DETERMINING FLAKY TESTS FROM TEST FAILURES

BY

LAMYAA ELOUSSI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Associate Professor Darko Marinov

ABSTRACT

Automated regression testing is widely used in modern software development. Whenever a developer pushes some changes to a repository, tests are run to check whether the changes broke some functionality. When previously passing tests fail, the most recent changes are typically suspected, and developers invest time and effort to debug those changes. Unfortunately, new test failures may not be due to the latest changes but due to non-deterministic tests, popularly called *flaky tests*, that can pass or fail even without changes to the code under test. Many projects have such flaky tests, which can cause developers to lose confidence in test results. Therefore, developers need techniques that can help them determine whether a test failure is due to their latest changes and warrants their debugging, or whether it is due to a flaky test that should be potentially debugged by someone else.

The most widely used technique for determining whether a test failure is due to a flaky test is to rerun the failing test multiple times immediately after it fails: if some rerun does pass, the test is definitely flaky, but if all reruns still fail, the status is unknown. This thesis proposes three improvements to this basic technique: (1) postponing the reruns, (2) rerunning in a new runtime environment (e.g., a new JVM for Java tests), and (3) intersecting the test coverage with the latest changes. The thesis evaluates the cost of (1) and (2) and evaluates the applicability of (3) on 15 projects with a total of 2715 test classes, 10 of which contain previously known flaky tests. The results show that the proposed improvements are highly applicable and would be able to determine that more failures are due to flaky tests for the same or somewhat higher cost as rerunning failing tests immediately after failure.

To my parents.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Darko Marinov, for his guidance and support. The past two years working with Darko have been intense, often challenging, but at the same time very productive and full of learning. I truly admire his dedication to helping his students grow and reach their full potential.

I would like to thank my dearest office mate, Alex Gyori, for always lifting my spirits and making our office a fun place to work. I am also grateful to my other group mates, August Shi, Farah Hariri, and Owolabi Legunsen, who have made my time at Illinois more enjoyable. I would like to extend special thanks to Milos Gligoric who was a great mentor to me. I have learned a lot from Milos and I am sure he will become a phenomenal professor.

I would also like to thank Professor Vikram Adve and Alexandros Tzannes who offered me a broader experience of academic research. I would like to thank my other collaborators: Rohan Sharma, Rupak Majumdar, and especially Qingzhou Luo who provided help with this project. Thanks to Ashwini Joshi and Xinyue Xu who also provided help with the initial stages of this project.

This material is based on work partially supported by NSF Grant Nos. CNS-0958199, CCF-1012759, CCF-1421503, and CCF-1439957. My research assistantships were funded by Grant Nos. CNS-0958199, CCF-1012759, and CCF-1439957. I am also grateful for financial support from the Siebel Scholarship.

Finally, I would like to thank my friends and family. Special thanks to my friends Hajar Abbadi and Soumaya Graine for supporting me and keeping our friendship alive despite the distance, and to Eric for his love and support. Lastly, I would like to thank my parents for being supportive and always believing in me. I would not have come this far without their care and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Automated regression testing is widely used in modern software development. Whenever a developer pushes some changes to a repository, tests are run to check whether the changes broke some functionality. If all the tests pass, the developer continues with development and makes further changes. However, if some test fails, the developer needs to inspect more. Ideally, every test failure would be due to the latest changes that the developer made, so the developer could focus on debugging these changes.

Unfortunately, some test failures are not due to the latest changes at all but instead due to non-deterministic tests, popularly called *flaky tests*, that can pass or fail even for the same code under test. There are various sources of non-determinism which can cause tests to be flaky; our recent study [1] of a sample of over 200 flaky tests from 51 Apache projects showed that common sources include asynchronous communication, concurrency, test-order dependency, GUI events, network, time dependency, resource leaks, and more. Flaky tests are frequent (not only in Apache projects but in most large software) and can create big problems in development, as described by several researchers and practitioners [1–11].

The key problem with flaky tests is that the developers cannot rely on the simple pass/fail outcome of test runs to decide how to proceed with their development: *when a test fails, should the developer debug the failure or not*? A test failure usually indicates some fault, not necessarily in the code under test but potentially in the test code itself [12, 13]. However, if the developer knows that the failure is due to a flaky test, the developer may decide to not debug it, potentially reporting it to be debugged by someone else. In fact, if the developer knows that the failure is definitely *not* due to the latest changes, the developer could more confidently ignore the failure. Without that information, the developer could spend substantial time debugging the failure only to find that it is an unrelated flaky test [11].

It is important to not underestimate how even a small number of flaky tests can affect the development process, as the proverbial "bad apple(s) spoiling the whole bunch". For example, consider a test suite with 10,000 tests of which only 10 are flaky. Further assume that each flaky test can fail in just 0.1% of its runs, and that these failures are independent (although we will discuss later that, for many tests, the failures for a test are *not* independent but "bursty"). The chance of at least one flaky test failing in a run of the entire test suite is about 1% (more precisely $1 - 0.999^{10}$), hence one in hundred test-suite runs would be disturbed by some flaky test(s), although only one in thousand tests is flaky, and each can fail in only one in thousand runs. For an example of more concrete numbers about failures, the Google TAP system for regression testing had 1.6M test failures on average each day during a 15-month period between 2013 and 2014, and flaky tests caused 73K out of those 1.6M failures (4.56%) [1].

As a result, developers need techniques that help them determine whether a test failure is due to a flaky test or not. Currently, the most widely used technique is to rerun the failing test multiple times immediately after it fails: if some rerun does pass, the test is definitely flaky, but if all reruns still fail, the status is unknown. For example, several open-source testing frameworks have annotations to require reruns upon failure (e.g., Android has `@FlakyTest` [14], Jenkins has `@RandomFail` [15], and Spring has `@Repeat` [16]). The proprietary Google TAP system also has the `@flake` annotation that requires a test to be rerun, by default, up to three times upon failure [17, 18]. Even build systems offer such features, e.g., `rerunFailingTestsCount` was added in the recent Maven Surefire 2.18.1 [19]. (Chapter 7 discusses more about the existing systems and other related work.)

The number of reruns is influenced by a simple cost-benefit analysis. The cost of reruns is that they take time, and if all reruns fail, that time is wasted, providing no additional information. However, the benefit of reruns can be great: if some rerun passes, the test failure is labeled to be due to a flaky test and helps the developer to decide how to proceed. For this benefit, the developers are willing to pay the cost of reruns. A higher number of reruns increases the chance to find more failures to be due to flaky tests, but a higher number of reruns also increases the time that is wasted for test failures that do not stem from flaky tests but from real faults.

We propose several improvements to the basic technique of rerunning failing tests immediately after failure. We will refer to this basic technique as RERUN. Our improvements are motivated by two insights from our earlier study of flaky tests [1]. The first insight is that for many (but not all) flaky tests, which we call *bursty flaky tests*, the failures of a bursty flaky test t are not independent in time but rather occur nearby; in other words, if t fails once, t is very likely to fail again if rerun immediately. We discuss examples in Chapter 2, but for intuition, consider a flaky test that fails if some network service is down: if the service is down at some point in time, it is likely to be down in the next point in time. Moreover, for some bursty flaky tests, the failures depend on the program state: if t fails once, t is almost guaranteed to fail again unless the state is changed. The second insight is that developers can benefit not only from knowing whether a test is flaky or not but also whether the flaky test is affected by the latest changes or not. Namely, if a test t passed in the previous code revision and failed for the current revision, but the failed test execution does not depend on any of the changes between the revisions, then the test is definitely flaky.

More precisely, we make the following contributions:

(1) POSTPONE: We propose to postpone (some) reruns of failing tests to the end of the test-suite run. Rerunning a test somewhat later than it failed the first time can increase the chance to properly label the test as a flaky test (especially for bursty flaky tests). Moreover, at the end of the test-suite run, more information is available, e.g., the total number of failures: if a large fraction of tests failed, it is more likely that the latest changes broke something, so it may not be worthwhile to rerun the failing tests at all.

(2) FORK: We propose to rerun (some) failing tests in a new runtime environment to get a different starting state. For example, many Java projects run all their tests in the same Java Virtual Machine (JVM) [8]. However, a (bursty) flaky test that depends on the JVM state [1,7–9] can repeatedly fail when run multiple times, whether immediately after the first failure or at the end of the test-suite run. In contrast, running such a test in a new JVM can find that it passes and is thus a flaky test with some test-order or state dependency [1,7,9].

(3) INTERSECT: We propose a novel, simple technique that can find *both* that a test failure is due to a flaky test *and* that the failure does not depend on the latest code changes. That additional information allows the developers to make a better informed decision about

debugging the test failure. Without knowing that a failure depends on the latest changes, a developer may make the wrong decision to ignore some failure for a flaky test, even if it is actually affected by a change and may be failing due to some real fault.

(4) We evaluate the proposed extensions on 15 projects with a total of 2715 test classes, 10 of which contain previously known flaky tests. Specifically, we evaluate how the cost of Postpone and Fork compares to the cost of Rerun. We also compare the cost of Fork and Intersect, and evaluate how often Intersect can find that a test execution does not depend on the latest code changes. Our results show that the proposed improvements are highly applicable and can provide additional benefit for the same or somewhat higher cost than rerunning failing tests immediately after they fail.

CHAPTER 2

Examples of Flaky Tests

In this chapter, we show real examples of three types of flaky tests—non-bursty tests, bursty tests that do not depend on state, and bursty tests that do depend on the state. We also discuss how our proposed technique can be used to find different types of flaky tests, and especially how intersecting coverage with the recent changes can help to determine that a test failure is flaky.

2.1 Non-Bursty Flaky Test

Figure 2.1 shows a code snippet for an example non-bursty flaky test from Apache Hadoop. This test is one of several tests from the class `TestAuditLogs` that were flaky. The test reads a file created with the method `DFSTestUtil.createFile()`; the relevant snippet of code shows that the file content is generated through the call to `Random.nextBytes()`, where the value of every byte can be between -128 and 127 (inclusive). The test then reads these bytes as unsigned integers, so the value can be between 0 and 255 (inclusive). However, the test asserts that the first byte is strictly greater than 0. Hence, the test fails when the first byte is 0. This test remained flaky over several revisions (since it was written in revision f4555669 until it was fixed in revision 06d635cd). At any of the revisions in between, the test had 1/256 chance of failure. (This is one of the very rare flaky tests where we can precisely compute the chance of failure.)

Note that, for this example flaky test, the failures are completely *independent* (assuming that the random number generator generates all byte values uniformly): if a test failed once, it is not more or less likely to fail again if rerun immediately after the failure; the chance of failure remains the same, 1/256. Therefore, a technique that simply reruns the failed test

```
public class TestAuditLogs {
  ...
  @Test
  public void testAuditAllowed() throws Exception {
    ...
    // calls DFSTestUtil.createFile to create the file
    ...
    InputStream istream = userfs.open(file);
    int val = istream.read(); // reads as unsigned
    ...
    assertTrue("failed to read from file", val > 0);
  }
  ...
}

public class DFSTestUil {
  ... createFile(...) {
    ...
    byte[] toWrite = new byte[bufferLen];
    Random rb = new Random(seed); // seed depends on time!
    ...
    while (bytesToWrite>0) {
      rb.nextBytes(toWrite);
      ...
      out.write(toWrite, ...);
      ...
    }
    ...
  }
}
```

Figure 2.1: Code Snippet of an Example Non-Bursty Flaky Test

multiple times immediately after it fails has a very good chance to have the test pass and thus find that this failure is due to a flaky test.

2.2 Bursty Flaky Test

There are many flaky tests whose failures are *not* independent, meaning that if the test fails once, there is a high chance that it will fail again if it is rerun immediately. We call such tests *bursty*. Figure 2.2 shows a code snippet for an example bursty flaky test from Apache

```
public class TestWebHCatE2e {
  ...
  // setup for the test class
  @BeforeClass
  public static void startWebHcatInMem() {
    templetonServer = new Main(new String[] {"-D" + AppConfig.UNIT_TEST_MODE +
        "=true"});
    ...
    templetonServer.run();
    ...
  }
  ...
}

public class Main {
  public static final int DEFAULT_PORT = 8080;
  ...
  public void run() {
    int port = conf.getInt(AppConfig.PORT, DEFAULT_PORT);
    try {
      ...
      runServer(port);
      System.out.println("templeton: listening on port " + port);
      ...
    } catch (Exception e) {
      System.err.println("templeton: Server failed to start: " + e.getMessage());
      ...
    }
  }
}
```

Figure 2.2: Code Snippet of an Example Bursty Flaky Test

Hive. This test is one of several tests from the class `TestWebHCatE2e` that were flaky. The setup method for this test class starts a server (by creating an object of the class `Main()` and later calling the method `run()` on it) on the default port, set to 8080. However, if this port is in use, the server cannot be started, and the tests that try to connect to it fail.

Rerunning these failing tests immediately would probably not help in determining that the failures are due to a flaky test, because the port is likely to be still in use immediately after the tests fail. In other words, failures for a bursty flaky test usually come in "bursts": if a test fails once, it has a very high chance to fail again if run right away. However, if

```
public class TestRPCCompatibility {
  ...
  @Test // Compatible new client & old server
  public void testVersion2ClientVersion1Server() throws Exception {
    ... // no resetCache() here!
    Version2Client client = new Version2Client();
    ...
    assertEquals(3, client.echo(3));
  }
  ...

  @Test // equal version client and server
  public void testVersion2ClientVersion2Server() throws Exception {
    ProtocolSignature.resetCache();
    ...
    Version2Client client = new Version2Client();
    ...
    assertEquals(-3, client.echo(3));
  }
  ...
}
```

Figure 2.3: Code Snippet of an Example State-Dependent Bursty Flaky Test

reruns are postponed to the end of the test-suite execution, there is a higher chance that the test passes, in this example, the port could be freed.

While we discuss the intuition for the chance of failure and base the design of our technique on that intuition, attempting to precisely measure the chances of failure (both the overall chance of failure at any given time and also the conditional chance of failure assuming that another failure happened within some time δ) would be rather challenging. In this example, the chance of failure would depend on how the machine on which the tests are run is used, what other tests or processes may be running in parallel, which of them also use the same port number, how long they keep that port number, etc. We can artificially create scenarios where the conditional chance is 0% (e.g., make the system release port 8080 as soon as some test attempts to access it) or 100% (e.g., make the system always return that the port 8080 is in use after some tests attempts to access it). But the actual chance depends on the overall system that is not under the control of the test code in this example.

2.3 State-Dependent Bursty Flaky Test

Some flaky tests actually do depend on the state that is under the control of the test code. The most common case is of tests that depend on the test order [7–9]. Figure 2.3 shows a code snippet for an example flaky test of this category from Apache Hadoop. The test `testVersion2ClientVersion1Server` (t_1 for short) in the class `TestRPCCompatibility` became flaky when another test `testVersion2ClientVersion2Server` (t_2 for short) was added in revision 1210208. Both tests t_1 and t_2 call the method `echo()` in `Version2Client`, which transitively calls the `ProtocolSignature` class. This class keeps a cache that t_1 requires to be clear; however, when this test was written, it did not clear the cache before executing. The test t_2 that was added later clears the cache before executing, then pollutes this cache, but does not clear it after executing. Hence, the test t_1 passes if run before the test t_2, but t_1 fails whenever it runs after t_2. (Because t_2 clears the cache before executing, it does not fail whether it is run the first or the second.)

State-dependent flaky tests are generally bursty: when they fail once, there is a high chance that they will fail on subsequent reruns. In fact, in this example, t_1 is very likely to fail when run after t_2 in the same JVM, regardless of whether t_1 is run immediately after its first failure or at the end of the test-suite run. The only way for t_1 to pass in the same JVM is that another test, t_3, is run in between the first failure of t_1 and the subsequent rerun, and this test t_3 happens to clear the cache after executing. In contrast, rerunning t_1 in a new JVM creates a fresh state, with the clear cache, and t_1 passes right away.

2.4 Benefits of Coverage Intersection

While some bursty flaky tests are much more likely to pass in a new JVM, not all bursty flaky tests are like that. Let us again consider the example flaky test described in Section 2.2. The test fails whenever the default port 8080 is in use; if the port remains in use when the test is rerun, it will still fail, regardless of whether the test is rerun immediately after failing (RERUN), at the end of the test-suite run (POSTPONE), or even in a new JVM (FORK). However, our proposed technique can still determine in many cases that a test is flaky *even if it fails all the reruns.*

For example, consider the changes that developers made to the Apache Hive repository at revision f02a544d. The changes are to two classes in the code under test (`org.apache.hadoop.hive.ql.exec.FunctionRegistry` and `org.apache.hadoop.hive.ql.udf.generic.GenericUDFBaseCompare`) and to one test class (`org.apache.hadoop.hive.ql.exec.TestFunctionRegistry`) in a module different from where the test class `TestWebHCatE2e` is. However, when the developers make their changes and run[1] `TestWebHCatE2e`, it would still fail if port 8080 is in use, and even if the test is rerun, but the port remains in use during reruns, `TestWebHCatE2e` would fail all the reruns.

The developer who is unaware that the test failure is due to a flaky test could spend substantial time trying to debug how the latest changes affected the failures; after all, `TestWebHCatE2e` passed in revision d23d8502 and failed just now in the next revision f02a544d, so it could be that the changes indeed broke the test. Debugging would only reveal to the developer that the failure is unrelated to the changes. Our technique automates that step; when the failing test is rerun in a new JVM, our technique collects the coverage of what the test executes: if the failing test does not depend on any change, then it cannot be failing because of changes and must be a flaky test. This example illustrates how intersecting coverage and changes can help developers.

[1]We discuss in Chapter 7 the relationship with regression test selection that may not run all the tests.

CHAPTER 3

Technique

We next describe our technique for detecting which test failures are due to flaky tests. Figures 3.1 and 3.2 show the pseudo-code of the technique. Given some tests to run, `flakyDetector` returns a set of tests that failed and a subset of those failures that are due to flaky tests. The function also has several parameters that we discuss in Section 3.1. We present our technique for tests run on a Java Virtual Machine (JVM), although the technique is general and applies to any similar runtime environment, e.g., .NET CLR.

The first `for` loop corresponds to the existing RERUN technique that reruns failing tests immediately after the failure to determine if they are flaky. The function `rerun` embodies this simple technique: it takes a test `t` and an integer `N`, runs `t` up to `N` times, and if `t` passes in one of those reruns, it is labeled flaky. For example, this exact functionality has been recently included in the Maven Surefire Plugin (version 2.18.1) [19]. We propose three improvements to this simple technique.

(1) POSTPONE: The second `for` loop postpones the rerun until the end of the test-suite run. This loop can be skipped altogether if there are too many failures. Note that this rerun of failed tests still uses the same JVM runtime environment. Also note that the user can independently control the number of reruns performed immediately after the failure (`C_IMM`) and the number of reruns performed at the end of the test-suite run (`C_END`). As discussed earlier, postponing reruns until the end can find more flaky tests whose failures are bursty (e.g., if a test depends on a network that is down): while these tests may not be detected as flaky if they are rerun immediately, by the time the test is rerun later, the cause of failure might be resolved (e.g., the network connection might be back up).

The cost of each postponed rerun is expected to be similar as the cost of the corresponding immediate rerun, although in general postponed reruns could be more expensive (e.g., maybe

the test depends on some server being set up, and the immediate rerun could reuse the server from the immediate previous failure, whereas the postponed rerun needs to restart the server) or less expensive (e.g., maybe the test initially failed when a lot of memory was used, and by the end of the test-suite execution most of that memory is garbage collected, so the postponed reruns can be faster). Section 4.2 experimentally compares the cost of postponed and immediate reruns.

(2) FORK: The third `for` loop reruns the failed tests in a new JVM. Again, as discussed earlier, this helps detect flaky tests that would not be found if rerun multiple times in the same JVM, e.g., flaky tests that are due to test-order dependencies. If a test fails because another test ran before it and polluted the state, the failing test would likely keep failing no matter how many times it is rerun in the same JVM.

The cost of rerunning on a forked new JVM is higher than the cost of rerunning in the same JVM. In particular, the new JVM needs to start, it needs to load the classes that the test depends on, and it needs to set up the state before it starts the test. However, setting up the new state is precisely what provides the benefit: these reruns can find some failures to be flaky that would not be found otherwise. Section 4.3 evaluates the cost of reruns that use a new JVM.

(3) INTERSECT: The special case of rerunning in a new JVM is to collect the coverage of the test being rerun. If the test passes, it can be immediately concluded that the test is flaky. If the test fails again but its coverage does *not* intersect with the latest changes, it can be again concluded that the test is flaky, because the code that is executed by the test did not change between the two revisions, while the test changed behavior. If the test fails and its coverage *does* intersect with the latest changes, then nothing can be concluded definitely: maybe the test failure is due to a real fault in the latest changes, or maybe the test failure is due to some flaky cause and just by chance executed some changes.

The cost of running the new forked JVM *with* collecting coverage is expected to be higher than the cost of running the new forked JVM *without* collecting coverage, especially if one wanted to collect coverage at a fine granularity level. In principle, any coverage granularity level (e.g., classes, methods, statements, etc.) can be used. The challenge is to choose the granularity of the coverage (and thus code changes) such that the coverage is cheap to

collect and yet does not intersect with the changes relatively often. We propose to use class coverage: for each test, we collect what classes it depends on [20]. This granularity is good for several reasons. First, class coverage is cheap to collect compared to coverage of finer granularity such as methods or statements. Second, using finer granularity may not be safe, because it may not capture all dependencies [20–22]. Third, we have previously developed a publicly available tool to efficiently collect class coverage [20]. Section 4.4 evaluates the cost of reruns that collect class coverage. Moreover, that section also evaluates how often covered classes do not intersect with the changed classes, i.e., how often the intersection could conclude that a test is flaky if it were to fail.

3.1 Parameters

Figure 3.1 summarizes the technique with our proposed improvements. The technique has several parameters: C_IMM is the number of times to rerun each failing test immediately after failure (can be 0); THRESHOLD is the percentage of failures for which the failing tests need not even be rerun because the latest changes are likely broken and the rerun would only waste time without finding relevant flaky tests; C_END is the number of times to rerun the failing tests at the end of the test-suite run (can be 0); C_NEW is the number of times to rerun each failing test in a new JVM (can be 0); and MEASURE_COVERAGE indicates whether to collect coverage when rerunning the test in the first new JVM.

While the parameters can be set to arbitrary values, a good default configuration can have the values as follows. C_IMM can be a small number, say 1 or 2; for flaky tests that are not bursty, there is a high chance that they would pass in one of these runs for a relatively small number of C_IMM. THRESHOLD can be set to some relatively higher ratio, say 1%, if the reruns are more expensive, and relatively lower ratio, say 0.1%, if the reruns are cheaper. If a test still fails for all immediate reruns, C_END can be also a small number, say 1 or 2; even if the test is a bursty flaky test, there would be some higher chance for it to pass at this time rather than immediately after the first failure. Tests that are still failing at this point will be rerun in a new JVM (each), and C_NEW can be another small number, say 1 or 2. This can suffice to detect flaky tests that were failing because of state pollution. While the test is run

in a new JVM for the first time, its coverage can be collected with MEASURE_COVERAGE set to `true`, because our evaluation shows that there is very little extra cost to collecting test coverage (*at the class granularity*) over forking a new JVM. If a test keeps failing after all these reruns, and its coverage does intersect with the latest changes, the test may still be flaky, but it is more likely a non-flaky (deterministic) failure due to the latest changes.

```
// Parameters for the algorithm
int C_IMM; // number of times to rerun test right after failure; can be 0
float THRESHOLD; // ratio of failures for which to rerun tests at the end
int C_END; // number of times to rerun test at the end; can be 0
int C_NEW; // number of times to rerun test in a new JVM; can be 0
boolean MEASURE_COVERAGE; // should the first rerun in a new JVM measure coverage

// Input is a list of tests to run
// Outputs are a set of failed tests and a (sub)set of definitely flaky tests
Pair<Set<Test>, Set<Test>> flakyDetector(List<Test> tests) {
  Set<Test> failures = emptySet;
  Set<Test> flakies = emptySet;

  // run tests and rerun failures immediately
  jvm = new JVM();
  for (Test t : tests) {
    if (!jvm.run(t)) { // FAIL
      failures.add(t);
      if (rerun(t, C_IMM, jvm)) { // PASS
        flakies.add(t);
      }
    }
  }

  // may not rerun if too many failures
  if (failures.size() / tests.size() >= THRESHOLD) {
    return new Pair(failures, flakies);
  }

  // rerun at end failures with unknown flaky status
  for (Test t : failures.minus(flakies)) {
    if (rerun(t, C_END, jvm)) {
      flakies.add(t);
    }
  }

  // rerun in new JVM failures with unknown flaky status
  for (Test t : failures.minus(flakies)) {
    if (rerun_fork(t, C_NEW, MEASURE_COVERAGE)) {
      flakies.add(t);
    }
  }
  return new Pair(failures, flakies);
}
```

Figure 3.1: Flaky Detector Technique

```
// reruns test t up to N times on the given JVM
boolean rerun(t, N, jvm) {
  for (i = 0; i < N; i++) { // could be parallel for
    if (jvm.run(t)) { // PASS
      return true;
    }
  }
  return false;
}

// reruns test t up to N times in a new JVM,
// potentially measuring coverage on the first rerun
boolean rerun_fork(t, N, MEASURE_COVERAGE) {
  for (i = 0; i < N; i++) { // could be parallel for
    if (MEASURE_COVERAGE && i == 0) { // measure coverage on first rerun
      if (run_fork_coverage(t)) { // PASS
        return true;
      }
    } else {
      if (run_fork(t)) { // PASS
        return true;
      }
    }
  }
  return false;
}

// run test t in a new JVM
boolean run_fork(t) {
  jvm = new JVM();
  return jvm.run(t);
}

// reruns test t in a new JVM and collects its coverage
boolean run_fork_coverage(t) {
  jvm = new JVM_with_code_coverage();
  if (jvm.run(t)) { // PASS
    return true;
  }
  covered_entities = jvm.get_coverage();
  changed_entities = compute_changes_from_VCS();
  intersection = intersect(covered_entities, changed_entities);
  return intersection.isEmpty();
}
```

Figure 3.2: Helper Functions for Rerunning in Flaky Detector

CHAPTER 4

Evaluation

We next evaluate the trade-off that our proposed improvements offer compared to rerunning failing tests immediately after failure. We first give an overview of the projects and tests used in our evaluation (Section 4.1). We then evaluate our three proposed extensions: Postpone (Section 4.2), Fork (Section 4.3), and Intersect (Section 4.4).

We ran the experiments that measure time (sections 4.2 and 4.3) on an Intel Xeon E5 CPU with 2GB of RAM running Scientific Linux 7.1, OpenJDK 64-Bit Server 1.7.0_79, and Apache Maven 3.0.5.

4.1 Experimental Objects

Table 4.1 lists the flaky tests used in our evaluation. For each flaky test, we tabulate the Apache project that had the test, the test name, the revision when the test was introduced, the revision when it was fixed to not be flaky, the number of revisions where the test was flaky, the number of those revisions where the project can compile, and the number of test classes in the last revision. We selected these tests from the Apache projects that we studied previously [1]. However, in our previous study we did not build the projects and did not run the flaky tests but only reasoned about the code changes. In this study we want to run the tests, but many old project revisions cannot be easily compiled due to their build dependencies [38]. As a result, we selected only a subset of flaky tests that compile in the majority of revisions in which they were flaky. All these flaky tests were flaky from the first revision when they were written (in general, a test may not be flaky from the first revision but could become flaky later on due to code changes), and all these flaky tests were eventually fixed. These flaky tests come from various categories and from projects in various

Project	Test Class. Test Method	Test ID	First SHA	Last SHA	Revisions	Buildable	# of Test Classes
Ambari [23]	TestActionQueue. testConcurrentOperations	Ambari_f1	b87dc45e	be1d871d	1061	1008	167
Hadoop [24]	TestFairScheduler. testContinuousScheduling	Hadoop_f1	4fe912df	cae1ed9f	50	48	151
Hadoop	TestRMContainerAllocator. testBlackListedNodes	Hadoop_f2	ba66ca68	9692cfc9	103	103	42
Hadoop	TestAuditLogs. testAuditAllowed	Hadoop_f3	f4555669	06d635cd	120	120	286
Hadoop	TestUnderReplicatedBlocks. testNumberOfBlocksToBeReplicated	Hadoop_f4	d26334b4	3c6e5b90	24	24	351
HBase [25]	TestHRegion. testWritesWhileScanning	HBase_f1	e593f0ef	f8ca192f	56	54	137
HBase	TestFromClientSide. testRegionCachePreWarm	HBase_f2	d92c4962	ca2f1678	25	23	121
HBase	TestMasterWrongRS. testRsReportsWrongServerName	HBase_f3	982a15f2	dc641719	46	45	116
HttpCore [26]	TestConnPool.testStateful ConnectionRedistributionOnPerRouteMaxLimit	HttpCore_f1	49247d20	66d43a02	34	34	68
Oozie [27]	TestPartitionDependencyManagerEhcache. testEvictionOnTimeToIdle	Oozie_f1	426d13fc	85e70e19	113	75	277

Table 4.1: Flaky Tests Used in Evaluation

domains (networking, databases, etc.), so they offer a relatively representative sample of flaky tests.

To compare the results of flaky tests with the results of non-flaky tests, we use two additional datasets. Table 4.2 lists several non-flaky tests from the projects for which we had some flaky tests. We randomly selected a few tests that existed in the first SHA where the flaky test was added. Table 4.3 lists several additional Java projects used in Section 4.4; we selected popular projects from GitHub that could compile and run tests in most of their latest 100 revisions at the time of our download in March 2015. We tabulate similar information as in Table 4.1; in our experiments, all these projects for all revisions (that build) had their tests pass, but they may still have unknown flaky tests that just by chance never failed in 100 runs (e.g., even our first example from Chapter 2 has the probability of $1 - (255/256)^{100} = 32.4\%$ to not fail in any of 100 runs).

4.2 Postponing the Reruns

The first improvement we propose is to postpone some reruns to the end of the test-suite execution. We measure the *cost* of these reruns for the tests listed in Section 4.1. As

Project	TestClass.TestMethod	Test ID
Ambari	TestHeartbeatMonitor.testHeartbeatStateCommandsEnqueueing	Ambari_t1
Ambari	TestHeartbeatMonitor.testHeartbeatLoss	Ambari_t2
HBase	TestMemStore.testSnapshotting	HBase_t1
HBase	TestMemStore.testMultipleVersionsSimple	HBase_t2
HBase	TestMemStore.testScanAcrossSnapshot	HBase_t3
HttpCore	TestConnPool.testEmptyPool	HttpCore_t1
HttpCore	TestConnPool.testLeaseRelease	HttpCore_t2
Oozie	TestCoordinatorEngine.testCustomDoneFlag	Oozie_t1
Oozie	TestCoordinatorEngine.testDoneFlag	Oozie_t2

Table 4.2: Non-Flaky Tests Used in Evaluation

Project	First SHA	Last SHA	Buildable	# of Test Classes
Closure Compiler [28]	9cc8d234	16a92448	100	258
Commons Lang [29]	d1c24733	53577f2f	100	125
Commons Math [30]	e11c0008	68e6de35	98	484
Commons Net [31]	4c63aa0a	e17d89b3	100	40
Cucumber [32]	f8ecfb49	3fc886e8	98	100
Dropwizard [33]	dd70ee6c	5add60b1	100	142
GraphHopper [34]	75a6b9c9	65a496d4	100	98
JodaTime [35]	51ca3165	b9fe534c	100	124
Phoenix [36]	ba0409f2	e0a81a09	96	107
Retrofit [37]	a9c1f415	763fe163	100	23

Table 4.3: Other Projects Used in Evaluation

discussed previously, postponing the reruns offers two benefits: (1) more flaky tests (whose failures are bursty) may be detected and (2) more information is available at the end of the run so if too many tests fail, reruns can be avoided altogether. However, we do not attempt to precisely measure the benefit; as discussed in Chapter 2, measuring the exact frequency of failures, and especially the "burstiness" and distribution of failures in time, would be challenging because they depend on sources of non-determinism that vary from one environment to another.

To compare the cost of Postpone and rerunning failing tests immediately after failure, we compare the execution time for the tests shown in tables 4.1 and 4.2; *a priori* one cannot tell if the reruns at the end would be slower or faster than immediate reruns, although we

Test ID	Reruns	Test-Suite Time (s)	
		RERUN	POSTPONE
Ambari_f1	10	369.31	370.98
Ambari_t1	10	363.70	362.14
Ambari_t2	10	366.68	364.75
Hadoop_f1	10	670.22	710.19
Hadoop_f2	10	140.71	138.52
Hadoop_f3	10	4912.29	5542.81
Hadoop_f4	10	7145.82	7107.83
HBase_f1	100	19160.57	18770.84
HBase_f2	n/a	n/a	n/a
HBase_f3	n/a	n/a	n/a
HBase_t1	100	40720.70	38515.12
HBase_t2	100	40620.96	38988.08
HBase_t3	10	40828.90	40537.51
HttpCore_f1	100	9.28	9.06
HttpCore_t1	100	8.99	8.67
HttpCore_t2	100	9.12	8.70
Oozie_f1	10	5408.33	5406.75
Oozie_t1	100	4536.35	5409.13
Oozie_t2	100	2245.49	2073.24

Table 4.4: Rerunning Immediately vs. Rerunning at the End

could reasonably expect these reruns to take about the same time. For this experiment, we have implemented a new option in the Maven Surefire Plugin to rerun failing tests at the end of the test-suite execution in the same JVM. The latest version of Maven Surefire already provides an option for rerunning failing tests immediately after the failure. We modify the projects' Maven configuration files (`pom.xml`) to use our modified version of Maven Surefire, and we set the maximum number of reruns to 10 for tests that run for more than 10 ms, and to 100 for tests that run for less than 10 ms, so that the contribution of the reruns to the overall test-suite execution time is not insignificant.

For each test, we run two experiments. In both experiments, we change the test to deterministically fail, so that it will be rerun exactly the maximum number of times. We also enforce an execution order on the test classes such that the class containing the failing test is the first to execute in the test suite. To limit the time needed, we restrict the experiments to the tests in the same module as the failing test. In the first experiment, we use the available option in Maven Surefire (`rerunFailingTestsCount`) to rerun failing tests

immediately after failure (C_IMM=r, where r is either 10 or 100, and C_END=0). In the second experiment, we use our added option to rerun failing tests at the end of the test-suite execution (C_IMM=0 and C_END=r, where r is either 10 or 100).

These two experiments, with all reruns at the very beginning or at the very end of the test-suite execution, allow us to compare the difference at the extremes. (In general, a flaky test could have its first natural run, and a failure, anywhere in the test suite, not at the beginning of the test suite.) We perform these experiments on all the tests listed in tables 4.1 and 4.2 except the tests HBase_f2 and HBase_f3 that cannot be rerun because they do not properly clean up the state after their first run and during the attempted reruns throw an exception when they try to start up a server that has already been started in the initial run. We repeat the experiments five times and report the average test-suite execution time.

Table 4.4 shows the results, with all times in seconds. For each test, we tabulate the number of reruns and the entire test-suite execution time (as reported by Maven) that includes the reruns in both scenarios. Our inspection of the differences shows that they are largely irrelevant in the context of the entire test-suite execution. Some individual reruns can be slower or faster because of class loading or JIT compilation in Java. However, the overall test-suite execution time is largely unaffected. As a result, rerunning the tests immediately after the failure or at the end of the test-suite execution has about the same cost. To derive the highest benefit for this cost, it seems the most prudent to split the number of reruns, e.g., for a total of three reruns, it is better to have one immediately and two more at the end, rather than having all three immediately or all three at the end.

4.3 Running in a New JVM

The second improvement we propose is to rerun each failing test in a new JVM. This rerunning also offers an option to collect coverage. In our experiments, we use the publicly available Ekstazi tool [20, 39] that collects coverage to perform regression test selection. While Ekstazi can work in multiple modes, we use the lowest-cost mode that simply reports all the classes that were loaded during a JVM run for a test. (The next section discusses the precision that this mode achieves.)

Test ID	Base (s)	Ekstazi (s)	Reruns (s)					N_E
			1	5	10	20	50	
Ambari_f1	1.11	1.36	1.74	8.61	20.43	43.90	114.62	1.28
Ambari_t1	8.41	8.29	10.67	16.84	23.67	32.22	54.85	4.99
Ambari_t2	7.58	7.70	9.89	16.14	21.04	30.97	54.72	4.37
Hadoop_f1	12.36	12.78	14.53	17.59	27.68	51.12	63.00	8.53
Hadoop_f2	8.19	8.19	9.03	11.34	14.67	20.33	36.99	13.02
Hadoop_f3	14.07	14.03	17.08	29.17	43.00	70.80	148.63	4.66
Hadoop_f4	12.91	12.99	14.41	18.38	23.78	32.41	57.22	12.36
HBase_f1	2.99	2.97	5.00	14.54	32.63	90.73	494.35	1.41
HBase_f2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HBase_f3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HBase_t1	0.45	0.46	0.46	0.52	0.63	0.80	0.99	35.79
HBase_t2	0.45	0.47	0.48	0.52	0.56	0.63	0.72	110.00
HBase_t3	0.70	0.71	0.95	1.42	1.75	2.11	3.15	4.83
HttpCore_f1	0.47	0.50	0.50	0.55	0.68	0.84	1.17	29.09
HttpCore_t1	0.43	0.46	0.45	0.49	0.55	0.67	0.96	39.66
HttpCore_t2	0.47	0.51	0.50	0.54	0.61	0.80	1.12	33.13
Oozie_f1	23.54	23.69	26.91	35.76	46.95	65.97	122.82	10.06
Oozie_t1	22.57	22.28	25.25	32.12	40.53	58.54	109.56	12.56
Oozie_t2	22.38	22.44	24.22	32.08	40.52	59.79	109.49	12.20

Table 4.5: Rerunning in the Same JVM vs. Forking a New JVM

To compare the cost of rerunning in a new forked JVM with rerunning in the same JVM, we compare the execution time for 17 tests, i.e., all tests from tables 4.1 and 4.2, excluding HBase_f2 and HBase_f3 for the same reason described in the previous section. To obtain more precise results, we focus on one test at a time. For each test, we collect its execution time (from the JVM execution, without the Maven build overhead) when run *alone* once in a JVM, when executed with Ekstazi collecting coverage, and when rerun in one JVM with a varying numbers of reruns (between 1 and 50). We repeat each of these executions five times and compute the average.

Table 4.5 shows the results, with all times in seconds. The cost of rerunning the test once in a new forked JVM is the execution time of its base run; the cost of rerunning it in a new JVM while collecting coverage is the execution time with Ekstazi; and the cost of rerunning the test in the same JVM n times is the difference between the column labeled n and the base. We can see that the time for rerunning is often non-linear. For example, for Ambari_f1, rerun with $n = 1$ gives $1.74 - 1.11 = 0.63s$, whereas with $n = 50$ gives $114.62 - 1.11 = 113.51$, and

$113.51/50 = 2.27s$; in contrast, for Hadoop_f1, rerun with $n = 1$ gives $14.53 - 12.36 = 2.17s$, whereas with $n = 50$ gives $63.00 - 12.36 = 40.72$, and $40.72/50 = 0.81s$. While some of the differences are due to the imprecision of time measurements (especially more affected for smaller values of n), some others are due to several reruns of the same test interacting (e.g., later test reruns can get slower if the test leaks some memory/resources as likely Ambari_f1 does, or later test reruns can get faster if the executed code gets JIT compiled with more optimizations as Hadoop_f1 likely does).

For a high-level comparison of techniques, we can still approximate the missing values for n by using a linear interpolation (and, in the case of HBase_t2, using extrapolation). Specifically, we are interested in the number of test reruns that can be executed in the same JVM for the time needed to rerun the test once in a new JVM. In terms of parameters from Section 3.1, we are interested in the values for C_IMM and C_END (and we know from the previous section that the time to execute the test for those is about the same) that give the same test execution time as for C_NEW=1. We call this value N_E and compute it as follows. We find values n_1 and n_2 such that $r_{n_1} \leq t_{Base} \leq r_{n_2}$, where $r_n = t_n - t_{Base}$. We then compute $N_E = (t_{Base} - r_{n_1}) \cdot (n_2 - n_1)/(r_{n_2} - r_{n_1}) + n_1$. The last column in Table 4.5 shows N_E. We can see that the value ranges from close to 1 (e.g., for Ambari_f1 and HBase_f1), which means that the cost of rerun in a new JVM is about the same as the cost of rerun in the same JVM, up to over 100 (for HBase_t2), which means that the cost of rerun in a new JVM is much higher than the cost of rerun in the same JVM.

For half the tests considered, running them in a new JVM would take more time than rerunning them 10 times in the same JVM. Although this may seem to show that simply rerunning the test many times in the same JVM would be much better than rerunning once in a new JVM, one must keep in mind that it is very unlikely for a test that failed 9 times in a row to suddenly pass in the 10th run with no change to the environment. However, the test may pass in a clean environment (a new JVM). Therefore, even if a test runs extremely fast, it is almost never beneficial to rerun it more than *a few* times in the same JVM and then potentially rerun in a new JVM.

It is important to note that the difference in execution times between the base run and the Ekstazi run is very small for all the tests considered. This shows that whenever developers

are willing to pay the cost of forking a new JVM to check flaky tests, it is a good idea to also collect coverage during the run. While a finer level of granularity (e.g., method or statement) could add a lot more overhead, using Ekstazi to collect coverage at the class level of granularity does not add much overhead.

4.4 Intersecting Coverage

The third improvement we propose is to intersect test coverage of a failing test with the latest changes. If a test that passed in the previous revision fails in this revision, and the test execution does not depend on the changes introduced in this revision, we can definitely conclude that the test is flaky (because the test can both pass and fail without executing the change, it depends on some source of non-determinism). If the test does depend on the change, we cannot conclude whether the test is flaky or not.

Therefore, the key question to evaluate is how often this Intersect technique could provide a definite conclusion. Yet again, it would be rather challenging to simulate when developers observed real failures due to flaky tests and what exact changes they made in the revisions when those failures happened. Instead, we use a large number of tests and revisions to evaluate how often test runs do not intersect with the latest changes; if one of those tests was a flaky test that failed in that revision, the technique would find it. For instance, recall the first example from Chapter 2; the coverage for this test does *not* intersect with the changes made in 80% percent of the revisions (96 out of 120 revisions) between the revision when this test was written and the revision when this test was fixed to not be flaky. In any one of those 120 revisions, the test had a chance of 1/256 to fail, and for 96 of those revisions, the test would fail without executing any of the changed code. In this case, only 20% of the test runs of the flaky test would intersect with the changes. If a large fraction of test runs intersect with the latest changes, our technique would not be beneficial: it would only increase the cost, however slightly, by collecting coverage, but it would not help determine whether the test is flaky or not.

To evaluate the applicability of Intersect, we collect coverage data for 2715 test *classes* (i.e., we do not collect test coverage separately for each test method in a test class, which

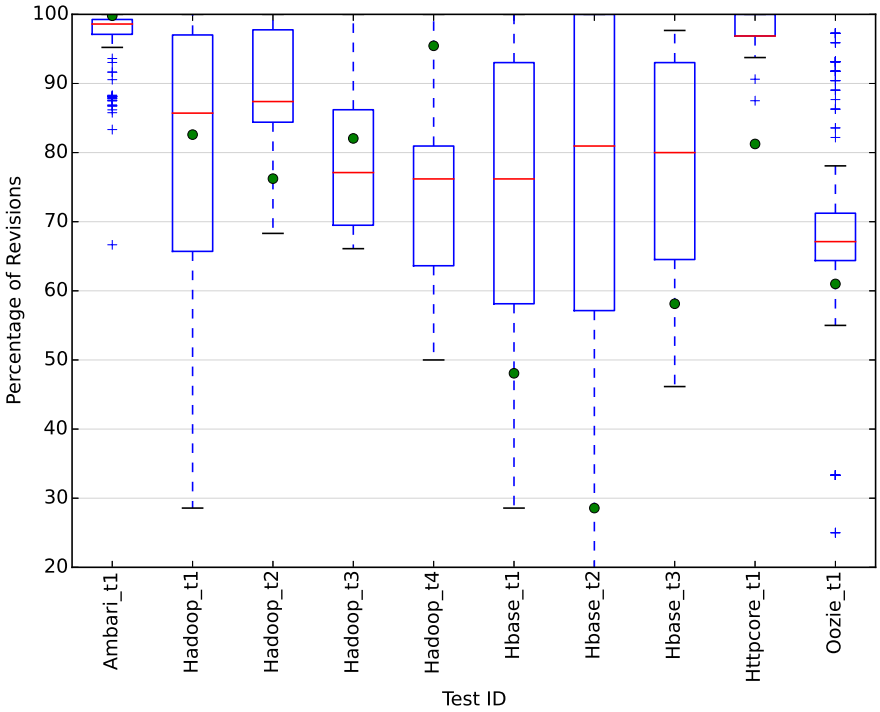

Figure 4.1: Percentage of Revisions with Disjoint Coverage in Projects w/ Flaky Tests

could make the results even more precise) over many revisions and determine the percentage of revisions in which this coverage does not intersect with the changes. Our evaluation includes both test classes with known flaky tests and test classes for which we never obtained a test failure (although they might still have flaky tests). Specifically, our evaluation includes 1214 test classes (test classes that were included in more than one range of revisions are counted only once) from 5 different projects that had at least one flaky test in our prior study [1] (Table 4.1 shows their detailed statistics). Because we used only 5 projects that could successfully build in most revisions between when the test was first written and when it was fixed, our evaluation also uses 1501 test classes from 10 additional projects (Table 4.3 shows their detailed statistics). While we did not encounter any flaky failure in these additional projects, their coverage data is still relevant to obtain a better insight into the applicability of our INTERSECT technique.

For each project and all revisions from both tables 4.1 and 4.3, we run the EKSTAZI tool for regression test selection. At each revision, EKSTAZI only selects tests whose behavior

may be affected by the changes (i.e., tests whose coverage intersects with the changes). We record which test classes Ekstazi selects and which not. We then compute, for each class, the percentage of revisions in which each test class is *not* selected (i.e., its coverage does not intersect with the changes).

Figure 4.1 plots the distribution of this percentage for the different flaky tests used to choose the range of revisions from their projects. We show each distribution as a boxplot, where whiskers mark 10th and 90th percentile, and the red horizontal line marks the median. It can be seen from the figure that the median in all cases but one (Oozie_f1) is above 70%. Moreover, most data points are over 60%. It means that for the majority of test classes and revisions, test coverage does not intersect with the changes. However, the flaky tests themselves (shown as green dots) often have percentages below the median. The reason for this is not that the flaky tests are in general more likely than non-flaky tests to have their coverage intersect with the changes; instead, *newer* (or more recently modified) tests are more likely than older (or less recently modified) tests to have their coverage intersect with changes. The revision range in each case includes revisions right after the flaky test was written, which likely means that the code it depends on is probably being changed at that point, and hence the test gets selected. Similarly, the range also includes revisions towards the end where developers were trying to fix the flaky tests, and sometimes developers make several commits while fixing a test (changing the test code itself and/or the code under test that this test depends on), and hence the test gets selected.

Figure 4.2 plots the distribution of the percentage for the projects and revision ranges from Table 4.3. We ran the test suite for each project over the last (buildable) 100 revisions at the time of download, and again used Ekstazi to only select the test classes that are affected by the changes in each revision. For these ranges, all the medians but one are above 80%, and except for two projects (Closure Compiler and Phoenix), most data points are above 80%. These averages are somewhat higher than for the ranges with known flaky tests, but in general the selection percentages vary because of the choice of revisions, type of changes, maturity of the project, and other factors.

Overall, these results show that the coverage intersection would be applicable in more than 70% of cases. The main takeaway is that the intersection of test coverage and latest

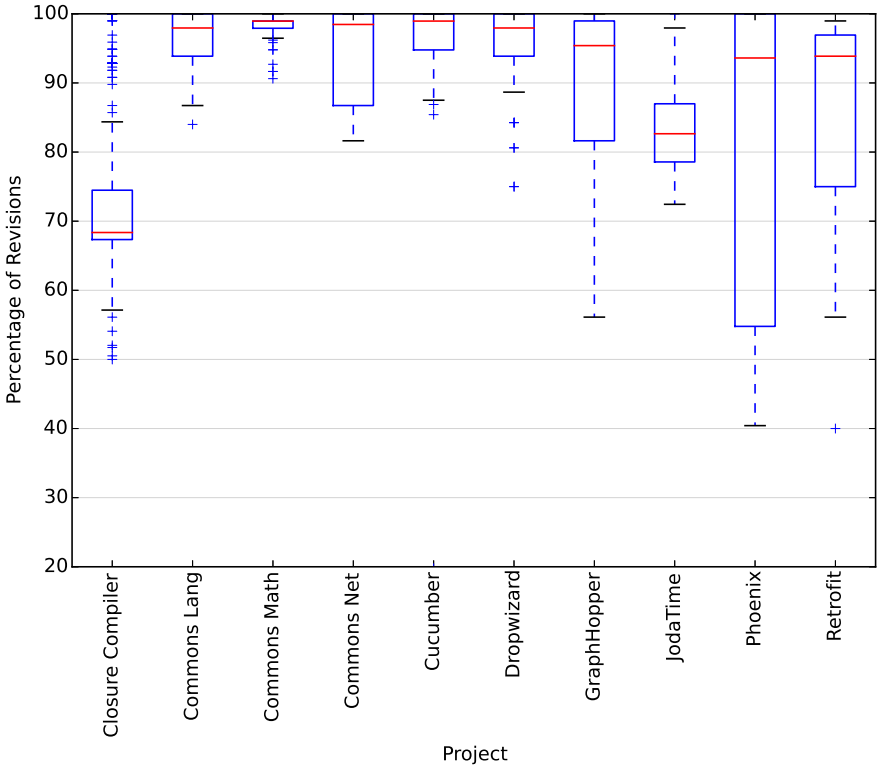

Figure 4.2: Percentage of Revisions with Disjoint Coverage in Additional Projects

changes is often empty, and therefore it is, on average, cost-beneficial to always pay the cost of collecting coverage in order to sometimes obtain the benefit of determining that some test failure is definitely due to a flaky test.

CHAPTER 5

Discussion

We next discuss several other factors that could influence the cost of detecting flaky tests.

For some projects, the developers already configure the build system to fork a new JVM for each test class (generally to avoid state pollution) [8]. For such projects, reruns would already run the failing test in a new JVM, so collecting coverage all the time by default would be definitely beneficial, without introducing almost any additional overhead. Moreover, for some projects, the build system already collects test coverage all the time (especially if the tests are run in a continuous integration system such as Jenkins). In such cases, coverage is obtained for free, with no extra cost.

Besides the cost collecting the coverage, there is also the cost of intersecting the coverage with the changes. Coverage intersection can be done in several ways with different trade-offs between time and space. One way is to get the changes from the version-control system (e.g., `git diff`); using this approach requires computing a map from the test coverage entities (computed at the level of compiled binary, e.g., Java `.class` or `.jar` files) to the changed entities (computed at the source level). This is easy to obtain if the classes are compiled with debug options (i.e., `javac -g`) as is the most common case in Java (because it allows easier debugging, e.g., by providing source information in the stack traces). Another approach would be to store some additional information to detect changes more easily. For example, `.class` files from the previous revision could be saved and a simple checksum can be used to detect class files that changed; Ekstazi already uses this approach, and the computation of checksum is extremely efficient [20]; the only issue is that the old `.class` files need to be saved for at least one revision.

In case of projects that already use regression test selection, Intersect would not need to be run, because a test that is not affected would not be even selected to run, and hence

it could not fail if flaky. However, not all projects use regression test selection. Moreover, using it requires collecting coverage for *all* tests all the time. In contrast, our INTERSECT technique requires collecting coverage only after some test(s) failed, and it requires collecting coverage only for the test(s) that failed.

CHAPTER 6

Threats to Validity

Out experimental evaluation shows that our proposed technique could determine that more failures are due to flaky tests without significantly increasing the cost of detection. However, the evaluation was performed on a limited number of cases, and there are threats to generalizing our findings.

External: The projects used in our evaluation may not be representative. To alleviate this threat, we consider a large number of projects from different application domains, with different code sizes, and number of test classes. We evaluate our proposed improvements on 15 projects, of which 5 have known flaky tests. We selected flaky tests with different characteristics and evaluate Postpone and Fork on the revision in which those tests were written. Our results could differ for different revisions. Similarly, we evaluate the applicability of Intersect for the revisions in which the tests were flaky, from the revision in which a flaky test was written to the revision in which it was fixed. Our results could differ for a different range of revisions. We chose this range because it shows most precisely the percentage of revisions in which coverage intersection would have been beneficial in detecting those specific flaky tests.

Internal: The tools we use in our evaluation may have bugs. To increase confidence in our experiments, we use tools that are adopted by the open-source community; the Maven Surefire Plugin is widely used, and the Ekstazi tool, although very recent, has already been adopted by several open-source projects [20].

Construct: To evaluate the applicability of Intersect, we compute the percentage of revisions in the coverage of a test class does not intersect with changes. This is only one part of the condition for coverage intersection to be beneficial. First, the test needs to show flaky behavior and then, if its coverage does not intersect with changes, it is flaky. Although

the probability of a test coverage intersecting with the changes between any two revisions is low, the conditional probability given that the test has failed might be higher. However, we expect that this probability is largely independent of the failures, because the pass/fail outcome of a flaky test depends on some source of non-determinism that is not related to the code changes.

We evaluate the costs and benefits of the proposed technique on both known flaky tests and (presumably) non-flaky tests. The results collected for non-flaky tests may not be representative of flaky tests. Because we had a limited number of flaky tests whose projects we could build over several old revisions, we include non-flaky tests in order to get more extensive results.

To evaluate the trade-off involved in postponing reruns to the end of the test suite, we compare the test-suite execution times when failing tests are rerun immediately and when they are rerun at the end. Measuring time in this way may be misleading as tests that are run earlier may have a higher cost (e.g., due to loading classes, JIT compiling, etc.). We mitigate this threat by showing the total execution time of the test-suite rather than the individual test-method execution times.

To compare the cost of rerunning tests in the same JVM multiple times and rerunning them once in a new JVM, we collect the execution times for n reruns where n ranges between 1 and 50 and then use a linear interpolation/extrapolation to approximate the number of reruns that can be executed in the time needed for running the test once in a new JVM. Though this offers a rough approximation, our experiments show that the increase in execution time is not always linear.

CHAPTER 7

Related Work

7.1 Flaky Tests

We recently published the first extensive study of flaky tests [1]. We studied more than 200 flaky tests from 51 Apache projects and categorized their root causes, common fixes, and ways to manifest them. During this earlier study, we came across different types of flaky tests that we now characterize as non-bursty and bursty (where bursty can be state-dependent or not). That study identified all of the flaky tests used in Chapter 4.

The most common approach to detecting flaky tests is to immediately rerun failing tests, which we call Rerun. Several systems mentioned in Chapter 1 provide support for this immediate rerun. For example, Google TAP can rerun tests immediately (it has the `@flake` annotation which indicates that a test should be rerun, by default up to 3 times, immediately after failure), but as a notable exception, it can also rerun (by default up to 10 times) at night all the tests that failed during a day (including those with the `@flake` annotation that failed on all reruns) [17, 18]. Rerunning at night waits for the build/test machines to be less busy but provides the feedback to the developers hours after their tests failed (and after they may have already wasted their time debugging a flaky test). To the best of our knowledge, TAP (or any other system for that matter) does not offer all the options we propose. Our options can provide the feedback much faster at the same cost (for postponing to the end) or somewhat higher cost (for new JVM) than `@flake`.

Specific techniques have been recently proposed for handling order-dependent tests. Zhang et al. propose several methods to detect such tests by rerunning them in different orders [7]. Huo and Clause also propose a technique that can be used to detect such tests [40], although their technique was originally proposed to detect brittle assertions (that may cause non-

deterministic failures). Bell and Kaiser [8] propose an approach to tolerate the effects of order-dependent tests by isolating them in the same JVM. However, none of these techniques focuses on detecting whether a given test failure is due to a flaky test or not, and none of the techniques handle general case of arbitrary flaky tests.

7.2 Change-Impact Analysis

Change-impact analysis (CIA) techniques aim to determine the effects of source code changes, using static analysis, dynamic analysis, or combined approaches [21, 41, 42]. For example, Chianti is a CIA technique, proposed by Ren et al. [21], which uses a static analysis to decompose the difference between two program versions into independent atomic changes and uses dynamic call graphs to determine the set of tests whose behavior might be affected by these changes. It also uses these call graphs to determine for each affected test, which subset of changes can affect its behavior. Our coverage intersection technique also collects coverage but fully dynamically and does not require any static analysis; it has a much lower overhead because it focuses on only the tests that failed and not on all tests.

7.3 Regression Test Selection

Regression Test Selection (RTS) techniques determine which tests can be affected by a code change and only run those to speed up regression testing. Many RTS techniques have been proposed [43–49], and are summarized in two literature reviews [50,51]. Most RTS techniques collect coverage, first for all the tests, and then recollect coverage only for the tests that are run as potentially affected by the code changes. Our experiments use the Ekstazi RTS tool that we recently developed and made publicly available [39]. Ekstazi collects for each test class which *files* it depends on (be they `.class`, `.jar`, or other files) [52].

If a project uses RTS, it is most likely already intersecting test coverage with the changes between revisions. Therefore, using the coverage intersection technique we propose for flaky tests would be redundant in such cases (as the flaky test may not be run in the first place to even fail once). However, projects that use RTS will pay the cost of collecting coverage

and analyzing changes for *all* tests at *every* revision. In contrast, our INTERSECT technique would only collect coverage and analyze changes if there are test failures (that did not pass after reruns). Also, our POSTPONE and FORK techniques for rerun are still relevant: even if the test coverage intersects with the changes, the failing test can still be flaky.

CHAPTER 8

Conclusions and Future Work

Automated regression testing is a valuable and widely practiced activity for improving software quality, but its value is lowered by flaky tests that can non-deterministically fail or pass for the same code revision. Knowing whether a test failure is due to a flaky test or not helps developers to make better decisions about debugging and development. The most widely used technique to determine whether a test failure is due to a flaky test is to rerun the failing test multiple times immediately after it fails. We have proposed and evaluated several improvements: (1) postponing the reruns, (2) rerunning in a new runtime environment (e.g., a new JVM for Java tests), and (3) intersecting the test coverage with the latest changes. Our results are promising, and we hope that our proposed improvements will be added to many testing frameworks and build systems to help developers combat flaky tests.

To perform the experiments presented in this thesis, we have already extended the Maven Surefire Plugin with an option to rerun failing tests at the end of the test-suite execution (this is our first proposed improvement, Postpone). In the future, we plan to further extend this plugin to also include our other two proposed improvements, namely rerunning failing tests in a new JVM (Fork) and intersecting their coverage with the code changes (Intersect). We are also hoping to release these contributions to the open-source community. Moreover, this thesis has only considered cases with one test failure at a time, but in general, multiple tests can fail in the same test-suite execution. It would be interesting to consider not just "temporal" correlation of failures for one test (i.e., being bursty or not) but also "spatial" correlation of failures among multiple tests. For example, multiple tests in the same test class may all depend on some network service, and when one test fails (if the service is down), all other tests are likely to fail as well. In fact, one can even consider *postponing the first test run* in such cases. Similarly, if multiple tests fail and are rerun in a new JVM, one can either rerun all failing tests at once of rerun each test in a separate JVM.

REFERENCES

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, 2014.

[2] "TotT: Avoiding flakey tests," http://goo.gl/vHE47r.

[3] M. Fowler, "Eradicating non-determinism in tests," http://goo.gl/cDDGmm.

[4] "Flakiness dashboard HOWTO," http://goo.gl/JRZ1J8.

[5] P. Sudarshan, "No more flaky tests on the Go team," http://goo.gl/BiWaE1.

[6] T. Lavers and L. Peters, *Swing Extreme Testing*, 2008.

[7] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, M. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis*, 2014.

[8] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *International Conference on Software Engineering*, 2014.

[9] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2011.

[10] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: models, tools, and controlling flakiness," in *International Conference on Software Engineering*, 2013.

[11] F. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *Agile*, 2009.

[12] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *International Conference on Automated Software Engineering*, 2009.

[13] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" in *European Conference on Object-Oriented Programming*, 2013.

[14] "Android FlakyTest annotation," http://goo.gl/e8PILv.

[15] "Jenkins RandomFail annotation," http://goo.gl/tzyC0W.

[16] "Spring Repeat Annotation," http://goo.gl/vnfU3Y.

[17] P. Gupta, M. Ivey, and J. Penix, "Testing at the speed and scale of Google," 2011, http://goo.gl/2B5cyl.

[18] J. Micco, "Continuous integration at Google scale," 2013, http://goo.gl/uq4eoB.

[19] "Surefire rerunFailingTestsCount option," http://goo.gl/wdvpzI.

[20] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *International Conference on Software Engineering, Demonstrations Track*, 2015, to appear.

[21] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2004.

[22] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *International Conference on Software Maintenance*, 2011.

[23] "Ambari," https://github.com/apache/ambari.git.

[24] "Hadoop," https://github.com/apache/hadoop.git.

[25] "HBase," https://github.com/apache/hbase.git.

[26] "HttpCore," https://github.com/apache/httpcore.git.

[27] "Oozie," https://github.com/apache/oozie.git.

[28] "Closure Compiler," https://github.com/google/closure-compiler.git.

[29] "Commons Lang," https://github.com/apache/commons-lang.git.

[30] "Commons Math," https://github.com/apache/commons-math.git.

[31] "Commons Net," https://github.com/apache/commons-net.git.

[32] "Cucumber," https://github.com/cucumber/cucumber-jvm.git.

[33] "Dropwizard," https://github.com/dropwizard/dropwizard.git.

[34] "GraphHopper," https://github.com/graphhopper/graphhopper.git.

[35] "Joda_Time," https://github.com/JodaOrg/joda-time.git.

[36] "Phoenix," https://github.com/apache/phoenix.git.

[37] "Retrofit," https://github.com/square/retrofit.git.

[38] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Internation Symposium on the Foundations of Software Engineering*, 2014.

[39] "Ekstazi," http://ekstazi.org/.

[40] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *International Symposium on Foundations of Software Engineering*, 2014.

[41] S. A. Bohner, "Software change impact analysis," 1996.

[42] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Workshop on Program analysis for software tools and engineering*, 2001.

[43] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *Transactions on Software Engineering and Methodology*, 2001.

[44] M. J. Harrold, D. S. Rosenblum, G. Rothermel, and E. J. Weyuker, "Empirical studies of a prediction model for regression test selection," *Transactions on Software Engineering*, 2001.

[45] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, 1997.

[46] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[47] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "Applying regression test selection for COTS-based applications," in *International Conference on Software Engineering*, 2006.

[48] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," in *International Conference on Software Maintenance*, 1988.

[49] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *International Symposium on Empirical Software Engineering and Measurement*, 2008.

[50] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica (Slovenia)*, 2011.

[51] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, 2012.

[52] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *International Symposium on Software Testing and Analysis*, 2015, to appear.