Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM

Nikita Koval[1](✉), Alexander Fedorov[1,2], Maria Sokolova[1], Dmitry Tsitelov[3], and Dan Alistarh[2]

[1] JetBrains, Prague, Czech Republic
ndkoval@ya.ru
[2] IST Austria, Klosterneuburg, Austria
[3] Devexperts, Munich, Germany

Abstract. This paper presents `Lincheck`, a new practical and user-friendly framework for testing concurrent algorithms on the Java Virtual Machine (JVM). `Lincheck` provides a simple and declarative way to write concurrent tests: instead of describing *how* to perform the test, users specify *what to test* by declaring all the operations to examine; the framework automatically handles the rest. As a result, tests written with `Lincheck` are concise and easy to understand. The framework automatically generates a set of concurrent scenarios, examines them using stress-testing or bounded model checking, and verifies that the results of each invocation are correct. Notably, if an error is detected via model checking, `Lincheck` provides an easy-to-follow trace to reproduce it, significantly simplifying the bug investigation.

To the best of our knowledge, `Lincheck` is the first production-ready tool on the JVM that offers such a simple way of writing concurrent tests, without requiring special skills or expertise. We successfully integrated `Lincheck` in the development process of several large projects, such as Kotlin Coroutines, and identified new bugs in popular concurrency libraries, such as a race in Java's standard `ConcurrentLinkedDeque` and a liveliness bug in Java's `AbstractQueuedSynchronizer` framework, which is used in most of the synchronization primitives. We believe that `Lincheck` can significantly improve the quality and productivity of concurrent algorithms research and development and become the state-of-the-art tool for checking their correctness.

1 Introduction

Concurrent programming is known to be notoriously hard and error-prone. Writing a good and robust test for a concurrent data structure may be even more challenging than implementing it. Programmers produce many such stress tests every day, but they often are nondeterministic, cover only specific cases, and do not catch all the bugs. Both the industry and academia need a tool that would simplify writing reliable tests for concurrent data structures.

In this paper, we present `Lincheck` [1], a new practical framework for JVM-based languages (such as Java, Kotlin, and Scala), which simplifies writing reliable concurrent tests. While most existing tools require writing the algorithm in a special language [2], specifying all possible concurrent scenarios and their outcomes [3–6], or learning a large amount of theory [7,8], `Lincheck` provides a more pragmatic *declarative* approach. It requires users only to list the data structure operations, thus, specifying *what to test* instead of *how*. Taking these operations, `Lincheck` generates a set of concurrent scenarios and examines them via stress testing or model checking, verifying that the outcome results are correct. The default correctness property is linearizability [9], but various relaxations [10–12] are also supported. One may think of `Lincheck` as a mix of a fuzzer (that generates concurrent scenarios) and a model checker or stress runner (which examines these scenarios) equipped with an automatic outcome verifier.

Lincheck by Example. The "classic" way to write a concurrent test is to manually run parallel threads, invoking the data structure operations in them and checking that some sequential history can explain the produced results. Such tests typically contain hundreds of lines of boilerplate code and cover only easy-to-verify scenarios. `Lincheck` automates the machinery, making tests short and declarative. To illustrate that, we present a test for the `ConcurrentLinkedDeque` collection (double-ended queue, which supports insertions and removals at both ends) of the standard Java library in Listing 1.

The initial state of the testing data structure is specified in the constructor; here, we simply create a new empty deque at line 2. The following lines 4–9 declare the deque operations; they should be annotated with `@Operation`. Finally, we run the analysis by invoking `ModelCheckingOptions.check(..)` on the testing class at line 11. Replacing `ModelCheckingOptions` with `StressOptions` switches to stress testing, which essentially runs parallel threads.

```
 1 class DequeTest {
 2  val deque = ConcurrentLinkedDeque<Int>()
 3
 4  @Operation fun addFirst(e: Int) = deque.addFirst(e)
 5  @Operation fun addLast(e: Int)  = deque.addLast(e)
 6  @Operation fun pollFirst()      = deque.pollFirst()
 7  @Operation fun pollLast()       = deque.pollLast()
 8  @Operation fun peekFirst()      = deque.peekFirst()
 9  @Operation fun peekLast()       = deque.peekLast()
10
11  @Test fun runTest() = ModelCheckingOptions()
12                          .check(this::class)
13 }
```

Listing 1. Concurrent test via Lincheck for Java's `ConcurrentLinkedDeque`. The code is written in Kotlin; `import` statements are omitted.

After executing the test, we get an error presented in Fig. 1. Surprisingly, this class from the standard Java library has a bug; the error was originally detected via Lincheck by the authors [13] (notably, there were several unsuc-

```
= Invalid execution results =                        Comment: this text is a Lincheck output,
| addLast(-6)       | addFirst(-8)    |             while the scheme is drawn by the authors
| peekFirst(): -8   | pollLast(): -8  |

= The following interleaving leads to the error =
|                   | addFirst(-8)                                          |
|                   | pollLast()                                            |
|                   |   pollLast(): -8 at DequeTest.pollLast(DequeTest.kt:35) |
|                   |     last(): Node@1 at CLD.pollLast(CLD.java:936)      |
|                   |     item.READ: null at CLD.pollLast(CLD.java:938)     |
|                   |     prev.READ: Node@2 at CLD.pollLast(CLD.java:946)   |
|                   |     item.READ: -8 at CLD.pollLast(CLD.java:938)       |
|                   |     next.READ: null at CLD.pollLast(CLD.java:940)     |
|                   |     switch                                            |
| addLast(-6)       |                                                       |
| peekFirst(): -8   |                                                       |
|                   |     item.CAS(-8,null): true at CLD.pollLast(CLD.java:941) |
|                   |     unlink(Node@2) at CLD.pollLast(CLD.java:942)      |
|                   |   result: -8                                          |
```

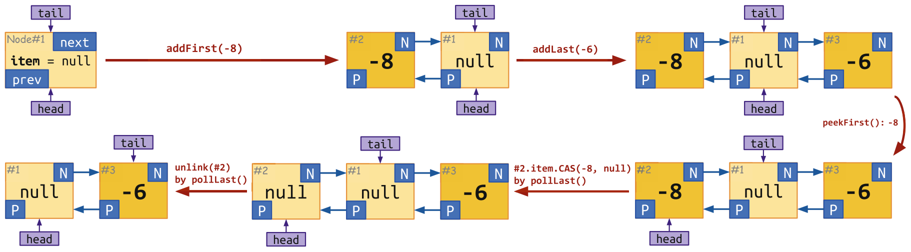

Fig. 1. The incorrect execution of the Java's `ConcurrentLinkedDeque` identified by the `Lincheck` test from Listing 1 and illustrated by a pictured diagram. To narrow the test output, `ConcurrentLinkedDeque` is replaced with `CLD`.

cessful attempts to fix the incorrectness before that [14,15]). Obviously, the produced results are non-linearizable: for `pollLast()` in the second thread to return `-8`, it should be called before `addLast(-6)` in the first thread; however, that would require the following `peekFirst()` to return `-6` instead of `-8`. While `Lincheck` always prints a failing scenario with incorrect results (if found), the model checker also provides a detailed *interleaving trace* that reproduces the error.

Providing a detailed and informative trace is a game-changer. With it, we can easily understand why `ConcurrentLinkedDeque` is incorrect. The underlying data structure forms a doubly-linked list, with `head` and `tail` pointers approximating its first and last nodes. Initially, `head` and `tail` point to a logically removed `(Node.item == null)` node. After `addFirst(-8)` in the second thread is applied, a new node is added to the beginning; `head` and `tail` remain unchanged. Then, `pollLast()` starts; it finds the last non-empty node (the previously added one) and gets preempted before extracting the element. (The procedure linearizes on changing the `Node.item` value to `null` via atomic `Compare-and-Set (CAS)` instruction.) After invoking `addLast(-6)` in the first thread, a new node is added to the end of the list. The following `peekFirst()` does not change the data structure logically but advances the `head` pointer.

Finally, the execution switches back to the second thread. The `pollLast()` operation successfully removes the node containing `-8` (which is no longer the last element), extracting the item via `CAS` followed by unlinking the node physically. These twelve lines of straightforward code easily find a bug in the standard library of Java and provide a detailed trace that leads to the error, reducing the investigation time from hours to minutes. We also believe that with such an instrument as `Lincheck`, the bug would not have been released in the first place.

Practical-Oriented Design. `Lincheck` was designed as a tool for testing real-world concurrent code. The following its properties are crucial in practice:

- **Declarative testing.** `Lincheck` takes only a list of operations and optional configuration parameters (we discuss them further), which results in short and intuitive tests — no need to learn a new language or technology.
- **No implementation restrictions.** `Lincheck` can test any real-world implementations, including those that utilize low-level JVM constructs like `Unsafe` or `VarHandle`, without imposing any restrictions.
- **No false positives.** `Lincheck` reports only reproducible errors, which is vital for using the framework in continuous integration (CI/CD) and unit tests.
- **User-friendliness.** `Lincheck` streamlines bug investigation by providing a thorough trace of the discovered error, saving programmers countless hours.
- **Flexibility.** `Lincheck` supports popular constraints, such as the single-producer/consumer workload, as well as a range of linearizability relaxations, enabling custom scenario generation and verification when necessary.

Real-World Applications. We have successfully integrated `Lincheck` in the development processes of Kotlin Coroutines [16] and JCTools [17] libraries, enabling reliable testing of their core data structures, which are often complex and several thousand lines of code long. `Lincheck`'s support of popular workload constraints and linearizability relaxations and its ability to handle blocking operations, such as those of `Mutex` and `Channel`, were crucial for these tests. Furthermore, for over five years, we have successfully used `Lincheck` in our "Parallel Programming" course to automate the verification of more than 4K student solutions annually.

We have also detected several new bugs [18] in popular libraries, including the previously discussed race in Java's `ConcurrentLinkedDeque` [13], non-linearizabi-lity of `NonBlockingHashMapLong` from JCTools [19], and liveness bugs in Java's `AbstractQueuedSynchronizer` [18] and `Mutex` in Kotlin Coroutines [20].

In conclusion, `Lincheck` is a powerful and versatile tool for testing complex concurrent programs. It provides non-trivial features in terms of generality, ease of use, and performance. We provide a comprehensive overview of `Lincheck` in the rest of the paper and believe that it will greatly save time and (mental) energy tracking down concurrency bugs.

2 Lincheck Overview

We now dive into `Lincheck` internals, presenting its key features as we go along. The testing process can be broken down into three stages, as depicted in the diagram below. `Lincheck` generates a set of concurrent scenarios and examines them via either model checking or stress testing, verifying that each scenario invocation results satisfy the desirable correctness property (linearizability [9] by default). If the outcome is incorrect, the invocation hangs, or the code throws an unexpected exception, the test fails with an error similar to the one in Fig. 1.

Minimizing Failing Scenarios. When an error is detected, it is often possible to reproduce it with fewer threads and operations [21]. `Lincheck` automatically "minimizes" the failing scenario in a greedy way: it repeatedly removes an operation from the scenario until the test stops failing, thus finding a *minimal* failing scenario. While this approach is not theoretically-optimal, we found it working well in practice[1].

User Guide. This section focuses mainly on the technical aspects behind the `Lincheck` features. For those readers who are interested in using the framework in their project, we suggest taking a look at the official `Lincheck` guide [22].

2.1 Phase 1: Scenario Generation

`Lincheck` allows to tune the number of parallel threads, operations in them, and the number of scenarios to be generated when creating `ModelCheckingOptions` or `StressOptions`. The framework then generates a set of concurrent scenarios by filling threads with randomly picked operations (annotated with `@Operation`) and generating (by default random) arguments for these operations.

Operation Arguments. Consider testing a concurrent hash table. If it has a bug, it is more likely to be detected when accessing the same element concurrently. To increase the probability of such scenarios, users can narrow the range of possible elements passed to the operations; Listing 2 illustrates how to configure the test in a way so the generated elements are always between `1` and `3`.

```
@Param(name = "elem", gen = IntGen::class, conf = "1:3")
@OpGroupConfig(name="writer", nonParallel=true)
class SingleWriterHashSetTest {
 val s = SingleWriterHashSet<Int>()

 @Operation(group = "writer"
) // never executes concurrently
 fun add(@Param(name = "elem") e: Int) = s.add(e)
 @Operation
 fun contains(@Param(name = "elem")
 e: Int) = s.contains(e)
```

[1] Finding the *minimum* failing scenario is a highly complex problem, as it could be not based on any of the generated scenarios.

```
  @Operation(group = "writer"
) // never executes concurrently
  fun remove(@Param(name = "elem")  e: Int) = s.remove(e)

  @Test fun runTest() = ModelCheckingOptions()
                           .check(this::class)
}
```

Listing 2. Testing single-writer set with custom argument generation (highlighted with yellow) and single-writer workload constraint (highlighted with red).

Workload Constraints. Some data structures may require a part of operations not to be executed concurrently, such as single-producer/consumer queues. `Lincheck` provides out-of-the-box support for such constraints, generating scenarios accordingly. The framework API requires grouping such operations and restricting their parallelism; Listing 2 illustrates how to test a single-writer set.

2.2 Phase 2: Scenario Running

`Lincheck` uses stress testing and model checking to examine generated scenarios. The stress testing mode was influenced by JCStress [3], but `Lincheck` automatically generates scenarios and verifies outcomes, while JCStress requires listing both scenarios and correct results manually. The main issue with stress testing is the complexity of analysing a bug after detecting it. To mitigate this, `Lincheck` supports bounded model checking, providing detailed traces that reproduce bugs, similar to the one in Fig. 1. The rest of the subsection focuses on the model-checking approach, discussing the most significant details.

Bounded Model Checker. The model-checking mode has drawn inspiration from the CHESS (also known as Line-Up) framework for C# [5]. It assumes the sequentially consistent memory model and evaluates all possible schedules with a limited number of context switches. Unlike CHESS, `Lincheck` bounds the number of schedules rather than context switches, which makes testing time independent of scenario size and algorithm complexity.

In some cases, the specified number of schedules may not be enough to explore all interleavings, so `Lincheck` studies them evenly, probing logically different scenarios first. For instance, imagine a case where Lincheck is analyzing interleavings with a single context switch and has previously explored only one interleaving, which originated from the first thread containing four atomic operations. Under these circumstances, Lincheck presumes that 25% of the interleavings have been explored when starting from the first thread, while the second thread remains unexplored. As a result, Lincheck becomes more inclined to select the second thread as the starting point for the next exploration.

Switch Points. To control the execution, `Lincheck` inserts internal method calls at shared memory accesses by on-the-fly byte-code transformation via ASM framework [23]. These internal methods serve as *switch points*, enabling manual context switching. Notably, `Lincheck` supports shared memory access through

`AtomicFieldUpdater`, `VarHandle`, and `Unsafe` and handles built-in synchronization via `MONITORENTER/MONITOREXIT`, `park/unpark`, and `wait/notify`. Internally, it replaces there synchronization primitives with custom implementations, thus, enabling full control of the execution.

Progress Guarantees. While exploring potential switch points, `Lincheck` can detect active synchronization, handling it similarly to locks. This capability to detect blocking code enables `Lincheck` to verify the testing algorithm for *obstruction-freedom*[2], the weakest non-blocking guarantee [10]. Although more popular lock- and wait-freedom are part of `Lincheck`'s future plans, the majority of practical liveness bugs are caused by unexpected blocking code, making the obstruction-freedom check fairly useful for lock-free and wait-free algorithms.

Optimizations. `Lincheck` uses various heuristics to speed up the analysis and increase the coverage. The most impactful one excludes `final` field accesses from the analysis, as their values are unchanging. Our internal experiments indicate a reduction in the number of inserted switch points by over $\times 2$ in real-world code. Another important optimization tracks objects that are not shared with other threads, excluding accesses to them from the analysis. This heuristic eliminates an additional 10–15% of switch points in practice.

Happens-Before. When an operation starts, `Lincheck` collects which operations from other threads are already completed to establish the "happens-before" relation; this information is further passed to the results verifier.

Modular Testing. When constructing new algorithms, it is common to use existing non-trivial data structures as building blocks. Considering such underlying data structures to be correct and treating their operations as atomic may significantly reduce the number of possible interleavings and check only meaningful ones, thus increasing the testing quality. `Lincheck` makes it possible with the modular testing feature; please read the official guide for details [22].

Limitations. For the model checking mode, the testing data structure must be deterministic to ensure reproducible executions, which is a common requirement for bug reproducing tools [24]. For the algorithms that utilize randomization, `Lincheck` offers out-of-the-box support by fixing seeds for `Random`; thus, making the latter deterministic. To our experience, `Random` is the only source of nondeterminism in practical concurrent algorithms.

Model Checking vs Stress Testing. The primary benefit of using model checking is obtaining a comprehensive trace reproducing the detected error, as demonstrated in Fig. 1. However, the current implementation assumes the sequentially consistent memory model, which can result in missed bugs caused by low-level effects, such as an omitted `volatile` modifier in Java. We are in the process of incorporating the GenMC algorithm [6,25] to support weak memory models and increase analysis coverage through the partial order reduction

[2] The *obstruction-freedom* property ensures that any operation completes within a limited number of steps if all other threads are stopped.

technique. In the meantime, we suggest using stress testing in addition to model checking.

2.3 Phase 3: Verification of Outcome Results

Once the scenario is executed, the operation results should be verified against the specified correctness property, which is linearizability [9] by default. In brief, `Lincheck` tries to match the operation results to a sequential history that preserves the order of operations in threads and does not violate the “happens-before” relation established during the execution.

LTS. Instead of generating all possible sequential executions, `Lincheck` lazily builds a *labeled transition system (LTS)* [26] and tries to explain the obtained results using it. Roughly, LTS is a directed graph, which nodes represent the data structure states, while edges specify the transitions and are labeled with operations and their results. Execution results are considered valid if there exists a finite path in the LTS (i.e., sequential history) that leads to the same results. `Lincheck` lazily builds LTS by invoking operations on the testing data structure in one thread. Thus, the sequential behavior is specified implicitly. Figure 2 illustrates an LTS lazily constructed by `Lincheck` for verifying incorrect results of `ConcurrentLinkedDeque` from Fig. 1.

Sequential Specification. By default, `Lincheck` sequentially manipulates the testing data structure to build an LTS. It is possible to specify the sequential behavior explicitly, providing a separate class with the same methods as those annotated with `@Operation`. It allows for a single `Lincheck` test instead of separate sequential and concurrent ones. For API details, please refer to the guide [22].

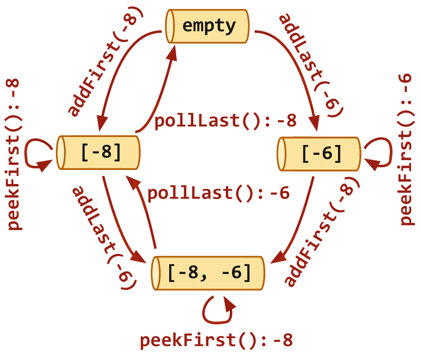

Fig. 2. An LTS constructed for verifying `ConcurrentLinkedDeque` results from Fig. 1.

Validation Functions. It is possible to validate the data structure invariants at the end of the test, adding the corresponding function and annotating it with `@Validate`. For example, we have uncovered a memory leak in the algorithm for removing nodes from a concurrent linked list in [27] by validating that logically removed nodes are unreachable at the end.

Linearizability Relaxations. Additionally to linearizability, `Lincheck` supports various relaxations, such as quiescent consistency [10], quantitative relaxation [11], and quasi-linearizability [12].

Blocking Operations. Some structures are blocking by design, such as the case of `Mutex` or `Channel`. Consider a *rendezvous channel*, also known as “synchronous queue", as an example: senders and receivers perform a rendezvous handshake as a part of their protocol (senders wait for receivers and vice versa). If we run `send(e)` and `receive()` in parallel, they both succeed. However, executing the operations sequentially will result in suspending the first one. To reason about correctness, the *dual data structures* formalism [28] is usually used. Essentially, it splits each operation into two parts at the point of suspension, linearizing these parts separately. We extend this formalism by allowing suspended requests to cancel and by making it more efficient for verification.

3 Evaluation

`Lincheck` has already gained adoption in Kotlin and Java communities, as well as by companies and universities. It has been integrated into the development processes of Kotlin Coroutines [16] and JCTools [17], enabling reliable testing of their core data structures, and was used to find several new bugs in popular concurrency libraries and algorithms published at top-tier conferences. Furthermore, for over five years, we have successfully used `Lincheck` in our "Parallel Programming" course to automate the verification of more than 4K student solutions per year. Notably, many users appear to especially appreciate `Lincheck`'s low entry threshold and its ability to "explain" errors with detailed traces.

***Novel Bugs Discovered with* `Lincheck`.** We have uncovered multiple new concurrency bugs in popular libraries and authors' implementations of algorithms published at top conferences. These bugs are listed in Table 1 and include some found in the standard Java library. `Lincheck` not only detects non-linearizability and unexpected exception bugs, but also liveliness issues. For example, it identified an obstruction-freedom violation in Java's `AbstractQueuedSynchronizer` framework, which is a foundation for building most synchronization primitives in the standard Java library.

Notably, the tests that uncover the bugs listed in Table tab1 are **publicly available** [18], allowing readers to easily reproduce these bugs.

Running Time Analysis. We have designed `Lincheck` for daily use and expect it to be fast enough in interactive mode. Various factors, including the complexity of the testing algorithm and the number of threads, operations, and invocations, can impact its performance. We suggest using two configurations for the best user experience and robustness: a *fast* configuration for local builds to catch simple bugs quickly and a *long* configuration to perform a more thorough analysis on CI/CD (Continuous Integration) servers:

- **Fast:** 30 scenarios of 2 threads × 3 operations, 1000 invocations per each;
- **Long:** 100 scenarios of 3 threads × 4 operations, 10000 invocations per each.

We assess the performance and reliability of `Lincheck` with these *fast* and *long* configurations by measuring the testing times and showing whether the expected bugs were detected. We run the experiment on the buggy algorithms listed in Table 1, along with `ConcurrentHashMap` and `ConcurrentLinkedQueue` from

Table 1. Novel bugs discovered with `Lincheck`; tests are publicly available [18].

Source	**Data structure**	**Description**
Java	`ConcurrentLinkedDeque`	Non-linearizable [13]; see Fig. 1
Java	`AbstractQueuedSynchronizer`	Liveliness error
Kotlin Coroutines [16]	`Mutex`	Liveliness error [20]
JCTools [17]	`NonBlockingHashMapLong`	Non-linearizable [19]
Concurrent-Trees [29]	`ConcurrentRadixTree`	Non-linearizable [30]
PPoPP'10 [31]	`SnapTree`	Unexpected internal exception
PPoPP'14 [32]	`LogicalOrderingAVL`[a]	Deadlock
ISPDC'15 [33]	`CATree`	Deadlock
Euro-Par'17 [34]	`ConcurrencyOptimalTree`	Unexpected internal exception

[a] The deadlock in the `LogicalOrderingAVL` algorithm was originally found by Trevor Brown and later confirmed with `Lincheck`.

Table 2. Running times of `Lincheck` tests with *fast* and *long* configurations using both stress testing and model checking (MC) for the listed data structures. Failed tests, which detect bugs, are highlighted with **red**. Notably, finding a bug may take longer than testing a correct implementation due to scenario minimization.

Data Structure	Fast Configuration		Long Configuration	
	Stress	MC	Stress	MC
`ConcurrentHashMap` (Java)	0.3 s	2.7 s	38.1 s	1 m 44 s
`ConcurrentLinkedQueue` (Java)	0.4 s	1.7 s	1 m 26 s	1 m 41 s
`LockFreeTaskQueue` (Kotlin Coroutines)	1.1 s	1.4 s	39.6 s	54.8 s
`Semaphore` (Kotlin Coroutines)	2.1 s	3.6 s	22.3 s	1 m 44 s
`ConcurrentLinkedDeque` (Java)	0.4 s	**1.2 s**	**19.7 s**	**10.7 s**
`AbstractQueueSynchronizer` (Java)	1.6 s	**0.5 s**	18.2 s	**8.6s**
`Mutex`(Kotlin Coroutines)	0.9 s	**2.6 s**	23.6 s	**8.7 s**
`NonBlockingHashMapLong` (JCTools)	**0.6 s**	**1.3 s**	**4.4 s**	**7 s**
`ConcurrentRadixTree` ([29])	2.9 s	10.6 s	40.9 s	**2 m 30 s**
`SnapTree` [31]	1.7 s	5.8 s	38.4 s	**5 m 6 s**
`LogicalOrderingAVL` [32]	1.5 s	4.2 s	17.1 s	**36.9 s**
`CATree` [33]	**20.1 s**	**0.8 s**	**41.3 s**	**6.5 s**
`ConcurrencyOptimalTree` [34]	**0.4 s**	**1.5 s**	**3 s**	**7.3 s**

the Java standard library and a quasi-linearizable `LockFreeTaskQueue` with `Semaphore` from Kotlin Coroutines. The results are available in Table 2. The experiment was conducted on a Xiaomi Mi Notebook Pro 2019 with Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz and 32 Gb RAM. The results show that the *fast* configuration ensures short running times, being suitable for use as unit tests without slowing down the build and able to uncover some bugs. However, some bugs are detected only with the *long* configuration, emphasizing the need for more operations and invocations to guarantee correctness. Despite this, the running time remains practical and acceptable.

4 Related Work

Several excellent tools for linearizability testing and model checking have been proposed, e.g. [4,5,35–41], and some even support relaxed memory models [6,25,42,43] and linearizability relaxations [36,44]. Due to space limitations, we focus our discussion on the works that shaped `Lincheck`.

Inspiration. `Lincheck` was originally inspired by the JCStress [3] tool for JVM, which is designed to test the memory model implementation. However, JCStress does not offer a declarative approach to writing tests. The bounded model checker in `Lincheck` was influenced by CHESS (Line-Up) [5] for C#, which is also non-declarative and does not support linearizability extensions. `Lincheck` offers several novel features and usability advantages compared to these inspirations, making it a versatile platform for research in testing and model checking. Although other tools such as GenMC [6,25,43] have superior features, `Lincheck` is designed to be extensible and can integrate new tools. In particular, we are working on incorporating the GenMC algorithm into `Lincheck` at the moment of writing this paper.

Lincheck Compared to Other Solutions. To the best of our knowledge, no other tool offers similar functionality. In particular, `Lincheck` allows certain operations to never execute in parallel (supporting single-producer/consumer constraints), detects obstruction-freedom violations (which is crucial for checking non-blocking algorithms), provides a way to specify sequential behavior explicitly (enabling oracle-based testing), and supports blocking operations for Kotlin Coroutines. Furthermore, `Lincheck` is a highly user-friendly framework, featuring a simple API and easy-to-understand output, which we have found users to highly appreciate.

5 Discussion

We introduced `Lincheck`, a versatile and expandable framework for testing concurrent data structures. As `Lincheck` is not just a tool but a platform for incorporating advancements in concurrency testing and model checking, we plan to integrate cutting-edge model checkers that support weak memory models. Written in Kotlin, `Lincheck` is also interoperable with native languages such as Swift or C/C++. Our goal is to extend `Lincheck` testing to these languages, making it the leading tool for checking correctness of concurrent algorithms. We believe that `Lincheck` has the potential to significantly improve the quality and efficiency of concurrent algorithms development, reducing time and effort to write reliable tests and investigate bugs.

References

1. Lincheck - A framework for testing concurrent data structures on JVM. https://github.com/Kotlin/kotlinx-lincheck
2. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6
3. The Java Concurrency Stress tests. https://openjdk.java.net/projects/code-tools/jcstress
4. Lindstrom, G., Mehlitz, P.C., Visser, W.: Model checking real time java using java pathfinder. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 444–456. Springer, Heidelberg (2005). https://doi.org/10.1007/11562948_33
5. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. SIGPLAN Not. **42**(6), 446–455 (2007)
6. Kokologiannakis, M., Vafeiadis, V.: GenMC: a model checker for weak memory models. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 427–440. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_20
7. Jung, R., et al.: Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In: Conference Record of the Annual ACM Symposium on Principles of Programming Languages 2015, pp. 637–650 (2015)
8. Coq - a formal proof management system. https://coq.inria.fr
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)
10. Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The art of multiprocessor programming. Newnes (2020)
11. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In ACM SIGPLAN Notices, vol. 48, pp. 317–328. ACM (2013)
12. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17653-1_29
13. [JDK-8256833] ConcurrentLinkedDeque is non-linearizable. https://bugs.openjdk.java.net/browse/JDK-8256833
14. [JDK-8188900] ConcurrentLinkedDeque linearizability. https://bugs.openjdk.java.net/browse/JDK-8188900
15. [JDK-8189387] ConcurrentLinkedDeque linearizability continued. https://bugs.openjdk.java.net/browse/JDK-8189387
16. Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines
17. JCTools - Java Concurrency Tools for the JVM. https://github.com/JCTools/JCTools
18. Lincheck: A Practical Framework for Testing Concurrent Data Structures on JVM. Zenodo (2023)
19. Race in NonBlockingHashMapLong in JCTools. https://github.com/JCTools/JCTools/issues/319
20. MutexLincheckTest.modelCheckingTest detects non lock-free execution path in Mutex #2590. https://github.com/Kotlin/kotlinx.coroutines/issues/2590
21. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–339 (2008)

22. Lincheck User Guide. https://kotlinlang.org/docs/lincheck-guide.html
23. ObjectWeb ASM. https://asm.ow2.io
24. Elmas, T., Burnim, J., Necula, G., Sen, K.: Concurrit: a domain specific language for reproducing concurrency bugs. ACM SIGPLAN Not. **48**, 06 (2013)
25. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 96–110. Association for Computing Machinery, New York (2019)
26. Tretmans, J.: Conformance testing with labelled transition systems: implementation relations and test generation. Comput. Netw. ISDN Syst. **29**(1), 49–79 (1996)
27. Koval, N., Alistarh, D., Elizarov, R.: Scalable fifo channels for programming via communicating sequential processes. In European Conference on Parallel Processing. Springer, Heidelberg (2019). http://pub.ist.ac.at/dalistar/Scalable_FIFO_Channels_EuroPar.pdf
28. Scherer III, W.N., Scott, M.L.: Nonblocking concurrent objects with condition synchronization. In: Proceedings of the 18th International Symposium on Distributed Computing, pp. 2121–2128 (2004)
29. Concurrent Radix and Suffix Trees for Java. https://github.com/npgall/concurrent-trees
30. Race in ConcurrentSuffixTree in Concurrent-Trees library. https://github.com/npgall/concurrent-trees/issues/33
31. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. SIGPLAN Not. **45**(5), 257–268 (2010)
32. Drachsler, D., Vechev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2014, pp. 343–356. Association for Computing Machinery, New York (2014)
33. Sagonas, K., Winblad, K.: Contention adapting search trees. In: 2015 14th International Symposium on Parallel and Distributed Computing, pp. 215–224 (2015)
34. Aksenov, V., Gramoli, V., Kuznetsov, P., Malova, A., Ravi, S.: A concurrency-optimal binary search tree, pp. 580–593 (2017)
35. Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 521–530 (2012)
36. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 330–340 (2010)
37. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, pp. 162–180 (2019)
38. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2003, pp. 167–178. Association for Computing Machinery, New York (2003)
39. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 308–319. Association for Computing Machinery, New York (2006)
40. Sen, K.: Race directed random testing of concurrent programs. In PLDI 2008 (2008)

41. Huang, J., O'Neil Meredith, P., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. SIGPLAN Not. **49**(6), 337–348 (2014)
42. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Proceedings of the 25th International Conference on Computer Aided Verification, vol. 8044, pp. 141–157 (2013)
43. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for c/c++ concurrency. Proc. ACM Program. Lang. **2**(POPL) (2017)
44. Emmi, M., Enea, C.: Violat: generating tests of observational refinement for concurrent objects. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 534–546. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_30