An Empirical Analysis of Flaky Tests

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi and Darko Marinov

11/20/2014
FSE 22
Hong Kong

NSF Grant Nos. CNS-0958199 and CCF-1012759
DARPA grant FA8750-12-C-0284
Typical Testing Scenario

Test Suite
Typical Testing Scenario
Typical Testing Scenario

Test Suite

Tests Outcomes

PASS

FAIL
Typical Testing Scenario

Test Suite

Tests Outcomes

PASS

FAIL

No further inspection needed
Typical Testing Scenario

Test Suite

Tests Outcomes

PASS

FAIL

No further inspection needed

Inspect and fix bugs
Typical Testing Scenario

Key Assumption: Test outcomes are reliable
Typical Testing Scenario

Test Suite → Tests Outcomes

PASS → No further inspection needed
FAIL → Inspect and fix bugs

Key Assumption: Test outcomes are reliable
Definition: Test Outcome Non-determinism

Test outcome non-determinism:
● Same code revision
● Same input and configuration
● Passes/fails non-deterministically

Such tests are a.k.a. flaky tests.
Flaky Test Example HADOOP-6933

```java
@Test
public void testDirectory() throws IOException {
    ...
    itor = fs.listFiles(DIR1, false);
    ...
    assertEquals(fs.makeQualified(FILE2), stat.getPath());
    itor.next();
    assertEquals(fs.makeQualified(FILE3), stat.getPath());
    ...
}
```

“TestListFiles assumes a particular order of the files returned by the directory iterator. There's no such guarantee made by the underlying API, so the test fails on some hosts.”
Flaky Test Fix Example

```java
+ Set<Path> filesToFind = new HashSet<Path>();
+ filesToFind.add(fs.makeQualified(FILE1));
+ filesToFind.add(fs.makeQualified(FILE2));
+ filesToFind.add(fs.makeQualified(FILE3));

+ itor = fs.listFiles(TEST_DIR, true);
+ stat = itor.next();
+ assertTrue(stat.isFile());
- assertEquals(fs.makeQualified(FILE2), stat.getPath());
+ assertTrue("Path " + stat.getPath() + " unexpected",
+     filesToFind.remove(stat.getPath()));

+ stat = itor.next();
+ assertTrue(stat.isFile());
- assertEquals(fs.makeQualified(FILE3), stat.getPath());
+ assertTrue("Path " + stat.getPath() + " unexpected",
+     filesToFind.remove(stat.getPath()));

+ stat = itor.next();
+ assertTrue(stat.isFile());
- assertEquals(fs.makeQualified(FILE1), stat.getPath());
+ assertTrue("Path " + stat.getPath() + " unexpected",
+     filesToFind.remove(stat.getPath()));
assertFalse(itor.hasNext());
+ assertTrue(filesToFind.isEmpty());
```
Flaky Tests are Harmful

- Undermine the value of test suite
  - Test failures no longer always indicate bugs

- Hide real bugs
  - Flaky test failures often get ignored

- Hard to reproduce and debug
Flaky Tests are Everywhere

“If you do not have a flaky functional tests build, you are not doing anything real”

-- A ThoughtWorks Developer

TAP system at Google has 1.6M test failures in last 15 months, 73K (4.56%) are flaky failures

Our study found hundreds of distinct flaky tests from Apache projects
Contributions of Our Work

- Raise awareness of flaky tests
- Provide 13 findings and implications for avoiding/manifesting/fixing flaky tests
- Propose research for handling flaky tests
- Provide a public dataset of flaky tests
  - Passed artifact evaluation
  - mir.cs.illinois.edu/farah/studied_flaky_commits.csv
How Did We Find Flaky Tests?

- Search commit logs of all 151 Apache projects for “flak” and “intermit” keywords
  - 1129 commit messages

- Manually label likely distinct fixed flaky tests
  - 486 fixed flaky tests

- Sample and inspect 161 commits in more detail
Research Questions

● Causes of flakiness:
  ○ Q1: What are the root causes of flaky tests?

● Introduction of flakiness:
  ○ Q2: How are flaky tests introduced?

● Manifestation:
  ○ Q3: How to manifest flaky tests?

● Fix strategy:
  ○ Q4: Does fixing flaky tests also change code under test (CUT)?
  ○ Q5: How to fix flaky tests?

More in our paper!
Q1: What are the Root Causes of Flaky Tests?
Async Wait

Test makes async calls but doesn’t wait for the result properly; example HBASE-2684:

```java
@Test
public void testRsReportsWrongServerName() throws Exception {
    MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
    MiniHBaseClusterRegionServer firstServer =
        (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
    HServerInfo hsi = firstServer.getServerInfo();
    firstServer.setHServerInfo(...);

    // Sleep while the region server pings back
    Thread.sleep(2000);
    assertTrue(firstServer.isOnline());
    assertEquals(2, cluster.getLiveRegionServerThreads().size());
    ... // similarly for secondServer
}
```
Concurrency

- Flakiness caused by buggy thread interleavings (excluding Async Wait)
  - Data races
  - Atomicity violations
  - Deadlocks

- Non-determinism could either come from test code or code under test
Test Order Dependency

Dependency between tests and the result depends on running order; example HBASE-7113:

```java
@Test
public void testGzipFilter() throws Exception {
    String path = "/" + TABLE + "/" + ROW_1 + "/" + COLUMN_1;
    ...
    Response response = client.put(path, headers, value_1_gzip);
    ...
}

@Test
public void testScannerResultCodes() throws Exception {
    ...
    Response response = client.post("/" + TABLE + "/scanner", headers, 
    
    "<Scanner/>".getBytes());
    
    assertEquals(response.getCode(), 204);
    ...
}
```
Root Causes Distribution

78%
Other Root Causes

- Resource leak
- Network
- Time
- I/O
- Randomness
- Floating point operations
- Unordered collections
Implication 1:
Researchers Can Focus on the Top Categories of Flaky Tests First
Q2: How are Flaky Tests Introduced?
Collect Evolution Info

- Find out the first time the flaky test was written in VCS
- Manually reason about whether the test was flaky at that time
- If not, track changes in history to see how the test became flaky
Flaky Tests Introduction

- Most (126 out of 161) flaky tests are flaky the first time they are written.

- Flakiness is later introduced when:
  - A new test introduces dependency on old tests
  - Patching a bug/refactoring/adding new functionality
Implication 2: Researchers Can Focus on Checking New Tests Extensively for Flakiness
Q3: How to Manifest Flaky Tests?
Manifestation of Async Wait Flaky Tests

- Tests fail when the desired orderings are violated
  - One ordering VS multiple orderings

- `sleep/waitFor` are used to enforce orderings
  - `W/ time parameter` VS `w/o time parameter`

- Waiting for external resources VS resources controlled by the program
W/ Time Parameter VS W/O Time Parameter

- W/ time parameter: 66%
- W/o time parameter: 34%
Implication 3.a: Many Async Flaky Tests Can be Manifested by Changing Time Parameters to Order Enforcing Methods
One Ordering VS Multiple Orderings

- Multiple Orderings: 7%
- One Ordering: 93%
External Resources VS Internal Resources

- External Resources: 91%
- Internal Resources: 9%
One Ordering and Internal Resources VS Others

- One ordering and internal resources: 85%
- Others: 15%
Implication 3.b: Most Async Wait Flaky Tests Can be Manifested by Adding One Time Delay in Program
Manifestation of Test Order Dependency

Flaky Tests

- Various sources of dependency

- Existing techniques focus on in-memory objects [Bell+Kaiser ICSE’14] or shuffling test runs explicitly [Zhang et al. ISSTA’14]
Implication 3.c: New Techniques for Modeling/Checking External Dependency Can be Useful
Q4:
Does Fixing Flaky Tests Also Change Code under Test (CUT)?
Fixing Code Under Test

- 24% (38 out of 161) flaky tests are fixed by changing both test and CUT
- Changes to CUT:
Implication 4: Flaky Tests Are Still Valuable For Catching Bugs and Should Not be Ignored or Removed
Q5: How to Fix Flaky Tests?
Flaky Tests Fixes

● We studied how flaky tests got fixed
  ○ Fix strategies for top three categories

● How effective was each fix?
  ○ Remove - remove its flakiness completely
  ○ Decrease - decrease probability of test flakiness

● Study outcome
  ○ Good practice for fixing flaky tests
  ○ Automated techniques for fixing flaky tests
Fix Async Wait Flaky Tests

Sleep and timed waitFor only decrease flakiness probability
Implication 5.a For Developers: Use waitFor to Fully Synchronize Code
Implication 5.b For Researchers: Automatically Generate Order Enforcing Code by Comparing Events Order Between Passing and Failing Runs
Test Order Dependency Fixes

- Setup/clean up state: 74%
- Remove dependency: 16%
- Merge tests: 10%
Implication 5.c For Developers: Identify Shared States in Test Execution and Maintain Them Clean
Implication 5.d For Researchers: Model and Compare Program States and Automatically Generate Code in setUp/tearDown to Restore Shared States
Threats to Validity

● Choice of projects
  ○ All Apache projects

● Selection criteria
  ○ Commit logs
  ○ Keywords “flak” and “intermittent”
  ○ Fixed flaky tests

● Manual inspection
  ○ Peer review for each flaky test
Related Work

● Non-deterministic bugs and tests
  ○ GUI flaky tests [Memon+Cohen ICSE’13]
  ○ Test order dependency [Zhang et al. ISSTA’14, Bell+Kaiser ICSE’14]
  ○ Concurrency bugs study [Lu et al. ASPLOS’08]

● Bug fixes
  ○ Bug fixes study [Bachmann et al. FSE’10, Murphy-Hill et al. ICSE’13]
  ○ Automatically fixing concurrency bugs [Jin et al. PLDI’11]

● Test fixes
  ○ Automatically repair broken tests [Daniel et al. ASE’09]
Conclusions

● Flaky tests are harmful and pervasive in practice

● We studied and summarized common characteristics of flaky tests
  ○ Common root causes
  ○ Common manifestation methods
  ○ Common fixing strategies

● We believe our results provide both research insights and practice guidelines