CONSTRUCTOR
UNIVERSITY

Studying compilation schemes for shared memory accesses in Kotlin/Native

by

Gleb Solovev

Bachelor Thesis in Computer Science

Submission: May 16, 2023 Supervisor: Adjunct Prof. A. Podkopaev

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Solovev, Gleb
Matriculation number	30006599
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

16.05.2023 [illegible]
....
Date, Signature

Abstract

This study investigates the necessity and effectiveness of replacing *NotAtomic* memory accesses with *Unordered* in Kotlin/Native, a contemporary Kotlin programming language technology that enables cross-compilation with native languages. The current compiler implementation uses *NotAtomic* mode, which is incorrect according to LLVM documentation, and may lead to vulnerabilities in multi-threaded programs.

Through rigorous benchmarking, we demonstrate that the *Unordered* mode implementation in LLVM is not currently optimized enough to facilitate a replacement of the *NotAtomic* mode. Although no vulnerabilities were observed during our experiments, it is still important to report the discovered issue to developers, as future versions of LLVM may cause unexpected program errors.

The findings of this research offer valuable insights for the development of future versions of Kotlin/Native and other implementations of programming languages that rely on LLVM for compilation.

Contents

1 Introduction

A programming language memory model is a formal specification that describes which outcomes are possible in multi-threaded programs. Moreover, it should not be too strict to leave room for different optimizations to be applied by the compiler. Therefore, the memory model defines a balance between the performance of programs and the strictness of guarantees provided to the developer.

One significant aspect of the memory model's implementation is the compilation scheme for shared memory accesses. This compilation scheme defines how the source code for reading and writing shared variables is compiled into assembly instructions for the processor. Our work is focused on examining several possible compilation schemes for the Kotlin programming language.

Kotlin is a modern programming language that is widely used in software development. Like its predecessor, Java, Kotlin positions itself as a safe language, where undefined behavior found in C-like languages is strictly unacceptable.

One of the key technology of Kotlin is the Kotlin Multiplatform project [14], which enables developers to compile Kotlin code for various target platforms, including Android, iOS, Web, Desktop, and Server-side. Kotlin Multiplatform consists of three versions of Kotlin: Kotlin/JVM, Kotlin/JS, and Kotlin/Native.

The Kotlin/JVM version compiles Kotlin code to execute on the Java Virtual Machine. This allows Kotlin to be interoperable with Java, meaning that Java code can be called from Kotlin and vice versa. Furthermore, Kotlin/Native [15] is another version of Kotlin that allows Kotlin code to be compiled using the LLVM framework [34] into a binary file that can be directly executed on a computer. This provides the ability to interop with other languages such as C and Swift, opening up more opportunities for cross-platform development.

The compilation processes of Kotlin/JVM and Kotlin/Native are fundamentally similar, as illustrated in Figure 1. First, the Kotlin compiler translates the code into either JVM bytecode or LLVM bitcode, respectively, which is then subject to a series of checks and optimizations by the corresponding backend. Finally, the backend compiles the code into executable files.

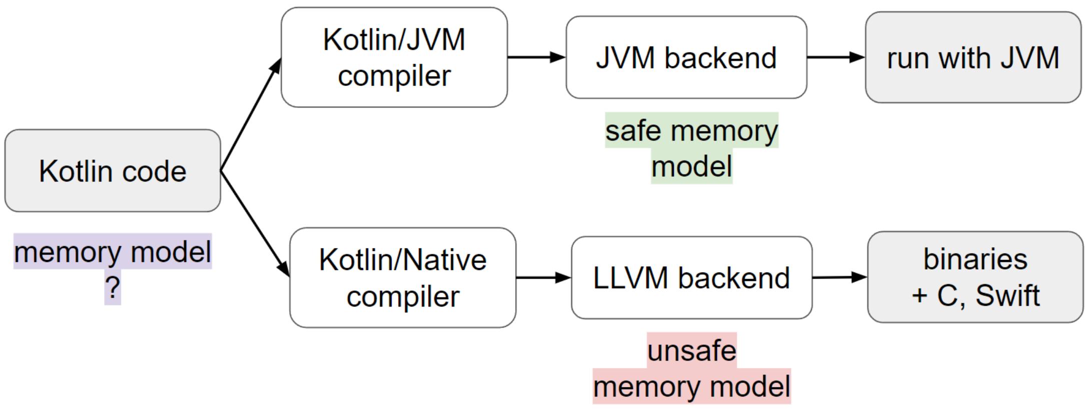
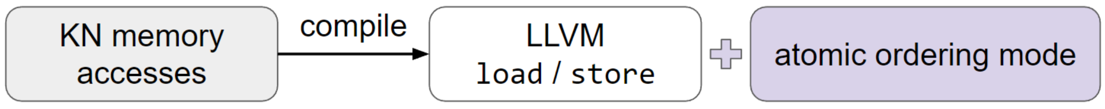


Figure 1: Kotlin Multiplatform compilation scheme

Similar to programming languages, both JVM and LLVM possess memory models that determine the possible outcomes and permitted optimizations of the programs. The key difference between the two is safety, with JVM guaranteeing no undefined behavior in any situation, while LLVM does not. Notably, undefined behavior is unacceptable in Kotlin, which emphasizes safety.

Thus, there arises a challenge: despite the stark differences in memory models between Kotlin/JVM and Kotlin/Native backends, a safe and uniform memory model is still needed for Kotlin. This is a complex problem, and the development of a suitable memory model for Kotlin/Native is currently ongoing [7, 8, 28]. Our research aims to contribute to this process.

The compilation of memory accesses in Kotlin/Native involves the translation of these accesses into either `load` or `store` instructions in LLVM bitcode, with the atomic ordering mode [21] specified to determine the ordering guarantees of the instruction. For instance, as shown in Figure 2, a simple Kotlin read from variable `x` on the left may be compiled into a `load` instruction with the `unordered` mode on the right.



```
val a = x
```

```
%31 = load atomic i32, i32* %x
      unordered, align 4
                    ...
```

Figure 2: Kotlin to LLVM bitcode translation example.

Of the various atomic orderings available, our research focuses on two: *NotAtomic* and *Unordered*. The *NotAtomic* mode is the weakest, allowing for maximum optimizations but also permitting undefined behavior in the case of a data race. This results in unsafe semantics. The stricter *Unordered* mode, on the other hand, restricts the number of optimizations but provides safe semantics. According to the LLVM documentation [21], this mode should be used in safe languages such as Java and, by extension, Kotlin. However, the current Kotlin/Native compiler uses the unsafe *NotAtomic* mode instead. Furthermore, *Unordered* mode is not present in any other known implementations of safe languages that uses LLVM, such as GraalVM for Java [11], GHC for Haskell [33] and Swift implementation for Swift [32].

Therefore, our project aims to investigate and compare the *Unordered* and *NotAtomic* compilation schemes for shared memory accesses in Kotlin/Native.

2 Statement and Motivation of Research

This section will outline the motivation for our study, as well as its purpose and objectives. Additionally, we will provide an overview of the sources and research required to comprehend our work.

2.1 Goal and Tasks

Any compiler must fulfill two fundamental requirements. First, it must be correct: the stated guarantees must hold for compiled programs. Second, it must be efficient: program performance is a crucial characteristic in numerous areas of development. Thus, the motivation for our research is to examine how the Kotlin/Native compiler meets both these criteria.

The aim of our work is to investigate the correctness and performance of a compilation scheme that uses *Unordered* instead of *NotAtomic* in Kotlin/Native.

To accomplish this goal, several tasks must be carried out. Firstly, it is essential to assess the correctness of the current implementation of the Kotlin/Native compiler in practice. This involves verifying whether the theoretically described vulnerability is reproduced in reality and whether it is necessary to replace *NotAtomic* accesses at all.

Next, the approach must be implemented in the Kotlin/Native compiler and tested in practice. This requires studying the large compiler code base and implementing a new compiler pass similar to the original ones. The correctness of the implementation must also be tested. Besides, it is important to examine the approach on different architectures, since different hardware memory models lead to different effects on the behavior of programs. To accomplish this, tools for executing the modified compiler and analyzing its work in various environments, such as Linux x64 and MacOS Arm64, must be established.

Finally, the impact of the approach on the performance of the compiled programs should be measured through benchmarking its implementation. To do this, a suitable test corpus should be gathered and adapted, and it should be executed in different environments. The results obtained should then be analyzed to determine the effectiveness of the approach.

2.2 Related Work

In this subsection, we will conduct a review of background and related works. Firstly, we will discuss the general design of the LLVM Project and its key features that are relevant to our study. Secondly, we will delve into the concept of atomic orderings in LLVM, their various types, and their applications. Finally, we will conduct a comprehensive analysis of the literature in our field of study, specifically, compilation schemes for programming languages.

2.2.1 The LLVM Project

We will commence with an overview of the LLVM project, which ”is a collection of modular and reusable compiler and toolchain technologies” [34]. The LLVM project enables the development of compilers for diverse programming languages, such as Kotlin/Native, Java, C/C++, Rust, Objective-C, Swift, Haskell, etc. Additionally, it makes it possible to compile, debug, and analyze C/C++/Objective-C programs while providing a broad range of libraries for interpretation, optimization, and linking code.

For our research, two components of LLVM are particularly relevant. Firstly, the LLVM Core libraries allow for code generation, compilation, and optimization of LLVM intermediate representation (LLVM IR), thereby serving as a backend for implementing other languages, particularly Kotlin/Native. Secondly, the Clang compiler is a widely used and efficient front-end compiler for C/C++/Objective-C to LLVM IR, backed by the same LLVM Core libraries. In other words, Clang enables C-like languages to be compiled with the same optimization pipeline as Kotlin/Native due to the same backend. Hence, we can experiment using the C language and the Clang compiler to test the behavior of the LLVM backend and, consequently, the optimizations applied in Kotlin/Native.

2.2.2 LLVM Atomic Orderings and Memory Model

Moving forward, let us explore the topic of atomic orderings in LLVM. Within the LLVM, both the `load` and `store` instructions are marked with an atomic ordering parameter. This parameter defines the instructions' semantics and provides guarantees regarding the memory model. From a practical standpoint, atomic orderings impact the way in which instructions are compiled. In exchange for stronger guarantees, they may introduce additional synchronization instructions, such as memory barriers, and limit applicable optimizations. Figure 3 illustrates examples of instructions annotated with various atomic orderings in LLVM IR [23].

```
; Examples of load/store-s annotated with appropriate orderings

; Load %ptr from memory to %val
%val = load i32, ptr %ptr                         ; NotAtomic
%val = load atomic i32, ptr %ptr unordered        ; Unordered
%val = load atomic i32, ptr %ptr monotonic        ; Monotonic
%val = load atomic i32, ptr %ptr acquire          ; Acquire
%val = load atomic i32, ptr %ptr seq_qst          ; SequentiallyConsistent

; Store 3 to %ptr
store i32 3, ptr %ptr                         ; NotAtomic
store atomic i32 3, ptr %ptr unordered        ; Unordered
store atomic i32 3, ptr %ptr monotonic        ; Monotonic
store atomic i32 3, ptr %ptr release          ; Release
store atomic i32 3, ptr %ptr seq_qst          ; SequentiallyConsistent
```

Figure 3: Example of atomic ordering annotations in LLVM IR

In total, there are seven types of atomic orderings, arranged in ascending order of guarantee strength (excluding the relationship between *Acquire* and *Release*): *NotAtomic*, *Unordered*, *Monotonic*, *Acquire*, *Release*, *AcquireRelease* and *SequentiallyConsistent*. *NotAtomic*, the weakest of them, does not formally represent an atomicity level since it denotes its absence. However, for the sake of convenience and consistency with the LLVM specification [21], we will consider it as an atomic ordering. Next, we will briefly analyze each of the orderings, although the primary focus of our study is on *NotAtomic* and *Unordered*.

NotAtomic is the weakest mode, used for the most common `load` and `store` instructions which are not atomic in principle. They lack thread safety and place minimal restrictions on the optimizer, resulting in optimal performance. However, it is important to note that this mode assumes undefined behavior: ”If there is a race on a given memory location, loads from that location return `undef`” [21]. As a result, they are unsuitable for use in safe languages that do not permit undefined behavior. The LLVM specification suggests that for safe languages such as Java, the *Unordered* mode should be used for loading and storing any shared variables [21].

The *Unordered* mode is the weakest atomic mode that guarantees the absence of any undefined behavior compared to its predecessor, *NotAtomic*. According to the LLVM specification, ”a load cannot see a value which was never stored” [21]. However, the specification cautions that the use of the *Unordered* mode may result in performance overhead, particularly when working with 64-bit instructions on ARM. Despite this, *Unordered* is the bare minimum for safe languages that ”need to guarantee that the generated code never exhibits undefined behavior” [21]. Hence, we tend to consider the use of *NotAtomic* in implementations of Kotlin/Native, GraalVM for Java [11], GHC for Haskell [33], and Swift implementation for Swift [32] as incorrect.

Finally, let us conclude our discussion of atomic orderings by briefly describing the stronger modes. The *Monotonic* mode represents the weakest mode that can be used to implement synchronization primitives. It guarantees a consistent order of operations for a specific memory address, also known as the coherence property [1]. The *Acquire* and *Release* modes are paired modes that allow the acquisition and release of resources, respectively, in order to access other memory with normal loads and stores [21]. Typically, the *Acquire* semantics is used to obtain a resource, while the *Release* semantics is used to publish a resource. The *AcquireRelease* ordering provides both the acquire and release semantics, which is typically used for operations that both read and write to memory. Finally, the *SequentiallyConsistent* ordering provides the strongest guarantees for shared memory access instructions. As stated in [21], it ”guarantees that a total ordering exists between all *SequentiallyConsistent* operations,” thus ensuring sequential consistency.

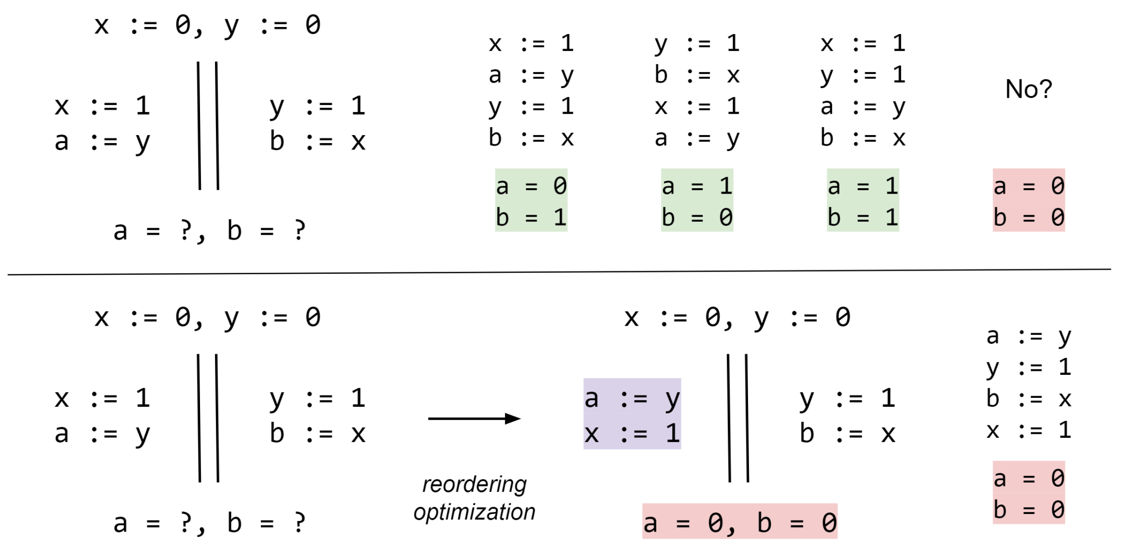


Figure 4: Store Buffering litmus test

Demonstration of how atomic ordering guarantees work can be conveniently illustrated using litmus tests, which are small programs that exhibit different outcomes depending on the memory model used. One of the classic litmus tests in this area is the Store Buffering test [25]. In this example, we analyze the effect of the *SequentiallyConsistent* ordering guarantees compared to *Unordered*.

Consider the program in Figure 4. The program initializes two memory cells, `x` and `y`, with zeros. Then, the thread on the left writes `1` to the cell `x` and later reads `y` into the local variable `a`. In parallel, the thread on the right stores `1` to the cell `y` and reads `x` into the local variable `b`. Running such a program can result in three different outcomes for the variables `a` and `b`: `0` and `1`, `1` and `0`, and `1` and `1`, respectively, depending on the interleaving of the threads. In fact, there is no interleaving of threads that can result in two zeros. However, when using *Unordered* instructions, surprisingly, this outcome may be observed. The most straightforward occurrence that may happen is the optimization of instruction reordering, where with a specific variant of thread interleaving, the result of two zeros is achieved. In contrast, using *SequentiallyConsistent* ordering, instruction reordering will be prohibited, thereby making it impossible to achieve the weak outcome of two zeros.

2.2.3 Studies on Compilation Schemes for Programming Languages

A brief review of similar studies to our own research is conducted in this section. Although we were unable to locate any studies on the issue of replacing *NotAtomic* with *Unordered* in safe languages, research on compilation schemes for various languages is widely and actively studied. In this regard, we examine a few of these studies.

In the paper [27], the researchers propose a method of preserving load-store ordering to eliminate the Out-of-Thin-Air (OOTA) [3] problem in C/C++. They accomplish this by modifying the LLVM compiler and then measuring the performance of the implementation on single-threaded C/C++ SPEC CPU2006 benchmarks on ARMv8. It is worth noting that the authors intended to test the suggested compilation scheme on other architectures but could not do so due to the complexity of porting the implementation. Additionally, the authors chose to evaluate the performance of the benchmarks under a single-threaded scenario, as this represents a worst-case scenario due to the higher relative cost of additional instructions and the absence of thread contention that may affect the results of benchmarks in an uncontrollable manner.

In another work [19], the authors aimed to introduce sequential consistency in Java by default. To accomplish this goal, they made modifications to Oracle's HotSpot JVM for the x86 architecture and evaluated their implementation using Da Capo and Apache Spark benchmarks on an x86 machine. Similar to the study mentioned above, the authors intended to conduct tests on a different hardware architecture, specifically ARM, but postponed it to another article [20] due to the complexity of the implementation process. One noteworthy aspect of their approach is the use of litmus tests for testing a modified compiler's correctness. These small programs facilitate external validation of the memory model's properties, making them an ideal way to verify the correctness of the proposed compilation scheme's implementation. Lastly, the authors observed performance costs in single-threaded benchmarks. They suggest that this is because multi-threaded programs already have synchronization, which implies that adding new synchronization instructions by a modified compiler results in smaller performance difference.

Finally, the article [6] proposes and theoretically proves a novel approach of local data race freedom semantics, which allows for a clear definition of the behavior of many problematic programs. The authors implemented their proposed approach in the OCaml compiler and evaluated its effectiveness using OCaml benchmarks on ARM and POWER machines. In this work, the authors successfully conducted measurements on both x86 and ARM architectures, which is essential in the study of compilation schemes. Furthermore, the authors present theoretical examples of programs with unexpected outcomes resulting from current vulnerabilities in memory models, supporting their argument for the development of a new theorem and compilation scheme.

Let us summarize a brief analysis of the existing work in the field of compilation schemes. First of all, we note that none of the articles reviewed address the specific problem at the core of our research, namely, the compilation of *NotAtomic* in safe programming languages. As far as we are aware, this problem has not been previously investigated. While the aforementioned works offer compilation schemes for addressing other memory model issues in other programming languages, they are not directly applicable to our study.

However, the majority of the research in the field of compilation schemes follows a similar research plan, as demonstrated in the aforementioned studies. Specifically, researchers initially propose and describe compilation approaches and schemes, subsequently implementing them in the target language compiler, and finally evaluating the performance of the implementation on target benchmarks. As can be observed from the research tasks stated earlier, our study follows the same research plan.

Additionally, our analysis of the related literature enabled us to identify several widely accepted practices in the field, which we adopted in our study. For instance, when investigating compilation schemes, it is common practice among researchers to perform tests on multiple hardware architectures, as these can significantly impact the performance of the memory model implementation. Another prevalent practice involves presenting theoretical examples of programs that exhibit unexpected, potentially harmful behavior to justify the need for a new compilation scheme. Additionally, litmus tests are often employed when working with memory models due to their simplicity and demonstrative nature, including their usefulness in testing implementations. Single-threaded benchmarks are also frequently used to showcase the most significant performance changes, such as worst-case scenarios. We applied all of these prevalent practices in our study.

3 Assessing Kotlin/Native Compiler Correctness

This section aims to evaluate the correctness of the Kotlin/Native compiler. To achieve this goal, we will first investigate the possibility of encountering undefined semantics in the safe Kotlin/Native language when using *NotAtomic* instead of *Unordered*. Then, we will describe our methodology to reproduce these vulnerabilities and report the results. Finally, we will draw conclusions regarding the correctness of the current Kotlin/Native compiler based on our findings.

3.1 NotAtomic as a Source of Undefined Semantics in the Safe Language

Safe languages are designed to prevent undefined semantics, but our research demonstrates that this guarantee can theoretically be violated in the safe Kotlin/Native language. In this section, we will examine this issue through the example of accessing an array by index.

An array is a contiguous segment of allocated memory, and accessing an incorrect index beyond this memory allows access to unallocated memory. This unallocated memory may contain either garbage data or be protected by the operating system. In unsafe languages, such incorrect memory access results in undefined behavior.

In contrast, safe languages enforce bounds checking when accessing an array by index. If the bounds are violated, an error is returned to the user. In Kotlin, this error is represented by the `IndexOutOfBoundsException` exception. This ensures safe semantics that protects developers from invalid memory access.

However, our research shows that the safe semantics of Kotlin/Native can still be theoretically violated. Specifically, the use of *NotAtomic* instead of *Unordered* can lead to invalid memory access. Figure 5a presents an example in pseudocode to illustrate this issue.

```
int[] a = new int[10];
int i = 0;

            ||
a[i] = 42;  ||  i = 100;
            ||
```

(a) Source

```
                       %r1 = new int[10];
                       store a, %r1;
                       memset(a, 0, 10);

%r2 = load i;               ||
if (0 <= %r2 < 10) {        ||  store i, 100;
    %r3 = load i;           ||
    a[~~%r2~~ %r3] = 42;    ||
} else {                    ||
    throw();                ||
}                           ||
```

(b) Compiled

Figure 5: Accessing an array by index example in pseudocode

First, the program initializes the array `a` of size `10` and index `i` is set to `0`. Then one thread attempts to access the array at index `i`, while another thread writes an overly large value to index `i`. A data race occurs, and depending on the interleaving of the threads, the first thread may access either the `0`-th or the `100`-th array cell. However, despite the presence of a data race, the safe Kotlin language ensures the absence of undefined behavior. In the latter case, an exception should be thrown.

As demonstrated in the compiled version of the program, presented in Figure 5b, array access by index is safeguarded by a bounds check. Thus, if the program is called at the correct index `0`, it will execute without issue, while if called at index `100`, it will throw an `IndexOutOfBoundsException` error. This behavior is expected and guaranteed by the language.

However, when the *NotAtomic* ordering is used in a compiler, it can lead to undefined semantics. By default, all accesses to variables are annotated with the weakest, *NotAtomic* ordering, which allows for dangerous optimizations such as load rematerialization [21]. This optimization allows the compiler to split one memory read into two. As shown in the compiled code of the first thread in Figure 5b, the value of the variable `i` is loaded from memory into the `%r2` register, after which bounds are checked and the array is accessed. However, in the case of optimization performed, an additional read occurs, marked in red in the image as `%r3 = load i;`. Depending on the interleaving of the threads, the second `load` may return the value `100`, leading to actual invalid access to memory, which results in undefined behavior.

The aforementioned example is more likely theoretical, as the load rematerialization optimization is typically employed only in more complex programs when there is a scarcity of free registers. Nonetheless, the LLVM specification formally permits the compiler to generate such code.

The rationale behind this behavior can be explained by the usage of the `undef` value within the LLVM memory model. In the event of a data race, a *NotAtomic* `load` operation returns an `undef` value. By employing this value, the compiler is effectively allowed to use any values it desires, even for distinct occurrences of the same `undef` value. Therefore, even the example presented in the article [4], which is:

```
if(t <= 1 && t > 1) printf("Hi");
```

can produce the output string `"Hi"` if the variable `t` holds an `undef` value. Upon each access, the `undef` value can be substituted with any number, either less than or greater than one.

From the perspective of the LLVM memory model, in our example in Figure 5b, there is a data race on the variable `i`. As such, reading it returns an `undef` value. Subsequently, a check for the array bounds, `if (0 <= undef < 10)`, may succeed, despite the subsequent potentially erroneous access to the array using the `undef` index.

In order to prevent the undefined semantics of `undef` values, LLVM provides the *Unordered* ordering. According to the LLVM memory model, *Unordered* `load`-s always return specific values, thereby providing safe semantics. From a practical perspective, *Unordered* is restricted from some of the dangerous optimizations that are permitted for *NotAtomic*. Specifically, the specification disallows any transformation that converts one `load` or `store` into multiple operations, narrows a `store`, or stores an impossible value. As an example, the ”narrowing an assignment into a bitfield, rematerializing a load, and turning loads and stores into a memcpy call” [21] optimizations are not permitted.

Therefore, with respect to the LLVM memory model and theoretical examples, the use of *NotAtomic* in safe Kotlin/Native may lead to undefined semantics. Conversely, the *Unordered* mode, due to its prohibition of certain optimizations, provides a guarantee of safe semantics.

3.2 Reproducing Undefined Semantics of NotAtomic in Practice

In the previous subsection, we demonstrated that the use of *NotAtomic* orderings in Kotlin/Native can result in undefined semantics. However, it is crucial to determine the practical impact of this vulnerability to evaluate the potential danger it poses. To achieve this, we employed various methods.

As we have discussed earlier, the cause of potential undefined semantics in Kotlin/Native is the dangerous optimizations that can be performed with *NotAtomic* operations. Hence, we initially attempted to establish the real-world list of optimizations applicable in practice. To achieve this, we carefully analyzed the LLVM codebase. Despite conducting extensive searches in both the primary codebase [35] and tests [24], and reaching out to a developer for assistance[1], this approach proved to be unsuccessful. The LLVM codebase is vast, intricate, and inadequately documented, rendering it challenging to extract relevant

[1] The post by john-brawn-arm on LLVM Discussion Forums.

information. Thus, further research is required, and this approach remains a prospective subject for future investigation.

Another method we pursued was to seek the views of the LLVM developer community regarding the use of potentially hazardous *NotAtomic* orderings. However, multiple attempts[2,3] resulted in varying opinions. Some developers were relatively confident that the vulnerability was reproducible, while others argued that the compiler was too conservative to make potentially dangerous optimizations.

Finally, an alternative approach was to conduct our own experiments to determine the existence of a vulnerability in practice. This involved creating code examples that would reproduce the undefined behavior that arises from using *NotAtomic* instead of *Unordered*.

To empirically examine undefined semantics in practice, it was necessary to focus on specific optimizations that trigger such behavior and try to implement code where such optimizations are applied. We selected the load rematerialization optimization as a prime candidate for experimentation, as it has been frequently cited in the literature as a potential source of undefined behavior [2, 6].

The load rematerialization optimization entails splitting a single *NotAtomic* read into two during the compilation process. This optimization can prove beneficial for the compiler when there is a shortage of available registers. By freeing up the register that initially held the value, it can be utilized for other operations, while the value can be read again from memory in the future if required. However, this optimization can also create a vulnerability in scenarios where multiple threads concurrently access the same memory location. In such cases, two separate reads may return different values instead of the expected identical value assumed by the compiler. Therefore, our primary objective in this study was to reproduce concurrent access to a variable with a data race, thereby inducing pressure on the register allocator.

The experimental design is illustrated in Listing 1, utilizing C-like pseudocode for simplicity. Two threads concurrently access a shared global variable. The writer thread stores a new value, while the reader thread reads the value and subsequently writes it to every element of a designated output array. The expected behavior is that all elements of this array are equal: either the reader thread reads the initial value of the global variable or the value stored by the writer, and then writes it to each cell of the array. However, when the load rematerialization optimization is applied, the reader thread's consecutive reads (first from the shared memory and then from the first read saved locally) may be split into two separate global reads, which allows the reader thread to read two different values – one before the writer's store and another after it.

Furthermore, to induce the compiler to apply the load rematerialization optimization, it was essential to create pressure on the register allocator. To achieve this, the reader thread performs some useless operations on the registers that the compiler will not consider redundant, such as copying the elements of one array to another through indices in local variables.

Additionally, to increase the likelihood of concurrent execution by both the reader and writer threads, each thread executed a useless loop of random length before starting its work. Finally, the output array for the reader was kept relatively short to ensure that the execution times of the reader and writer threads did not differ significantly.

[2]"Register Spilling, Rematerialization, and Racy Accesses Discussion" on LLVM Discussion Forums.
[3]"Atomic Ordering: Non-atomic vs Unordered Discussion" on LLVM Discussion Forums.

```
#define DUMP_N 4

int shared_var = 0;

void* reader_thread() {
    int dump[DUMP_N] = {0};

    // the first read from shared_var
    int t = shared_var;

    for (int i = 0; i < DUMP_N; ++i) {
        // put pressure on the register allocator
        // ...
        // read from t, which the compiler may replace with the second read of shared_var
        dump[i] = t;
    }
    for (int i = 0; i < DUMP_N; ++i) {
        if (dump[i] != dump[0]) printf("Gotcha!");
    }
}

void* writer_thread() {
    // write to shared_var
    shared_var = 42;
}
```

Listing 1: Pseudocode of the load rematerialization optimization experiment

We conducted an experiment using Kotlin/Native and executed it with compiler version 1.7.20 on an x86-64 machine [30]. Unfortunately, we were unable to reproduce the expected undefined behavior. Furthermore, upon analyzing the generated bitcode, we observed that regardless of the pressure on the registers, there was only one `load` operation for the global variable. Therefore, the desired optimization was not performed.

Subsequently, we attempted to replicate the experiment using a different setup, specifically, the C language. For C, we used the Clang LLVM compiler [5], which shares the same LLVM backend as Kotlin/Native. This similarity implies that the dangerous optimizations we aimed to detect in the Kotlin/Native compiler should also be present in Clang LLVM. Furthermore, the C language is closer to assembly language, enabling greater control and transparency when conducting an experiment. Lastly, the analysis of compiler-generated code can be performed at an even lower level, via a tool such as Godbolt Compiler Explorer [10]. Therefore, conducting the experiment in C appeared to be more promising.

However, similar to the Kotlin/Native experiment, the results of the experiment in C, compiled by Clang LLVM 15.0.0 on an x86-64 machine, demonstrated that the rematerialization optimization was not performed. An analysis of the assembler [26] revealed that when there was no pressure on the register allocator, the value of the global variable was read into the register once and used from there, as expected. In the case of a shortage of available registers, the value was read into a stack slot once and utilized from there. Thus, the LLVM compiler, in practice, proved to be more conservative than its own specification by abstaining from risky load rematerialization.

3.3 Conclusions

As a result, the experiment has confirmed the hypothesis of certain community developers regarding load rematerialization optimization: the LLVM compiler tends not to perform it in practice. As future work, further examination of other dangerous optimizations is necessary. Nonetheless, an intermediate conclusion about the conservatism of LLVM towards optimizations with *NotAtomic* can already be drawn. This finding, in turn, implies that the Kotlin/Native compiler does not exhibit problematic behaviors due to *NotAtomic* ordering in practice.

4 Implementation of the Atomic Ordering Pass in Kotlin/Native Compiler

Simultaneously with studying the correctness of the current compiler, the effectiveness of potentially replacing *NotAtomic* accesses with *Unordered* was investigated. To conduct this analysis, it was necessary to develop a compiler pass within the Kotlin/Native compiler.

4.1 Implementation Details

The Kotlin/Native compiler pipeline is composed of a series of sequential phases, each of which may include smaller phases and passes. A pass refers to a specific set of operations executed while traversing the internal representation of the input code. Figure 6 illustrates an example of a compiler pipeline arrangement, displaying the drop-down of phases. The top-level phase, `backendCodegen`, is responsible for the LLVM code generation of bitcode from the Kotlin IR (Intermediate Representation) and its processing. As shown in the figure, it comprises smaller phases, the last of which is the `bitcodePostprocessingPhase`. The latter primarily optimizes the initially generated bitcode and comprises even smaller phases, each serving a distinct purpose, which we discuss in the following paragraphs.

In order to replace all memory accesses of a given atomic ordering with another, specifically, replacing *NotAtomic* memory accesses with *Unordered*, an Atomic Ordering Pass was developed. The approach involved iterating through all suitable instructions, such as *NotAtomic* `load` and `store`, in a single pass of the internal code representation and modifying their ordering mode to the desired one, such as *Unordered*. It should be noted that the focus of this investigation was on evaluating the performance of compiled programs rather than the compilation speed, and thus, the implementation's efficiency was not a crucial factor. Nevertheless, the implementation did not result in any noticeable slowdown during compilation.

The primary implementation code for the Atomic Ordering Pass, which substitutes *NotAtomic* with *Unordered*, is depicted in Listing 2 within the `runOnModule` function. The implementation closely follows the algorithmic description outlined earlier. Specifically, the code first iterates over all instructions in the module, subsequently filters the relevant

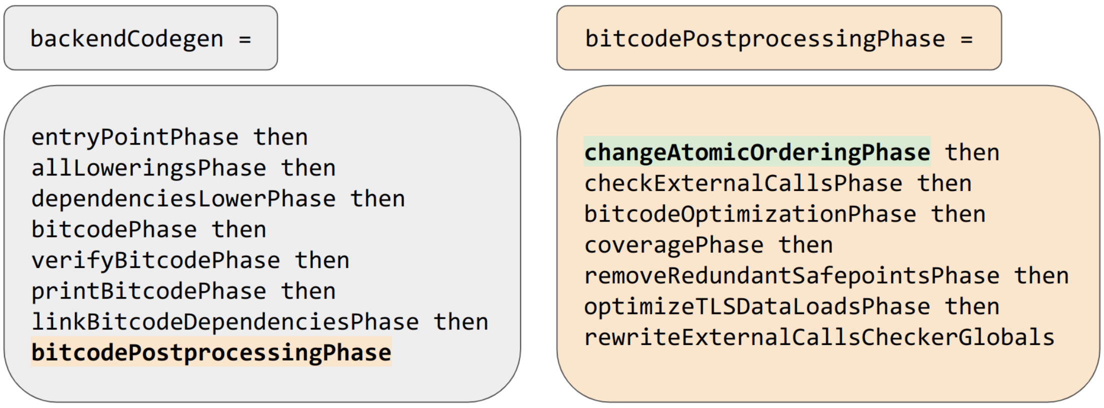
```kotlin
class ChangeAtomicOrdering() {
    private var replacedAccessesCount = 0

    fun runOnModule(module: LLVMModuleRef, llvmTargetData: LLVMTargetDataRef) {
        getFunctions(module)
            .flatMap { function -> getBasicBlocks(function) }
            .flatMap { block -> getInstructions(block) }
            .filter { inst ->
                isLoadOrStoreInst(inst) &&
                    isSuitableType(inst) &&
                    isByteSized(inst, llvmTargetData) &&
                    isPowerOfTwoSized(inst, llvmTargetData)
            }
            .forEach { inst ->
                if (LLVMGetOrdering(inst) ==
                        LLVMAtomicOrdering.LLVMAtomicOrderingNotAtomic) {
                    LLVMSetOrdering(inst, LLVMAtomicOrdering.LLVMAtomicOrderingUnordered)
                    replacedAccessesCount++
                }
            }
        println("Replaced accesses: $replacedAccessesCount")
    }
}
```

Listing 2: Simplified version of the Atomic Ordering Pass implementation

instructions, and finally replaces instances of *NotAtomic* with *Unordered* for those instructions. Additionally, the `replacedAccessesCount` counter is maintained to ensure the presence and number of modifications. Furthermore, the substitution of other atomic orderings requires the alteration of only a few lines of code that specify the desired orderings. For example, to replace *NotAtomic* with the strongest ordering that guarantees sequential consistency, one can simply replace `LLVMAtomicOrderingUnordered` with `LLVMAtomicOrderingSequentiallyConsistent` in the aforementioned listing. This feature proves particularly beneficial in the further testing of the implementation.

Furthermore, the filtering conditions are noteworthy. In addition to the self-explanatory `isLoadOrStoreInst` check that filters the `load` and `store` instructions, there are other checks such as `isSuitableType`, `isByteSized`, and `isPowerOfTwoSized`. The first condition ensures that the value in the instruction is either of floating-point, integer, or pointer type. The second checks that the size of the memory area in bits is a multiple of bytes. Finally, the third condition verifies that the bit length is also a power of two. All these checks were included out of necessity as the LLVM compiler generated errors with detailed requirements for instructions modified by the Atomic Ordering Pass. Namely, further compilation to machine code supports annotating with atomic orderings only instructions of suitable types and sizes. At present, the implemented filtering checks are adequate for performing correct replacements on all libraries and benchmarks studied in this research.

In order to ensure proper invocation of the implemented Atomic Ordering Pass, a trivial `changeAtomicOrderingPhase` phase was wrapped around the call to the `runOnModule` function, similar to the pre-existing code in the compiler. The resulting code was then



```
backendCodegen =

entryPointPhase then
allLoweringsPhase then
dependenciesLowerPhase then
bitcodePhase then
verifyBitcodePhase then
printBitcodePhase then
linkBitcodeDependenciesPhase then
bitcodePostprocessingPhase
```

```
bitcodePostprocessingPhase =

changeAtomicOrderingPhase then
checkExternalCallsPhase then
bitcodeOptimizationPhase then
coveragePhase then
removeRedundantSafepointsPhase then
optimizeTLSDataLoadsPhase then
rewriteExternalCallsCheckerGlobals
```

Figure 6: Kotlin/Native compiler pipeline with the pass being injected

injected into the appropriate top-level phase, as depicted in Figure 6. We will now provide a brief overview of the phases illustrated in the diagram to support the correctness of the Atomic Ordering Pass injection.

As previously mentioned, the top-level `backendCodegen` phase is responsible for generating LLVM bitcode from Kotlin IR and performing its subsequent processing. Therefore, any bitcode handlers, such as the Atomic Ordering Pass, must be included in this top-level phase. Moving one level deeper, the `backendCodegen` phase is comprised of several smaller phases. The first three phases prepare for bitcode generation, with the `bitcodePhase` subsequently generating the primary bitcode directly from Kotlin IR. The following two phases do not alter the bitcode; the first verifies it, while the second saves it at the user's request. During the `linkBitcodeDependenciesPhase`, all dependencies present in the generated bitcode are linked. Finally, the `bitcodePostprocessingPhase` completes the bitcode processing and invokes LLVM optimizations.

It is essential that the Atomic Ordering Pass runs on the most complete version of the bitcode, with all dependencies resolved, in order to have access to all program instructions. Thus, the `changeAtomicOrderingPhase` must be placed after the `bitcodePhase`, which generates the first bitcode version, and after the `linkBitcodeDependenciesPhase`, which resolves its dependencies. Therefore, `changeAtomicOrderingPhase` must be injected inside `bitcodePostprocessingPhase`, which performs the final bitcode processing.

Finally, let us briefly examine the `bitcodePostprocessingPhase` internals. The phases `checkExternalCallsPhase` and `rewriteExternalCallsCheckerGlobals` are paired together to process and verify external calls. The `bitcodeOptimizationPhase` then executes the LLVM optimization pipeline on the bitcode, while the subsequent phases, namely, `removeRedundantSafepointsPhase` and `optimizeTLSDataLoadsPhase`, perform additional optimizations related to eliminating unnecessary safepoints (points in the program that enable the garbage collector to perform) and reducing the number of thread data loads. If enabled, the `coveragePhase` instruments code to enable coverage data collection at runtime.

The primary effect of replacing atomic orderings is on the available optimizations, and thus, the Atomic Ordering Pass should be executed before the optimization phase, i.e., before the `bitcodeOptimizationPhase`. The remaining phases, as evident from their description, are independent of the atomic orderings of `load` / `store` instructions and do not affect them. Therefore, the final location of the `changeAtomicOrderingPhase`, as illustrated in Figure 6, permits the implementation to fully apply the new compilation scheme while remaining consistent with the rest of the compiler pipeline.

4.2 Testing the Implementation

The Atomic Ordering Pass implementation needed to be tested. This included verifying both that the replacement of *NotAtomic* with *Unordered* was actually executed on programs compiled by the modified compiler and that the intended changes impacted the behavior and performance of the compiled program. We accomplished this by analyzing the generated bitcode, measuring the execution time of simple programs, and conducting litmus tests to observe changes in the memory model. In the following section, we elaborate on the details of the testing process.

It is worth noting that testing was conducted in different environments, including Linux x64 and MacOS Arm64. As previously discussed, hardware memory models can affect program behavior, therefore it was essential to evaluate the implementation on distinct architectures. These environments were chosen due to their popularity in real-world systems, their common use in related studies, and their accessibility. We created a comprehensive guide [31] with instructions for building and running the modified compiler on these environments to facilitate the testing process.

The first step of our testing process was to validate the generated bitcode and ensure that the substitutions made to atomic orderings were indeed observable in the bitcode used for program compilation. Moreover, it was crucial to learn how to obtain the bitcode of a program after the optimization pipeline to analyze the changes in applied optimizations caused by the replacements of atomic orderings. This would also require obtaining a human-readable bitcode.

However, using only the available tools of the Kotlin/Native compiler, it was not possible to obtain the desired bitcode. Some tools did not support optimization mode, while others did not consider optimizations, and yet others returned the full version of the bitcode, which was unreadable because of large size of generated bitcode due to the linked libraries. To overcome this, we developed bash and Python scripts that use the LLVM disassembler utility [22] to convert intermediate compilation files into human-readable bitcode, and extract the section corresponding to the input program according to specific labels. As a result, we obtained a convenient way to extract the bitcode of small programs that is suitable for further analysis.

Using the developed tool, we analyzed the resulting bitcode for various simple programs that included loops, arithmetic, and array operations. Our analysis confirmed that the replacements of *NotAtomic* with *Unordered* were indeed performed.

The subsequent testing stage aimed to validate whether the replacements performed by the Atomic Ordering Pass were reflected in the resulting executable programs. This involved observing the impact on the executable programs themselves, such as alterations to memory access assurances and performance.

To test the change in memory access strictness, a well-known Store Buffering litmus test was compiled with two versions of the Kotlin/Native compiler: a regular one and a modified one that replaces all *NotAtomic* accesses with the strongest *SequentiallyConsistent* ones that guarantee sequential consistency. The result of running the litmus test compiled with a regular compiler showed weak behavior. In contrast, the sequentially consistent variant, which forbids such weak behavior, did not show weak results. Thus, the effect of changing the guarantees of memory access was observed.

To check changes in performance, a simple program was implemented that sums 10^9 array elements, and its execution time was measured on average over five runs using bash scripts. A program compiled with a regular compiler was compared to a modified one that replaces *NotAtomic* with *SequentiallyConsistent* as the strongest and, therefore, most performance-influencing mode. As a result, a difference of $5/26\%$ (with/without `-opt` flag) slowdown on the Linux x64 machine was observed, which seemed sufficient for this test.

Overall, we conducted tests on the Atomic Ordering Pass (AOP) using small programs and implemented scripts. Our testing approach enabled us to observe a direct alteration in atomic orderings within the generated bitcode and the expected impact on program behavior. Furthermore, the testing process involved configuring the building process, including the usage of Gradle and various compiler run modes and flags. Thus, a suitable environment for further experiments was created [31].

5 Evaluating Modified Kotlin/Native Compiler Performance

In this section, we will discuss the evaluation of performance changes resulting from replacement *NotAtomic* with *Unordered* made to the Kotlin/Native compiler. Firstly, we formulate the primary research questions and hypotheses that form the basis of the experiment. Subsequently, we delve into the experimental design, elaborating on the methodologies, testing framework and dataset employed to address the research questions. Furthermore, we discuss the process of adaption of the dataset selected for benchmarking purposes. Finally, we analyze the results obtained and conclude on the efficacy of replacing *Unordered* with *NotAtomic*.

5.1 Evaluation Research Questions and Hypotheses

Let us begin by defining the precise objective of the performance evaluation. This study primarily focuses on investigating the feasibility of replacing the *NotAtomic* ordering with *Unordered* in Kotlin/Native. Hence, the main goal of benchmarking is to quantify the performance degradation resulting from the modifications made to the compiler.

In addition, our secondary research objective is to identify the factors responsible for the performance slowdown associated with the compilation and optimization of code using *Unordered* . The findings of this investigation will allow us to draw conclusions about the implementation of *Unordered* in the LLVM compiler, which can be utilized to analyze compilation schemes in other LLVM-based language implementations

Let us now proceed to the hypotheses that form the basis of our experiments. As previously discussed, the distinction between the *NotAtomic* and *Unordered* orderings, as stated in the documentation, lies in the applied optimizations. Moreover, *NotAtomic* mode offers only a limited set of optimizations that are not accessible in the *Unordered* mode. Furthermore, these optimizations are localized, affecting only a subset of instructions.

Consequently, the distinction between *NotAtomic* and *Unordered* orderings is anticipated to be less pronounced in the context of large-scale real-world programs. Thus, the experiments will focus on micro-benchmarks, which are tiny programs designed to primarily evaluate the performance of individual language components. Examples of such micro-benchmarks include iterating over array elements and computing their sum, concatenating strings, and invoking a function with default arguments.

The subsequent hypothesis pertains to the multithreading of tests. It is crucial to highlight that the altered compiler substitutes *NotAtomic* with *Unordered* in any program, including single-threaded ones, as it lacks the ability to distinguish between them. Thus, we can utilize both single-threaded and multi-threaded programs for our experiments.

First and foremost, multi-threaded programs generally possess a higher number of synchronization elements, restrict the compiler from applying various optimizations. As the objective of measuring *NotAtomic* to *Unordered* change is to evaluate the impact of optimizations, it is anticipated that single-threaded programs would encounter fewer limitations and exhibit more significant performance degradation. Secondly, multi-threaded tests are commonly regarded as being susceptible to higher levels of noise arising from thread contention [27].

Based on the formulated hypotheses, it is expected that single-threaded micro-programs will display the most significant performance changes with lower levels of randomness. On the other hand, the performance of multi-threaded micro-programs and real-world macro-programs is anticipated to undergo relatively fewer changes.

Finally, we propose hypotheses regarding the expected outcomes of the experiments. The conclusions derived from the ”Assessing Kotlin/Native Compiler Correctness” section indicate that LLVM demonstrates a higher level of caution compared to the specification, as it tends to avoid employing potentially risky optimizations. Consequently, the actual disparity in optimizations employed in practice, and thus the performance variation between *Unordered* and *NotAtomic*, may not be as significant.

In contrast, during correspondence with Soham Chakraborty, the author of the article [4] and an active researcher in the field of the LLVM memory model, it was noted that, to the best of their knowledge, LLVM treats the *Unordered* ordering overly-conservatively. Specifically, LLVM prohibits all forms of reordering and elimination transformations on operations marked with the *Unordered* annotation. This suggests that substituting *NotAtomic* with *Unordered* would potentially impose more restrictions on optimizations than mandated by the specification, leading to a noticeable performance degradation.

5.2 Design of Evaluation Experiments

Let us now delve into the description of the experimental approach designed to address the research questions at hand. Based on the formulated hypotheses, our focus lies on single-threaded micro-benchmarks as the target tests for the experiments.

Furthermore, as the second phase of the experiment, a comparative analysis of the bitcode was conducted for programs exhibiting the most pronounced performance degradation. Specifically, we compared the bitcode version compiled by the baseline compiler with the version compiled by the modified compiler. Such a comparison aimed to reveal the difference in the applied optimizations, thereby identifying the underlying causes of the performance slowdown.

Hence, the primary experiment entails comparing the launch of single-threaded micro-benchmarks on the baseline and modified versions of the Kotlin/Native compiler, followed by an in-depth analysis of the bitcode of programs with the most noticeable performance degradation.

In order to validate our selection of single-threaded micro-benchmarks as target tests, we also planned to conduct performance experiments for multi-threaded tests and real-world macro-programs. Our intended outcome was to observe relatively smaller changes in performance during these experiments as compared to the target case.

5.2.1 Environment's Technical Profile

Prior to discussing the benchmarks, we present a concise overview of the technical characteristics of the utilized environment. The experiments were conducted using the Kotlin/Native compiler version 1.7.21, which was built from the `Jetbrains/kotlin` [13] repository on December 6, 2022. As previously discussed, when conducting experiments on compilation schemes, it is beneficial to consider different computer architectures. Two notable architectures in this context are *x86_64* and Arm, both of which are widely used and exhibit significant differences in their memory models. Thus, two machines were employed for the experiments, and their respective technical specifications are outlined in Figure 7.

Machine	CPU	RAM	OS version
Linux x64	AMD Ryzen 7 5800H with 8 cores	8 GB	Ubuntu WSL 2, 5.15.90.1-microsoft -standard-WSL2
MacOS Arm64	Apple M1 with 8 cores	16 GB	Darwin 22.1.0

Figure 7: Technical specifications of the machines used in the experiments

Furthermore, it is important to note that Ubuntu WSL 2 on Windows 11 was employed as the Linux x64 environment, as indicated in the aforementioned table. While using a modern and efficient WSL still incurs certain performance changes, it fully possesses the properties of the `x86_64`, making it suitable for our experimental purposes.

5.2.2 Exploring Kotlin/Native Benchmarks as a Testing Framework

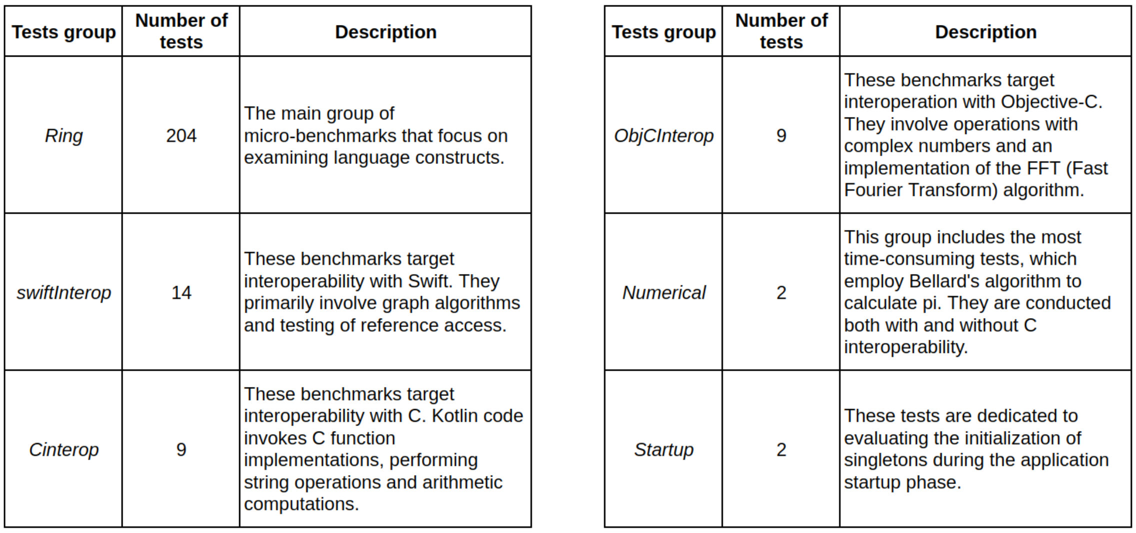
As the fundamental framework for conducting the experiments, the Kotlin/Native benchmarks were selected. These benchmarks consist of 240 performance tests along with the necessary infrastructure for their execution, which the Kotlin/Native compiler team uses to evaluate the project's performance. The benchmarks and tools required for their execution can be accessed in the open-source Kotlin repository on GitHub [16].

The Kotlin/Native benchmarks possess several qualities that make them the best choice for our experimental investigations. Firstly, they satisfy our target requirements as they already comprise a comprehensive set of micro-benchmarks, with the majority of them being single-threaded. Secondly, to the best of our knowledge, these benchmarks are the only open-source benchmarks specifically designed for Kotlin/Native, covering a wide range of language constructs. This extensive coverage enhances the likelihood of identifying programs that exhibit significant performance variations and facilitates drawing conclusions regarding the overall impact of modified compilation schemes on Kotlin/Native. Lastly, the Kotlin/Native benchmarks possess a well-established and thoroughly documented infrastructure that facilitates the execution of tests on different platforms, generates visual reports to analyze the results, and allows for the seamless integration of new tests.

The Kotlin/Native benchmarks framework has the following structure. Broadly speaking, it computes scores for each test in a sequential manner, recording the results in a report. The score calculation process involves the following steps. For a given number of attempts `attempts` (set to `20` by default), the framework measures the execution time of the test in nanoseconds. Subsequently, it calculates both the average and variation of these measurements and converts them to microseconds, which serves as the final score.

Delving deeper into a singular test measurement, the process unfolds as follows. Firstly, it initiates a warm-up phase, executing the test a predetermined number of times (defaulting to `10`). Following the warm-up phase, the framework restarts the test, and if the execution time is less than one second, the framework employs a specialized formula to calculate the desired number of iterations. Finally, the framework measures the execution time in nanoseconds for the computed number of test iterations and calculates the average of these measurements.

Thus, to enhance result accuracy, the framework employs repeated executions and warm-up runs, a well-established practice commonly employed in language benchmarking, such as Java [9]. Furthermore, the Kotlin/Native benchmarks themselves are categorized into distinct groups, and relevant information regarding these groups is presented in Figure 8.



Tests group	Number of tests	Description
Ring	204	The main group of micro-benchmarks that focus on examining language constructs.
swiftInterop	14	These benchmarks target interoperability with Swift. They primarily involve graph algorithms and testing of reference access.
Cinterop	9	These benchmarks target interoperability with C. Kotlin code invokes C function implementations, performing string operations and arithmetic computations.

Tests group	Number of tests	Description
ObjCInterop	9	These benchmarks target interoperation with Objective-C. They involve operations with complex numbers and an implementation of the FFT (Fast Fourier Transform) algorithm.
Numerical	2	This group includes the most time-consuming tests, which employ Bellard's algorithm to calculate pi. They are conducted both with and without C interoperability.
Startup	2	These tests are dedicated to evaluating the initialization of singletons during the application startup phase.

Figure 8: Test groups in Kotlin/Native benchmarks

However, the Kotlin/Native benchmarks suffer from a few drawbacks that we needed to address. The first limitation was the absence of readily available tools suitable for automating the experimentation process. While the project had all the requisite programs, configuring a pipeline that could launch benchmarks automatically with customizable configurations and report generation was necessary. To address this limitation, we created a series of bash scripts for both machines and created documentation for their usage [31].

The second drawback encountered in Kotlin/Native benchmarks was the issue of noise. Despite the implementation of the techniques within the project aimed at mitigating this problem, our initial runs still exhibited considerable levels of noise during experimentation. In order to assess the extent of this noise, we performed several tries of two separate runs of the entire framework on the same Kotlin/Native baseline compiler. Preliminary experiments showed that on a MacOS Arm64 machine, each run had approximately 5% of tests with results displaying differences in execution time ranging from 1% to a maximum of 17%, while on a Linux x64 machine, about 11% of tests exhibited differences from 1% to a maximum of 50%. Furthermore, this noise was observed randomly across different tests, with its absolute value being comparable to replacing *NotAtomic* with the strongest *SequentiallyConsistent* ordering, which caused differences in benchmark results up to 90% and 70% on MacOS Arm64 and Linux x64, respectively.

While the presence of noise did not affect the stage of analyzing the bitcode, it hindered the acquisition of reliable estimates regarding performance changes. To mitigate this problem, we employed several techniques to reduce noise in Kotlin/Native benchmarks, which will be elaborated on in the subsection 5.3.

5.2.3 Acquiring a Dataset of Multi-threaded Benchmarks

While there exist several widely recognized projects [17, 18], that support Kotlin/Native and employ multithreading, they primarily offer only correctness tests that are not well-suited for evaluating performance. Thus, we implemented custom micro-benchmarks that were specifically designed to assess concurrent behaviors. The Kotlin/Native benchmark framework facilitated the integration of these tests into a robust benchmarking infrastructure.

For our multi-threaded benchmarks, we adapted existing tests from the Kotlin/Native benchmark suite. Specifically, we utilized the single-threaded tests belonging to the `Ring::ForLoops` group, which computes the sum of elements in arrays of different primitive types. We selected these tests for several reasons. Firstly, they are simple and focused on the fundamental design of the language. Secondly, during our preliminary experiments, these tests consistently exhibited significant changes in performance when replacing *NotAtomic* with the *Unordered*. Thus, we regarded the `Ring::ForLoops` tests as a worst-case scenario, where the effects of the changes would be most pronounced. To further expand our test suite with similar behavior, we developed additional multi-threading micro-benchmarks that specifically targeted the optimizations available for *NotAtomic* in comparison to *Unordered*.

Our initial experiments have validated the effectiveness of the implemented benchmarks in executing the designated tasks. These tests demonstrated relatively low levels of noise compared to the only existing multi-threaded test within the Kotlin/Native benchmark suite. On average approximately 17% of the implemented tests exhibited a noise level of around 4-5%, with the highest observed noise level reaching 11%. In contrast, the multi-threaded `Ring::SplayWithWorkers` test consistently displayed a significantly higher noise level of 48%. Moreover, the specifically designed tests aimed to highlight the impact of variations in optimizations between *NotAtomic* and *Unordered*, revealed a significant performance change when executed using the modified compiler. Thus, we effectively designed and prepared an experiment involving multi-threaded micro-benchmarks.

Regrettably, our search for a suitable real-world macro-program for benchmarking purposes has not yielded satisfactory results thus far. Nonetheless, we intend to pursue this experiment in the future. We identify the `Ktor` framework [18], widely used for constructing server applications, as a promising candidate due to its popularity and high-performance capabilities.

5.3 Mitigating the noise in Kotlin/Native benchmarks

We will now delve into a discussion on the observed noises encountered throughout the study. Our observations revealed the presence of two distinct types of noise: consistent noise in certain tests and significant score discrepancies, reaching approximately 50% and representing isolated outliers. Importantly, occurrences of both noise types were mostly observed intermittently between different runs of all benchmarks. In contrast, when the framework executed repeated test attempts to refine the outcome, minimal deviations in execution times were observed, indicating their consistency. Consequently, it is reasonable to infer that the predominant cause of the observed noise can be attributed to global factors, such as CPU throttling or incomplete release of resources between tests, potentially arising from bugs in the garbage collector.

In order to address the issue of noise, we employed the technique of aggregation across multiple runs as our primary approach. Specifically, as previously described, noise predominantly arises between runs of the entire test suite. Consequently, we could mitigate this issue by averaging the results obtained from multiple such runs. This approach serves two primary purposes. Firstly, by utilizing an average, we are able to effectively eliminate sporadic random outliers. Secondly, through the calculation of variance, we can identify consistently noisy tests and interpret their results with enhanced caution. Despite the significant time cost associated with the proposed approach, it helped us to largely mitigate the adverse effects of noise on the final outcomes.

Nevertheless, we also made efforts to mitigate the impact of noise and minimize its absolute values in general by adjusting the internal parameters of the benchmarks. Specifically, in addition to the `attempts` parameter, which determines the number of runs for each test, we found the `benchmarkSize` parameter that establishes the size of each individual test within the main `Ring` suite. For instance, in tests belonging to the `Ring::ForLoops` subgroup, the `benchmarkSize` parameter specifies the size of the array used for calculating the sum, while in tests from the `Ring::Euler` subgroup, it represents the number of iterations for mathematical algorithms. We posited that increasing the number of runs for each test could further refine the scoring calculation, while change of the test sizes could enhance their resilience against external factors, thus reducing overall noise. This approach was considered potentially advantageous for consistently noisy tests that generate fluctuations owing to their intrinsic nature, which could be related to their size.

We conducted a preliminary investigation to assess the impact of the `attempts` and `benchmarkSize` parameters on the level of noise. Due to significant time constraints, we performed a single measurement for each unique combination of these parameters. Furthermore, we intentionally limited the range of `benchmarkSize` parameter variations to avoid potential occasions of extreme phenomena, such as resource scarcity or inadequately short runtime for accurate measurements. Notably, the MacOS Arm64 platform exhibited significantly lower levels of noise compared to a Linux x64 machine, and only negligible changes were observed in response to parameter adjustments. Therefore, the majority of experiments were conducted on a Linux x64 machine.

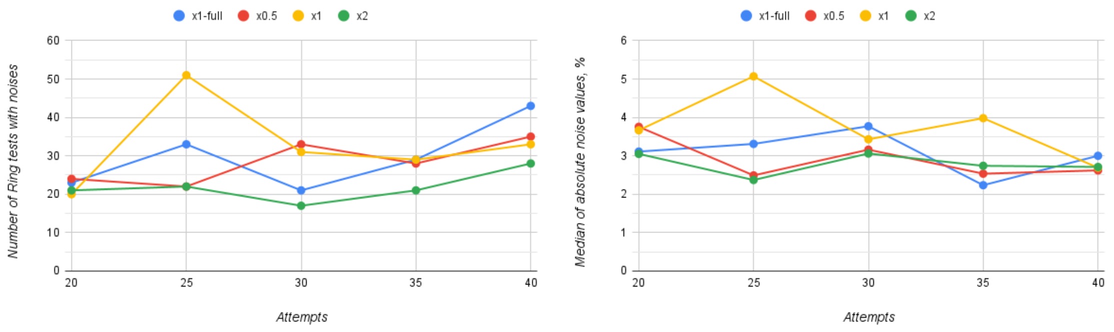


Figure 9: Impact of `attempts` and `benchmarkSize` parameters on a noise level of `Ring` tests on Linux x64 machine

The figures presented in Figure 9 depict the correlation between the `attempts` parameter and the number of `Ring` tests affected by noise (i.e., those exhibiting a difference greater than 1%), as well as the median absolute noise value. These variables demonstrate the average noise level, which we aimed to minimize in this experiment. Each line on the plot corresponds to a distinct configuration of the *benchmarkSize* parameter. The default value for `benchmarkSize` is set at `10,000` (`x1`), and two additional options we tried are to double (`x2`) or halve (`x0.5`) the value. Moreover, the `x1-full` option represents the execution of the entire test suite, including tests other than `Ring`, to validate the absence of inter-test influences on noise levels.

Despite the experiment's limitations, the variant with a `benchmarkSize` multiplied by 2 exhibited a noticeable trend of reduced noise occurrences and lower noise values on average. This trend reached its optimum at `attempts = 30`, leading to the selection of this particular configuration for the final experiments. It is important to note that altering the `benchmarkSize` and `attempts` parameters did not guarantee improved noise reduction in the final results, but it increased the likelihood heuristically.

The final approach for denoising Kotlin/Native benchmarks involved accurate error analysis. Each test is associated not only with a score calculated as the average of multiple `attempts` runs, but also with its spread. Throughout our experiments, it was observed that error calculations neglect the measurement accuracy, resulting in errors with absolute values below a nanosecond. As an initial measure, we implemented a minimum error threshold of 1 nanosecond. Consequently, certain exceptionally fast tests no longer exhibited noise that surpassed the measurement accuracy. In future work, it is essential to apply statistical techniques to accurately account for all types of errors and incorporate them in the analysis.

To mitigate noise, we propose employing averaging over complete benchmark runs, enabling the removal of outliers and identification of particularly noisy tests. Furthermore, we have determined the optimal parameter configuration and refined the error calculation methodology. These advancements are expected to effectively reduce the overall quantity and magnitude of noise in the final results.

5.4 Experimental Results Analysis and Discussion

In this subsection, we will present the outcomes of the conducted experiments, followed by providing their potential interpretation and drawing conclusions.

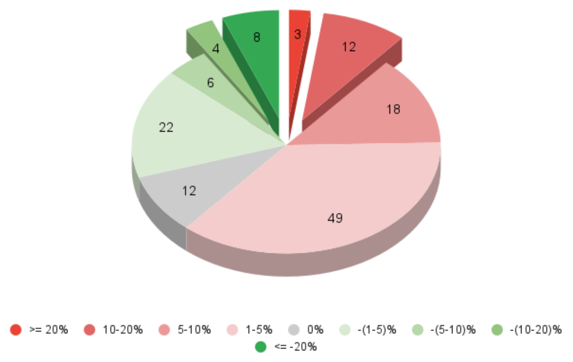
The focal point of our study was an experiment involving the execution of Kotlin/Native benchmarks, aimed at assessing the performance degradation of the modified compiler. As detailed in the preceding subsections, in order to mitigate noise, it was imperative to conduct multiple complete runs of all benchmarks and subsequently calculate the average results. We successfully executed a total of 20 runs for both the baseline and modified compilers on each of the machines. Furthermore, we employed the configuration parameters `benchmarkSize = 20,000` and `attempts = 30` as an optimal setup based on our earlier preliminary experiments.

We aggregated multiple reports on full benchmark runs using the following procedure. Firstly, we computed the median value for each test performed on each compiler. As previously mentioned, Kotlin/Native benchmarks often exhibit sporadic random outliers, and the application of the median value was intended to mitigate their impact on the final results. Secondly, we calculated the coefficient of variation for each test, which represents the ratio of the standard deviation to the mean. This statistical measure provides a relative estimation of the spread or measurement noise. By estimating the noise level in percentage, we could compare the noise levels of even very different tests in terms of scale, as well as provide error estimates for other metrics.

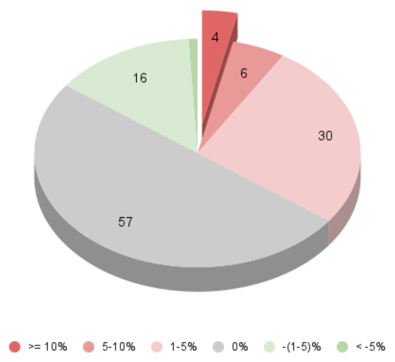
Following this aggregation process, we obtained the finalized dataset [29]. Initially, we computed the percentage change in the average score of each test when using the modified compiler compared to the baseline one. Subsequently, we filtered out changes that can be considered at least moderately significant. Specifically, we retained only those tests where the percentage of score change exceeded the noise estimation percentage.

Consequently, out of the total of 329 tests conducted on Linux x64 and 351 tests on MacOS Arm64, 135 and 115 tests respectively exhibited substantial performance differences. The majority of the remaining tests demonstrated no notable change in performance: 152 tests exhibited a score change of less than 1%, and an additional 38 filtered tests displayed a change of less than 5% on Linux x64. Likewise, on MacOS Arm64, 201 removed tests had a score change of less than 1%, and a further 33 tests exhibited a change of less than 5%. Among the remaining unfiltered tests, only 4 and 2 tests on Linux x64 and MacOS Arm64 respectively, displayed a level of noise comparable to the percentage of changes.

Therefore, our filtering methodology enabled us to retain nearly all tests with significant performance variations while safeguarding against the influence of noise.

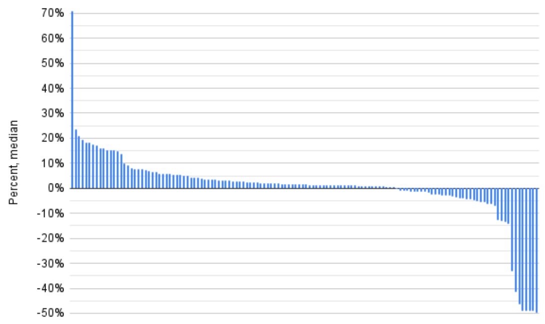


(a) Linux x64

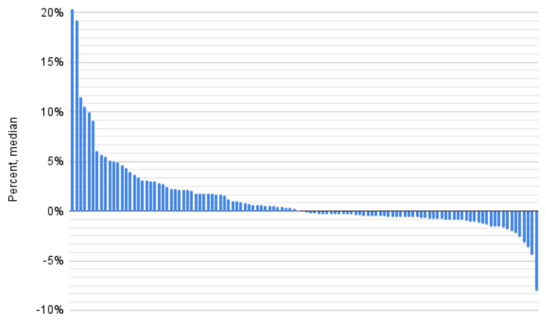


(b) MacOS Arm64

Figure 10: Distribution of tests with trusted performance changes by groups



(a) Linux x64



(b) MacOS Arm64

Figure 11: Score variations of tests with trusted performance changes

Now, let us proceed to the analysis of the tests that exhibited a noteworthy performance shift. Figures 10a and 10b illustrate the distribution of tests based on the magnitude of these changes. In this context, the gray shading labeled as "0%" indicates a significant but less than 1% alteration. Figures 11a and Figure 11b present the scores corresponding to all significant score changes, thereby highlighting the maximum and minimum observed values.

Performance degradation. The performance degradation resulting from the replacement of *NotAtomic* with *Unordered* exhibited a significant impact. Approximately 11% of all filtered tests exhibited changes exceeding 10% on the Linux x64 platform, while approximately 9% showed changes exceeding 5% on the MacOS Arm64 platform. Moreover, around one-third of the tests demonstrated slowdowns ranging from 1% to 5% on both machines.

Performance improvements. Surprisingly, our results reveal the presence of not only performance degradation but also performance improvement in some tests. However, the degradation of performance is more pronounced than the improvement, as demonstrated in the charts. Across both machines, slowed tests are more prevalent, with absolute values of changes reaching larger magnitudes. Specifically, on Linux x64, performance degradation of up to 71% was observed, as opposed to improvement of only up to 50%. On MacOS Arm64, the slowed tests exhibited performance degradation of up to 20%, while performance improvement was only up to 8%.

Comparison between different hardware. Furthermore, it is crucial to compare the results obtained on different hardware. The charts indicate that the Arm64 architecture, represented by the MacOS machine, experienced a notably lesser impact when *NotAtomic* was replaced with *Unordered*, compared to the $\mathrm{x86\_64}$ architecture represented by the Linux machine. This observation holds true for both the number of tests exhibiting score differences and the absolute magnitudes of the changes. Notably, on the Linux x64 platform, twice as many tests displayed a significant non-zero change, and the maximum slowdowns and accelerations differed in magnitude by more than 3 and 5 times, respectively.

Multi-threaded benchmarks. Regarding the multi-threaded benchmarks, they generally substantiated the corresponding hypothesis. Although they exhibited noticeable changes of approximately 15% on Linux x64 and changes ranging from 3% to 5% on MacOS Arm64, the single-threaded micro-benchmarks still yielded the most substantial score changes, as anticipated.

To investigate the underlying causes behind the obtained results,we conducted a preliminary analysis of the bitcode on several tests exhibiting significant score variations. The bitcode of two program versions, one compiled with a baseline compiler and the other with a modified compiler, was compared using a previously developed tool.

In our initial analysis, no significant differences were found in the applied optimizations within the main bitcode of the tests. However, it is possible that these optimizations, in fact, were employed in the Kotlin standard library code, extensively used in the tests. Thus, further investigation into the bitcode of the standard library is required to confirm this hypothesis. However, performance changes may not be entirely due to bitcode optimizations alone, as differences between *Unordered* and *NotAtomic* could potentially arise at a lower level, such as generated instructions during the translation from bitcode to machine code.

Upon further examination of the bitcode in the performance-improved tests, it became evident that the majority of instructions with modified ordering were related to accessing the garbage collector and its associated structures. The garbage collector, as a sophisticated subroutine tightly integrated into the main program, operates dynamically during runtime, closely tied to the program's actions. As a result, even slight modifications introduced by *Unordered* compilation to the operations of the garbage collector may inadvertently lead to positive effects on its functionality, consequently impacting the overall performance of the entire test.

Nevertheless, analyzing the bitcode of only a limited number of benchmarks does not provide a conclusive explanation for all the observed effects. As part of future work, we intend to expand our analysis to encompass a broader range of tests, including the examination of the standard library linked to them. Additionally, we aim to explore programs at a deeper level of machine instructions to gain a more comprehensive understanding of the underlying mechanisms.

In summary, the performance evaluation experiments on the modified Kotlin/Native compiler yielded the following findings. The replacement of *NotAtomic* with *Unordered* led to noticeable and consistent performance degradation, observed in approximately 60% of tests on Linux x64 and 35% of tests on MacOS Arm64. Quantitatively, this degradation manifested as a slowdown of about 10-20% in roughly 11% of the tests, reaching up to 70% on Linux x64. On MacOS Arm64, the slowdown ranged from 5-10% in approximately 9% of the tests, reaching a maximum of 20%.

Based on the obtained results, it is evident that replacing *NotAtomic* with *Unordered* is still accompanied by significant performance loss and cannot be implemented in the current Kotlin/Native compiler. Furthermore, these findings support the hypothesis regarding the insufficient performance of the *Unordered* implementation in LLVM.

6 Conclusions and Future Work

In this concluding section, we summarize our research findings. Initially, we outline the tasks that were set at the outset of this work and report on their completion. Next, we present the overall results obtained, along with their interpretation. Finally, we identify the tasks that still require attention in future work.

6.1 Completed Tasks

The purpose of this study was to determine the necessity and effectiveness of replacing *NotAtomic* memory accesses with *Unordered* ones in Kotlin/Native. The following tasks were completed to achieve this goal.

Firstly, the correctness of the current Kotlin/Native compiler implementation was tested in practice. This involved studying the LLVM sources, collecting information from the developer community, and attempting to reproduce the incorrect behavior with specially implemented code snippets.

Next, the approach was implemented in the Kotlin/Native compiler by understanding the compiler code base, how different data types accesses are handled, and the order of compilation phases. The modified compiler was tested by implementing tools for analyzing the final bitcode of the program and measuring the performance of a mock project. Furthermore, the implementation was set up to be run in different environments of Linux x64 and MacOS Arm64.

Finally, the impact of the approach on performance was measured by collecting a broad test corpus consisting of single-threaded and multi-threaded micro-benchmarks. The tests were adapted: their noise was reduced, execution in different environments was automated, and the most important tests for analysis were chosen. The benchmark results were then obtained and analyzed.

6.2 Results and Discussion

Based on the research, the following conclusions were drawn. Despite the hypothetical possibility of optimizations leading to undefined behavior, we could not reproduce the vulnerability in practice. Such dangerous optimizations are currently not used in LLVM, and the compiler acts more conservatively than the specification allows. Replacing *NotAtomic* accesses with *Unordered* ones results in a performance slowdown of approximately 10-20% on Linux x64 and 5-10% on MacOS Arm64 across approximately 10% of the benchmark samples collected. This slowdown can potentially reveal itself in some real-world programs.

Thus, it is currently unnecessary to replace *NotAtomic* accesses with *Unordered* ones. Nevertheless, according to the specification, *NotAtomic* accesses remain incorrect. Future versions of LLVM may introduce more aggressive optimizations, potentially leading to undefined behavior. Hence, it is essential to inform the LLVM developers about this issue and to discuss it with other compiler developers, such as Graal VM.

6.3 Future Work

Further research could involve the ongoing investigation of vulnerabilities in practice, as well as the exploration of other potentially dangerous optimizations and scenarios. Improvements to the test corpus can be made by correcting the Kotlin/Native benchmarks, better parameter tuning, and applying advanced techniques found in state-of-the-art benchmarking libraries, such as JMH [12]. Additionally, further investigation into other access modes, such as *Monotonic* and *SequentiallyConsistent*, involving the evaluation of their performance impact, may be required to develop the Kotlin/Native memory model.

As a continuation of this study, it is also necessary to explore other controversial compilation schemes that arise due to the differences in the semantics of the JVM and LLVM memory models.

References

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014). ISSN: 0164-0925. DOI: 10.1145/2627752. URL: https://doi.org/10.1145/2627752.

[2] Hans-J. Boehm. “How to miscompile programs with benign data races”. In: *HotPar* (May 2011).

[3] Hans-J. Boehm and Brian Demsky. “Outlawing Ghosts: Avoiding out-of-Thin-Air Results”. In: MSPC ’14 (2014). DOI: 10.1145/2618128.2618134. URL: https://doi.org/10.1145/2618128.2618134.

[4] Soham Chakraborty and Viktor Vafeiadis. “Formalizing the Concurrency Semantics of an LLVM Fragment”. In: CGO ’17 (2017), 100–110.

[5] *Clang Compiler User’s Manual*. URL: https://clang.llvm.org/docs/UsersManual.html.

[6] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. “Bounding Data Races in Space and Time”. In: *ACM SIGPLAN Notices* (June 2018), pp. 242–255. DOI: 10.1145/3192366.3192421. URL: https://doi.org/10.1145/3192366.3192421.

[7] Roman Elizarov. *Kotlin/Native Memory Management Roadmap*. The Kotlin Blog. July 2020. URL: https://blog.jetbrains.com/kotlin/2020/07/kotlin-native-memory-management-roadmap/.

[8] Roman Elizarov. *Kotlin/Native Memory Management Update*. The Kotlin Blog. May 2021. URL: https://blog.jetbrains.com/kotlin/2021/05/kotlin-native-memory-management-update/.

[9] Andy Georges, Dries Buytaert, and Lieven Eeckhout. “Statistically Rigorous Java Performance Evaluation”. In: *ACM SIGPLAN Notices* 42 (Oct. 2007). DOI: 10.1145/1297027.1297033.

[10] Matt Godbolt. *Compiler Explorer*. URL: https://godbolt.org/.

[11] *GraalVM Repository*. URL: https://github.com/oracle/graal.

[12] *Java Microbenchmark Harness (JMH) Repository*. URL: https://github.com/openjdk/jmh.

[13] *Jetbrains/kotlin Repository*. URL: https://github.com/JetBrains/kotlin.

[14] *Kotlin Multiplatform Documentation*. URL: https://kotlinlang.org/docs/multiplatform.html.

[15] *Kotlin Native Documentation*. URL: https://kotlinlang.org/docs/native-overview.html.

[16] *Kotlin/Native in JetBrains/kotlin Repository*. URL: https://github.com/JetBrains/kotlin/tree/master/kotlin-native.

[17] *kotlinx.coroutines Repository*. URL: https://github.com/Kotlin/kotlinx.coroutines.

[18] *Ktor Repository*. URL: https://github.com/ktorio/ktor.

[19] Lun Liu, Todd Millstein, and Madanlal Musuvathi. “A Volatile-by-Default JVM for Server Applications”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133873. URL: https://doi.org/10.1145/3133873.

[20] Lun Liu, Todd Millstein, and Madanlal Musuvathi. “Accelerating Sequential Consistency for Java with Speculative Compilation”. In: PLDI 2019 (2019), 16–30. DOI: 10.1145/3314221.3314611. URL: https://doi.org/10.1145/3314221.3314611.

[21] *LLVM Atomic Orderings Documentation*. URL: https://llvm.org/docs/Atomics.html#atomic-orderings.

[22] *LLVM Disassembler Documentation*. URL: https://llvm.org/docs/CommandGuide/llvm-dis.html.

[23] *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.

[24] *LLVM Test-suite Repository*. URL: https://github.com/llvm/llvm-test-suite.

[25] E. Moiseenko, A. Podkopaev, and D. Koznova. “A Survey of Programming Language Memory Models”. In: *Program Comput Soft* 47 (May 2021), pp. 439–456. DOI: 10.1134/S0361768821060050. URL: https://doi.org/10.1134/S0361768821060050.

[26] Evgeniy Moiseenko. *Load Rematerialization Test Implemented in C*. Godbolt Compiler Explorer. URL: https://godbolt.org/z/PdnGqesM6.

[27] Peizhao Ou and Brian Demsky. “Towards Understanding the Costs of Avoiding Out-of-Thin-Air Results”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276506. URL: https://doi.org/10.1145/3276506.

[28] Ekaterina Petrova. *Try the New Kotlin/Native Memory Manager Development Preview*. The Kotlin Blog. Aug. 2021. URL: https://blog.jetbrains.com/kotlin/2021/08/try-the-new-kotlin-native-memory-manager-development-preview/.

[29] Gleb Solovev. *Aggregated results of Kotlin/Native benchmark experiment*. URL: https://rb.gy/231mi.

[30] Gleb Solovev. *Load Rematerialization Test Implemented in Kotlin/Native*. URL: https://rb.gy/5rxbh.

[31] Gleb Solovev and Denis Lochmelis. *Atomic Ordering Pass Benchmarking Readme*. URL: https://rb.gy/944s4.

[32] *Swift Programming Language Repository*. URL: https://github.com/apple/swift.

[33] *The Glasgow Haskell Compiler Repository*. URL: https://github.com/ghc/ghc.

[34] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[35] *The LLVM Compiler Infrastructure Repository*. URL: https://github.com/llvm/llvm-project.