FlakeSync: Automatically Repairing Async Flaky Tests

Shanto Rahman
The University of Texas at Austin
Austin, Texas, USA
shanto.rahman@utexas.edu

August Shi
The University of Texas at Austin
Austin, Texas, USA
august@utexas.edu

ABSTRACT

Regression testing is an important part of the development process but suffers from the presence of flaky tests. Flaky tests nondeterministically pass or fail when run on the same code, misleading developers about the correctness of their changes. A common type of flaky tests are async flaky tests that flakily fail due to timing-related issues such as asynchronous waits that do not return in time or different thread interleavings during execution. Developers commonly try to repair async flaky tests by inserting or increasing some wait time, but such repairs are unreliable.

We propose FlakeSync, a technique for automatically repairing async flaky tests by introducing synchronization for a specific test execution. FlakeSync works by identifying a critical point, representing some key part of code that must be executed early w.r.t. other concurrently executing code, and a barrier point, representing the part of code that should wait until the critical point has been executed. FlakeSync can modify code to check when the critical point is executed and have the barrier point keep waiting until the critical point has been executed, essentially synchronizing these two parts of code for the specific test execution. Our evaluation of FlakeSync on known flaky tests from prior work shows that FlakeSync can automatically repair 83.75% of async flaky tests, and the resulting changes add a median overhead of only 1.00X the original test runtime. We submitted 10 pull requests with our changes to developers, with 3 already accepted and none rejected.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

flaky test repair, async flaky tests

ACM Reference Format:
Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639115

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639115

1 INTRODUCTION

Regression testing is an important part of the software development process, but it suffers from the presence of flaky tests. A *flaky test* is a test that can nondeterministically pass or fail when run on the same version of code [29]. After a developer makes some changes, if the flaky test fails, the developer cannot trust whether that failure indicates a true fault introduced in their changes. Flaky tests can mislead developers, forcing them to waste time debugging nonexistent faults in changes. Flaky tests are prevalent both in open-source projects [9, 16, 23, 29] and in industry [19–22, 30].

Luo et al. [29] found that the most prominent type of flaky tests in open-source projects are async-wait flaky tests, which are tests that fail due to making an asynchronous call but does not wait sufficiently for the computations to complete. The second most prominent type of flaky tests are concurrency flaky tests, which are tests that fail due to different thread interleavings. Both such flaky tests are dependent on nondeterministic timing issues during test execution, e.g., the service being waited on takes too long to complete or some thread takes longer than another, leading to an unexpected thread interleaving. In total, Luo et al. found that 65% of the flaky tests they studied are of these two categories [29]. Other empirical studies similarly found both these types of flaky tests to be the most prominent ones [13, 22]. In this paper, we refer to such tests as *async flaky tests*, because these tests fail due to lack of proper synchronization between relevant parts of code.

The most common way developers repair async flaky tests, particularly for async-wait flaky tests, is by modifying wait times in the test code and code-under-test [22, 29]. For example, Lam et al. found that developers commonly repair flaky tests at Microsoft by adding additional wait times [22] to reduce flaky failures. However, they found developers did not have a systematic way in determining the proper wait time, waiting longer than necessary and increasing the cost of testing. Further, modifying wait times can only mitigate the issue, as the test may still fail when run on different machines. Luo et al. [29] also found that developers in open-source would add wait times to mitigate async-wait flaky tests, but this strategy does not provide guarantees concerning flakiness while making tests run slower. They recommend doing some synchronization that waits for a condition to be satisfied as a better repair strategy.

We propose FlakeSync, a technique for automatically repairing async flaky tests by introducing synchronization for a specific test execution. Our intuition is that the flakiness comes from the test execution not synchronizing properly between concurrently executing code, e.g., across different threads. For async-wait flaky tests, the test does not wait long enough for another service to complete its execution. By ensuring the test does wait for the service to complete, i.e., synchronizing with the completion of that

service, it would no longer fail. For concurrency flaky tests, the failure comes from one thread interleaving differently from another, so synchronizing one thread to wait until the other thread finishes executing a critical part of code can repair the flakiness.

At the high level, FlakeSync is composed of two main components. For a given async flaky test to repair, the first component, CritSearch, searches for the *critical point* that must be executed early relative to other concurrently running code. If the execution slows down such that the critical point is executed too late, the test would fail. The second component, BarrierSearch, searches for the *barrier point* corresponding to the identified critical point. The barrier point is the part of code that needs to wait until the critical point has been executed before proceeding, or else the test fails (we use the term "barrier" due to similarities to barriers for synchronization in concurrent programming). With both the critical point and barrier point, FlakeSync can repair the async flaky test by (1) modifying the critical point to keep track of whether it gets executed and how many times it gets executed when the test runs, and (2) modifying the barrier point to continuously wait and yield execution to other threads until the critical point has been executed sufficiently. These changes together introduce the synchronization that ensures the async flaky test does not flakily fail.

We evaluate FlakeSync on 176 known flaky tests taken from an existing dataset of known flaky tests [4]. Initially, we find that FlakeSync can repair 67 flaky tests. From further inspection, we find that most of the tests that FlakeSync cannot repair are actually flaky due to their reliance on exact timing, e.g., assertions that check some code takes less than a constant amount of time to run or a strict timeout built into the test. While such tests can indeed be flaky due to timing issues on the machine, e.g., the machine being busy and therefore runs the tests slower, they are not flaky due to synchronization issues that async flaky tests suffer from. Such tests can be repaired by increasing the existing timeout bounds encoded in the tests in case of slower machines. Filtering out such flaky tests results in 80 async flaky tests, and FlakeSync can repair 83.75% of these async flaky tests. The median amount of time FlakeSync takes to repair a flaky test is 58.77 minutes. Further, after we apply the changes to repair the test, we find that these changes lead to a median overhead of only 1.00X the original test runtime. This low overhead shows how proper synchronization does not necessarily increase the testing cost, unlike inserting/increasing wait times.

We also submit pull requests where we repair the known async flaky tests on the latest versions of the projects based on FlakeSync's proposed changes. We only submit pull requests for async flaky tests that are still flaky on the latest version of projects that we could build, and we only submit one pull request per project as to avoid spamming developers with pull requests until they confirm they would like to see such changes to their tests. We submitted 10 pull requests to 10 projects. So far, 3 pull requests have already been accepted, and no pull request has been rejected.

The contributions of our paper are:

- We propose a technique FlakeSync for automatically repairing async flaky tests by introducing synchronization points during test execution.
- We implement FlakeSync for Java projects and evaluate on known flaky tests from a prior dataset [4]. Our results show that FlakeSync can repair 83.75% of the async flaky tests, and the changes make the test runtime on median 1.00X the runtime before the changes.
- We submitted 10 pull requests to developers for repairing their async flaky tests. So far, 3 pull requests have already been accepted, and no pull request has been rejected.

```
1  public class GrpcServerTest {
2    @Test
3    public void testGrpcExecutorPool() {
4      GRPCMetrics gm = GRPCMetrics.getEmptyGRPCMetrics();
5      GrpcThreadPoolExecutor executor =
6        new GrpcServer.GrpcThreadPoolExecutor(gm);
7      ...
8      executor.submit(...);
9      ...
10 +    while (!GGrpcThreadPoolExecutor.hasExecuted) {
11 +      Thread.yield();
12 +    }
13     Thread.sleep(120);
14     double activeThreads = gm.getGaugeMap().get(THREADS);
15     assertEquals(2, activeThreads);
16     double queueSize = gm.getGaugeMap().get(QUEUE);
17     assertEquals(1, queueSize);
18     ....
19   }
20 }
21
22 public GrpcThreadPoolExecutor {
23   ...
24   public GrpcThreadPoolExecutor {
25     private final GRPCMetrics gm;
26     public GrpcThreadPoolExecutor(GRPCMetrics gm) {
27       this.gm = gm;
28     }
29   @Override
30   protected void beforeExecute(Thread t, Runnable r) {
31     gm.incGauge(THREADS);
32     gm.setGauge(QUEUE, getQueue().size());
33 +    hasExecuted = true;
34     super.beforeExecute(t, r);
35   }
36 }
```

Figure 1: Example flaky test from apache/incubator-uniffle

2 EXAMPLE

Figure 1 illustrates a simplified version of an example flaky test from project apache/incubator-uniffle, used in our evaluation. The flaky test is `GrpcServerTest#testGrpcExecutorPool`, a previously known flaky test from a public dataset [4], meaning prior work observed the test to both pass and fail when rerun on the same version of code many times.

This test first creates a `GRPCMetrics` object (Line 4), which is then passed in as an argument when creating an instance of `GrpcThreadPoolExecutor` (Line 6). This `executor` is used to spawn new threads to run (Line 8). We see that each new thread first executes the `beforeExecute` method of the `GrpcThreadPoolExecutor` class (Line 30), which modifies the shared `GRPCMetrics` object passed in as an argument (Lines 31-32). After spawning the new thread to run, the test proceeds to wait for

```
1 def FlakeSync(test):
2   crit_points, delays = CritSearch(test)
3   barrier_points = BarrierSearch(test, crit_points, delays)
4   return crit_points, barrier_points
```

Figure 2: Pseudocode for FlakeSync.

120ms (Line 13), expecting that other threads will finish their task within this time period. Finally, the test asserts on some values based on the shared `GRPCMetrics` object (Lines 15 and 17).

This pattern of waiting for a constant amount of time for some other code to finish running is a classic example of an async-wait flaky test [29]. This test's assertions can fail if the computations on another thread that modify the shared `GRPCMetrics` object do not complete within a reasonable amount of time. Interestingly, the developers had previously noticed this test to be flaky and attempted to repair it by modifying the wait time (Line 13); the wait time was previously 100ms, so they increased the time, but the test still remains flaky.

The key issue is that the test should only assert on values in the shared `GRPCMetrics` object once the other thread has finished setting those values. As such, a better way to repair this test is to force the test to wait until the other thread has finished executing the critical computations that modify the shared `GRPCMetrics` object before proceeding to the assertions. We can enforce this type of synchronization between threads by inserting a variable to check whether execution has made it past those critical computations (Line 33) and then introducing a loop in the test code to continuously yield execution to other threads until that point has been executed on the other thread (Lines 10-12).

Using our technique, we were able to identify these parts of the code to modify to enforce such synchronization, constructing a patch for this flaky test as shown in Figure 1. We sent this patch as a pull request to the developers of apache/incubator-uniffle, and the pull request was accepted with positive feedback.

3 FLAKESYNC

We present FlakeSync, a technique for automatically repairing async flaky tests. The intuition behind FlakeSync is that async flaky tests' flaky failures often involve concurrently executing code across multiple threads where code on one thread needs to be executed first relative to code on another thread, such as the async-wait flaky test presented in Figure 1. FlakeSync repairs async flaky tests by identifying the key parts of code during test execution that should be synchronized with one another. FlakeSync can then modify the code execution to introduce synchronization mechanisms that ensure the execution on one thread does not proceed until the other thread has executed the other critical code section.

At the high-level FlakeSync takes as input an async flaky test and returns the relevant locations where a developer should synchronize. FlakeSync consists of two main components. The first component is the CritSearch. This component aims to identify the *critical point* for the async flaky test, which is the part of code that must be executed early; if the execution slows down such that the critical point is executed too late relative to other executed code on a different thread, the test would fail. The second component is the BarrierSearch. This component aims to identify the *barrier point*, which is the code that needs to wait until the corresponding critical point has been executed. Essentially, the execution should not proceed past the barrier point until the critical point is executed, or else the test fails (we use the term "barrier" as it is similar to the concept of barriers for synchronization in concurrent programming). Figure 2 shows pseudocode illustrating how FlakeSync leverages these two components to propose a patch for a given async flaky test. Essentially, FlakeSync first uses CritSearch to find critical points along with the amounts of delay to inject at each critical point that can reproduce a flaky-test failure (Line 2). FlakeSync then passes those critical points and corresponding delays to BarrierSearch so it identifies the corresponding barrier points to the critical points (Line 3). The final output from FlakeSync are two lists, the critical points and barrier points (the critical points and barrier points in the same position in the two lists correspond to each other).

3.1 CritSearch

Given an async flaky test, CritSearch searches for the critical points. The intuition is that, if the thread that executes a critical point takes too long, then the test fails. In other words, if we inject a long enough delay before a critical point, then the test fails. As such, CritSearch identifies critical points by searching through possible locations in which to inject some delay that can lead to the test to reliably fail after execution. Figure 3 illustrates pseudocode for how CritSearch runs on a given async flaky test.

3.1.1 Delay Injection. CritSearch first needs to inject delays throughout the test execution to see whether any injected delays can lead to a test failure. As opposed to randomly sampling delay locations or simply injecting delays everywhere, CritSearch relies on the heuristic that the most relevant delay locations are going to be those related to concurrent methods, namely methods that execute concurrently with others during test execution. As such, CritSearch first looks for concurrent methods (Line 5). Similar to prior work on finding concurrent methods for detecting concurrency bugs [33], we find concurrent methods by dynamically instrumenting the entry-point and exit-points of each executed method in the code-under-test, and we identify a method to be a concurrent method if its execution has not reached an exit-point while execution has reached the entry-point of another method. Due to the nondeterministic nature of flaky test executions, we run the test 10 times and combine all the concurrent methods detected throughout those runs. Note that our approach over-approximates the set of concurrent methods to track, because we do not track which exact pairs of tests actually ran concurrently with one another, instead grouping together all methods that at some point ran concurrently with another method.

With the set of concurrent methods for an async flaky test, CritSearch searches for all the locations (i.e., lines in code) where there are calls to a concurrent method (Line 6). The intuition is that any method invocation can potentially take more time than expected, e.g., due to any stress on the machine or other processes running [40], so we want to simulate slowdowns, particularly for these methods that run concurrently with others. CritSearch then runs

```
1  def CritSearch(test):
2    crit_points = []
3    crit_points_delays = []
4
5    conc_methods = find_conc_methods(test)
6    locs = find_delay_locs(test, conc_methods)
7    delay = INITIAL_DELAY
8    while delay <= MAX_DELAY:
9      test_res = run_with_delay(test, locs, delay)
10     if test_res == FAIL:
11       break
12     delay = delay*2
13   if not test_res == FAIL:
14     return crit_points, crit_points_delays # Unsuccessful
15   min_locs, delay = minimize(test, locs, delay, 2)
16   root_methods_and_delays = find_root_methods(test, delay,
17     min_locs)
18
19   for root_method, delay in root_methods_and_delays:
20     in_boundary = False
21     start_line = start_line_number(root_method)
22     end_line = end_line_number(root_method)
23     for line in range(start_line, end_line):
24       test_res = run_with_delay(test, line, delay)
25       if test_res == FAIL:
26         in_boundary = True
27         cp = line
28       elif test_res != FAIL and in_boundary:
29         crit_points.append(cp)
30         crit_points_delays.append(delay)
31         in_boundary = False
32         break
33     # Last line is critical point
34     if in_boundary:
35       crit_points.append(end_line)
36       crit_points_delays.append(delay)
37   return crit_points, crit_points_delays
38
39 def minimize(test, locs, delay, n):
40   if len(locs) == 1:
41     return locs, delay
42   # Split n equal subsets and their complements
43   subsets = split_locs(locs, n)
44   _d = delay
45   while _d <= MAX_DELAY:
46     for s in subsets:
47       if run_with_delay(test, s, _d) == FAIL:
48         return minimize(test, s, _d, 2)
49     _d = 2*_d
50   # Trying finer granularity if possible
51   if len(locs) < 2*n:
52     return locs, delay
53   else:
54     return minimize(test, locs, delay, 2*n)
55
56 def find_root_methods(test, delay, min_locs):
57   callstacks = get_callstacks(test, min_locs, delay)
58   root_methods_and_delays = set()
59   for callstack in callstacks:
60     _d = delay
61     for callsite in get_callsites(callstack):
62       while _d <= MAX_DELAY:
63         test_res = run_with_delay(test, callsite, _d)
64         if test_res == FAIL:
65           root_method = containing_method(callsite)
66           break
67         else:
68           _d = _d*2
69     root_methods_and_delays.add((root_method, _d))
70   return root_methods_and_delays
```

Figure 3: Pseudocode for CritSearch.

the test with delays injected at all such locations (Line 9), i.e., dynamically instrumenting code to call `Thread.sleep` with a specified amount of delay (initially a configurable `INITIAL_DELAY`), right before all locations. As part of this process, if the test still passes with delays injected, then CritSearch doubles the amount of delay injected at each delay location (Line 12) before running the test again. It continues doubling the amount of delay up until the test fails or reaches the maximum amount of delay (a configurable `MAX_DELAY`). If the test cannot fail up to this threshold, CritSearch is unsuccessful at finding any critical points.

3.1.2 Minimizing Delay Locations. CritSearch uses delta-debugging [43] to minimize the set of delay locations found from the previous step, represented as function `minimize` presented in Figure 3. Given a set of delay locations, `minimize` returns a subset of those delay locations that can still result in a test failure. In standard delta-debugging methodology, `minimize` splits the input set into equal subsets (Line 43) and runs the test with a delay injected at each delay location within a subset, minimizing that subset further if the test still fails (Line 48). If the algorithm cannot split the set of delay locations into any finer granularity, it simply returns the current set as the minimal set (Line 52); otherwise, it recursively tries to minimize the set further but with a finer split (Line 54).

Normally, when delta-debugging finds it cannot split the current set anymore while satisfying the criterion, it terminates and reports that set as the minimal set. In our case, we make a change to the delta-debugging process by instead increasing the amount of delay injected at the current delay locations (Line 49), up until a maximum `MAX_DELAY`, retrying to run with the new delay amount. The reasoning is that a test may only fail when its execution slows down for a sufficient total amount of time, which is summed across all the delays injected at multiple delay locations. As such, when we reduce the number of delay locations, we would be reducing the overall delay time during test execution, which in turn could decrease the chance of test failure. While we potentially can increase the cost of the search by running with increased amounts of delay, we find that this modification helps reduce the set of delay locations.

3.1.3 Root Method Search. The minimal set of delay locations, even if it consists of just one delay location (indeed, our results show that often we can obtain just one delay location), may still not represent the critical point. There may be many minimal sets of delay locations that all independently can lead to the test to fail. For example, it could be the case that injecting a delay anywhere in the same method can still result in the same failure. In such a scenario, instead of injecting that delay anywhere within the method, we could instead inject a delay within the caller of that method, right before the call site. In fact, we could keep going up this call stack until reaching the "highest" call site where injecting a delay there results in the same failure. We consider this method that is highest on the call stack from the minimal set of delay locations where injecting a delay can still make the test fail (i.e., we cannot inject a delay at any caller of this method and make the test fail) as the *root method*.

Function `find_root_methods` in Figure 3 returns the root methods with their corresponding delays for the minimal delay locations for an async flaky test. For each delay location in the minimal set, we compute the call stacks for that location collected

across all executions of the location when running the test (Line 57); this process also removes any repeated call stacks for the same location. Each call stack starts from the delay location and ends at a call site within the async flaky test itself. `find_root_methods` handles each call stack independently, iterating through the call sites in each call stack, going up starting from the delay location. Each iteration keeps track of a current call site, where it runs the test while injecting delay at that call site (Line 63). If the test fails, then `find_root_methods` sets the method containing the call site as the current root method (Line 65). On the other hand, if the test passes, we also attempt to increase the amount of delay injected at the current call site (Line 68), with the similar reasoning that a higher call site may work if there is a longer delay.

The next iteration of the loop then tries the direct caller in the call stack (Line 61). The final root method along with the amount of delay to inject is included in a set before moving on to the next call stack (Line 69). The final output of this function is the set of root methods and their corresponding amounts of delay to be injected.

3.1.4 Identifying Critical Point. Finally, CritSearch searches through the root methods to identify the critical points. The intuition is that there is a region of code in a root method where injecting a delay anywhere in that region manifests the test failure. As such, the "boundary" of this region, namely the location right after this region, would be the critical point.

To search through the root method, CritSearch starts from the beginning of the method, injecting a delay on a location-by-location basis and running the test (Lines 23-32). The boundary starts when injecting a delay leads the test to start failing. CritSearch then checks whether there is then a switch from test failing to test passing (Lines 28-32). The final location where injecting a delay leads to a failure is then the critical point for the root method and is saved. The final output is the list of all critical points.

Example Critical Point. In the example shown in Figure 1, the critical point is identified to be Line 32, as after executing this line all computations modifying the shared data have finished. We can introduce a field to keep track of when the critical point has been executed for use later (Line 33).

3.2 BarrierSearch

Given the critical points identified by CritSearch, BarrierSearch searches for the corresponding barrier points. Figure 4 shows pseudocode for BarrierSearch.

For each critical point, BarrierSearch first runs the test with the injected delay at the key delay location right before the critical point as to force the test to fail. In Java/JUnit, a test failure results in a thrown exception, which includes an exception call stack that shows the point the exception occurred along with all the methods called in-between. We run the test with the injected delay to obtain this exception call stack (Line 4).

BarrierSearch uses function `search` to iterate through the call sites in the call stack, starting at the location where the exception occurred (Line 14). The location right before this call site could be a barrier point, and BarrierSearch starts from that location, going back location-by-location up until the beginning of the containing method (Line 18), checking whether any of these locations are a barrier point. The function `run_test_with_crit_and_yield`

```
1  def BarrierSearch(test, crit_points, delays):
2    barrier_points = []
3    for cp, delay in zip(crit_points, delays)
4      callstack = get_failure_callstack(test,
5        cp, delay)
6      bp = search_bp(test, cp, delay, callstack, 1)
7      if bp == None:
8        thres = count_crit_executions(test, cp)
9        bp = search_bp(test, cp, delay, callstack, thres)
10       barrier_points.append(bp)
11   return barrier_points
12
13 def search_bp(test, cp, delay, callstack, thres):
14   for callsite in get_callsites(callstack):
15     containing_meth = get_method_name(callsite)
16     start_line = start_line_number(containing_meth)
17     end_line = callsite
18     for line in range(end_line, start_line):
19       test_res = run_with_crit_yield(test, cp,
20         line, delay, thres)
21       if test_res == PASS:
22         return line # Identified barrier point
23   return None # Unsuccessful
```

Figure 4: Pseudocode for BarrierSearch.

runs the test with a delay injected at the critical point while inserting at the potential barrier point a loop that continuously calls `Thread.yield`, which hints to the system that this thread will relinquish execution to other threads, allowing the other thread time to execute the critical point (Line 19). This loop is conditioned to keep calling `Thread.yield` until the critical point has been executed. Lines 10 to 12 in Figure 1 illustrate what this loop looks like. If the test passes after this change, then we have identified the barrier point (Line 22 in Figure 4). If the test still fails or the test times out (we use a timeout value of three minutes), BarrierSearch moves on to the preceding location. The intuition is that the exception may have been thrown based off of wrong data computed earlier in the code. As such, no matter how much the execution pauses at the later location, there is no way to recover from the wrong computation having already occurred. If none of the locations before the location where the exception originates are barrier points, the function moves up the exception call stack to the caller and repeats the search. It continues checking potential barrier points through the prior locations relative to that call site, moving up the call stack again if necessary, until identifying a barrier point.

3.2.1 Tracking Multiple Executions of Critical Point. Note that this search process assumes that the barrier point should wait until the critical point is executed just once. However, sometimes the critical point should be executed multiple times, and the execution should not proceed past the barrier point until a sufficient number of executions have occurred. If we cannot identify a barrier point by the time we have searched through the entire exception call stack, the next step is to count how many times the critical point is executed in a passing test execution without any injected delays. (Line 8). The count `thres` can be used by `search` to wait at potential barrier points until the critical point has been executed that many times. If the test still does not pass, then FlakeSync is

unsuccessful at repairing the async flaky test corresponding to the critical point.

3.3 Final Output

Figure 1 illustrates a repaired flaky test. FlakeSync introduces a field on Line 33 to keep track of when the critical point has been executed, and FlakeSync introduces a loop to wait until the critical point is executed at the barrier point on Lines 10 to 12.

FlakeSync performs these transformations dynamically as the test runs to confirm that using the critical point and barrier point can repair the async flaky test. A developer can use the same information to modify their source code with such transformations to form a patch for their flaky test. However, both critical point and barrier point must be in the developer's own code for them to apply this patch. A developer would not be able to modify third-party library code. Further, given that the goal of these changes is to synchronize execution for a specific async flaky test, making changes to general library code that could be used by many other projects is undesirable. In such a scenario, a developer can continue to use FlakeSync as a runtime environment that dynamically modifies code to ensure proper synchronization using the critical point and barrier point.

4 EXPERIMENTAL SETUP

4.1 Research Questions

To evaluate the effectiveness of FlakeSync, we address the following research questions:

- **RQ1:** How many async flaky tests can FlakeSync repair?
- **RQ2:** What are the characteristics of the repaired tests?
- **RQ3:** What is the runtime cost of running FlakeSync?
- **RQ4:** How much runtime overhead is there from using the repaired test compared to the original test?
- **RQ5:** What are the developers' reactions to the patches that FlakeSync proposed?

We address RQ1 to see whether the intuition behind FlakeSync's repair strategies are effective for repairing async flaky tests. We address RQ2 to analyze the characteristics of the repaired async flaky tests, in particular whether they can be easily directly applied to the project code. We address RQ3 to show the cost of running FlakeSync. We address RQ4 to check how much overhead is incurred from using the repaired test compared to running the original test; there might be some trade-offs in terms of removing flakiness and cost of testing. Finally, we address RQ5 to discuss whether developers find the type of patches that FlakeSync proposes acceptable.

4.2 Evaluation Dataset

For our evaluation, we use known flaky tests from a public dataset of flaky tests, IDoFT [4]. These flaky tests come from open-source Maven projects on GitHub and are categorized based on the flaky-test detection techniques that found them. The most prominent categories are order-dependent (OD) and non-order-dependent (NOD) flaky tests, found using iDFlakies [3, 23]. OD tests' outcomes depend on the test order in which they are run [29, 45], while NOD tests' outcomes do not. FlakeSync is designed to repair async flaky tests, so FlakeSync would unlikely work for OD tests. As such, we start with the 300 flaky tests marked as NOD. For each NOD test, we attempt to build and run the test at the commit where the flaky test was detected. We filter out tests that we cannot build or run due to out-of-date or missing library dependencies, leading to 221 tests. We then run each test individually and track whether there are any concurrent methods during execution (Section 3.1); we filter out tests that do not have any concurrent methods, as they are also unlikely to be async flaky tests. Ultimately, our evaluation dataset consists of 176 flaky tests from 23 open-source Maven projects and 37 modules[1]. Table 1 shows the breakdown of the flaky tests per module (column "# Flaky Tests") and the commit SHA on which we evaluate (taken from IDoFT). We provide an ID per module for ease of presentation in future tables. The table also shows for each module the number of lines of code ("KLOC"), average runtime per flaky test ("Test Runtime (sec)"), and average number of concurrent methods we found per flaky test ("# Conc Meth").

```
1  public class TestTaskExecutor {
2    ....
3    @Test
4    public void shouldBeFasterWhenRunningMultipleSlowTasks() {
5      ...
6      final long start = System.currentTimeMillis();
7      victim.submit(callables);
8      final long end = System.currentTimeMillis();
9
10     final long delta = end - start;
11     LOG.debug("Execution took: {}", delta);
12     assertTrue(delta < delay * times);
13   }
14 }
```

Figure 5: Timeout flaky test from wro4j/wro4j

4.3 Running Environment

We run FlakeSync on each flaky test individually. FlakeSync uses ASM [1] to instrument code to inject delays and to modify code for synchronizing critical point and barrier point. For our experiments, we set a maximum timeout of 12 hours for FlakeSync to run on each test. After FlakeSync repairs a test, we rerun the repaired test 100 times to confirm that it always passes. We configure the `INITIAL_DELAY` to be 100ms and the `MAX_DELAY` to be 51200ms (Figure 3). We set a 3 minutes timeout for each run for barrier point search. We run our experiments in a Ubuntu 20.04 Docker container configured to use 4 CPU and 4GB RAM.

5 RESULTS

5.1 RQ1: Repairing Async Flaky Tests

Table 2 shows the results of running FlakeSync on the 176 tests that we use in our evaluation. This table shows for each module the number of flaky tests for which FlakeSync could identify the critical point (column "# CSS") and the number of flaky tests for which FlakeSync could identify the barrier point and therefore repair the flaky test (column "# Fixed Tests"). Note that FlakeSync can only identify a barrier point for the tests where it was able to identify a critical point. Overall, FlakeSync could repair 67 flaky tests.

[1] A Maven project can have multiple modules

Table 1: Breakdown of projects and async flaky tests used in evaluation

ID	Project	Module	SHA	KLOC	# Flaky Tests	Test Runtime (sec)	# Conc Meth
M1	Accenture/mercury	platform-core	8586dc7	19.27	1	15.00	622.50
M2	Alluxio/alluxio	.	78c063a	59.68	17	5.42	893.25
M3	TooTallNate/Java-WebSocket	.	fa3909c	10.92	52	4.30	103.59
M4	activiti/activiti	activiti-engine	b11f757	111.80	12	10.30	2219.41
M5	activiti/activiti	activiti-spring	b11f757	3.04	1	7.56	2842.00
M6	alibaba/wasp	.	b2593d8	153.50	1	5.67	59.00
M7	apache/dubbo	dubbo-config-api	737f7a7	8.34	4	6.33	498.50
M8	apache/dubbo	dubbo-remoting-netty	737f7a7	1.56	1	13.38	265.00
M9	apache/dubbo	dubbo-rpc-dubbo	737f7a7	4.69	5	4.12	364.20
M10	apache/dubbo	dubbo-rpc-http	737f7a7	0.39	1	3.81	134.00
M11	apache/dubbo	dubbo-rpc-rest	737f7a7	1.03	3	4.54	94.00
M12	apache/httpcore	httpcore	49247d2	21.05	2	2.16	18.50
M13	apache/httpcore	httpcore-nio	49247d2	19.19	9	2.15	327.66
M14	apache/incubator-uniffle	common	6fb2a9a	10.68	1	6.32	46.00
M15	cescoffier/vertx-completable-future	.	011d3cd	2.10	1	2.67	9.00
M16	davidmoten/rxjava2-extras	.	7663d3b	13.69	3	7.06	207.00
M17	doanduyhai/Achilles	integration-test-2_1	e3099bd	8.97	12	15.04	6042.08
M18	doanduyhai/Achilles	integration-test-2_2	e3099bd	0.86	7	13.37	6066.00
M19	doanduyhai/Achilles	integration-test-3_10	e3099bd	0.51	2	12.27	6053.00
M20	doanduyhai/Achilles	integration-test-3_7	e3099bd	0.19	6	12.36	6068.16
M21	doanduyhai/Achilles	integration-test	f52f7ec	2.56	1	8.88	4802.00
M22	elasticjob/elastic-job-lite	elasticjob-infra-common	9afe466	2.12	1	4.91	6.00
M23	feroult/yawp	yawp-testing-appengine	b3bcf9c	0.21	1	3.51	516.00
M24	flaxsearch/luwak	luwak	c27ec08	7.53	2	3.60	44.00
M25	fluent/fluent-logger-java	.	2e5fdf2	1.67	1	5.39	602.00
M26	javadelight/delight-nashorn-sandbox	.	da35edc	2.49	1	3.07	17.00
M27	kagkarlsson/db-scheduler	db-scheduler	0e9f2a8	10.39	4	3.53	32.25
M28	kagkarlsson/db-scheduler	.	4a8a28e	3.74	2	7.67	434.00
M29	nlighten/tomcat_exporter	client	bc6a2d2	0.95	1	7.13	1111.00
M30	qos-ch/logback	logback-classic	0f57531	21.63	3	4.05	380.66
M31	qos-ch/logback	logback-core	0f57531	25.96	2	2.87	36.50
M32	square/okhttp	okhttp-tests	129c937	11.55	1	2.78	129.00
M33	undertow-io/undertow	core	ac7204a	49.28	4	4.03	817.75
M34	undertow-io/undertow	websockets-jsr	d0efffa	6.98	4	4.77	1376.50
M35	vmware/admiral	registry	e4b0293	1.59	5	3.96	1864.20
M36	vmware/admiral	compute	e4b0293	40.47	1	8.13	2475.00
M37	wro4j/wro4j	wro4j-core	7e3801e	23.34	1	2.48	9.00
Total/ Mean				**17.9**	**176**	**6.40**	**1434.28**

Of the two tests for which FlakeSync could not identify the critical point, we found that these two tests crashed due to hitting memory limits after running a long time with the injected delays. If given more resources, it is possible FlakeSync would identify the critical point for these tests. Of the remaining 174 tests for which FlakeSync could identify a critical point, FlakeSync identified a barrier point, and therefore could repair 67 tests, i.e., 38.5% of all flaky tests in our dataset. However, when we inspected test code and the failure logs of these flaky tests, we found that many of them are written in such a way that they rely on absolute runtime. For example, Figure 5 shows a flaky test from module M37 that measures the exact time taken to run a block of code (Lines 6-8) and asserts that this time is less than a constant value (Line 12). While this test is flaky due to timing issues, it does not fail due to lack of synchronization between code. Rather, as long as some part of execution simply runs slower, e.g., on a slow machine, the test would fail. Another example of such flaky tests are those that fail due to a constant timeout set by the developer, so the test may fail if the machine is slower than expected. FlakeSync is not designed to repair such flaky tests.

Our inspection shows that 94 tests rely on exact timing; the remaining 80 tests are async flaky tests (column "# Async Flaky

Table 2: Breakdown of tests that FlakeSync repairs

ID	# CSS	# Fixed Tests	# Async Flaky Tests	Overhead (X)
M1	1	1	1	1.00
M2	15	6	9	1.08
M3	52	7	7	1.11
M4	12	0	2	n/a
M5	1	1	1	1.02
M6	1	1	1	1.05
M7	4	0	0	n/a
M8	1	1	1	0.53
M9	5	2	2	0.97
M10	1	0	0	n/a
M11	3	0	0	n/a
M12	2	1	1	1.33
M13	9	0	5	n/a
M14	1	1	1	0.88
M15	1	0	0	n/a
M16	3	2	2	1.12
M17	12	12	12	1.25
M18	7	7	7	1.40
M19	2	2	2	1.90
M20	6	6	6	1.01
M21	1	0	1	n/a
M22	1	1	1	0.95
M23	1	1	1	0.86
M24	2	1	2	1.20
M25	1	1	1	0.99
M26	1	1	1	0.91
M27	4	0	0	n/a
M28	2	1	1	1.04
M29	1	1	1	1.03
M30	3	2	2	1.02
M31	2	1	1	1.04
M32	1	1	1	1.05
M33	4	0	0	n/a
M34	4	3	3	1.09
M35	5	3	4	1.06
M36	1	0	0	n/a
M37	1	0	0	n/a
Total/ Mean/ Median	**174**	**67**	**80**	**2.26 1.00**

Tests" in Table 2). Therefore, FlakeSync repairs 83.75% async flaky tests. A developer could repair the 94 other tests that rely on exact timing simply by increasing the hard-coded time values in the tests.

We inspected further the remaining 13 async flaky tests that FlakeSync does not repair. We find that there are six tests whose failures were nondeterministic, meaning that we could not reliably reproduce the failure while searching for the critical point. For two tests, we reach the maximum time of 12 hours that we allocated for running FlakeSync for our experiments (Section 4). For five tests, our inspection found that they fail due to network I/O operation errors. These errors occur when a socket connection times out and is reset. While there is no constant waiting time in the code, conceptually these tests are similar in nature given that the socket connection times out after some default time.

> ***RQ1:*** *FlakeSync can repair 67 out of the 80 (83.75%) async flaky tests in our dataset.*

5.2 RQ2: Characteristics of Repairs

A developer can only apply the changes that FlakeSync proposes to repair an async flaky test as a patch to their project code if both the critical point and barrier point are in their own code (Section 3.3). As such, we characterize the code changes that FlakeSync proposes for the 67 async flaky tests that FlakeSync could repair based on the locations of the critical points and barrier points.

We find that only six tests have the critical point in third-party library code. As such, we see that async flaky tests generally need to synchronize on critical computations that occur in their own project code and do not need to rely on such computations in third-party code. Having the critical point in the same project code also means it can be easier for developers to understand the interactions during test execution and how exactly the flaky failure can occur. Meanwhile, we find that 31 tests have their barrier point in third-party library code. The reason that so many barrier points are outside project code is that FlakeSync starts its search for the barrier point from the origin of the exception, which usually ends up in third-party library code (Section 3.2). As such, it usually ends up identifying a valid barrier point within third-party library code before reaching the project code that calls into the library. If we adjust BarrierSearch to only consider potential barrier points in project code, we find that 61 tests can have both their critical point and barrier point to be in the project code itself, meaning a patch involving the repair can be directly applied.

Finally, we analyze how often we need to have the barrier point wait until the critical point is executed more than once (Section 3.2.1). We find that seven tests require waiting for the critical point to be executed multiple times, with the mean number of times to be 3.19. Most tests require only waiting until the critical point is executed just once, so it is reasonable that FlakeSync prioritizes its search to first consider only needing the critical point executed just once.

> ***RQ2:*** *FlakeSync can repair 61 of 67 tests with changes that can be directly applied as a patch to the project code. Most async flaky tests also only need the barrier point to wait until the critical point is executed just once (60 out of 67).*

5.3 RQ3: FlakeSync Runtime

Figure 6 shows the amount of time that FlakeSync takes to repair tests in each module. Overall, FlakeSync on average takes 126.62 minutes per async flaky test (whether successfully repaired or not). FlakeSync often takes longer to run if it cannot successfully repair the test. Considering only the tests that FlakeSync successfully repairs, the average runtime is 103.52 minutes. From the figure, we observe a few tests that require much higher runtime, as median

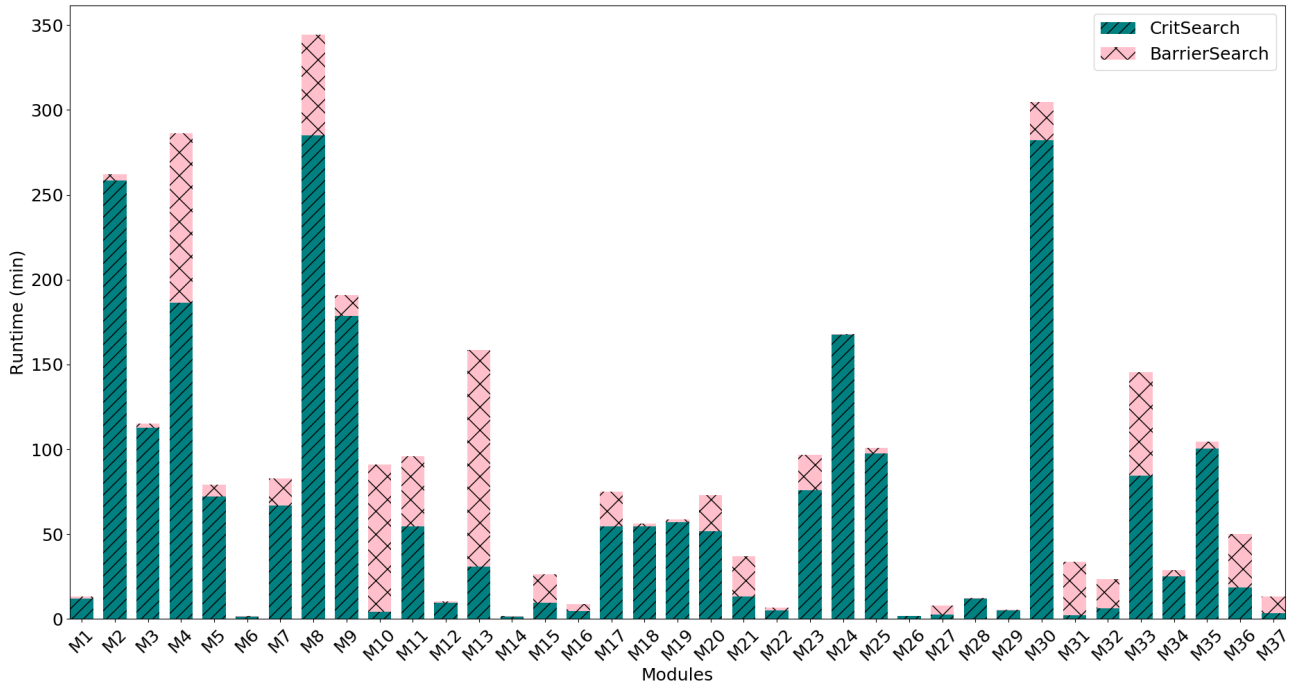

Figure 6: Runtime breakdown for FlakeSync per module

runtime per test is 58.77 minutes. While this amount of time to run FlakeSync is very high compared against the time to run a single test, note that this search process for critical point and barrier point is completely automated. A developer could run FlakeSync just once to obtain a permanent way to repair their async flaky test.

The figure also shows the breakdown of FlakeSync's runtime across CritSearch and BarrierSearch. On average, CritSearch takes 104.39 minutes (median 30.73 minutes) to run per test while the BarrierSearch takes 22.23 minutes (median 7.00 minutes) per test. CritSearch tends to contribute the most to the runtime cost of using FlakeSync, likely due to its minimization and root-method search steps. For example, in module M30 that on average take the most time, we see that the main reason for the high runtime cost is due to the large delay amount needed, with on average 8000ms at each delay location per test to reproduce the failure. In addition, the call stacks obtained from those delay locations have many call sites, so searching for the root method takes longer as well (Section 3.1.3). Future work may consider how to speed up this process by more quickly jumping to the likely best delay locations and root method.

There are a few cases where BarrierSearch takes much more time than CritSearch. For example, in module M13, BarrierSearch takes much more time because it cannot obtain a barrier point. In such cases, BarrierSearch has to essentially run twice across all candidate barrier points, since if it cannot find a barrier point with threshold 1 then it has to run again with a different threshold (Section 3.2.1), ultimately still not identifying a barrier point.

> ***RQ3:*** *We find that on average FlakeSync takes 126.62 minutes and on median 58.77 minutes to repair an async flaky test.*

5.4 RQ4: Overhead of Repaired Tests

Table 2 also shows per module under the column "Overhead (X)" the average overhead of the repaired async flaky tests, measured as the ratio of the runtime of the repaired test over the runtime of the original test, e.g., how many times slower is the repaired test. We report "n/a" for modules where FlakeSync could not repair any async flaky tests. Overall, the average overhead of a repaired test is 2.26X of the original test runtime (shown in the final row). However, there are a few modules with particularly high overhead, and median overhead is just 1.00X (also shown in the final row).

However, for most tests, the changes add barely any overhead. We even observe some cases where the overhead is seemingly less than 1.00X, but these differences are likely due to noise. Overall, the low overheads highlight how repairing by synchronization does not substantially slow down test execution.

> ***RQ4:*** *The repaired async flaky test takes on median 1.00X the runtime of the original test.*

5.5 RQ5: Developers' Reactions

To evaluate how developers react to FlakeSync's patches for their flaky tests, we send pull requests to the developers based on the proposed patches. Since our evaluations thus far are on known flaky tests found on old commits, we first rerun FlakeSync on these tests found in the latest commit of the projects. In moving to the latest commit, we encountered several issues. First, we find some async flaky tests have already been fixed by this latest version of the project, so we do not need to submit any pull request. Second,

we find that the known flaky test no longer exists, likely already deleted. We attempted to match tests based on the test name in the latest commit, but we could not always find a matching test. Third, for some projects, we could no longer build them at this latest commit. Furthermore, some projects shifted from Maven to Gradle; our current implementation of FlakeSync only supports Maven projects, so we could not run FlakeSync on these projects. Finally, for some projects, FlakeSync does not identify a critical point and barrier point both in project code, so we cannot prepare a patch to their codebase. After filtering out projects based on these constraints, we obtain 10 projects on which we could run FlakeSync and send pull requests. As to not overwhelm developers with many pull requests before confirming they indeed want such patches, we limit to creating one pull request per flaky test per project.

We submitted a total of 10 pull requests for 10 projects. Out of these 10 submitted pull requests, 3 pull requests have already been accepted [2, 5, 6]; no pull request has been rejected yet.

For one of the accepted pull requests, the developers merged the changes without too much feedback. For another pull request, we found that developers recognized that incorporating our patch would eliminate the necessity of maintaining hardcoded sleep durations. Consequently, upon merging our proposed changes, they took the initiative to remove the predetermined sleep times that had previously been implemented. For another pull request (shown in Figure 1), we found that the developers were actually already working on trying to repair the flaky test via increased wait time. However, they found the test would still sometimes fail in a continuous integration environment, as waiting a constant amount of time is unreliable. When we submitted our pull request, the developers commented that this patch that waits on a specific condition is much more desirable than changing the existing sleep time. The developers merged this pull request with the comment "LGTM, thanks, it's really great work". These comments showcase how FlakeSync's strategy for repairing async flaky tests is more desirable than simply increasing wait times.

> ***RQ5:*** *We submitted 10 pull requests that repair async flaky tests, with 3 accepted and the rest pending; no pull requests have been rejected thus far.*

5.6 Limitations

FlakeSync is designed to only repair async flaky tests and not other flaky tests. A developer would have to know ahead of time to only use FlakeSync for such tests, or rely on some automated techniques to determine the category of flaky test [8, 34]. FlakeSync assumes it should repair async flaky tests by making them pass, but it is possible the failure is due to a real concurrency bug, so a developer would want a consistent failure. Currently, FlakeSync cannot distinguish whether the failure is due to test flakiness or a concurrency bug that needs to be fixed. Future work can investigate how to automatically distinguish the two. Note that a developer could use CritSearch only to just reproduce the failure consistently. The delay locations may indicate where the bugs are located.

FlakeSync heavily relies on being able to reproduce a flaky-test failure to proceed; it cannot repair an async flaky test if injecting delays does not reproduce the failure. Part of our process to reproduce the flaky-test failure involves identifying concurrently executing methods within the same JVM, so our current implementation deals specifically with multithreading. We currently do not track concurrently executing code across different processes. After we can reproduce the failure, if the async flaky test is failing due to reliance on some absolute runtime, e.g., fails if some part of the code does not execute within a constant set amount of time, then FlakeSync cannot synchronize code as to prevent the failure. FlakeSync also cannot handle cases where an async flaky test needs to synchronize on multiple critical points at once. Finally, a developer can only directly apply a patch on their code using FlakeSync's reported critical point and barrier point if both are in their own code, not in third-party libraries. In this case, developers can use FlakeSync's runtime framework to dynamically insert the critical point and barrier point during execution of the async flaky test.

6 THREATS TO VALIDITY

External Validity. We construct our evaluation dataset by filtering from a previous dataset of known flaky tests. Our filtering is systematic but excludes flaky tests that FlakeSync was never designed to repair, e.g., OD tests. One part of our filtering is that we remove any tests that do not execute concurrent methods. It is possible that we exclude some tests that actually do execute concurrent methods due to bugs in our concurrent method search. We develop our approach to find concurrent methods based on past work [33], and we filtered out 45 tests out of an initial set of 300 tests due to lack of concurrent methods. Ultimately, we believe our evaluation set is still representative of async flaky tests.

FlakeSync may find different delay locations and amount of delay to inject when run on different machines, leading to different critical points and barrier points. Our manual inspection and modification of code confirms that the identified critical points do make the tests fail with the specified delays, and that the critical points and barrier points also allow the test to pass with the injected delays.

Internal Validity. Our implementation of FlakeSync involves instrumentation that may introduce changes beyond delaying execution, and failures may be due to these unintentional changes. To check, we manually modify source code to inject the delays at the identified critical points, and we also check that we can apply the patches modifying critical point and barrier point when possible.

Construct Validity. We confirm a test to be repaired if the test always passes after 100 times even with injected delays. While it is possible there may still be failures after more reruns, we believe this check to be sufficient given the large number of reruns and that the test always fails with injected delays without the patch. Further, we also send patches as pull requests to developers to confirm.

7 RELATED WORK

Luo et al. presented the first empirical study on the root causes of flaky tests in open-source projects [29]. They found the top reasons for flaky tests include async-wait and concurrency, which we call async flaky tests. Separate groups conducted additional studies to better understand flaky tests [13, 18, 22, 24, 25] with similar

findings on top causes. Companies also report problems with flaky tests [19–21, 30, 32].

Prior work on automatically repairing flaky tests has focused mostly on those with easily reproducible flaky behavior. Shi et al. [39] proposed iFixFlakies for fixing order-dependent (OD) flaky tests in Java, where the failure can be easily reproduced by running in a specific test order. Li et al. [27] followed up with technique ODRepair to fix OD flaky tests that iFixFlakies cannot, and Wang et al. [42] adapted iFixFlakies to fix Python OD flaky tests. Beyond OD flaky tests, Zhang et al. [44] proposed DexFix to fix flaky tests that are dependent on specific implementations of library API. Dutta et al. [12] proposed FLEX to fix flaky tests in machine-learning applications whose assertions are too strict. Our work focuses on specifically automatically repairing async flaky tests.

Prior work found developers would try to repair async flaky tests by inserting wait times [22, 29, 31]. For example, Lam et al. [22] found that Microsoft developers would need to guess at how long to wait to mitigate flakiness, but their wait times may be longer than needed. Lam et al. then proposed technique FaTB to adjust the existing wait times to reduce testing time while still ensuring the tests do not fail. FlakeSync proposes a different way to repair async flaky tests by instead synchronizing the test execution. Such a strategy avoids issues with inserting or changing wait times, where machine load variance can influence the effectiveness of the constant wait times. Furthermore, FlakeSync automatically identifies the critical points and barrier points, whereas FaTB relies on a developer already having identified where to insert the wait time (akin to the barrier point that FlakeSync identifies). It is possible to then use FlakeSync to identify the barrier point and insert the wait time instead of synchronizing and waiting for the critical point to execute. However, this strategy still runs into the other issues involving constant wait times, namely not knowing exactly how long to wait and slowing down test execution more than necessary.

A wide range of work focuses on detecting flaky tests. Lam et al. [23] proposed iDFlakies [3] for detecting OD flaky tests. Shi et al. [38] proposed NonDex [7, 17] for detecting flaky tests that are dependent on specific implementations of library API. Flaky tests detected by both tools constitute a large part of the IDoFT dataset [4] that we use in our evaluation. We focus on tests marked NOD, because they are likely async flaky tests. Alshammari et al. [9] proposed FlakeFlagger, a machine-learning model to predict whether a test is flaky. Silva et al. [40] proposed Shaker to detect async flaky tests by purposefully introducing large amounts of stress on the machine's CPU and memory while running tests, simulating a machine with constrained resources. Other tools that detect flaky tests include DeFlaker [10], which detects flaky tests as those that newly fail after changes yet do not cover changed code, and FLASH [11], which detects flaky tests in machine learning applications by manipulating underlying random number generators. In our work, we focus on repairing flaky tests once they have been detected.

Researchers have also proposed tools to help debug flaky tests. Lam et al. [21] proposed RootFinder to find the root cause of flakiness by comparing passing and failing execution traces of the same test. Researchers at Microsoft proposed FlakeRepro [26] to reproduce concurrency-related flaky tests in C# applications by injecting delays at locations related to shared memory accesses, checking whether the test can reliably fail. Rahman et al. also proposed FlakeRake [36] to reproduce flaky-test failures in Java applications by injecting delays at timing-related Java API calls. FlakeSync also relies on injecting delays as part of identifying the critical point, but it also identifies the barrier point to repair the test.

Our work is related to the area of concurrency bug detection, and we borrow ideas from prior work in this area. Pradel et al. [33] proposed automatically generating tests to detect concurrency bugs by first executing the code with some initial inputs to detect concurrent methods and then generating tests that target executing pairs of methods found to be concurrent to each other. We rely on similar logic to determine concurrent methods. Much prior work in concurrency bug detection rely on injecting delays at key locations within code [14, 15, 28, 37, 41]. For example, Li et al. [28] proposed TSVD to search for thread-safety violations by injecting delays at supposedly thread-safe read/write API calls in C# applications. Rahman et al. [35] implemented TSVD for Java while also enhancing it to consider read/write operations to object fields. We also inject delays in code, but at calls to concurrent methods. Our goal is not to detect concurrency bugs but rather to make tests fail more consistently to aid in repair. The async flaky tests we focus on may be flaky due to concurrency issues, but we assume the failure does not indicate a true concurrency bug in code, and we add synchronization to ensure the test passes.

8 CONCLUSIONS

We propose FlakeSync, a technique for automatically repairing async flaky tests. The intuition is that such tests fail due to lack of synchronization between concurrently running code. FlakeSync repairs async flaky tests by identifying a critical point and barrier point, where the barrier point waits until the critical point has been executed before proceeding. Our evaluation on known flaky tests from a prior dataset shows that FlakeSync can automatically repair 83.75% of the async flaky tests. The repaired async flaky tests have a low median runtime overhead compared to the original test runtime. We submitted 10 pull requests based on FlakeSync's patches, with 3 accepted pull requests and none rejected thus far.

In the future, we plan to improve FlakeSync's CritSearch through better heuristics that allow for more efficient search for the delay locations, leading to finding the critical point. We plan to similarly improve the runtime of BarrierSearch through prioritization of which locations to check for valid barrier points. We also plan on improving the FlakeSync tool to also automatically modify source code using the critical point and barrier point when applicable. Finally, we plan on extending FlakeSync's core algorithm to consider multiple critical points and barrier points that together are needed to repair the flaky test.

DATA AVAILABILITY

Our data is available at https://sites.google.com/view/flakesync/home, containing code, scripts, and raw data. An artifact to reproduce our results is available at https://zenodo.org/records/10460139.

ACKNOWLEDGMENTS

We would like to acknowledge NSF grant no. CCF-2145774 and the Jarmon Innovation Fund.

REFERENCES

[1] ASM. https://asm.ow2.io/.
[2] elasticjob-2242. https://github.com/apache/shardingsphere-elasticjob/pull/2242.
[3] iDFlakies. https://github.com/idflakies/iDFlakies.
[4] IDoFT. http://mir.cs.illinois.edu/flakytests.
[5] incubator-uniffle-1023. https://github.com/apache/incubator-uniffle/pull/1023.
[6] mercury-136. https://github.com/Accenture/mercury/pull/136.
[7] NonDex. https://github.com/TestingResearchIllinois/NonDex.
[8] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon. FlakyCat: Predicting flaky tests categories using few-shot learning. In *International Conference on Automation of Software Test*, 2023.
[9] A. Alshammari, C. Morris, M. Hilton, and J. Bell. FlakeFlagger: Predicting flakiness without rerunning tests. In *International Conference on Software Engineering*, 2021.
[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*, 2018.
[11] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *International Symposium on Software Testing and Analysis*, 2020.
[12] S. Dutta, A. Shi, and S. Misailovic. FLEX: Fixing flaky tests in machine-learning projects by updating assertion bounds. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.
[13] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer's perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
[14] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1), 2002.
[15] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.
[16] M. Gruber, S. Lukasczyk, F. Krois, and G. Fraser. An empirical study of flaky tests in python. In *International Conference on Software Testing, Verification, and Validation*, 2021.
[17] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *International Symposium on Foundations of Software Engineering (Tool Demonstrations Track)*, 2016.
[18] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *International Conference on Software Testing, Verification, and Validation*, 2022.
[19] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*, 2018.
[20] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon. Modeling and ranking flaky tests at Apple. In *International Conference on Software Engineering, Software Engineering in Practice*, 2020.
[21] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *International Symposium on Software Testing and Analysis*, 2019.
[22] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. A study on the lifecycle of flaky tests. In *International Conference on Software Engineering*, 2020.
[23] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*, 2019.
[24] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *International Symposium on Software Reliability Engineering*, 2020.
[25] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell. A large-scale longitudinal study of flaky tests. *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 4(OOPSLA), 2020.
[26] T. Leesatapornwongsa, X. Ren, and S. Nath. FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.
[27] C. Li, C. Zhu, W. Wang, and A. Shi. Repairing order-dependent flaky tests via test generation. In *International Conference on Software Engineering*, 2022.
[28] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Symposium on Operating Systems Principles*, 2019.
[29] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*, 2014.
[30] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *International Conference on Software Engineering, Software Engineering in Practice*, 2019.
[31] J. Malm, A. Causevic, B. Lisper, and S. Eldh. Automated analysis of flakiness-mitigating delays. In *International Conference on Automation of Software Test*, 2020.
[32] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*, 2017.
[33] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation*, 2012.
[34] S. Rahman, A. Baz, S. Misailovic, and A. Shi. Quantizing large-language models for predicting flaky tests. In *International Conference on Software Testing, Verification, and Validation*, 2024.
[35] S. Rahman, C. Li, and A. Shi. TSVD4J: Thread-safety violation detection for Java. In *International Conference on Software Engineering (Tool Demonstrations Track)*, 2023.
[36] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell. Automatically reproducing timing-dependent flaky-test failures. In *International Conference on Software Testing, Verification, and Validation*, 2024.
[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.
[38] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *International Conference on Software Testing, Verification, and Validation*, 2016.
[39] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
[40] D. Silva, L. Teixeira, and M. d'Amorim. Shake it! Detecting flaky tests caused by concurrency with Shaker. In *International Conference on Software Maintenance and Evolution*, 2020.
[41] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.
[42] R. Wang, Y. Chen, and W. Lam. iPFlakies: A framework for detecting and fixing Python order-dependent flaky tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*, 2022.
[43] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 2002.
[44] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *International Conference on Software Engineering*, 2021.
[45] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*, 2014.