotlinIntSupplier, java.lang.Integer>);
3    public static final void foo();
4      Code:
5        0: nop
6        1: new           #17 // class KotlinIntSupplier
7        4: dup
8        5: invokespecial #21 // Method KotlinIntSupplier."<init>":()V
9        8: astore_0
10       9: aload_0
11       10: invokevirtual #53 // Method KotlinIntSupplier.getInt:()I
12       13: istore_0
13       14: getstatic     #39 // Field java/lang/System.out:Ljava/io/PrintStream;
14       17: iload_0
15       18: invokevirtual #45 // Method java/io/PrintStream.println:(I)V
16       21: nop
17       22: return
18 }
```

Figure 5.3: Bytecode for the Kotlin code in Figure 5.1 if the function higherOrderFunction was inlined.

Regarding the Java approach, in line 9 in Figure 5.4, we observe that a higher-order function is called with a lambda expression as an argument. The generated bytecode for this method call can be seen starting from line 14 in Figure 5.5, where we observe that Java makes use of the invokedynamic instruction. The invokedynamic instruction is associated with a bootstrap method, that is part of the LambdaMetaFactory class and returns a CallSite object, which can be seen in line 4 and 29 in Figure 5.5). In the same figure, we notice that Java does not generate bytecode for an anonymous inner class.

We do also observe, in line 12 in Figure 5.5, that the use of a specialized functional interface leads to that a primitive integer is returned from the lambda invocation instead of a boxed integer.

```
1  public class JavaNonCapturingLambda {
2    void higherOrderFunction(ToIntFunction<JavaIntSupplier> lambda) {
3      if (lambda != null) {
4        int num = lambda.applyAsInt(new JavaIntSupplier());
5        System.out.println(num);
6      }
7    }
8    void foo() {
9      higherOrderFunction(x -> x.getInt());
10   }
11 }
```

Figure 5.4: Java lambda (non-capturing)

```
1  public class JavaNonCapturingLambda
2  Constant pool:
3    ...
4    #7 = InvokeDynamic #0:#50 // #0:applyAsInt:()Ljava/util/function/
         ↪ ToIntFunction;
5    ... // Bytecodes for printing
6  {
7    public JavaNonCapturingLambda();
8
9    void higherOrderFunction(java.util.function.ToIntFunction<JavaIntSupplier>)
         ↪ ;
10     Code:
11     ...
12     12: invokeinterface #4, 2 // InterfaceMethod java/util/function/
           ↪ ToIntFunction.applyAsInt:(Ljava/lang/Object;)I
13     ... // Bytecode for printing
14   void foo();
15     Code:
16     0: aload_0
17     1: invokedynamic #7, 0 // InvokeDynamic \#0:applyAsInt:()Ljava/util/
           ↪ function/ToIntFunction;
18     6: invokevirtual #8 // Method higherOrderFunction:(Ljava/util/function/
           ↪ ToIntFunction;)V
19     9: return
20   private static int lambda$foo$0(JavaIntSupplier);
21     Code:
22     0: aload_0
23     1: invokevirtual #9 // Method JavaIntSupplier.getInt:()I
24     4: ireturn
25 }
26 InnerClasses:
27      public static final #75= #74 of #78; //Lookup=class java/lang/invoke/
          ↪ MethodHandles$Lookup of class java/lang/invoke/MethodHandles
28 BootstrapMethods:
29   0: #46 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/
         ↪ lang/invoke/MethodHandles$Lookup;Ljava/lang/String;Ljava/lang/
         ↪ invoke/MethodType;Ljava/lang/invoke/MethodType;Ljava/lang/invoke/
         ↪ MethodHandle;Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/
         ↪ CallSite;
30     Method arguments:
31       #47 (Ljava/lang/Object;)I
32       #48 invokestatic JavaNonCapturingLambda.lambda$foo$0:(LJavaIntSupplier
           ↪ ;)I
33       #49 (LJavaIntSupplier;)I
```

Figure 5.5: Bytecode for the Java lambda expression.

Discussion

The fact that Kotlin generated an anonymous inner class with three extra methods and instantiated it does not necessarily mean that the approach always is worse than Java's approach of using invokedynamic and generating only one method. The performance for the Java approach is heavily depending on how

the runtime decides to translate the lambda. However, with the Kotlin approach, it is always the case that generating bytecode for anonymous inner classes which leads to bigger DEX files, loading of more class files at run time, and more interpretation, can eventually add up to a non-negligible performance cost. Even if it turns out that the Java lambda is implemented as an anonymous inner class at run time, it will probably occur much faster compared to interpreting the bytecode for an anonymous inner class.

5.1.2 Accessing a Class Variable

For the creation of a companion object, Kotlin generates a new class for the companion object and stores a singleton instance of it within the main class. The members of the companion object get stored as fields of the main class, which allows the companion object members to be accessed from the main class with a single $getstatic$ operation, as can be seen in line 6 in Figure 5.7.

```
1 class ClassWithCompanion {
2   companion object {
3     private val TAG = "Hello"
4   }
5   fun hello() {
6     println(TAG)
7   }
8 }
```

Figure 5.6: An example of accessing a constant class variable in Kotlin.

```
1  public final class ClassWithCompanion {
2    private static final java.lang.String TAG;
3    public static final ClassWithCompanion$Companion Companion;
4    public final void hello();
5      Code:
6         0: getstatic     #10 // Field TAG:Ljava/lang/String;
7         3: astore_1
8         4: getstatic     #16 // Field java/lang/System.out:Ljava/io/
               ↪ PrintStream;
9         7: aload_1
10        8: invokevirtual #22 // Method java/io/PrintStream.println:(Ljava/lang
               ↪ /Object;)V
11       11: return
12   public ClassWithCompanion();
13     ...
14   static {};
15     Code:
16        0: new #47 // class ClassWithCompanion$Companion
17        ...
18        5: invokespecial #50 // Method ClassWithCompanion$Companion."<init>":(
               ↪ Lkotlin/jvm/internal/DefaultConstructorMarker;)V
19        8: putstatic     #52 // Field Companion:LClassWithCompanion$Companion;
20       11: ldc           #29 // String Hello
21       13: putstatic     #10 // Field TAG:Ljava/lang/String;
22       16: return
23 }
24 public final class ClassWithCompanion$Companion {
25   private ClassWithCompanion$Companion();
26     ...
27   public ClassWithCompanion$Companion(kotlin.jvm.internal.
       ↪ DefaultConstructorMarker);
28     ...
29 }
```

Figure 5.7: Content of the two class files that were generated for the class with the companion object.

For the Java case, nothing that cannot be seen in the source code in Figure 5.9 occurs. Private class variables can get accessed with a single *getstatic* operation, as can be seen in line 11 in Figure 5.9.

```
1 public class JavaClassVariable {
2   private static String TAG = "Hello";
3   void hello() {
4     System.out.println(TAG);
5   }
6 }
```

Figure 5.8: An example of accessing a constant class variable in Java.

```
1  public class JavaClassVariable {
2    private static java.lang.String TAG;
3    public JavaClassVariable();
4      Code:
5         0: aload_0
6         1: invokespecial #1 // Method java/lang/Object."<init>":()V
7         4: return
8    void hello();
9      Code:
10        0: getstatic     #2 // Field java/lang/System.out:Ljava/io/PrintStream
              ↪ ;
11        3: getstatic #3 // Field TAG:Ljava/lang/String;
12        6: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/
              ↪ String;)V
13        9: return
14   static {};
15     Code:
16        0: ldc           #5 // String Hello
17        2: putstatic     #3 // Field TAG:Ljava/lang/String;
18        5: return
19 }
```

Figure 5.9: Java bytecode for accessing a constant class variable in Java.

Discussion

Regarding the implementation of Kotlin's companion objects, it seems like the underlying logic has been improved over time since Kotlin 1.1. From having to make two or three extra method calls compared to Java in order to access a constant class variable, to only having to make one method call like Java has done since Day 1. Based on this, it could be argued that companion objects do not have a hidden cost anymore when it comes to accessing constant variables.

5.1.3 Local Functions

As can be observed from the compiled version of the source code in Figure 5.10, Beyls' observation that local functions get converted to Function objects, like lambdas do as well, still holds. However, the use of local functions do not get compiled exactly like the use of lambdas. Since the instance of the local function is known to the caller, the caller can directly call the function without first calling a generic, synthetic function that boxes the integer — just as Beyls observed in 2017. This can be seen in line 5 in Figure 5.11 where the method with the method descriptor $I(I)$ is called instead of the boxing method with the $Ljava/lang/Object;(Ljava/lang/Object)$ method descriptor.

```
1  fun someMath(a: Int): Int {
2    fun sumSquare(b: Int) = (a + b) * (a + b)
3    return sumSquare(128)
4  }
```

Figure 5.10: A local function in Kotlin.

```
1  public final class LocalFunctionKt {
2    public static final int someMath(int);
3      Code:
4         ...
5         11: invokevirtual #15 // Method LocalFunctionKt$someMath$1.invoke:(I)I
6         14: ireturn
7  }
8  final class LocalFunctionKt$someMath$1 extends kotlin.jvm.internal.Lambda
       ↪ implements kotlin.jvm.functions.Function1<java.lang.Integer, java.
       ↪ lang.Integer> {
9  public java.lang.Object invoke(java.lang.Object);
10 Code:
11    ...
12    8: invokevirtual #18 // Method invoke:(I)I
13   11: invokestatic  #24 // Method java/lang/Integer.valueOf:(I)Ljava/lang/
         ↪ Integer;
14   14: areturn
15 public final int invoke(int);
16 LocalFunctionKt$someMath$1(int);
17 }
```

Figure 5.11: Bytecode for a Kotlin local function

Since there does not exist syntactic support for local functions in Java, the Java solution for creating a local function is to declare a local lambda expression.

It can be observed that when the source code in Figure 5.12 is compiled, similar bytecode as in the lambda and higher-order function case is generated, which can be seen in Figure 5.13. Like in the higher-order function case, we observe that Java is able to utilize a specialized functional interface, $ToIntFunction$, to avoid boxing and then immediate unboxing, which can be seen on line 11 in Figure 5.13.

```
int someMath(int a) {
  ToIntFunction<Integer> sumSquare = (b) -> (a + b) * (a + b);
  return sumSquare.applyAsInt(128);
}
```

Figure 5.12: The Java solution for a local function.

```
public class JavaLocalFunction {
  public JavaLocalFunction(); // Constructor
  int someMath(int);
    Code:
     0: iload_1
     1: invokedynamic #2,  0 // InvokeDynamic #0:applyAsInt:(I)Ljava/util/
         ↪ function/ToIntFunction;
     6: astore_2
     7: aload_2
     8: sipush        128
     11: invokestatic  #3 // Method java/lang/Integer.valueOf:(I)Ljava/lang/
         ↪ Integer;
     14: invokeinterface #4,  2 // InterfaceMethod java/util/function/
         ↪ ToIntFunction.applyAsInt:(Ljava/lang/Object;)I
     19: ireturn
  private static int lambda$someMath$0(int, java.lang.Integer);
}
```

Figure 5.13: Bytecode for the Java code in Figure 5.12

Discussion

Since the results show that Kotlin implements local functions by generating an anonymous inner class, it could be argued that local functions should be avoided if one does not desire the extra methods and the instantiated object that follows with it. However, it should be noticed that they bring one less hidden cost compared to normal lambda expressions. They do not box primitives and immediately unbox it since the signature of the local function is known by the caller.

5.1.4 View Binding

This section presents how the binding of a text view was compiled. For the Kotlin approach, functionality from the Kotlin Android extension was used for accessing the view.

The Java Way

In line 6 in Figure 5.14 an object, that represents a text view, is accessed and then gets its text field modified. The generated bytecode for this logic can be found in Figure 5.15, where Java does nothing else than what can be seen in the source code.

```
1 public class JavaViewBinding extends Activity {
2     @Override
3     protected void onCreate(@Nullable Bundle savedInstanceState) {
4         super.onCreate(savedInstanceState);
5         setContentView(R.layout.test_layout);
6         ((TextView) findViewById(R.id.textView)).setText("Hello");
7     }
8 }
```

Figure 5.14: Java code for finding a text view and setting its text.

```
1  public class com.jqhan.kotlin_benchmarks.JavaViewBinding extends android.app.
       ↪ Activity {
2    public com.jqhan.kotlin_benchmarks.JavaViewBinding(); // Constructor
3    protected void onCreate(android.os.Bundle);
4      Code:
5        ...
6        12: ldc           #7 // int 2131165337
7        14: invokevirtual #8 // Method findViewById:(I)Landroid/view/View;
8        17: checkcast     #9 // class android/widget/TextView
9        20: ldc           #10 // String Hello
10       22: invokevirtual #11 // Method android/widget/TextView.setText:(Ljava/
             ↪ lang/CharSequence;)V
11       25: return
12 }
```

Figure 5.15: Bytecode for Java's way of finding a text view and setting its text.

The Android Kotlin Extension Way

In line 6 in Figure 5.16 an object, that represents a text view, is accessed and then gets its text field modified. The generated bytecode for this logic can be found in Figure 5.17, where Kotlin does more things than what written in the source code.

As can be observed in line 7 in Figure 5.17, Kotlin calls a synthetic method that checks if the wanted view exists in a view cache that belongs to the class.

If the view does not exist in the cache, Kotlin calls the method findViewById like Java did in Figure 5.14.

```
class KotlinViewBinding: Activity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.test_layout)
    textView?.text = "Hello"
  }
}
```

Figure 5.16: Kotlin code for finding a text view and setting its text element.

```
public final class com.jqhan.kotlin_benchmarks.KotlinViewBinding extends
    ↪ android.app.Activity {
  private java.util.HashMap
  protected void onCreate(android.os.Bundle);
    Code:
      ...
      12: getstatic #20 // Field com/jqhan/kotlin_benchmarks/R$id.textView:I
      15: invokevirtual #24 // Method _$_findCachedViewById:(I)Landroid/view/View;
      18: checkcast     #26 // class android/widget/TextView
      ...
      25: ldc           #28 // String Hello
      27: checkcast     #30 // class java/lang/CharSequence
      30: invokevirtual #34 // Method android/widget/TextView.setText:(Ljava/
          ↪ lang/CharSequence;)V
      ...
  public com.jqhan.kotlin_benchmarks.KotlinViewBinding(); // Constructor
  public android.view.View _$_findCachedViewById(int) {
    if(_$_findViewCache == null)
      _$_findViewCache = new HashMap();
    View view = (View)_$_findViewCache.get(Integer.valueOf(i));
    if(view == null) {
      view = findViewById(i);
      _$_findViewCache.put(Integer.valueOf(i), view);
    }
    return view;
  }
  public void _$_clearFindViewByIdCache();
}
```

Figure 5.17: Bytecode and Java code for Kotlin's the code in 5.16. The bytecode for the method _$_findCachedViewById was decompiled into Java for better readability. The full bytecode can be found in the appendix.

Moreover, it can be observed that the view cache and its related methods will be generated even though if there is no view binding occurring in the source code. An example of this can be seen in Figure 5.18 and Figure 5.19,

where the view cache and its methods are generated for an empty Activity class. This is also the case for Fragment classes, which can be seen in Figure 5.20 and Figure 5.21.

```
1 class KotlinViewBindingEmptyActivity: Activity() {}
```

Figure 5.18: Kotlin code for an empty Activity class.

```
1 public final class com.jqhan.kotlin_benchmarks.KotlinViewBindingEmptyActivity
      ↪ extends android.app.Activity {
2   private java.util.HashMap _$_findViewCache;
3   public com.jqhan.kotlin_benchmarks.KotlinViewBindingEmptyActivity();
4   public android.view.View _$_findCachedViewById(int);
5   public void _$_clearFindViewByIdCache();
6 }
```

Figure 5.19: Bytecode for the Kotlin code in Figure 5.18.

```
1 class KotlinViewBindingEmptyFragment: Fragment() {}
```

Figure 5.20: Kotlin code for an empty Fragment class.

```
1 public final class com.jqhan.kotlin_benchmarks.KotlinViewBindingEmptyFragment
      ↪ extends androidx.fragment.app.Fragment {
2   private java.util.HashMap _$_findViewCache;
3   public com.jqhan.kotlin_benchmarks.KotlinViewBindingEmptyFragment();
4   public android.view.View _$_findCachedViewById(int);
5   public void _$_clearFindViewByIdCache();
6   public void onDestroyView();
7 }
```

Figure 5.21: Bytecode for the Kotlin code in Figure 5.20.

Discussion

The results for bytecode analysis of the view binding, which shows that Kotlin generates up to two extra objects and two synthetic methods, do not necessarily mean that the performance will be impacted negatively. Due to the nature

of this generated object, the view cache, the direction of the impact on the performance depends on the number of times that the view will be accessed. For the first access of the view, the cache logic will bring an overhead to performance, but for future accesses of the same view, it will probably start to pay off.

Moreover, since the view cache and its related methods are generated even for empty Activity or Fragment classes, the use of the Android Kotlin extension way of binding views should not be avoided if one uses Kotlin for Android development. It could be considered pointless to call the regular findViewById method to bind a view if the logic for view cache is already implemented — one might as well use the cache.

5.1.5 Null Safety

This section shows how Kotlin implements some of its null safety features.

For the Kotlin code that is regarding the null safety features in Figure 5.22, we observe that no new objects have been instantiated, that no synthetic methods have been generated, and that there is no boxing of primitives occurring. However, as Beyls noticed, Kotlin makes a call to a static function Intrinsics.checkParameterIsNotNull, which can be observed in line 6 in Figure 5.23, if the function is public. The functions fooPublic and fooPrivate are identical except when it comes to their visibility modifier. We notice that the private function did not make this method call.

Other than what has already been observed, nothing else was generated that we could not expect given that we are aware of how the call safe and the not-null assertion operators are implemented.

```
class KotlinNullSafety() {
  public fun fooPublic(x: String) {
    println(x)
  }
  private fun fooPrivate(x: String) {
    println(x)
  }
  fun fooNullable(x: String?) {
    println(x?.length)
    println(x!!.length)
  }
}
```

Figure 5.22: Kotlin functions that get handled with different null safety measures.

```
public final class KotlinNullSafety {
  public final void fooPublic(java.lang.String);
    Code:
    0: aload_1
    1: ldc           #9 // String x
    3: invokestatic #15 // Method
        kotlin/jvm/internal/Intrinsics.checkParameterIsNotNull:(Ljava/lang/Object;Ljava/lang/String;)V
    ... // Bytecodes for printing
  private final void fooPrivate(java.lang.String);
    Code:
    ... // Bytecodes for printing
  public final void fooNullable(java.lang.String);
    Code:
    0: aload_1
    1: dupk
    2: ifnull        14
    5: invokevirtual #39 // Method java/lang/String.length:()I
    8: invokestatic  #45 // Method java/lang/Integer.valueOf:(I)Ljava/lang/
       ↪ Integer;
    11: goto          16
    14: pop
    15: aconst_null
    16: astore_2
    ... // Bytecodes for printing
    24: aload_1
    25: dup
    26: ifnonnull     32
    29: invokestatic  #49 // Method kotlin/jvm/internal/Intrinsics.throwNpe
       ↪ :()V
    32: invokevirtual #39 // Method java/lang/String.length:()I
    35: istore_2
    ... // Bytecodes for printing
    43: return
  public KotlinNullSafety(); // Constructor
```

Figure 5.23: Bytecode for the Kotlin code in 5.22.

The equivalent Java code for the Kotlin code in Figure 5.22 would be like in Figure 5.24. The bytecode for the Java code in Figure 5.24 can be seen in 5.24 and contains nothing that could not be understood from reading the source code.

```
1  public class JavaNullSafety {
2    public void fooPublic(String x) {
3      System.out.println(x.length());
4    }
5    private void fooPrivate(String x) {
6      System.out.println(x.length());
7    }
8    void fooNullable(String x) {
9      System.out.println(x != null ? x.length() : null);
10     System.out.println(x.length());
11   }
12 }
```

Figure 5.24: Java methods for the null safety related examples.

```
1  public class JavaNullSafety {
2    public JavaNullSafety(); // Constructor
3    public void fooPublic(java.lang.String);
4      Code:
5      ... // Bytecodes for printing
6    private void fooPrivate(java.lang.String);
7      Code:
8      ... // Bytecodes for printing
9    void fooNullable(java.lang.String);
10     Code:
11     0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
12     3: aload_1
13     4: ifnull        17
14     7: aload_1
15     8: invokevirtual #4 // Method java/lang/String.length:()I
16     11: invokestatic  #5 // Method java/lang/Integer.valueOf:(I)Ljava/lang/
           ↪ Integer;
17     14: goto          18
18     17: aconst_null
19     18: invokevirtual #6 // Method java/io/PrintStream.println:(Ljava/lang/
           ↪ Object;)V
20     ... // Bytecodes for printing
21     31: return
22 }
```

Figure 5.25: Bytecode for the Java code in Figure 5.24.

Discussion

The only interesting finding, that public functions with non-nullable parameters get compiled to code that involves making a method call to check if the parameter is null, is a questionable hidden cost. Given that it is possible to instruct the compiler to not generate this hidden method call for release builds, in which you should be sure that no non-Kotlin class can call the public function with null arguments, it should not be considered a hidden cost.

5.1.6 Extension Functions

When one declares an extension function like in line 5 in Figure 5.26, it can be observed that a public method, belonging to the class that the extension function was declared in, is generated, which can be seen in line 2–11 in Figure 5.27. This method will act as the extension function.

```
1  class KotlinFoo(){
2    public val x = 0
3  }
4  class KotlinExtensions{
5    fun KotlinFoo.extensionFunction() = x + 128
6    fun printInt() {
7      val a = KotlinFoo()
8      println(a.extensionFunction())
9    }
10 }
```

Figure 5.26: Kotlin code that makes use of an extension function.

```
1  public final class KotlinExtensions {
2    public final int extensionFunction(KotlinFoo);
3      Code:
4        0: aload_1
5        1: ldc           #9 // String receiver$0
6        3: invokestatic  #15 // Method kotlin/jvm/internal/Intrinsics.
             ↪ checkParameterIsNotNull:(Ljava/lang/Object;Ljava/lang/String;)V
7        6: aload_1
8        7: invokevirtual #21 // Method KotlinFoo.getX:()I
9        10: sipush        128
10       13: iadd
11       14: ireturn
12   public final void printInt();
13     Code:
14       0: new #17 // class KotlinFoo
15       3: dup
16       4: invokespecial #30 // Method KotlinFoo."<init>":()V
17       7: astore_1
18       8: aload_0
19       9: aload_1
20       10: invokevirtual #32 // Method extensionFunction:(LKotlinFoo;)I
21       ... // Bytecode for printing
22       21: return
23   public KotlinExtensions();
24 }
25 public final class KotlinFoo {
26   private final int x;
27   public final int getX();
28   public KotlinFoo();
29 }
```

Figure 5.27: Bytecode for the Kotlin code in Figure 5.26

Since Java does not have syntactic support for extension functions, the decorator design pattern was utilized for the Java extension function approach, as shown in Figure 5.28. It can be observed that this required an extra class to be declared and an extra object to be instantiated. The instantiation of the extra object can be seen in line 5 in Figure 5.29.

Regarding the method count, it can be observed that 5 methods, including constructors, were required for the task for both approaches.

```
1  public class JavaExtensions {
2    void printInt() {
3      JavaFooDecorator a = new JavaFooDecorator(new JavaFoo());
4      System.out.println(a.extensionFunction());
5    }
6  }
7  public class JavaFoo {
8    final int x = 0;
9  }
10 public class JavaFooDecorator extends JavaFoo {
11   private JavaFoo mJavaFoo;
12   JavaFooDecorator(JavaFoo javaFoo) {
13     mJavaFoo = javaFoo;
14   }
15   int extensionFunction() {
16     return mJavaFoo.x + 128;
17   }
18 }
```

Figure 5.28: Java code that makes use of an "extension function" by utilizing the decorator design pattern.

```
1  public class JavaExtensions {
2    public JavaExtensions();
3    void printInt();
4      Code:
5      0: new #2 // class JavaFooDecorator
6      3: dup
7      4: new #3 // class JavaFoo
8      7: dup
9      8: invokespecial #4 // Method JavaFoo."<init>":()V
10     11: invokespecial #5 // Method JavaFooDecorator."<init>":(LJavaFoo;)V
11     ... // Bytecode for printing
12     25: return
13 }
14 public class JavaFoo {
15   final int x;
16   public JavaFoo(); // Constructor
17 }
18 public class JavaFooDecorator extends JavaFoo {
19   private JavaFoo mJavaFoo;
20   JavaFooDecorator(JavaFoo); // Constructor
21   int extensionFunction();
22     Code:
23     0: aload_0
24     1: getfield      #2 // Field mJavaFoo:LJavaFoo;
25     4: invokevirtual #3 // Method java/lang/Object.getClass:()Ljava/lang/
           ↪ Class;
26     7: pop
27     8: iconst_0
28     9: sipush        128
29     12: iadd
30     13: ireturn
31 }
```

Figure 5.29: Bytecode for the Java code in Figure 5.26

Discussion

If the Java alternative for using an extension function is to utilize the decorator pattern, it could be argued that Kotlin's extension functions can bring an opportunity when it comes to performance because it is not required to declare an extra class and instantiate an extra object.

Regarding the number of generated functions for the both approaches, even if it was observed that both approaches had a method count of five, it should be realized that one of the methods for the Kotlin case was a getter method for the integer instance variable. It is questionable if this method should have counted as an addition to the method count, since the generation of the method could have been prevented if one would have provided the annotation "@JvmField" above the instance variable or changed its visibility to public.

5.1.7 Bytecode Analysis Summary

In Table 5.1, the number of extra instantiated objects, extra generated methods, and occurances of boxing and unboxing of short-lived objects, are listed for the set of analyzed features.

Feature	**Number of instantiated objects, Kotlin (Java)**	**Number of generated methods, Kotlin (Java)**	**Boxes primitives, Kotlin (Java)**
Non-capturing lambda	3 (1)	3 (1)	Yes (No)
Local functions	2 (1)	3 (1)	No (No)
Class variable access with companion object	1 (0)	0 (0)	N/A
View binding, the Kotlin Android extension approach	2 (0)	2 (0)	N/A
Parameter null safety in a public function	1 (1)	0 (0)	N/A
Extension function	1 (2)	0 (0)	N/A

Table 5.1: A summary of the observations for the Kotlin and Java solutions.

It can be observed that the bytecode produced by Kotlin can instantiate up to two extra objects and generate up to two extra methods compared to Java for the set of six feature implementations. It can also be observed that Kotlin performs systematic boxing and unboxing for short-lived objects if primitive values are passed as arguments to, or returned from, a lambda expression that is passed to a higher-order function. However, for the case of a Kotlin local function, which gets compiled to an anonymous inner class, like a Kotlin lambda, the boxing does not occur. Lastly, in the extension function case, it can be observed that Kotlin instantiates one less object than Java if the Java approach for an extension function is to utilize the decorator pattern.

5.2 Microbenchmark results

The boxplots in Figures 5.30–5.35 present the statistical material of the microbenchmarks for each feature. In each boxplot, the whiskers show the lowest

and the highest values that represent the execution times in milliseconds. The ends of the box mark the first and the third quartile such that the box contains 50 % of the values. The gap between the quartiles and the whiskers contain 25 % of the values, respectively. Outliers, values that are located more than 1.5 box lengths away from the box, are presented as dots.

The average execution times and the differences between the non-optimized microbenchmarks can be seen in Table 5.2.

The microbenchmarks showed a variety of time differences between the performance of the Java and Kotlin implementations. The differences ranged from Kotlin extension functions being 53.3 % faster than the Java implementation, to capturing Kotlin lambdas being 18.15 % slower than Java's. A student's t-test was performed, which can be found in Appendix B, on the time measurement distributions of each of the six microbenchmarks to see if the execution time differences were statistically significant. The results showed that the differences of the Kotlin and Java execution times, in all microbenchmarks, were statistically significant at a $p \leq 0.05$ significance level. Thus, the null hypothesis, that the arithmetic means of the microbenchmarks for each feature are the same, was rejected.

From the results of the microbenchmarks with optimizations, it could be observed that the manual optimization did not result in better performance than in the case when the code had been optimized by bytecode optimization tools, except for the case of the view-binding microbenchmark in which the bytecode optimization tool approaches performed worse than the manual optimization.

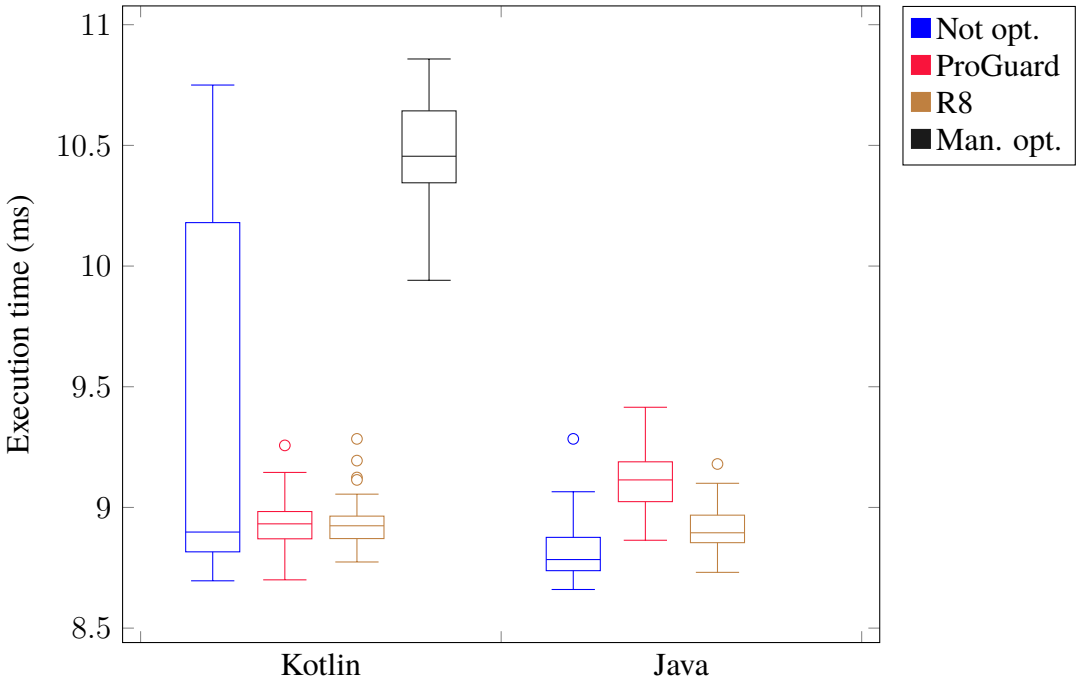

Figure 5.30: Results for the non-capturing lambda microbenchmark.

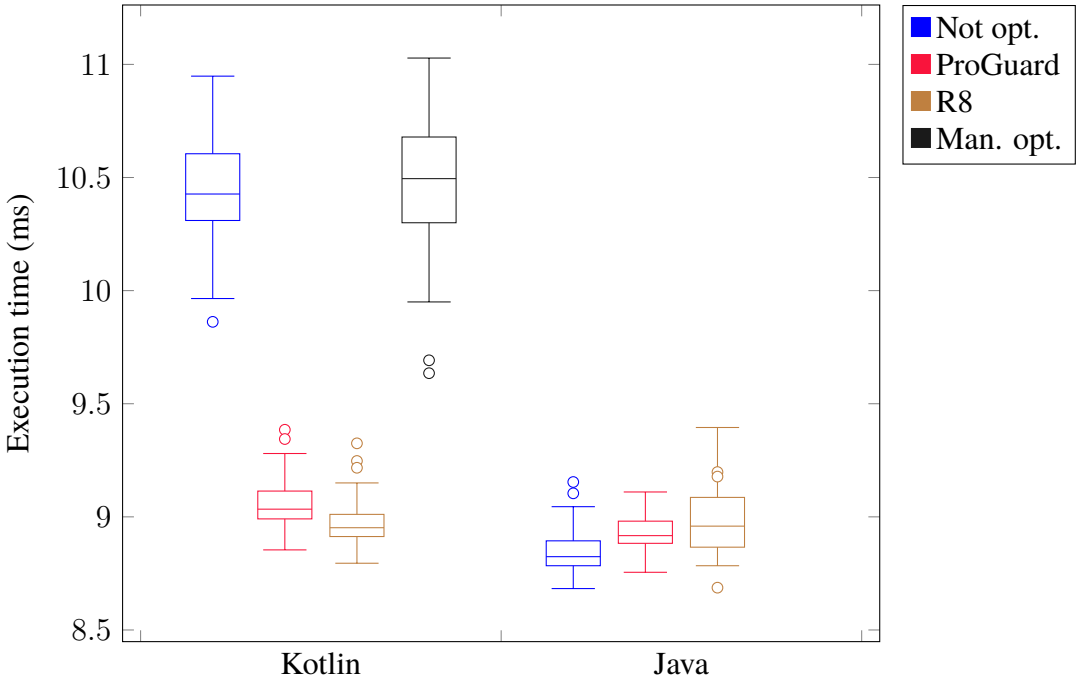

Figure 5.31: Results for the capturing lambda microbenchmark.

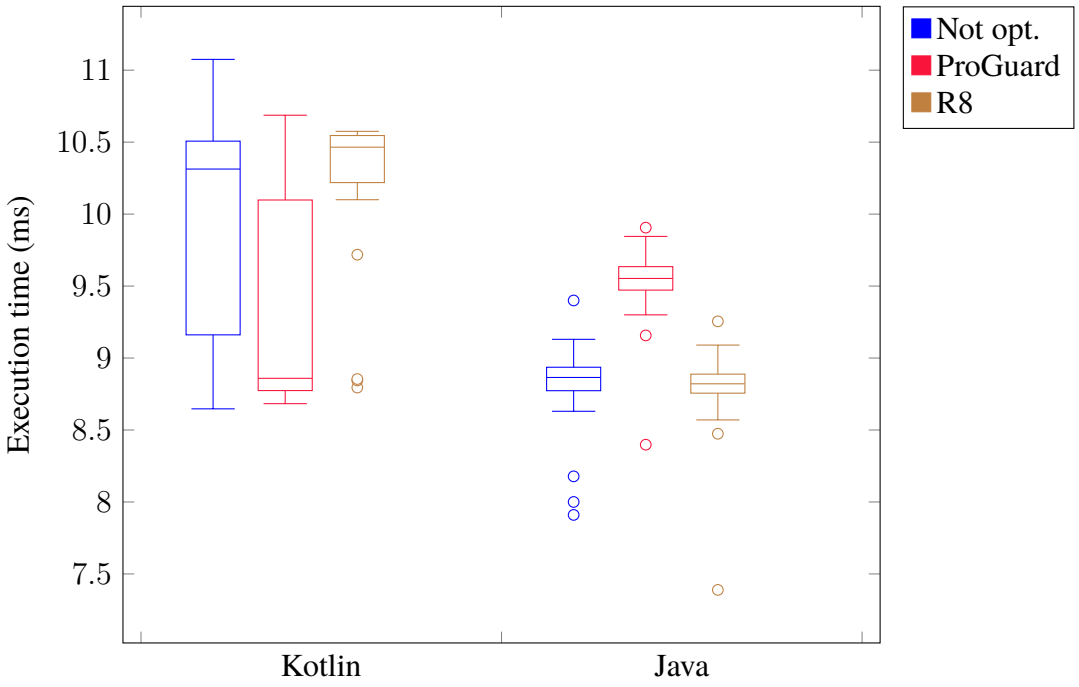

Figure 5.32: Results for the class variable access microbenchmark.

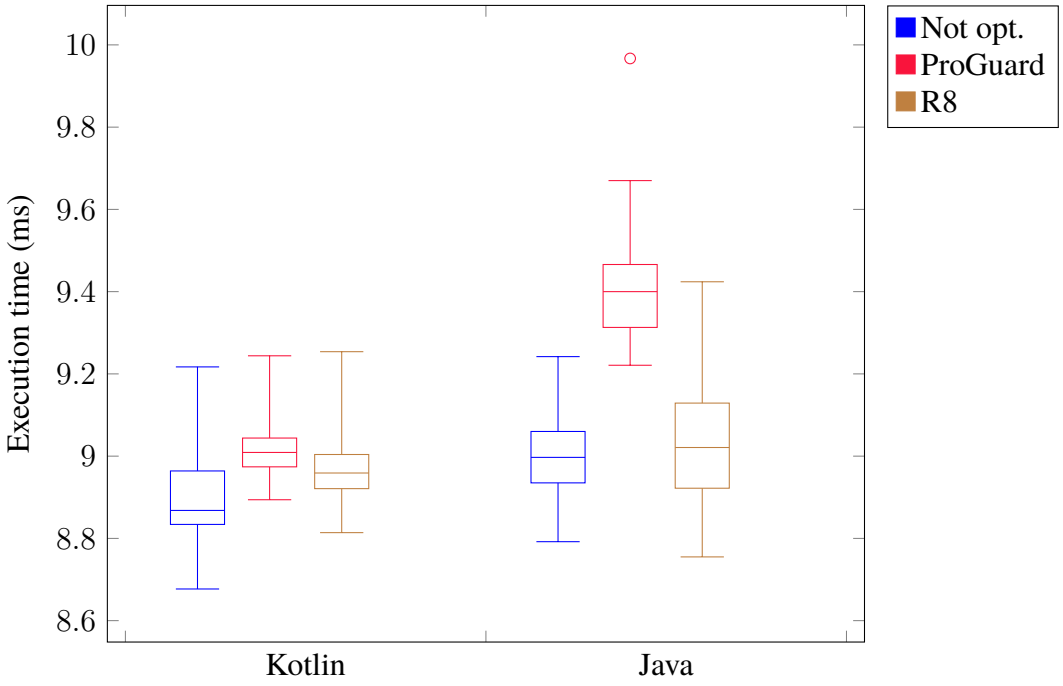

Figure 5.33: Results for the local function microbenchmark.

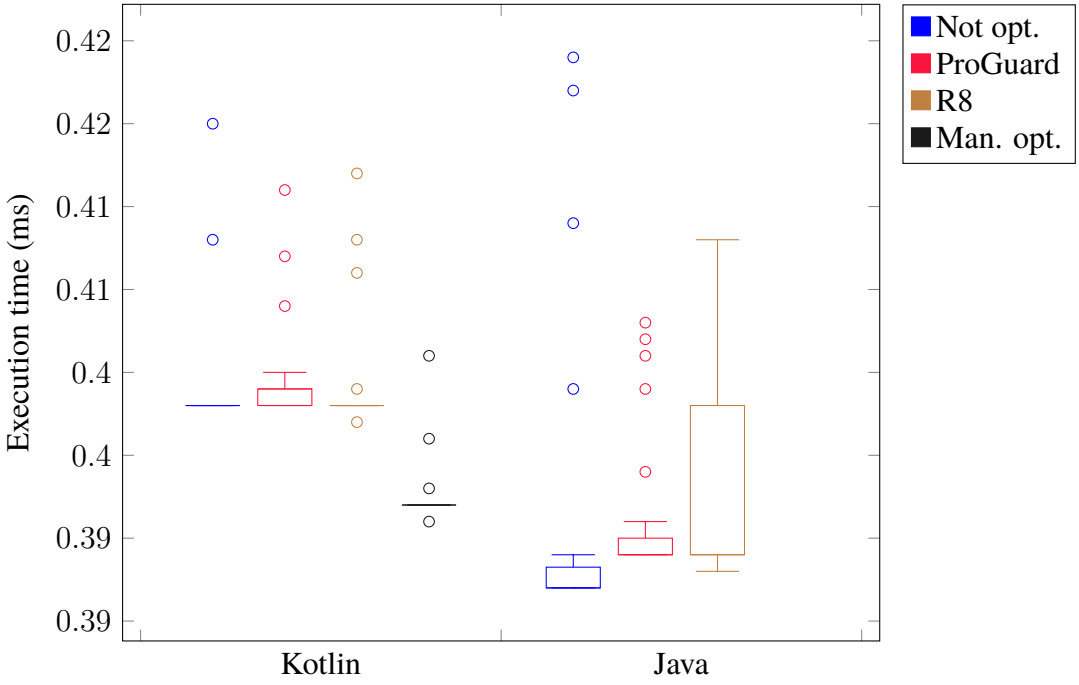

Figure 5.34: Results for the view binding microbenchmark.

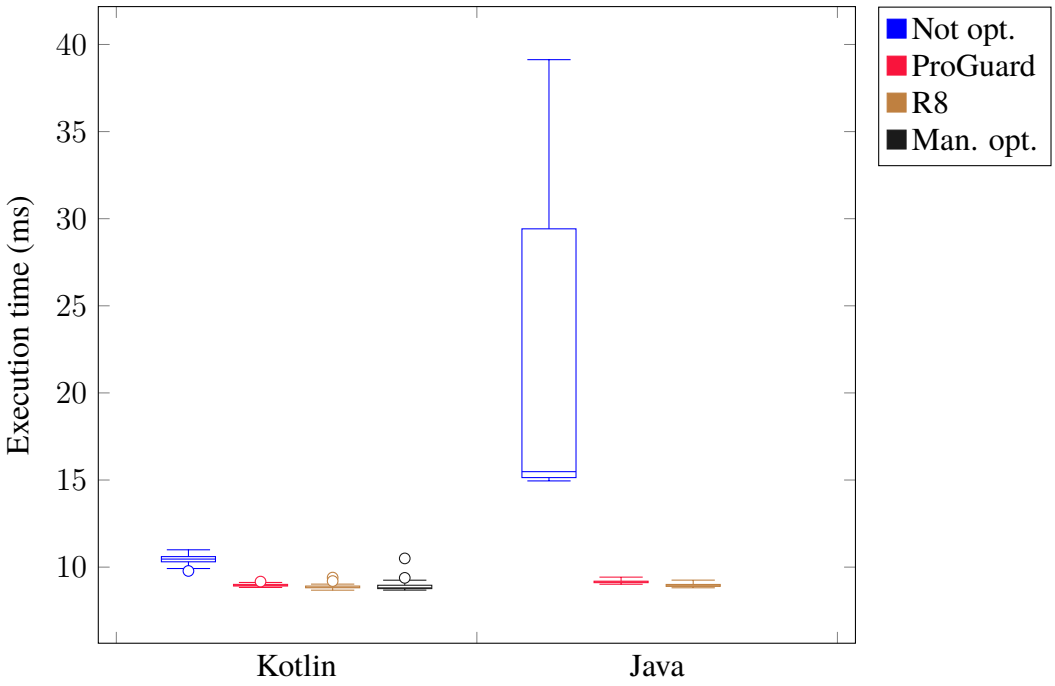

Figure 5.35: Results for the extension function microbenchmark.

Feature	Lang.	Not opt.	Opt. with ProGuard	Opt. with R8	Man. opt.	Difference, not opt.	Difference, ProGuard
Non-capturing lambda	Java	8.8	9.1	8.9	N/A	—	—
	Kotlin	9.4	8.9	8.9	10.5	6 %	1.9 %
Capturing lambda	Java	8.9	8.9	9.0	N/A	—	—
	Kotlin	10.5	9.0	9.0	10.5	18.1 %	1.4 %
Class variable access	Java	8.8	9.5	8.8	N/A	—	—
	Kotlin	10.0	9.3	10.2	N/A	13.2 %	2.5 %
Local function	Java	9.0	9.4	9.0	N/A	—	—
	Kotlin	8.9	9.0	9.0	N/A	−1.1 %	−4.2 %
View binding	Java	0.39	0.39	0.39	N/A	—	—
	Kotlin	0.4	0.4	0.4	0.39	1.7 %	2.2 %
Extension function	Java	19.7	9.0	10.4	N/A	—	—
	Kotlin	10.5	9.0	9.4	10.1	−53.3 %	0 %

Table 5.2: Average execution times in **milliseconds** for the microbenchmarks.

5.2.1 Discussion of the Microbenchmark Results

Overall, the results of the microbenchmarks, that were optimized with bytecode optimization tools, showed no large differences in execution speed between the Java and Kotlin implementations. Therefore, a developer should not avoid using these features for performance reasons, if they were to be used in a similar fashion. However, since the features for the microbenchmarks were implemented in one, specific way, it is difficult to say how the features would perform in broader contexts.

The largest difference, the 53.3 % difference of the non-optimized extension function microbenchmarks, could probably be explained by realizing that, as part of the microbenchmark, the Java case had to instantiate an extra object — the decorator instance.

The results of the view binding microbenchmarks, which showed that the manual optimization approach led to better performance than the bytecode optimization tool approaches, is questionable. It is worth noticing that the manual optimization consisted of directly calling the view binding API instead of dealing with the view cache. Given that the microbenchmarks ran in isolation, every time with an empty view cache, these results can be considered misleading in a real-world scenario as the benefits of a cache will start to become more palpable once the cache is populated.

Regarding the reliability of the microbenchmark, hypotheses can be made about what was actually microbenchmarked. Given that each microbenchmark had a warm-up phase and that the results of the warm-up were discarded, it is worth to realize that the results of the microbenchmarks probably only were the results of executing JIT-compiled native code. The results would probably have been different if the execution instead solely consisted of interpreting the bytecode.

Chapter 6

Threats to Validity

This chapter covers the possible sources of error that could have affected the validity of this study's answer to the research question.

6.1 Selection of Features

The methodology of the selection of features that were assessed in this study could have been improved. When it came to deciding which features to assess based on popularity, their assumed popularity was based on intuition and not based on statistics from modern projects.

6.2 Microbenchmarks

Due to the sophistication of ART and both Java's and Kotlin's compilers, the results of the microbenchmarks of this study should not be accepted as reliable numbers. The problem of divorcing the execution of Java or Kotlin code from JIT-compiling, garbage collection, and the effects of other running systems that affect performance, requires a wide and deep understanding of many complex areas. Even though precautions were taken in this study, there are probably still elements that affect the microbenchmark results and that were not taken into consideration when the microbenchmarks were created. However, the microbenchmarks that were run in this study are not completely useless, as they can still indicate that it is worth to investigate the efficiency of a given feature implementation.

Also, because only one device was used for the microbenchmarks, it limits the number of insights that could be obtained from the microbenchmarks. Given

that the device ran Android 9, it means that only the Android 9 implementation of ART was evaluated — and it is probably not the case that a majority of Android devices that are being used today run Android 9. Therefore, it could be argued that the perspective for the overall runtime performance of Kotlin on ART, which was obtained from the microbenchmarks, is **narrow** and does not reflect the performance for Kotlin on ART as a whole. However, since a majority of the Android devices, as of the time this study was written, run an Android version equal to or newer than version 7, most devices' runtimes do at least have a JIT compiler.

Lastly, regarding the microbenchmarks, it can be the case that hardware-related factors also could have affected the results. Factors like CPU clock frequencies, the number of CPU cores, CPU caches, CPU architectures, scheduling algorithms, RAM size, and translation lookaside buffers could have affected the results depending on if some of the features were more suitable to run on a specific hardware configuration. For instance, if one was to benchmark concurrency, the results would probably have varied significantly if the benchmark was run on a single core CPU and then on a multi-core CPU.

6.3 Need for Manual Optimization

Even if the manual optimization did not result in better performance, for most cases, than in the case when the compiled code had been optimized by bytecode optimization tools, it might not mean that manual optimizations are futile. With the methodology of this study, it was not verified if the bytecode optimization tools did the same optimizations that could be done manually. The hypothesis, that the combination; of manually optimizing the source code and processing the compiled code with bytecode optimizations tools; could lead to further improved performance, was not evaluated in this study.

Chapter 7

Conclusion

The results of the qualitative bytecode analysis showed that the bytecode produced by Kotlin can instantiate up to two extra objects and generate up to two extra methods compared to Java for a set of six feature implementations. It was also observed that Kotlin performs systematic boxing and unboxing for short-lived objects if primitive values are passed as arguments to, or returned from, a lambda expression for an example involving a lambda passed to a higher-order function. Lastly, it was observed that an extension function in Kotlin instantiated one less object than Java if the Java approach for an extension function was to utilize the decorator pattern.

Microbenchmarks were run for the set of six features, which compared the execution speed of idiomatic Java and Kotlin examples that utilized a feature in a minimal and isolated manner. For each feature, in both languages, the microbenchmarks were run three to four times. First without any optimizations, then after having been optimized by the bytecode optimization tool ProGuard, a third time after having been optimized by the bytecode optimization tool R8, and lastly, if applicable, after only having been manually optimized. The microbenchmarks showed a variety of time differences between the performance of the Java and Kotlin implementations. The differences ranged from Kotlin extension functions being 53.3 % faster than the Java implementation, to capturing Kotlin lambdas being 18.15 % slower than Java's.

Further, from the results of the microbenchmarks with optimizations, it could be observed that, in most cases, the manual optimization did not result in better performance than in the case when the code had been optimized by bytecode optimization tools. Overall, the ProGuard-optimized microbenchmarks

showed no differences over $5\ \%$ in execution times between the Java and Kotlin implementations.

Chapter 8

Future Study

This chapter covers what can be done to complement or extend the work of this study.

8.1 Better Selection of Features to Assess

It could be interesting to perform the same analysis as this study did for another set of features that get picked based on statistics of their usages in modern projects. A web crawler can be implemented to look for — and list — usages of Kotlin features in Kotlin projects that are hosted on software project hosting sites such as GitHub. Then, if the popularity is known, it would be possible to rank the impact of a hidden cost or opportunity. For instance, a cost that usually occurs at multiple places in a project could be considered more costly than a cost that only occurs once.

8.2 Different Implementations of Android Runtime

It could be interesting to perform the same microbenchmarks that were done in this study but on more Android devices with different Android Runtime implementations. It would give a more complete picture of the Kotlin features performance on Android Runtime overall.

8.3 Asynchronous Programming

Due to the popularity of Kotlin coroutines, it could be interesting to evaluate the performance of them in a quantitative manner to get insights about in what situations the cost of handling existing threads in a runtime environment surpasses the cost of allocating resources for new threads that are only to be handled by the operating system.

Bibliography

[1] Andrey Breslav. *Kotlin 1.0 Released*. [Online; accessed 2019-12-04]. JetBrains. URL: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/.

[2] Chet Haase. *Google I/O 2019*. [Online; accessed 2019-11-08]. URL: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html.

[3] "Mobile Web Stress - The Impact of Network Speed on Emotional Engagement and Brand Perception". In: Radware, 2013, p. 4.

[4] Patrik Schwermer. *Performance Evaluation of Kotlin and Java on Android Runtime*. [Online; accessed 2019-11-18]. 2018. URL: http://www.nada.kth.se/~ann/exjobb/patrik_schwermer.pdf.

[5] Google. *Android docs: Platform Architecture*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/guide/platform.

[6] Google. *Android docs: Overview of memory management*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/topic/performance/memory-overview.

[7] Google. *Android docs: Performance management*. [Online; accessed 2019-11-18]. URL: https://source.android.com/devices/tech/power/performance.

[8] Tanuj Mittal, Lokesh Singhal, and Divyashikha Sethia. "Computer Networks Communications (NetCom)". In: Springer, 2013. Chap. Optimized CPU Frequency Scaling on Android Devices Based on Foreground Running Application, p. 827.

[9] Dawei Li and Jie Wu. “Energy-aware Schedulingon Multiprocessor Platforms”. In: ISBN: 978-1-4614-5223-2. Springer, 2013, pp. 1–2.

[10] *Android Version Distribution*. [Online; accessed 2019-01-15]. Google, Android Docs. URL: https://developer.android.com/about/dashboards.

[11] Jetbrains. “Kotlin docs”. In: (). [Online; accessed 2019-11-08], p. 345. URL: https://kotlinlang.org/docs/kotlin-docs.pdf.

[12] *Coroutine Context and Dispatchers*. [Online; accessed 2019-01-11]. JetBrains. URL: https://kotlinlang.org/docs/reference/coroutines/coroutine-context-and-dispatchers.html.

[13] *Kotlin Census Report 2018*. [Online; accessed 2019-01-11]. JetBrains. URL: https://www.jetbrains.com/research/kotlin-census-2018/.

[14] JetBrains. *Objects and companion objects*. [Online; accessed 2019-11-14]. URL: https://kotlinlang.org/docs/tutorials/kotlin-for-py/objects-and-companion-objects.html.

[15] *Function types*. [Online; accessed 2019-12-08]. Jetbrains. URL: https://kotlinlang.org/docs/reference/lambdas.html#function-types.

[16] *spec-docs: function-types*. [Online; accessed 2019-01-09]. JetBrains. URL: https://github.com/JetBrains/kotlin/blob/master/spec-docs/function-types.md.

[17] *Null Safety*. [Online; accessed 2019-12-08]. Jetbrains. URL: https://kotlinlang.org/docs/reference/null-safety.html.

[18] *Extensions*. [Online; accessed 2019-01-11]. JetBrains. URL: https://kotlinlang.org/docs/reference/extensions.html.

[19] JetBrains. *Kotlin Android Extensions*. [Online; accessed 2019-12-01]. URL: https://kotlinlang.org/docs/tutorials/android-plugin.html.

[20] Torben Ægidius Mogensen. “Introduction to Compiler Design, Second Edition”. In: Springer, 2017. Chap. Preface, pp. vii–viii.

[21] Dong-Heon Jung, Soo-Mook Moon, and Sung-Hwan Bae. “Evaluation of a Java Ahead-of-TimeCompiler for Embedded Systems”. In: (2011), p. 1.

[22] Dick Grune et al. “Modern Compiler Design, Second Edition”. In: Springer, 2012, pp. 450–451.

[23] Google. *Android docs*. [Online; accessed 2019-11-08]. URL: https://source.android.com/devices/tech/dalvik/jit-compiler.

[24] Brian Goetz. *Dynamic compilation and performance measurement*. [Online; accessed 2019-11-08]. URL: https://www.ibm.com/developerworks/java/library/j-jtp12214/.

[25] Hugh Glaser, Pieter Hartel, and Herbert Kuchen. “Programming Languages: Implementations, Logics, and Programs”. In: Springer, 1997. Chap. Inline expansion: when and how?, pp. 143–144.

[26] James E. Smith and Ravi Nair. “Virtual Machines - Versatile Platforms for Systems and Processes”. In: ScienceDirect, 2007. Chap. 3, pp. 83–84.

[27] Tim Lindholm, A. Buckley, and G. Bracha. “The Java Virtual Machine Specification, Java SE 8 Edition”. In: Addison-Wesley Professional, 2014. Chap. 2, 3.

[28] Oracle. *Java Platform, Standard Edition Java Virtual Machine Guide*. [Online; accessed 2019-11-08]. URL: https://docs.oracle.com/javase/10/vm/java-hotspot-virtual-machine-performance-enhancements.htm.

[29] Oracle. *Chapter 4. The class file format.* [Online; accessed 2019-12-01]. Oracle. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html.

[30] Fransico Ortin et al. *The Runtime Performance of invokedynamic:An Evaluation with a Java Library*. [Online; accessed 2019-12-08]. IEEE COMPUTER SOCIETY, pp. 81–82.

[31] Google. *Android docs: Implementing ART Just-In-Time (JIT) Compiler*. [Online; accessed 2019-11-08]. URL: https://source.android.com/devices/tech/dalvik/jit-compiler.

[32] Dan Bornstein. *Dalvik VM Internals*. [Online; accessed 2019-11-08]. Google I/O slides, 2008. URL: http://sites.google.com/site/io/dalvik-vm-internals.

[33] P Sabanal. *Hiding Behind ART*. [Online; accessed 2019-11-08]. IBM X-Force Advanced Research, 2014. URL: https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf.

[34] Ryan elfmaster O'Neill. *Learning Linux Binary Analysis*. ISBN: 9781782167105. Packt Publishing, 2016. Chap. 2.

[35] Google. *Android docs: d8*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/studio/command-line/d8.

[36] Google. *Android docs: Configure your build*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/studio/build.

[37] *Shrink, obfuscate, and optimize your app*. [Online; accessed 2019-01-11]. Google, Android Docs. URL: https://developer.android.com/studio/build/shrink-code.

[38] Guardspace. *Optimizations*. [Online; accessed 2019-11-08]. URL: https://www.guardsquare.com/en/products/proguard/manual/usage/optimizations.

[39] Eric Lafortune. "Comparison of ProGuard vs. R8: October 2019 edition". In: (). [Online; accessed 2019-11-08]. URL: https://www.guardsquare.com/en/blog/comparison-proguard-vs-r8-october-2019-edition.

[40] Google. *Android docs: Multidex*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/studio/build/multidex.

[41] M Ma. *Android's multidex slows down app startup*. [Online; accessed 2019-11-08]. URL: https://medium.com/groupon-eng/android-s-multidex-slows-down-app-startup-d9f10b46770f.

[42] Oracle. *Java Language Specification SE8*. [Online; accessed 2019-11-15]. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.7.

[43] Google. *Android docs: Benchmark*. [Online; accessed 2019-11-08]. URL: https://developer.android.com/jetpack/androidx/releases/benchmark.

[44] Christophe Beyls. *Kotlin's hidden costs part 1*. [Online; accessed 2019-11-08]. 2017. URL: https://medium.com/@BladeCoder/exploring-kotlins-hidden-costs-part-1-fbb9935d9b62.

[45] Christophe Beyls. *Kotlin's hidden costs part 2*. [Online; accessed 2019-11-08]. 2017. URL: https://medium.com/@BladeCoder/exploring-kotlins-hidden-costs-part-2-324a4a50b70.

[46] David Gregg, James Power, and John Waldron. "Platform independent dynamicJava virtual machine analysis:the Java Grande Forumbenchmark suite". In: (2003), p. 472.

[47] Sergey Kusenko. *JDK 8: Lamda Performance study*. [Online; accessed 2019-12-08]. Oracle. URL: https://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf.

[48] Christophe Beyls. *Kotlin's hidden costs part 3*. [Online; accessed 2019-11-08]. 2017. URL: https://medium.com/@BladeCoder/exploring-kotlins-hidden-costs-part-3-3bf6e0dbf0a4.

[49] Svetlana Isakova. *Kotlin 1.3.50 released*. [Online; accessed 2019-01-08]. JetBrains. URL: https://blog.jetbrains.com/kotlin/2019/08/kotlin-1-3-50-released/.

[50] Oracle. *JDK 8u211 Update Release Notes*. [Online; accessed 2019-01-08]. Oracle. URL: https://www.oracle.com/technetwork/java/javase/8u211-relnotes-5290139.html.

[51] *FAQ - Kotlin Programming Language*. [Online; accessed 2019-01-08]. JetBrains. URL: https://kotlinlang.org/docs/reference/faq.html.

[52] Brian Goetz. *Anatomy of a flawed microbenchmark*. [Online; accessed 2019-11-08]. IBM. URL: https://www.ibm.com/developerworks/java/library/j-jtp02225/.

[53] Wikipedia. *Java bytecode instruction listings*. [Online; accessed 2019-12-01]. URL: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings.

[54] Oracle. *Chapter 6: The Java Virtual Machine Instruction Set*. [Online; accessed 2019-12-01]. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html.

Appendix A

Java Bytecode Information Resources

The following table presents common Java bytecode instructions.

Instruction	**Stack [before] → [after]**	**Description**
iload #index	→ value	load an int value from a local variable #index
istore #index	value →	store int value into variable #index
aload #index	→ objectref	load a reference onto the stack from a local variable #index
astore #index	→ objectref	store a reference into a local variable #index
iadd	value1, value2 → result	add two ints
ldc #index	→ value	push a constant index from a constant pool (String, int, float, Class, java.lang.invoke.MethodType, java.lang.invoke.MethodHandle, or a dynamically-computed constant) onto the stack
getfield #index	objectref → value	get a field value of an object objectref, where the field is identified by field reference in the constant pool index
getstatic #index	→ value	get a static field value of a class, where the field is identified by field reference in the constant pool index
invokespecial	objectref, [arg1, arg2, ...] → result	invoke instance method on object objectref and put the result on the stack (might be void)
invokevirtual	objectref, [arg1, arg2, ...] → result	invoke virtual method on object objectref and put the result on the stack (might be void)
invokestatic	[arg1, arg2, ...] → result	invoke a static method and put the result on the stack (might be void)
invokedynamic	[arg1, [arg2 ...]] → result	invoke a dynamic method and put the result on the stack (might be void); the method is identified by method reference index in constant pool
return	→ [empty]	return void from method

Table A.1: Common Java bytecode instructions with summarized descriptions from Wikipedia [53]. The full JVM instruction set for Java 8 can be found in Oracle's Java documentation [54]

Appendix B

Source code, class files, t-test, and microbenchmark results

The source code for the bytecode analysis and the microbenchmarks, the class files for the bytecode analysis, the microbenchmark result, and the microbenchmark difference t-test are hosted on GitHub, which can be accessed through the following link:

https://github.com/jqhan/msc-thesis

The repository contains a README file that explains where the relevant material is located.

TRITA-EECS-EX-2020:66