Testing Concurrent Algorithms on JVM with Lincheck and IntelliJ IDEA

Aleksandr Potapov
JetBrains
Berlin, Germany
aleksandr.potapov@jetbrains.com

Maksim Zuev
JetBrains
Amsterdam, Netherlands
maksim.zuev@jetbrains.com

Evgenii Moiseenko
JetBrains Research
Belgrade, Serbia
evgeniy.moiseenko@jetbrains.com

Nikita Koval
JetBrains
Amsterdam, Netherlands
nikita.koval@jetbrains.com

Abstract

This paper presents an Intellij IDEA plugin for Lincheck— a popular framework for testing concurrent data structures on JVM. The Lincheck framework automatically generates concurrent scenarios and examines them with a model checker, providing a detailed execution trace that reproduces the detected error. This trace includes all shared memory access and synchronization events. The Intellij IDEA plugin offers *record-and-replay debugging* to study the execution trace, providing native debugging experience in the IDE. The Lincheck plugin pauses the failed execution at the beginning and provides additional panels that visualize the failed scenario, the execution trace, and the current state of the data structure. One can step through the trace and reproduce the error, moving forward and backward and observing how the data structure changes. These novel capabilities significantly improve the debugging process, making identifying and fixing complex concurrency bugs easier.

CCS Concepts

• **Software and its engineering → Model checking**; **Software testing and debugging**.

Keywords

concurrency, model checking, record-and-replay debugging, IDE

ACM Reference Format:
Aleksandr Potapov, Maksim Zuev, Evgenii Moiseenko, and Nikita Koval. 2024. Testing Concurrent Algorithms on JVM with Lincheck and IntelliJ IDEA. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685301

1 Introduction

Concurrent programming is notoriously challenging. Bugs in concurrent programs can be elusive and often difficult to reproduce, particularly due to the non-deterministic nature of concurrent execution. As such, reliable tools for testing, verifying, and debugging concurrent programs are critically needed in industry, academia, and software engineering education.

One technique that has proven effective in restraining the non-deterministic nature of concurrent programs is *software model checking* [5, 8, 15, 22]. Model checking systematically and deterministically explores possible traces of a concurrent program in a controlled environment. In this way, the inherently non-deterministic execution of concurrent programs is turned into a deterministic process, enabling robust and reproducible testing.

For JVM-based languages, such as Java and Kotlin, the Lincheck framework [16] provides a declarative and reliable way to test concurrent data structures. Lincheck takes a list of data structure operations, automatically generates a set of concurrent scenarios, and examines them in the *model checking mode*, exploring different thread interleavings and executing each deterministically. In case the model checker discovers a bug, the framework provides a detailed execution *trace* that instructs how to reproduce the error. This trace represents the order of shared memory events in the execution, such as reads and writes to object fields and thread switches.

Although the execution trace is extremely useful for identifying the root cause of a bug, presenting it as plain text means that developers have to simulate the sequence of events from the trace in their minds. One may dream of a superior approach involving integration into standard development workflow inside IDE, leveraging the debugger for interactive stepping through the trace.

Our Contribution. To improve the debugging user experience, we developed an IDE plugin for Intellij IDEA that provides a smooth integration of the Lincheck framework with the IDE debugger.

The plugin complements the IDE with several new panels that visualize the failed scenario, the execution trace, and the current state of the tested data structure. By providing integration with the Intellij debugger in the form of *record-and-replay debugging* [2, 14, 23, 24], one can replay the execution trace right in the IDE, jumping both forward and backward and utilizing the full power of the Intellij IDEA debugger. Specifically, the execution trace view offers convenient navigation to the source code and the ability to pause at any trace event, making it easy to navigate the execution trace and match trace events with the corresponding source code.

Overall, the Lincheck integration with the Intellij IDEA enables a cohesive testing and debugging workflow.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685301

2 The Lincheck Framework

We begin by showcasing Lincheck on an example and explaining the framework's implementation details.

```kotlin
1  class SPSCQueue<E> {
2      @Volatile var head = Node(null)
3      @Volatile var tail = head
4
5      fun enqueue(element: E) {
6          val newTail = Node(element)
7          val curTail = tail
8          tail = newTail
9          curTail.next = newTail
10     }
11
12     fun dequeue(): E? {
13         if (tail == head) return null
14         head = head.next!! // <- can throw NullPpointerException
15         return (head.element as E)
16     }
17
18     private class Node(val element: Any?) {
19         @Volatile var next: Node? = null
20     }
21 }
22 class SPSCQueueTest {
23     val q = SPSCQueue<Int>()
24
25     @Operation(nonParallelGroup = "producer")
26     fun enqueue(element: Int) = q.enqueue(element)
27
28     @Operation(nonParallelGroup = "consumer")
29     fun dequeue() = q.dequeue()
30
31     @Test
32     fun test() = ModelCheckingOptions().check(this::class)
33 }
```

Listing 1: Buggy single producer single consumer concurrent queue implementation with a concurrent Lincheck test.

Lincheck by example. Listing 1 shows a simple single-producer single-consumer queue with a Lincheck test to verify its correctness. The data structure recalls the classic Michael-Scott queue [20], forming a linked list (lines 18–20) with head and `tail` referencing the first and the last nodes (lines 2–3). The empty queue is expressed with a linked list of a single node, both head and `tail` pointing to it. In this way, the first queue element is stored not in the head node but in the `head.next` one.

The `enqueue(..)` operation creates a new node with the specified element (line 6), reads the current `tail` (line 7), updates it to the created node (line 8), and then updates the `next` pointer of the previous `tail` (line 9), thus, linking the new node. To extract the first element, `dequeue()` checks whether the queue is empty by comparing head and `tail` references (line 13), followed by extracting the first node (line 14) and reading the element (line 15).

The following Lincheck test verifies whether this queue implementation is correct (lines 22–33). In the test constructor (lines 23), we specify how to create a new queue instance, followed by listing the tested operations (lines 25–25) — the corresponding functions should be marked with a special `@Operation` annotation provided by Lincheck. To instruct the framework to satisfy the single-producer single-consumer semantics, we put `enqueue(..)` and `dequeue()` into different non-parallel operation groups (see the `nonParallelGroup` annotation parameter). Finally, we run the test using the `JUnit` testing framework and instructing Lincheck to use the model checking mode (lines 31–32).

```
= Invalid execution results =
| ---------------------------------------------------- |
|             Thread 1             |     Thread 2      |
| ---------------------------------------------------- |
| dequeue(): NullPointerException #1 | enqueue(-1): void |
| ---------------------------------------------------- |

The following interleaving leads to the error:
| ------------------------------------------------------------- |
|             Thread 1             |          Thread 2          |
| ------------------------------------------------------------- |
|                                  | enqueue(-1)                |
|                                  |   q.enqueue(-1)            |
|                                  |     tail.READ: Node#1      |
|                                  |     tail.WRITE(Node#2)     |
|                                  |     switch                 |
| dequeue(): NullPointerException #1 |                          |
|                                  |     Node#1.setNext(Node#2) |
|                                  |   result: void             |
| ------------------------------------------------------------- |

Exception stack traces:
#1: java.lang.NullPointerException: null
    at SPSCQueue.dequeue(SPSCQueue.kt:14)
    at SPSCQueueTest.dequeue(SPSCQueue.kt:29)
```

Listing 2: Lincheck test output for the code in Listing 1.

The test output is presented in Listing 2. Lincheck reports a bug, providing the minimized *concurrent scenario* on which the bug manifests and the detailed execution trace that shows how to reproduce the error. The bug manifests when `enqueue(..)` and `dequeue()` are executed concurrently, leading to an unexpected `NullPointerException` in `dequeue()`.

The detailed execution trace makes it easy to analyze the error. The execution starts with `enqueue(-1)`, which creates a new node and updates `tail` but gets preempted before linking this node to the linked list. At this point, the execution switches to `dequeue()`, which notices that the `tail` reference has been updated and tries to extract the first node, which results in `NullPointerException` when reading `head.next` and expecting it to be non-null. A possible fix would be to either update the `next` reference before `tail` in `enqueue(..)`, or check `head.next` for `null` to decide whether the queue is empty in `dequeue()`.

How Lincheck works. Given the listed operations (annotated with `@Operation`), Lincheck automatically (1) generates a set of random concurrent scenarios, (2) examines them with a bounded model checker, and (3) verifies that the results of each invocation satisfy the required correctness property (linearizability is the default one). The model checker explores different thread interleavings, attempting to detect various bugs, such as unexpected exceptions, deadlocks, livelocks, and linearizability violations [7, 13]. When Lincheck discovers a bug, it replays the failed interleaving and collects the execution trace to provide it in the test output.

Bytecode Instrumentation. For the model checker to control the execution and study different thread interleaving, we need a way to pause threads before shared memory accesses (such as reads, writes, atomic, and synchronization instructions) and select the one that is currently active. For that, Lincheck instruments the JVM bytecode of the testing code, injecting calls to the model checker methods right before shared memory events. This technique helps to not only track these events but also to collect all the debug information required to provide an execution trace. Please refer to our paper about Lincheck for the technical details [16].

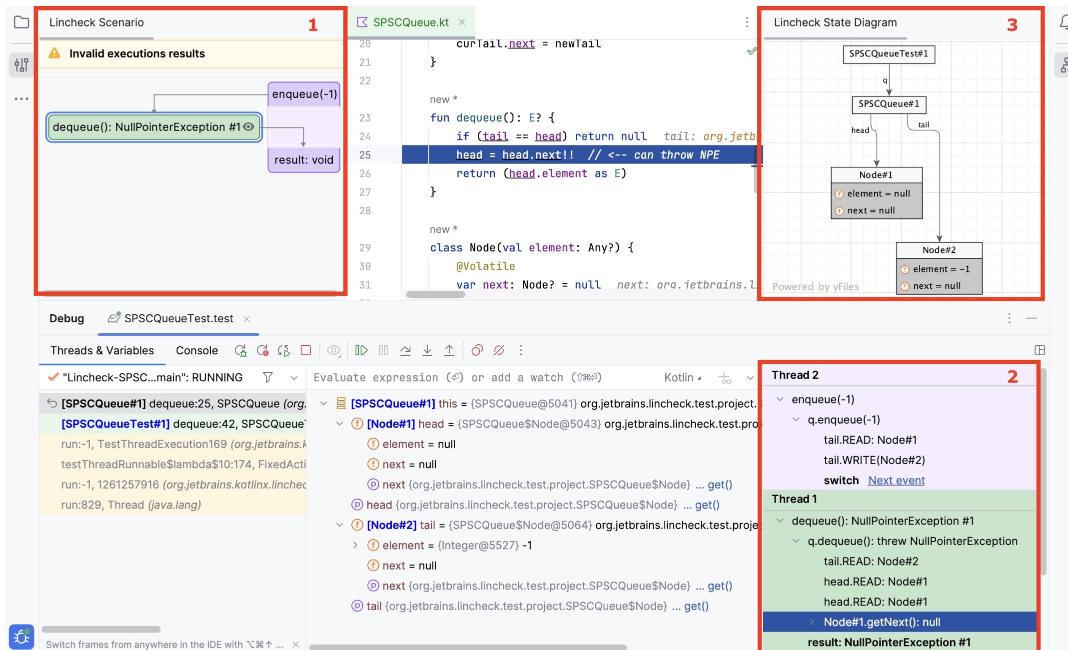

Figure 1: Screenshot of the Intellij IDEA with the Lincheck plugin for debugging the failed test from Listing 1. The plugin visualizes (1) the failed scenario, (2) the execution trace, and (3) the current state of the data structure.

3 IntelliJ IDEA Plugin for Lincheck

To improve the error debugging, we created a plugin [25] for IntelliJ IDEA, which provides native debugging experience for Lincheck. With the plugin, users can re-run a failed Lincheck test in the debug mode. The plugin pauses the failed execution at the beginning, providing additional Lincheck panels that visualize the failed scenario, the execution trace, and the current state of the data structure.

Figure 1 demonstrates the IntelliJ IDEA screen with the Lincheck plugin enabled and the execution paused before reading `null` from `head.next`; the debugging interleaving is the one from Listing 2.

Scenario Visualization. The plugin displays the failed scenario with the results in an intuitive way on the additional “Lincheck Scenario” panel (frame 1 on the screenshot). Concurrent threads are highlighted with different colors, and arrows indicate the order of operations. On Figure 1, the execution starts with `enqueue(-1)` in Thread 2, then switches in the middle of the operation, followed by executing `dequeue()`, switching back to Thread 1, and finishing the preempted `enqueue(-1)`.

Execution Trace. In addition to the scenario visualization, the plugin shows all trace events of the failed execution; see frame 2 on Figure 1. These trace events are also provided in the Lincheck test output (see Listing 2). Apart from the overview of the operations and the thread switches, this panel allows jumping between different trace points and stopping there with the debugger to examine the state of the data structure.

Data Structure Visualization. The final piece is the “Lincheck State Diagram” panel (frame 3 on the screenshot), which visualizes the current state of the tested data structure. This feature is crucial when debugging complex data structures, especially those with circular references, arrays, linked lists, or trees under the hood.

Notably, the plugin renders some concurrency-related constructs in a more intuitive way. For instance, the `AtomicReference` and similar classes are visualized as regular fields instead of separate objects, the fields related to coroutine internals or low-level atomic primitives (e.g., `AtomicReferenceFieldUpdaters`) are hidden, *etc.* This way, the developer's focus does not blur with unnecessary data, and they can track the logical state of the data structure.

Record-and-Replay Debugging. The most significant feature is that the plugin enables stopping at any interleaving point in the debug mode, making it possible to go both forward and backward in the execution.

To go forward in the execution, the debugger resumes the program until it reaches the selected event in the trace. For going to an event in the past, the plugin utilizes Lincheck's ability to reproduce the failed execution, re-running the scenario from the beginning until the execution reaches the selected event. Given that Lincheck is capable of minimizing the failed scenario, the resulting one is typically of a small size, so re-running it from the beginning does not lead to a noticeable delay.

Note that re-running the interleaving when stepping backward might change the object identifiers. To compensate for this effect, the plugin enumerates all objects with the deterministic enumeration that remains stable for different runs (see the node titles in the “Lincheck State Diagram” panel, the labels in blue color in the debugger panel, and the object names in the trace panel).

Deterministic replay of a failed execution is made possible through the bytecode instrumentation performed by Lincheck. The tool records all shared memory and synchronization events and later

replays them in the same order. While a program may also contain other sources of non-determinism, such as filesystem or network I/O, these are beyond the scope of Lincheck, as its primary use case is testing concurrent data structures.

In contrast to alternative record-and-replay debuggers [9, 11, 17, 18], our solution does not require code modification, and does not need additional memory to memorize the execution, but relies on the relatively small size of the execution. At the same time, fairly running the execution makes the plugin's implementation more straightforward and allows the debugger features to be utilized almost out of the box.

Integration with the IntelliJ Debugger. For the model checker to control the execution, Lincheck injects calls to the model checker methods right before shared memory events. To integrate with the debugger, we added additional `beforeEvent(eventId)` calls, enumerating all the execution trace events with ids. When the user selects a trace event in the plugin view, the plugin knows its event id and instructs the debugger to stop when `beforeEvent` with the specified `eventId` is called. The latter is achieved by installing synthetic breakpoints. This way, we created a communication protocol between the Intellij debugger and Lincheck.

With such a communication protocol, Lincheck can provide all the necessary information to the debugger, passing it as parameters to the injected method calls. That is how we provide the bug details and the current state of the data structure for further visualization. Notably, the Intellij debugger can communicate back by replacing the method return values via the "Force Return" functionality [3], which Lincheck uses to detect the plugin.

This strategy makes the integration straightforward but still brings challenges when using the standard debugger features, such as breakpoints and stepping. First, the user's breakpoints should not be triggered during the bug discovery phase. Second, Lincheck injects additional method calls into the bytecode, which must be skipped during stepping. Finally, Lincheck manages thread scheduling during the scenario execution, so any step in the debugger may lead to a hang, as Lincheck may pause the current thread.

We addressed these issues by integrating the debugger process with the Lincheck's model checker. Specifically, we turn off all the user breakpoints until the bug is reproduced, add Lincheck internal classes to the stepping filters in the Intellij debugger configuration, and notify the debugger about the logical thread switches performed by Lincheck, so the debugger can interrupt the current stepping operation and resume the execution in another thread.

With the help of these techniques, developers get the full power of the debugger, including evaluation and memory state analysis, as well as jumping in both forward and backward directions.

4 Tool Availability

The Lincheck framework is open-source and available on GitHub at github.com/JetBrains/lincheck. The Intellij plugin for Lincheck can be downloaded from the marketplace [25]. (The plugin requires IntelliJ IDEA Ultimate, which is free for academic use.)

For a video that showcases the plugin, please refer to jb.gg/lincheck-demo-ecoop24. The accompanying artifact, containing the code in Listing 1, several other examples, and detailed instructions on how to install the plugin, is available on GitHub at JetBrains/lincheck-/tree/ecoop24.

5 Related Work

Model Checking is a well-known software verification technique, implemented in tools such as CBMC [8] for the C language and PathFinder [12] for Java. The CHESS [7, 22] framework for C# was one of the pioneering tools to apply stateless model checking for verifying real-world concurrent programs. More recently, tools like GenMC [15] and Nidhugg [4, 5] have made major advancements in addressing the state space explosion problem encountered during the model checking of concurrent programs, using a technique known as *dynamic partial order reduction*. Additionally, these tools support verification under *weak memory models* [6], detecting more sophisticated bugs arising from insufficient synchronization.

Currently, Lincheck supports neither partial order reduction nor the JVM weak memory model, but we are actively working on integrating these functionalities into our framework.

Record-and-Replay Debugging, also referred to as *time-traveling debugging*, aims to capture and record all sources of non-determinism in a program's execution, with the ability to later deterministically replay the given execution. There could be different sources of non-determinism in a program, such as concurrency, I/O, or system calls. Different record-and-replay tools utilize a variety of techniques to capture the sources of non-determinism: code instrumentation [2, 14, 24], system calls tracing and interception [23], shared library interposition [19]. execution in a controlled environment (*e.g.,* running single thread at a time) [23], as well as special support on the virtual machine [10] or hardware [21] levels.

Lincheck's primary use case is testing and debugging concurrent data structures. Thus, we focus on concurrency and thread scheduling record-and-replay and utilize the code instrumentation approach. In the future, we aim to gradually improve the record-and-replay capabilities of our platform to support more use cases.

Debugging Visualization. Plugins that enhance the host IDE with various debugging visualization features have been developed in the past for Intellij IDEA [17, 18], VSCode [9], and Eclipse [11].

The visual debugger of the Lincheck plugin leverages the Intellij platform diagramming API (based on the `yFiels` library [1]) to achieve IDE-native look and familiar user experience. Moreover, the visualization functionality is integrated with other components of the plugin, such as record-and-replay debugging, enabling users to visually inspect and navigate through the execution trace.

6 Conclusion

This work presents the Intellij IDEA plugin for Lincheck [25], enabling outstanding debugging experience when testing concurrent data structures on JVM. The plugin provides intuitive scenario visualization and navigation through the execution trace, enabling it to stop at any trace point in the debug mode. Besides, it offers time-traveling debugging and visualizes the current data structure state as an object graph.

We believe the Lincheck plugin not only helps software engineers, researchers, and students design and develop concurrent algorithms but also serves as a strong proof-of-concept for IDE-native data structure visualization for IntelliJ IDEs and a reliable record-and-replay debugger for JVM. Both these applications cover broader cases and can be utilized for regular code.

References

[1] 2009. yFiles - the diagramming library. https://www.yworks.com/yfiles-overview Accessed: 2024-07-01.

[2] 2018. Reversible debugging tools for C/C++ on Linux & Android. https://undo.io/ Accessed: 2024-07-01.

[3] 2024. Force return from the current method | Alter the program's execution flow | IntelliJ IDEA Documentation. https://www.jetbrains.com/help/idea/altering-the-program-s-execution-flow.html#force_return Accessed: 2024-07-01.

[4] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless model checking for TSO and PSO. *Acta Informatica* 54 (2017), 789–818.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[6] Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *computer* 29, 12 (1996), 66–76.

[7] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 330–340.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*. Springer, 168–176.

[9] Henning Dieterichs. 2019. Debug Visualizer - VS Code Extension | Marketplace. https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer Accessed: 2024-07-01.

[10] Pavel Dovgalyuk. 2012. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging.. In *CSMR*. 553–556.

[11] Paul V Gestwicki and Bharat Jayaraman. 2004. Jive: Java interactive visualization environment. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 226–228.

[12] Klaus Havelund and Thomas Pressburger. 2000. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* 2 (2000), 366–381.

[13] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[14] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 207–216.

[15] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A model checker for weak memory models. In *International Conference on Computer Aided Verification*. Springer, 427–440.

[16] Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. 2023. Lincheck: A practical framework for testing concurrent data structures on JVM. In *International Conference on Computer Aided Verification*. Springer, 156–169.

[17] Tim Kräuter, Patrick Stünkel, Adrian Rutle, and Yngve Lamo. 2024. The Visual Debugger: Past, Present, and Future. *Proceedings of the 1st IDE Workshop (Co-located with International Conference on Software Engineering)* (2024).

[18] Tim Kräuter. 2024. Visual Debugger - IntelliJ IDEs Plugin | Marketplace. https://plugins.jetbrains.com/plugin/16851-visual-debugger Accessed: 2024-07-01.

[19] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards practical default-on multi-core record/replay. *ACM SIGPLAN Notices* 52, 4 (2017), 693–708.

[20] Maged M Michael and Michael L Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. 267–275.

[21] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 289–300.

[22] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs.. In *OSDI*, Vol. 8.

[23] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[24] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2–11.

[25] Aleksandr Potapov, Maksim Zuev, and Nikita Koval. 2024. Lincheck - IntelliJ IDEs Plugin | Marketplace. https://plugins.jetbrains.com/plugin/24171-lincheck Accessed: 2024-07-01.

Received 2024-07-05; accepted 2024-07-26