

# **Testing conformance of Kotlin compilers to memory model specifications**

by

**Denis Lochmelis**

Bachelor Thesis in Computer Science

Submission: May 16, 2023

Supervisor: Anton Podkopaev  
Industry advisor: Evgeniy Moiseenko

## Statutory Declaration

Family Name, Given/First Name	Lochmelis, Denis
Matriculation number	30006532
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  .....

Date, Signature

## **Abstract**

The Kotlin programming language can be compiled to multiple platforms, and a single piece of code is certainly expected to behave similarly on these platforms. However, making multithreaded programs behave similarly in different environments is a challenge due to different memory models. A common new memory model is being developed, and it should be tested whether the Kotlin compiler conforms to it.

In this work, a tool is developed to achieve that, using the concept of litmus tests. To implement the tool, first the existing litmus testing tools are surveyed. The tool is then written using the found techniques. Finally, it is used to run a number of tests on the most recent version of the Kotlin/Native compiler. Some of the discovered behaviors violate the Java memory model, and should therefore be considered a bug in the compiler.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related work</b>	<b>3</b>
2.1	Tools for litmus testing . . . . .	3
2.1.1	herdtools7 . . . . .	3
2.1.2	JCStress . . . . .	3
2.1.3	Lincheck . . . . .	4
2.2	Common techniques . . . . .	4
<b>3</b>	<b>Implementation</b>	<b>6</b>
3.1	Test interface . . . . .	6
3.2	Test runner . . . . .	7
<b>4</b>	<b>Tool evaluation</b>	<b>10</b>
<b>5</b>	<b>Testing the compiler</b>	<b>13</b>
<b>6</b>	<b>Conclusions and future work</b>	<b>16</b>

# 1 Introduction

One of the key features of the Kotlin programming language is Kotlin Multiplatform, which allows to run the same Kotlin code across different platforms, such as Android, iOS, Windows, and more. It is achieved through compiling Kotlin code in different ways, for example, into Java bytecode for running on JVM [6], or into a native executable via the LLVM [12] framework. Naturally, one code fragment is expected to behave in a similar way on any platform. However, different platforms can have major differences, both in software and hardware. This difference poses many challenges for the development of Kotlin compiler. In this work, we will focus particularly on the issue of different memory models.

For the purposes of this work, we will define “memory model” as follows: a memory model is the set of rules for the interaction of program threads via shared memory. In other words, a memory model defines which behaviours are allowed for a multi-threaded program and which are not. A memory model can be defined on many levels: for a particular processor, for a programming language, or for some modelling language. The simplest memory model would be the so-called sequential consistency [11], or SC for short. Under this memory model, the only allowed behaviours are the ones where the program performs as if all its instructions were executed sequentially in some order by a single executor. Here is an example:

$$\begin{array}{l} x, y = 0; \\ x = 1; \parallel y = 1; \\ a = y; \parallel b = x; \end{array}$$

Figure 1: Sample program that runs differently under different memory models

This notation describes a program where two threads run in parallel, as denoted with parallel lines, with code before parallel lines happening before launching the threads. Notice how under the SC model it is impossible for this program to end with  $a == 0$  and  $b == 0$  at the same time.

In the real world, however, the SC model is rarely used, because it limits the performance of the program by disallowing many compiler optimizations [19, 13, 14]. For example, the Java language has its own different memory model [15] (called the Java Memory Model, or JMM), and if the program above is run on JVM, it can in fact end with both  $a == 0$  and  $b == 0$ . The behaviors that violate SC are called weak behaviors. This example clearly demonstrates how the same multithreaded program can behave differently under different memory models.

Turning back to Kotlin Multiplatform, this change of behaviour poses a problem, because the main targeted platforms — namely, JVM and LLVM — have different memory models [15, 12]. This can be especially problematic when sharing libraries across platforms, as the behavior of the library may be dependent on the underlying memory model. For example, Kotlin Coroutines library, another key feature of Kotlin, was written based on JMM. Due to it relying on some aspects of that particular memory model, this library may require some changes in order to be fully functional on the Native platform. To address this issue of reusing code, a common memory model for Kotlin is currently being developed. It will ensure that Kotlin programs produce consistent results across different

platforms and under different circumstances.

Such a common memory model has to be guaranteed by the Kotlin compiler, which has to produce programs that conform to the defined memory model. Therefore, the development of a common memory model implies modifying the existing Kotlin compiler. Naturally, any new software has to be tested, and in this case we have to test if a given version of Kotlin compiler does indeed conform to a given memory model. This kind of testing is usually done either by providing a formal proof, by testing empirically, or by using both methods [9]. Formal proofs are deterministic and irrefutable, so they are used for maximum level of confidence. However, formal proofs for a large enough program can get quite complex and take too much effort to be practical [9]. The final version of the new memory model will most likely be verified with a formal proof (moreover, the it may even incorporate some features specifically for simplifying such a proof), but a similar proof would not be feasible for the entire Kotlin compiler. Therefore, we will focus on empirical testing of conformance to a memory model.

Empirically testing a program basically means running this program a large number of times and checking if no disallowed behaviors appeared. It is important to note that this kind of testing does not guarantee this program will never produce a disallowed behavior, but it still can find a number of bugs if they exist [21]. To test conformance to a memory model in particular, it is useful to run not just any programs, but special small code snippets designed to check some specific aspect of a memory model. These small snippets are referred to as "litmus tests" [1], and running them many times (anywhere from tens of thousands to tens of millions and beyond) is usually called "litmus testing". The short program previously shown in figure 1 is in fact a classic litmus test called "store buffering", or SB for short.

The topic of litmus testing a memory model has been explored in different contexts, such as programming language memory models or hardware memory models. Accordingly, there exist various tools for performing litmus testing in these contexts, such as the herdttools suite [7] or JCStress [8]. The key point is, though, that none of these tools provide the functionality necessary for testing Kotlin compiler memory model conformance. The JCStress tool does exactly what we would like to achieve, but only with Java and heavily relying on Java-specific features. Therefore, though it can help with testing Kotlin/JVM compiler, it cannot be used for testing Kotlin/Native compiler, which is just as important for checking the common memory model. The litmus7 of herdttools suite is focused on testing hardware memory model, and cannot be easily adapted for testing the Kotlin compiler, while other relevant tools from the suite operate on theoretical memory models.

Therefore, to test the conformance of Kotlin compiler to a memory model, the need for a custom Kotlin-oriented litmus testing tool arises. The goal of this work is to implement such a tool, and then use it to perform some tests on the Kotlin compiler.

## 2 Related work

Litmus testing is a widely used technique for reasoning about memory models and even more generally about concurrent programs. The tests themselves can either be actually run on some hardware, or be used for special analysis called model checking. The latter is a common technique for developing and testing theoretical memory models, whereas the former is more useful with working with implementations of these memory models.

To create a custom litmus testing tool, it is necessary to first properly understand what possibilities do existing tools offer, and which techniques they employ to achieve better results. This is what is going to be discussed in the following subsections.

### 2.1 Tools for litmus testing

Here is a short review of the tools commonly used for performing litmus testing. As it was mentioned previously, there are different kinds of memory models, and also the litmus testing approach can be applied with different goals. Therefore, each of these tools does its own thing and deserves a separate description.

#### 2.1.1 herdtools7

The herdtools7 tool suite [7] is the de-facto state-of-the-art tooling for operating with hardware-level litmus tests. It includes several separate tools aimed at different use-cases: for simulating theoretical memory models, for generating litmus tests according to specifications, and more. To provide some examples, it has been used to help develop ARMv8 memory model [16], explore behaviors of POWER processors [17], test memory models of various GPUs [3], formalize the Linux kernel memory model [2], or search for bugs in C compilers [20].

The one specific tool we are now most interested in is the `litmus7` tool [4], which can run litmus tests on hardware. The main aim of this tool is checking the behaviors that appear in hardware, and as such, it requires the test to be written in assembler-like syntax. The tool then converts the test description into a C program with some custom macros inserted. Running the resulting program runs the test defined by the given assembler description and outputs the encountered behaviors.

Unfortunately, this tool cannot be directly used for the purposes of testing Kotlin compiler, because it is only aimed at low-level memory models. Still, it is quite important to understand how does this tool achieve its efficiency and which of its approaches can be used in the case of Kotlin.

#### 2.1.2 JCStress

JCStress [8] is a part of the OpenJDK project<sup>1</sup> which, as stated in the project description, aims "to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware". Unlike herdtools7, JCStress is not intended for any usage outside of testing the behavior of JVMs (and by extension the hardware there JVMs run on). In practice, this tool is an advanced Java-oriented litmus test runner which comes with a large number of predefined tests that explore various aspects of JMM. With the

---

<sup>1</sup><https://openjdk.org/projects/code-tools/jcstress/>

help of JCStress, a number of bugs in the implementations of Java standard library was discovered [18], and further tools for concurrency testing [5] were created.

At the first glance, JCStress looks like the perfect candidate for testing the Kotlin compiler, since Kotlin can run on JVM. The problem is, we need a common Kotlin memory model, which requires testing it not only for Kotlin/JVM, but for Kotlin/Native as well. And even though it is generally possible to translate Java into Kotlin in order to then run the same code on a platform other than JVM, JCStress extensively uses JVM features, such as Reflection API. Therefore, it is not possible to use this tool for testing Kotlin/Native compiler. Nonetheless, since JCStress achieves a very similar goal in a different context, it can provide insight for our purposes. Its large set of custom memory model litmus tests is also very useful as a basis for our compiler testing.

### 2.1.3 Lincheck

Lincheck [10] is a Kotlin tool intended for testing concurrent algorithms implemented in JVM-based languages. It allows for stressing the algorithm in a similar fashion to what running a litmus test looks like, as well as for simple model checking.

While this tool is useful for testing algorithms, in the context of testing the Kotlin compiler it provides little value. Even though it is mostly written in Kotlin, it still uses a noticeable amount of Java code, which is the same issue as with JCStress. Moreover, since it operates on a higher abstraction level compared to litmus tests, it uses techniques that are not favorable for litmus testing, such as inserting extra memory fences into user code. In practice, this tool cannot consistently reproduce all behaviors of the SB test, which can be considered the most basic one of memory model litmus tests. Still, this tool deserves a mention when talking about litmus testing in general, since it is able to find bugs in some standard Java concurrent structures.<sup>2</sup>

## 2.2 Common techniques

The simplest litmus testing tool is just a loop which repeatedly runs the given litmus test. However, there are multiple techniques that can be used to achieve much higher frequency of observing more behaviors. They involve artificially inducing thread contention, trying different thread configurations, and more. By looking into [8], [18], [7], [4] and [9], we can outline the list of the techniques that can be used in the new tool:

- Allocating many tests at once and running them sequentially in one go. Doing so removes the need for extra operations like memory allocation during the run, which in turn decreases the number of memory fences. This approach also stresses cache hits.
- Occasionally synchronizing threads with a barrier. This prevents one thread from consistently running ahead of another, which would result in no concurrency at all. Moreover, it is important to use a barrier with actively waiting threads instead of, for instance, a pthread one [4]. This is due to some barriers introducing too much overhead, particularly compared to very lightweight code in litmus tests.
- Manipulating thread affinity. Thread affinity is the set of CPU cores, on which the thread is allowed to run. Actively manipulating this set allows to make sure that

---

<sup>2</sup><https://github.com/JetBrains/lincheck#Example>

some thread configurations are checked. To illustrate, some behaviors may occur only when the two interacting threads use the same CPU cache of a certain level [18].

- Addressing false sharing. False sharing happens when two independent variables are stored in the same cache line, which renders them not independent for some operations. In practice, this disallows many behaviors, and as such is undesirable for a litmus testing tool. False sharing can happen particularly often for sequentially declared and/or allocated variables, which is also the case with litmus tests. Hence it is important to address this problem. Usually it is done by shuffling the pointers to the allocated memory chunk.
- Parallelizing the same test. Firstly, this increases the number of tests run, which in turn increases the probability to see more behaviors. Secondly, this puts some additional stress on the memory subsystem, which may cause the behaviors that do not happen under usual conditions.

This certainly is not the full list, but most of the other optimizations are case-specific. For example, JCStress pays significant attention to forking JVMs with various parameter configurations, but this cannot be applied to Kotlin compiler case since it doesn't have as many concurrency-related parameters.

Aside from these, there are a couple of other common ideas that seem to relate to all litmus testing tools:

- The results of litmus tests are inherently non-deterministic. The impact of one or another approach on the resulting test behaviors is basically impossible to definitively predict in advance, or, as [18] puts it, the parameters are an "educated guess". Therefore, all of the options for litmus testing tools are usually left as runner parameters.
- Normally we only care *if* the behavior appears and not how often it appears. At the same time, we want to make sure that the results are reproducible. Given that, [9] shows that if a behavior appears just 3 times, the probability of it appearing again can be estimated a little over 95%. This idea removes the necessity for striving towards higher and higher frequency of rare behaviors, because there is no direct link between the frequency of weak behaviors and their count.

## 3 Implementation

Any litmus testing tool has at least two parts: a way to define new litmus tests — a test interface, and a way to run these tests — a runner. Advanced tools may include additional functionality like data visualization or statistical analysis, but these are not necessary. A general pipeline of using any litmus testing tool is similar: write some tests according to provided interface, and then run them with the provided runner, optionally setting run parameters. Here is how these two key parts are implemented in our new tool.

### 3.1 Test interface

In general, there is no universal way in which these tests should be defined, except that defining tests should be at least somewhat user-friendly. The specifics depend highly on the context: for example, the `litmus7` tool from `herdtools` requires tests to be written in some subset of assembly code, because it aims at testing hardware memory model. At the same time, `JCStress` requires tests to be written as Java classes with multiple custom annotations, since it aims at JMM and uses code generation under the hood. In our case, tests must be written in Kotlin, because we are targeting Kotlin compiler in our research. We also cannot use methods like Java Reflection, which `JCStress` uses extensively, as we are targeting Kotlin/Native, and so we cannot simply translate `JCStress` Java code into Kotlin.

Kotlin offers a variety of ways that can be used to declare a litmus test, namely via extending a class, via some DSL<sup>3</sup> (another Kotlin feature), or via code generation using KSP.<sup>4</sup> We needed to pick one of these methods, which allows for test readability, leaves room for different features, and does not introduce too much instrumentalization to user code:

- Extending some abstract class is the most straightforward way to define a test. The key advantage of this method is that local variables provide a simple mapping between a name and a memory region. However, the syntax is somewhat limited compared to DSLs.
- After some experimenting with DSLs, it became clear that this method introduces too much overhead for memory access, which practically reduces the frequency of weak behaviors even in the most basic case of SB test to zero.
- Code generation can allow to transform user test definition in any way, which gives maximum flexibility. However, it is also a significantly more complicated method, so its use has to be justified. After comparing it to class extension, the only reasonable advantages we found were possibly easier bytecode extraction and possibly more efficient memory shuffling.

Given the relatively short time span of the project, we decided to not spend time on code generation and went with classes. To compensate for rigid syntax, we can still use some of the DSL methods for defining test settings. Here is how the same SB test from 1 is implemented for the new tool:

---

<sup>3</sup><https://kotlinlang.org/docs/type-safe-builders.html>

<sup>4</sup><https://kotlinlang.org/docs/ksp-overview.html>

```

class TestSB : BasicLitmusTest("store_buffering") {
    // shared variables
    var x = 0
    var y = 0
    // "local" output variables
    var o1 = 0
    var o2 = 0

    override fun actor1() {
        x = 1
        o1 = y
    }

    override fun actor2() {
        y = 1
        o2 = x
    }

    override fun arbiter() {
        outcome = o1 to o2
    }

    init {
        setupOutcomes {
            interesting = setOf(0 to 0)
        }
    }
}

```

Figure 2: SB test defined for the new tool

The exact syntax is still inspired by JCStress, even though we cannot copy its syntax precisely. The shared variables are simply class properties. The `actorN()` functions are the ones that run in parallel, named after JCStress `@Actor` annotation. The `arbiter()` function is run after all actors to collect results, if necessary. The `outcome` variable (also named after JCStress `@Outcome` annotation) is defined in the parent class and can be also assigned in the actor functions. Lastly, the `init{}` block can be used to setup additional test parameters, such as outcome types.

### 3.2 Test runner

The test runner is the key part of any litmus testing tool, responsible for invoking as many test behaviors as possible. As mentioned before, there are some techniques to make the runner more effective, any they have to be used by the new tool. Some of them involve manipulating threads on a low level, which requires some thread API. Kotlin/Native, however, does not provide such an option, due to decisions made long before a new common memory model was proposed. There are two options to overcome this problem:

- Calling `pthread` API via C interoperability feature of Kotlin. This approach provides

full control over thread parameters, but there is one thing to be concerned about. To start a pthread thread, we have to pass in the function of the thread, which would be a wrapper around user-defined code, with all the data it requires, including the memory that is going to be stressed. The problem is that we are testing Kotlin compiler memory model, and after passing all of that data directly to C, it is not clear if there are no side effects that may influence testing results.

- Using Kotlin/Native `Worker` class. These Workers are wrappers around platform threads with very limited functionality, representing something more of an executor interface rather than a thread. For example, its function is not allowed to capture any outside values, or there are no functions to set thread affinity, which is one of the techniques used for litmus testing. However, there is one experimental `platformThreadId` property of a Worker, that normally allows to use its underlying pthread handle. It can be used for calling pthread functions via C interoperability, allowing almost as much freedom as just using pthread API with no Workers.

We decided to start with Workers, which turned out to provide enough functionality. It is worth noting that any of these approaches refers only to Kotlin/Native, and is unavailable in Kotlin/JVM. There is no common thread API for both of these platforms. However, it is possible to run litmus tests on Java using JCIStress, while Kotlin/Native does not have any such option. Therefore, we will only focus on Kotlin/Native for the time being.

Based on that, a runner was implemented using the techniques discussed in 2.2. There are some details worth mentioning:

- A custom barrier with actively waiting threads had to be implemented. It can be done using Kotlin/Native `AtomicInt`, but one concerning thing is that it has sequentially consistent semantics, which introduces additional memory fences, potentially reducing the probability of a weak behavior happening. This extra synchronization is relatively seldom though, and the net gain from any spin-waiting barrier is still very positive. However, this barrier can also be implemented via C interop, which allows to use C11 `_Atomic` variables with custom memory ordering. Using these atomics would introduce less fences than the former method, but at the same time, using the C interop mechanism may introduce its own overhead. In the end, both of these approaches ended up being implemented and moved to running parameters rather than being hardcoded into the runner.
- Setting thread affinity can be done in two ways: random assignment or manually predefined affinity maps. Concrete affinity maps can ensure that certain arrangements of threads on CPU cores are tested, and it also allows to assign a set of cores to a thread instead of one core. The disadvantage is that this approach does not work well with running tests in parallel, because any core can be already assigned to a thread, and parallel tests can interfere with each other by invoking extra thread switches and raising a lot of memory fences. Meanwhile, random affinity assignment may not try certain core arrangements, but it does allow for parallel running. Like with barriers, both approaches were implemented and left as parameters.
- There is one more problem with setting thread affinity which comes from MacOS. The part of pthread API responsible for setting thread affinity is not available there. Instead, there is a custom API,<sup>5</sup> which is not only different but also lacks some

---

<sup>5</sup><https://developer.apple.com/library/archive/releasenotes/Performance/RN-AffinityAPI/index.html>

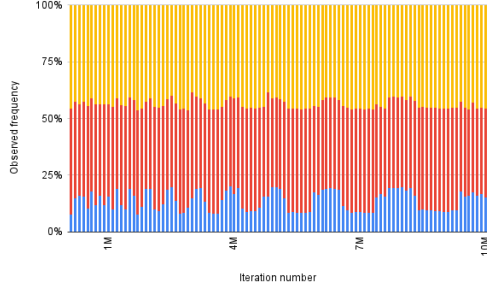
features compared to pthread, such as explicitly binding a thread to a specific core. And even then, it requires to set thread affinity before it starts, while the Worker API only allows to access the thread handle after the thread is started. Therefore, it is not possible to support affinity setting on MacOS with the current tool. One way to resolve it in the future may be to modify the Worker internal code in order to get access to thread handle before it starts. Note that this is not compatible with pthread API, which only generates the thread handle upon calling `pthread_create()`, also immediately launching the thread.

- False sharing is usually prevented by shuffling the locations of stored variables across a pre-allocated memory chunk. In our case this is problematic because of class variables being allocated by the compiler and not manually. The test interface could be extended to include access to an array, but this would clash with defining variables normally and make the interface less user-friendly. Another way to override variable allocation is to provide a property delegate (a Kotlin feature). This approach was implemented a couple of times in different ways, and while it does solve the false sharing problem, it introduces a lot of overhead in the form of unnecessary function calls, which in practice leads to large overhead and results in poor efficiency. The only sensible way to implement decent memory shuffling is most likely via code generation, which would radically complicate the tool, so the memory shuffling functionality was left for future work.
- As mentioned in [2.2](#), the number of parameters available to the user can be not just significant, but even overwhelming: for instance, if the user is allowed to set custom affinity maps, just their number alone can be on the order of millions. It is neither possible to iterate through all combinations of them, nor for the tool user to comprehend all reasonable combinations. Therefore, it is useful for the user to have some predefined parameter sets. Some of these sets were implemented for the new tool, based on various observations during the development and common sense.

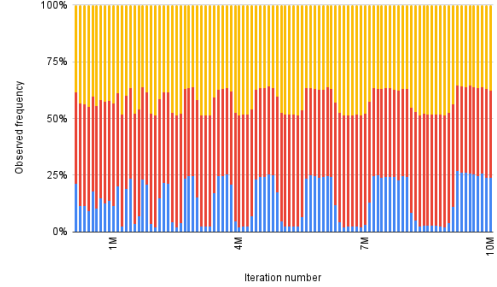
## 4 Tool evaluation

Assessing the performance of any litmus testing tool is quite challenging. The conceptual reason for that is that such a tool is intended for finding unexpected behaviors. To be more specific, one might assume that the frequency of certain behaviors can be used as a metric for assessing performance. However, the purpose of this tool is not to maximize the occurrences of some particular behaviors, but to maximize the number of different seen behaviors. This is a classic problem of quality versus quantity. Also, there is no connection between achieving high frequency of weak behavior for one test on one machine and for another test on another machine. The best parameters for one case may be significantly worse for another. Finally, as mentioned previously, if a behavior appears at least 3 times in a large enough run, it will very likely reproduce in subsequent runs. Hence, increasing its frequency basically does not help ensure the consistent occurrence of the same behavior. So overall, the raw frequency of any particular behaviors happening cannot be used as a decent evaluation metric.

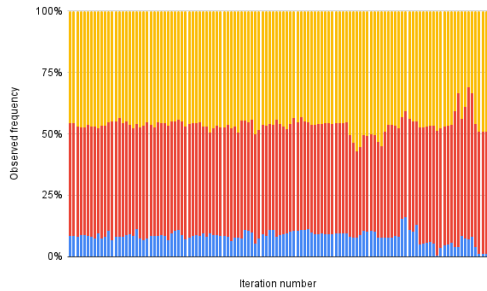
Speaking of frequencies, what we can do is look at how reproducible the test results are. If the frequencies are consistent between different runs, then we can gain confidence that there are no unaccounted parameters. The only test where measuring frequency is reasonable is the SB test, simply because all other tests that produce weak behaviors only do so extremely rarely. The results of this measurement can be seen in figure 3. The bars in the charts represent the frequency of all behaviors in the corresponding 100K iterations. Note that SB has four behaviors, but the fourth one can appear 3 orders of magnitude rarer, to the point where it is not visible on the charts.



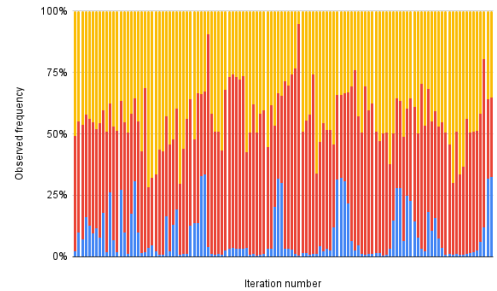
(a) x86, normal run, Var=0.19%



(b) Arm, normal run, Var=0.93%



(c) x86, parallel run, Var=0.06%



(d) Arm, parallel run, Var=0.97%

Figure 3: SB test, frequencies of observed behaviors vs. iteration number. Outcome color-coding: (0, 0) is blue (weak behavior), (0, 1) is red, (1, 0) is yellow, (1, 1) is too small to see. Point resolution of 100K iterations.

Visually it is clear that the stability of frequencies highly depends on the environment and on test parameters. If we suppose these distributions should be constant, then variance is a natural metric to quantitatively describe the stability of this distribution. Out of these four datasets, the most stable is the parallel x86 run with a variance of 0.06%, while the clearly least stable is the parallel Arm run with a variance of 0.97%. What this analysis tells us is not only that some environments are more stable than others, but also that due to uncontrollable environment changes the results of running a litmus test will not necessarily be quantitatively same. The best we can do to mitigate these effects is to run the test long enough for these perturbations to cancel out.

Another test we can do to more precisely check the stability of frequencies is to use a statistical test such as chi-squared test. These tests interpret data as some distribution of a random variable, and are widely used for validating these distributions in various ways. For instance, the chi-squared test can be used to check whether two independent data sets have the same distribution. We can use this to try find such a test duration, so that the perturbations of the unstable environment average out to a stable frequency distribution. This check was performed on SB test, but it failed to reach a stable distribution (in the sense that the longest tested duration still was not enough). The probable reason for that is poor performance of chi-squared test specifically when some of the frequencies are significantly lower than others, which is the case with SB. As mentioned previously, it makes no sense to try tests other than SB, so this chi-squared test gave no interpretable results. In the future, instead of the chi-squared test it is possible to try the Fisher exact test, which has its own shortcomings though.

Measuring quantities does not help, so we turn our attention to qualitative measurements. Since we want to ensure that the tool finds as many behaviors as possible, we can try running a set of tests and making sure that all the expected behaviors are found. The problem with this approach is that normally we don't know which behaviors are to be expected. It can be illustrated with another classic litmus test — load buffering, or LB for short.

$$\begin{array}{l} x, y = 0; \\ a = x; \parallel b = y; \\ y = 1; \parallel x = 1; \end{array}$$

Figure 4: Load buffering litmus test

The outcome of this test is given by the state of  $a$  and  $b$ . Three outcomes are straightforward, and the fourth  $(1, 1)$  outcome is weak. The problem though is that this fourth outcome is very rare to observe, to the point where it does not seem to appear at all. For instance, on an x86 machine, neither JCStress nor litmus7 can quickly confirm this weak behavior. Naturally, the new tool cannot spot it either, but it would be wrong to conclude this is due to some flaw. This idea is further supported by the message passing test (discussed later in details), which does not appear in its classic form, but suddenly does appear when a redundant `if` is inserted into it. This implies that it is not the tool being unable to detect some behavior, but rather the compiler doing something intricate and possibly prohibiting this behavior at all.

One further method of evaluating the new tool can be to directly compare it to existing tools like JCStress or litmus7. We can run some tests using each of these tools and check if they agree on produces behaviors. The problem with this approach is that the results in

fact do not have to agree on the behaviors, because each of these tools operates under its own memory model. We can expect some behaviors from the current Kotlin memory model, but once again, we do not know them for sure. So, since this comparison is not exactly representative, for now it has been left as future work.

To conclude, the evaluation of the tool showed ambiguous results. In some circumstances, the results visually seem to be relatively stable, which was seen in cumulative charts in figure 3. However, no method showed definitive evidence of tool reliability. Abstractly speaking, this is due to the nature of the context: the behaviors are not guaranteed to appear, different memory models induce different results, unstable environments render statistical tests less useful than they normally are. So in the end, the final test we can perform on our new tool is to simply use it as intended. If we are able to find some unexpected behaviors of the compiled programs with its help, then it can be declared that the tool achieved its goal.

## 5 Testing the compiler

First, a suite of litmus tests checking various aspects of Kotlin memory model had to be prepared. These tests were picked with the help of our industry advisor based on common existing litmus tests,<sup>6</sup> and then translated into a format to be used with the new tool, similarly to figure 2. Then, these tests were run on two different machines: one with a 6-core x86 AMD processor and one with an 8-core M1 processor in Arm mode. It is crucial to run these tests on multiple hardware and software platforms, since some tests only demonstrate weak behaviors on specific platforms. The version of the latest Kotlin compiler at the time is 1.8.21.

<b>Test name</b>	<b>Arm</b>	<b>x86</b>	<b>Expected</b>
ATOM	-	-	-
SB	+	+	+
SB+Vol	-	-	-
MUTEX	-	-	-
SB+Lock	-	-	-
MP	-	+	+
MP+Vol	-	-	-
MP+Lock	-	-	-
MP+DRF	-	-	-
CoRR	-	-	+
CoRR+CSE	-	-	+
IRIW	-	-	+
IRIW+Vol	-	-	-
UPUB	+	-	-
UPUB+Ctor	+	-	-
OOTA	-	-	-
LB	-	-	+
LB+FakeDEPS	-	-	-
LB+Vol	-	-	-

Table 1: Results of testing Kotlin/Native compiler on two machines. "Arm" and "x86" are machine CPU architectures. Plus symbol means this test exhibited some weak behavior. Deviations from expectation are marked with color: allowed deviations are green, invalid deviations are red.

The results of running the tests are presented in Table 1. Firstly, we see that most of the tests match our expectations, which means that the current implementation of the Kotlin/Native compiler is quite robust. We can also see here that some tests only give weak outcomes on one platform and not another. There is also a number of tests that were expected to behave weakly, but in reality did not, which is perfectly acceptable. Finally, some tests showed weak behavior even though they were not supposed to. We will now examine the interesting discrepancies in detail.

One interesting test is message passing, or MP, shown in figure 5. As it can be seen, it appeared on a stronger x86 architecture, without appearing on weaker Arm. One thing about this test is it never showed all four possible behaviors at the same time. It shows

<sup>6</sup>The detailed list can be found at: <https://github.com/JetBrains-Research/KotlinMemoryModelResearch/blob/master/litmus/litmus.md#guarantees-and-litmus-tests>

<pre> x, y = 0; x = 1;    a = y; y = 1;    b = x; </pre>	<pre> x = 0; volatile y = 0; x = 1;    if(y != 0) { y = 1;        b = x;               } </pre>	<pre> x, y = 0; a = y; x = 1;    if(a != 0) { y = 1;        b = x;               } </pre>
(a) MP	(b) MP+DRF	(c) MP+NoDRF

Figure 5: Message passing test and its variants

three behaviors when compiled in debug mode on x86, and also three behaviors when compiled in optimized mode on x86, except this time with the weak behavior. It appears that this test does not truly show weak behavior on hardware level, otherwise all four behaviors would be likely to show in the same run. This behavior can be easily explained, however, if the compiler makes a reordering of operations in either thread. This is a clear demonstration of how memory models of different levels can interact with each other.

There is one more interesting thing about this MP test. Figure 5 also shows two variants of MP. The MP+DRF is a classic test from our suite, and it behaves as expected. DRF stands here for data-race-freedom principle, which requires programs with no data races to be sequentially consistent. The interesting part is the third test, which was discovered in a different context and then reshaped to this form. Since it looks like the DRF test, short of the `volatile` modifier and hence with a data race, it was nicknamed NoDRF. While the other two showed no *true* weak behavior, this one did: it can end with `a==1 && b==0`. This tells us that the tool is in fact able to catch the weak behavior of MP test, and yet it does not. This effect is probably caused by some quirk regarding the `if` statement, and it might also be worth looking into for the compiler developers.

<pre> class Holder { x: Int } h: Holder = null; t = h; h = new Holder();    if (t != null) {                            outcome = t.x;                        } </pre>	<pre> class Holder { x: Int = 1; } h: Holder = null; t = h; h = new Holder();    if (t != null) {                            outcome = t.x;                        } </pre>
(a) UPUB	(b) UPUB+Ctor

Figure 6: Unsafe publication tests

Another two tests require additional attention. The UPUB and UPUB+Ctor tests demonstrated in figure 6 check which values can be read from class fields during the process of class instantiation and publication. As Kotlin is a memory-safe programming language, it must ensure that no "garbage" value can be read during this process. This follows the JMM principle as well. However, the tool has found that "garbage" values can in fact be read. This behavior is quite unexpected and should not be allowed for two reasons. One is that Kotlin is a memory-safe language, and reading uninitialized values should not be allowed. Another is that the common memory model cannot violate JMM in order to be common by definition.

Upon closer inspection, we find apparently random values being read. The explanation for this behavior is likely the following: publishing a new object requires writing to several variables. One is the reference to the object, other are filling its fields with default values.

outcome		type		count		frequency
null		ACCEPTED		26767684		65.607%
0		ACCEPTED		14032239		34.392%
71817408		FORBIDDEN		12		<0.001%
-1493360416		FORBIDDEN		1		<0.001%
-1560102576		FORBIDDEN		1		<0.001%
5830112		FORBIDDEN		1		<0.001%
21281392		FORBIDDEN		1		<0.001%
-1115771136		FORBIDDEN		1		<0.001%
...						

Figure 7: Sample tool output after running UPUB test

If no special restrictions are imposed on these writes, they can be reordered. When that happens, other threads can see the reference to the object, but its fields are not yet initialized, which results in "garbage" values being read. The fix is pretty straightforward: the compiler needs to add a release fence between these writes.

## 6 Conclusions and future work

In this work, a new litmus testing tool designed for testing Kotlin/Native compiler was implemented. It incorporates the common techniques from existing litmus testing tools. The tool was then evaluated to make sure that any results would be reproducible. Then it was used to run a number of tests against the latest version of Kotlin/Native compiler. Several unexpected behaviors were found, some of them are practically bugs in the compiler. The main goal of testing Kotlin compiler memory model has been achieved.

That said, there is more testing to be done: more machine architectures, more different tests that cover other aspects of memory models, testing the Kotlin/JVM compiler. The latter is especially important to ensure that the common memory model is indeed common. Also, not all tests that were expected to show weak behaviors did so, and some more improvements like code generation can be made in order to maximize the chance of covering all behaviors. Finally, the tool is also currently lacking a proper user interface such as a command line interface.

The implementation of the tool and the used litmus test suite can be found at <https://github.com/DLochmelis33/komem-litmus>.

## References

- [1] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding cats: Modelling, simulation, testing, and data mining for weak memory”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014), pp. 1–74.
- [2] Jade Alglave et al. “Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 405–418.
- [3] Jade Alglave et al. “GPU concurrency: Weak behaviours and programming assumptions”. In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 577–591.
- [4] Jade Alglave et al. “Litmus: Running Tests against Hardware.” In: *TACAS*. Vol. 11. Springer. 2011, pp. 41–44.
- [5] Michael Emmi and Constantin Enea. “Violat: generating tests of observational refinement for concurrent objects”. In: *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II* 31. Springer. 2019, pp. 534–546.
- [6] James Gosling et al. *The Java language specification*. Addison-Wesley Professional, 2000.
- [7] *herdtools7*. <https://github.com/herd/herdtools7>.
- [8] *Java Concurrency Stress (jckstress)*. <https://github.com/openjdk/jckstress>.
- [9] Jake Kirkham et al. “Foundations of Empirical Memory Consistency Testing”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: [10.1145/3428294](https://doi.org/10.1145/3428294). URL: <https://doi.org/10.1145/3428294>.
- [10] Nikita Koval et al. “Testing concurrency on the JVM with lincheck”. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2020, pp. 423–424.
- [11] Leslie Lamport. “How to make a multiprocessor computer that correctly executes multiprocess programs”. In: *IEEE Transactions on Computers* c-28 9 (1979), pp. 690–691.
- [12] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.
- [13] Lun Liu, Todd Millstein, and Madanlal Musuvathi. “A volatile-by-default jvm for server applications”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–25.
- [14] Lun Liu, Todd Millstein, and Madanlal Musuvathi. “Accelerating sequential consistency for Java with speculative compilation”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 16–30.
- [15] Jeremy Manson, William Pugh, and Sarita V Adve. “The Java memory model”. In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 378–391.
- [16] Christopher Pulte et al. “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–29.

- [17] Susmit Sarkar et al. “Understanding POWER multiprocessors”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 175–186.
- [18] Aleksey Shipilëv. “JCStress Workshop”. Hydra conference. 2021. URL: [\url{https://shipilev.net/talks/hydraconf-June2021-jcstress-workshop.pdf}](https://shipilev.net/talks/hydraconf-June2021-jcstress-workshop.pdf).
- [19] Abhayendra Singh et al. “End-to-end sequential consistency”. In: *ACM SIGARCH Computer Architecture News* 40.3 (2012), pp. 524–535.
- [20] Matt Windsor, Alastair F Donaldson, and John Wickerson. “C4: the C compiler concurrency checker”. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 670–673.
- [21] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.