Soundy Automated Parallelization of Test Execution

Shouvick Mondal*, Denini Silva†, and Marcelo d'Amorim†
*Indian Institute of Technology Madras, India
†Federal University of Pernambuco, Brazil
shouvick@cse.iitm.ac.in, {dgs, damorim}@cin.ufpe.br

***Abstract*—Regression Testing is an important quality assurance practice widely adopted today. Optimizing regression testing is important. Test parallelization has the potential to leverage the power of multi-core architectures to accelerate regression testing. Unfortunately, it is not possible to directly use parallelization options available in build systems and testing frameworks without introducing test flakiness. Tests can fail because of data races or broken test dependencies. Although it is possible to safely circumvent those problems with the assistance of an automated tool to collect test dependencies (e.g., PRADET), the cost of that solution is prohibitive, defeating the purpose of test parallelization. This paper proposes PASTE an approach to automatically parallelize the execution of test suites. PASTE alternates parallel and sequential execution of test cases and test classes to circumvent provoked test failures. PASTE does not provide the safety guarantee that flakiness will not be manifested, but our results indicate that the strategy is sufficient to avoid them. We evaluated PASTE on 25 projects mined from GitHub using an objective selection criteria. Results show that (i) PASTE could circumvent flakiness introduced with parallelization in all projects that manifested them and (ii) 52% of the projects benefited from test-parallelization with a median speedup of 1.59x (best: 2.28x, average: 1.47x, worst: 0.93x).**

***Index Terms*—Regression testing, test flakiness, parallelization.**

I. Introduction

Regression testing is the task of checking if functionalities implemented in the software remain intact during software evolution. Regression testing is widely adopted in industry [1]–[4]. Large codebases, large test suites, and high frequency of code changes make regression testing expensive. At large organizations, running regression suites often demands special infrastructure. For example, Google's TAP service [5] makes 150 million test runs every day. To support such a high workload, the TAP service runs a batch of tests every 45m –each period is called an epoch–, selecting which tests need to be executed in every epoch based on a lightweight change-impact analysis. Although the scale of the problem at Google is certainly higher compared to that of a typical IT company, the problem affects the software industry in general [6], [7].

Different approaches have been proposed in research to address the cost of regression testing [8]. Regression Test Selection (RTS), for example, uses an objective criterion to determine which tests need to be executed in every regression cycle. For example, Ekstazi [6], [9] tracks file dependencies to determine which tests need to be executed upon file changes.

Test parallelization [7], [10] is another approach to reduce cost of regression testing. It is complementary to traditional approaches to optimize regression testing, including test selection, prioritization, reduction [8]. Test parallelization leverages available machine resources to speed up the execution of tests. Although most build systems (e.g., Maven [11]) and test frameworks (e.g., JUnit [12]) provide features to accelerate the execution of tests (see Section II-A), using those features "out of the box" rarely works. Unintentional interference across tests, such as broken test dependencies and data races, often occurs when developers do not prepare the code to take into account the fact that tests will execute concurrently. Listing 1 shows an example, later detailed on Section II-B, where the test `randomServer` fails when executed before the test `randomServerFromMany`. In that case, the failures occurs because the test `randomServer` does *not* properly configure the EasyMock library before using mock objects. The test `randomServerFromMany` uses the EasyMock API properly. Consequently, the failure is only observed when the test `randomServer` is executed first. In this example, the test `randomServer` depends on the test `randomServerFromMany`. Test dependency is a problem for parallelization as test methods can, in principle, execute in any order. Parallel execution of tests can produce data races for the same reason: shared data reachable from the tests.

One way to address this problem –as to enable automated parallelization– is to forbid tests to share state [13], i.e., to detect and report test dependencies during evolution and to adopt strict policies on the use of static fields (to prevent data races). One issue with that approach is that it imposes a burden on developers. For instance, certain design patterns, such as Singleton [14], are implemented with static fields. Furthermore, developers may not feel the urgency to repair test cases if these test cases pass in regular sequential runs.

This paper proposes PASTE (PArallel-Sequential Test Execution), a lightweight approach to automate parallel execution of tests. PASTE builds on the observation that broken test dependencies that are manifested in parallel runs involve test cases from the same test class. PASTE leverages that observation to reinstate dependencies by running the tests from the same class in the same order. PASTE is organized as a pipeline of three stages. In the first stage, PASTE runs tests in parallel using any parallel configuration option provided by the build system. In the second stage, PASTE runs the test cases that failed at the first stage sequentially with the goal of circumventing failures due to data races. Finally, in the third stage, PASTE runs the test classes with any test that failed at the second stage sequentially. The intuition is that the test failures at the second stage are due to broken test dependencies manifested within the same test class. On the hypothesis that broken test dependencies involve tests declared in the same class that strategy would address the failures provoked by test parallelization that were not addressed on the second stage.

We evaluated PASTE on 25 projects mined from GitHub. Results show that (1) PASTE could circumvent flakiness introduced with parallelization in all projects that manifested them and that (2) the best parallel execution mode of PASTE was faster than the sequential execution mode by a factor of 1.59x. These results indicate that (i) our proposal of alternating parallel and sequential execution of test cases and classes is effective and that (ii) the overhead incurred from executing tests with PASTE is acceptable. To sum up, this paper provides initial yet strong evidence that PASTE can be used to assist developers in using parallel options of build systems "out of the box". The artefacts produced in our experiments are publicly available at the following URL: https://github.com/STAR-RG/paste.

The rest of this paper is organized as follows. Section II provide technical background for the following sections. Section III presents PASTE. Sections IV–VII report on our evaluation and Section VIII discusses implications. Finally, sections IX and X discuss related work and conclusions.

II. Background

A. Test Parallelization

Parallelism for test execution can be obtained at various levels. For instance, test parallelization can be obtained within a single machine or across several machines (e.g., using virtual machines from a cloud service). This paper focuses on parallelism within a single machine, where computation can be offloaded at different CPUs within that machine and at different threads within each CPU. This form of parallelism is enabled through build systems and testing frameworks. In the following, we elaborate relevant features of these tools for parallelization. We focused on Java, Maven, and JUnit but the discussion can be generalized to other languages and tools.

1) Testing Frameworks: Test frameworks are a piece of software to facilitate the construction of automated tests. A variety of testing frameworks support parallel test execution (e.g., JUnit [12], TestNG [15], and NUnit [16]) as to benefit from the available processing power in the machine that executes the tests. The list below shows the choices provided by JUnit for test parallelization:

- **`sequential`.** No parallelism is involved.
- **`methods`.** This configuration corresponds to running test classes sequentially, but running test methods from those classes concurrently.
- **`classes`.** This configuration corresponds to running test classes concurrently, but running test methods from those classes sequentially.
- **`classesMethods`.** This configuration corresponds to running test classes and test methods concurrently.

It is worth noting that these options are restricted to *one* Java Virtual Machine (JVM), the JVM that executes the tests. Notice that an important aspect in deciding which configuration to use (or in designing new test suites) is the possibility of race conditions on shared data during execution. That can occur, for example, through state that is reachable from statically-declared variables in the program or through variables declared within the scope of the test class or even through resources available on the file system and the network [17].

2) Build Systems: Forking OS processes to run test jobs is the basic mechanism of build systems to obtain parallelism at the machine space. For Java-based build systems, such as Maven and Ant, this amounts to spawning JVMs on a host CPU to handle test jobs and then aggregating test results when these jobs finish. Forking can only be combined with configuration `methods` (see Section II-A1) as Maven made the design choice to only accept one test class at a time per forked process. As such, parallelizing the execution of test classes within a forked JVM process would not make sense as there is only one test class to be executed per process. Maven offers an option to reuse JVMs (`reuseForks`) that can be used to attenuate the potentially high cost of spawning new JVMs, but that increases the chances of tests accessing polluted state created by other test runs [18], [19].

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artefactId>maven-surefire-plugin</artefactId>
    <configuration>
        <forkCount>1C</forkCount>
        <reuseForks>true</reuseForks>
        <parallel>methods</parallel>
        <threadCount>5</threadCount>
    </configuration>
</plugin>
```

Figure 1: Example Maven configuration file.

Figure 1 shows a fragment of a Maven configuration file, known as `pom.xml`, highlighting options to run tests in parallel. Maven implements this feature through its Surefire JUnit test plugin [20]. With this configuration, Maven forks one JVM per core (`forkCount` parameter) and uses five threads (`threadCount` parameter) to run test methods (`parallel` parameter) within each forked JVM. Recall that only one test class runs at a time per JVM. Maven reuses created JVMs on subsequent forks when the execution of a test class terminates (`reuseFork` parameter).

B. Test Dependencies

A test is said to depend on another test when it reads state previously written by another test. These dependencies can manifest through shared state [22], reachable from the fields declared in the test class or from the static fields declared in classes of the application.

Test case dependencies are undesirable. In a regular (non-parallel) execution, for example, they can result in inability to run tests in isolation [10], [13], a feature that is often important

```
 1 public class MasterReplicationCoordinatorTest {
 2   static AccumuloConfiguration config = DefaultConfiguration.getInstance();
 3
 4   @Test
 5   public void randomServer() {
 6     Master master = EasyMock.createMock(Master.class);
 7     ZooReader reader = EasyMock.createMock(ZooReader.class);
 8     ServerContext context = EasyMock.createMock(ServerContext.class);
 9     EasyMock.expect(context.getConfiguration()).andReturn(config).anyTimes();
10     EasyMock.expect(master.getContext()).andReturn(context);
11     EasyMock.expect(master.getInstanceID()).andReturn("1234");
12     EasyMock.replay(master, reader);
13
14     MasterReplicationCoordinator coordinator = new MasterReplicationCoordinator(master, reader);
15     TServerInstance inst1 = new TServerInstance(HostAndPort.fromParts("host1", 1234), "session");
16
17     assertEquals(inst1, coordinator.getRandomTServer(Collections.singleton(inst1), 0));
18   }
19   ...
20   @Test
21   public void randomServerFromMany() {
22     Master master = EasyMock.createMock(Master.class);
23     ZooReader reader = EasyMock.createMock(ZooReader.class);
24     ServerContext context = EasyMock.createMock(ServerContext.class);
25     EasyMock.expect(context.getConfiguration()).andReturn(config).anyTimes();
26     EasyMock.expect(context.getInstanceID()).andReturn("1234").anyTimes();
27     EasyMock.expect(context.getZooReaderWriter()).andReturn(null).anyTimes();
28     EasyMock.expect(master.getInstanceID()).andReturn("1234").anyTimes();
29     EasyMock.expect(master.getContext()).andReturn(context).anyTimes();
30     EasyMock.replay(master, context, reader);
31
32     MasterReplicationCoordinator coordinator = new MasterReplicationCoordinator(master, reader);
33     EasyMock.verify(master, reader);
34     TreeSet<TServerInstance> instances = new TreeSet<>();
35     TServerInstance inst1 = new TServerInstance(HostAndPort.fromParts("host1", 1234), "session");
36     instances.add(inst1);
37     TServerInstance inst2 = new TServerInstance(HostAndPort.fromParts("host2", 1234), "session");
38     instances.add(inst2);
39
40     assertEquals(inst1, coordinator.getRandomTServer(instances, 0));
41     assertEquals(inst2, coordinator.getRandomTServer(instances, 1));
42   }
43 }
```

Listing 1: Test `randomServer` depends on test `randomSeverFromMany` [21].

when the developer needs to debug code from a given test case. In a parallel execution, the infrastructure (i.e., build system and test framework) runs the tests concurrently. As result, tests can fail either (1) because of broken test dependencies or (2) because of data races. In both cases, the shared state reachable from the tests triggers these problems.

1) Example: Listing 1 shows a code snippet from the test class `MasterReplicationCoordinatorTest` from the Apache project `accumulo` [23], a distributed data store to support key-value information retrieval. Accumulo stores the data in multiple tables using a cluster of server instances. The class `MasterReplicationCoordinator` is responsible for coordinating server replicas. The method `getRandomTServer`, for instance, returns a random server instance from the cluster. The tests `randomServer` and `randomServerFromMany` are similar; they create the coordinator object and check if method `getRandomTServer` answers with a valid server instance. The difference between these tests is that `randomServer` tests the behavior of the class `MasterReplicationCoordinator` with one server instance whereas `randomServerFromMany` tests the behavior with two server instances.

The test `randomServer` fails when executed in isolation. That happens because `randomServer` depends on the test `randomServerFromMany`. In a scenario where tests are executed in parallel, `randomServer` may also fail because the test infrastructure can execute test methods in any order. The problem in this example is that the test `randomServer` does not use the EasyMock Mocking library [24] properly whereas the test `randomServerFromMany` does the proper setup. When `randomServerFromMany` executes first the problem is not observed as the setup of the library persists across tests.

More in detail, developers often define behavior of mock objects in Mocking libraries through expectations, such as the ones declared on Listing 1 using the method `expect`. For these expectations to have an effect, the developer needs to call the method `replay` after defining the expectations, passing the mock objects as argument. For example, note that the mock objects `master`, `context`, and `reader`, created at lines 22–24, are passed as argument to the call to method `replay` at line 30. In contrast, the mock object referred by variable `context`, declared and initialized at line 8, is *not* passed as argument to the method `replay` at line 12. Because of this mistake, any call made by the test during execution to the method `ServerContext.getConfiguration()` returns `null` instead of a reference to the object `config`, as the developer would certainly hope (as per line 9). The code of the application does not expect that method to ever return `null`[1] and attempts to dereference the re-

[1]In fact, this is only possible using mocks to bypass the original behavior.

turned (`null`) reference, raising a `NullPointerException` (NPE). That exception is raised in the execution of the constructor `MasterReplicationCoordinator` at line 14. When the test `randomServerFromMany` is executed first, the setup of the mocks is properly done and the mock library returns a non-null object upon calls to `ServerContext.getConfiguration()`. That object is stored in a static field declared in the application code, persisting across test executions. That same object is returned instead of `null` and the NPE exception is no longer triggered.

III. PASTE

This section presents PASTE (PArallel-Sequential Test Execution), a lightweight approach to introduce test parallelization in Java projects. PASTE builds on the observation that broken test dependencies that are manifested in parallel runs involve test cases from the same test class. Section II-B showed a representative example of that scenario. The failure manifested with the parallel execution of project `accumulo` involves a test declared in the class `MasterReplicationCoordinatorTest` that is dependent on another test declared in the same class. Intuitively, that occurs because developers tend to write related test cases in the same test classes. PASTE leverages that observation to reinstate dependencies by running the tests from the same class in the same order, if needed. PASTE makes the assumption that a relatively low number of test failures are manifested in a parallel execution of the test suite. Intuitively, the higher the number of test failures is –relative to the size of the test suite– the lower the chances of observing speedups with parallelization.

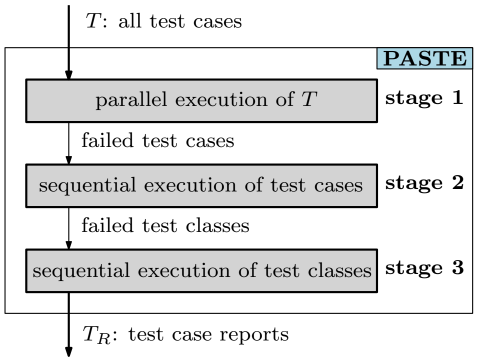

Figure 2: PASTE.

PASTE is organized as a pipeline of three stages. Figure 2 shows the pipeline, with each stage represented as a gray rectangle. PASTE takes as input the full set of test cases T and reports as output the outcomes of the test runs T_R. All stages of the pipeline consume a set of test cases on input and report a set of test cases on output. In the **first stage**, PASTE runs the test suite in a given parallel mode of execution provided by the build system. In Maven, that translates to selecting an option for the parameter `-Dparallel` (e.g., `classes`, `methods`, etc.). If all tests pass in this stage, execution stops. In the **second stage**, PASTE executes *sequentially the test cases that failed on the first stage*. Intuitively, a test that failed on the first stage because of data races and is not dependent on another test case should pass in this stage. If all tests pass in this stage, execution stops. In the **third stage**, PASTE executes *sequentially the test classes from the tests that failed on the second stage*. The failures that reached this point are due to broken test dependencies. As previously mentioned, the hypothesis is that test dependencies involve test cases declared in the same test class. As such, failed test cases will run in the same order —within a given test class— they ran in the original sequential setup. Executing the entire test class instead of individual tests reinstates the test dependencies at the expense of duplicate effort. More precisely, at this stage, PASTE executes test classes in isolation to workaround test failures due to broken test dependencies. At last, PASTE reports a list of test failures on output.

Tool availability and replication support

Our tool was implemented on top of bash and traditional UNIX utilities like `grep`, `sed`, `awk`, etc. It currently supports Maven projects. The artefacts are made publicly available at the following URL: https://github.com/STAR-RG/paste.

IV. Method and Objects of Analysis

This section details the method and objects we used to conduct our experiments.

A. Environment

We performed our experiments on a machine powered by a tenth-generation Intel Core i5-1035G1 CPU @ 1.00GHz (base frequency), 8 CPUs (4 cores, with 2 threads per core), 8 GB RAM, and with an SSD storage of 512 GB. Software used were `Linux` kernel version 5.4.0-42-generic, and `OpenJDK` 1.8.0_282 for Java. We implemented PASTE on top of GNU `bash` 5.0.17, and `maven` 3.6.3.

B. GitHub Projects

To build the dataset for our empirical study, we mined 25 Java projects that use Maven and have at least 200 stars and 300 tests. We focused on Apache projects written in Java [25], but found only 21 projects satisfying the criteria specified. The Apache foundation is recognized for their diligence in maintaining projects. To complete the list of 25 projects, we randomly selected 4 projects from GitHub that satisfied these requirements. Table I shows the projects we selected. The suffix "(A)" indicates an Apache project. Column "#" shows the id of the project, column "Name" shows the name of the project, column "# Stars" shows the number of stars of the project on GitHub, column "# Tests" shows the number of tests of that projects, and column SHA shows the first 7 digits of the hash of the commit we used to analyze that project.

Table I: Open-source Java projects. The suffix “(A)” indicates that the project is maintained by Apache.

#	Name	# Stars	# Tests	SHA
1	`accumulo(A)`	861	514	76247b1
2	`atlas(A)`	839	1422	acb9880
3	`avro(A)`	1807	10446	5bd7cfe
4	`biojava`	438	811	4d1cf58
5	`cayenne(A)`	250	2084	54cb1f9
6	`chronicle-queue`	2291	328	8754ad3
7	`commons-collections(A)`	475	16923	3aae82c
8	`commons-io(A)`	767	1840	c1ee777
9	`datasketches-java(A)`	706	1490	dab9542
10	`dubbo(A)`	34954	3519	b5c81d8
11	`httpcomponents-client(A)`	1040	1865	bde58d6
12	`iotdb(A)`	1255	422	6f7eac8
13	`kylin(A)`	3015	1057	d6073d2
14	`maven(A)`	2490	1053	276c6a8
15	`mina(A)`	776	371	daf2a33
16	`mina-sshd(A)`	394	1790	a0bbdf9
17	`opennlp(A)`	1024	791	7286f9c
18	`pdfbox(A)`	1403	1849	9daeaf6
19	`ranger(A)`	489	552	58b51a3
20	`ratis(A)`	460	444	0c9913f
21	`rocketmq(A)`	13740	372	3ae2517
22	`shiro(A)`	3419	856	a85dfcd
23	`strata`	603	16277	050745d
24	`soul`	3666	1081	a99c9fc
25	`wicket(A)`	551	2699	34f78c8
	Σ	-	**70856**	-

C. No Failures

To make sure the failures we observe during the execution of PASTE are result of parallelization as opposed to manifestations of bugs in the application or manifestations of flaky tests, we used the most recent release of the application (to avoid deterministic failures because of code “in flux”) and reran each test suite for 10 times to identify and eliminate flaky tests (to avoid non-deterministic failures). With this setup, we wanted to ascertain that all tests from the projects we selected pass when executed in regular (i.e., non-parallel) mode.

D. Parallel Configurations

We focused on parallel configurations provided by the test framework as opposed to configurations provided by the build system. As such, we fixed parameters the following parallel configuration parameters provided by the build system: `forkCount=1C`, `reuseForks=true`, and `threadCount=7`. Section II-A1 describes the configuration parameters of the test framework in detail. Recall that the test framework provides features to explore parallelism within the JVM. The rationale for our choice is that JVM parallelism is capable of manifesting the flakiness issues we discussed and is more fundamental compared to forking. For the sake of completeness, we focused on these configurations, but there is no fundamental reason why PASTE cannot be used with forking. Conceptually, forking reduces contention and can explore more effectively the cores of the machine as each JVM runs in a separate process. However, there is non-negligible cost in spawning different JVMs for execution [19].

V. Research Questions

We pose the following research questions to evaluate PASTE. The first three research questions address the feasibility of the approach whereas the last two questions address its effectiveness.

A. Feasibility

RQ1: Is it feasible to use parallelization options provided by the build system “out of the box” to run test suites? PASTE would be unnecessary if parallel and sequential runs consistently produce the same outputs.

RQ2: Is it practical to use a test dependency analyzer to partition test sets as to enable sound parallel execution? Simply rerunning the tests that failed during parallel execution *in isolation* is not an acceptable approach to circumvent failures provoked by parallel test runs. Test dependencies could be broken and tests would fail for that reason [13]. This research question evaluates the plausibility of an alternative strategy to parallelize test runs: (1) Run tests in parallel (in any execution mode), (2) Identify failed tests, (3) Topologically sort the dependency subgraph associated with the transitive closure of test dependencies from failed tests,[2] and (4) Rerun tests sequentially, according to that ordering. In contrast to PASTE, that strategy is sound. The goal of this question is to assess how computationally efficient is that approach.

B. Effectiveness

RQ3: How reliable is PASTE? This question evaluates whether one could reliably run test suites in parallel with PASTE. Conceptually, if relevant test dependencies are manifested across test classes then there should be persistent test failures that PASTE would be unable to circumvent. This research question evaluates whether such test dependencies arise during test execution. The goal is to demonstrate that the execution of a test suite with PASTE produces the same results compared with a sequential execution.

RQ4: What are the speedups obtained with PASTE? Finally, we investigate what are the speedups obtained with the various parallelization strategies used by PASTE. The rationale for the question is to show that the overhead introduced with PASTE’s infrastructure is not unacceptably high to defeat the purpose of parallel executions.

VI. Results

A. Answering RQ1: Is it feasible to use parallelization options provided by the build system “out of the box” to run test suites?

The rationale for this question is that PASTE would be unnecessary if developers could always find, for a given project, a parallel configuration whose test runs produce the same outputs as sequential runs. Given that we knew a priori that all tests in the sequential execution pass, consistency between parallel and sequential runs can be determined by verifying that all tests pass in parallel executions. To answer this question, we ran the test suite of each of the 25 projects for 5 times on each parallel configuration we considered in this study (Section IV-D). Column “#1” from Table II shows

[2]We used PRADET [22], [26] to compute test dependencies.

Table II: Number of failures after every stage of the pipeline. Columns "`classes`", "`methods`", and "`classesMethods`" show the options used for the parameter `-Dparallel` at stage #1. Stage #2 runs failing test cases sequentially whereas stage #3 runs test classes associated with failing tests cases sequentially. A dash indicates that a stage was skipped.

project	# tests	`classes`			`methods`			`classesMethods`		
		#1	#2	#3	**#1**	#2	#3	**#1**	#2	#3
`accumulo`	514	0.0	–	–	6.0	1.4	0.0	6.8	2.4	0.0
`atlas`	1422	10.4	0.0	–	147.2	6.0	0.0	0.0	–	–
`avro`	10446	0.0	–	–	0.0	–	–	0.0	–	–
`biojava`	811	2.6	0.0	–	8.6	1.0	0.0	17.6	0.0	–
`cayenne`	2084	8.2	0.0	–	22.8	0.0	–	26.4	0.0	–
`chronicle-queue`	328	3.2	0.0	–	74.4	0.0	–	65.4	0.0	–
`commons-collections`	16923	0.0	–	–	0.0	–	–	0.0	–	–
`commons-io`	1840	0.0	–	–	0.0	–	–	0.0	–	–
`datasketches-java`	1490	0.0	–	–	10.2	0.0	–	0.0	–	–
`dubbo`	3519	1.6	0.0	–	5.4	0.0	–	0.8	0.0	–
`httpcomponents-client`	1865	0.2	0.0	–	1.4	0.0	–	5.4	0.0	–
`iotdb`	422	30.6	0.0	–	TO	–	–	TO	–	–
`kylin`	1057	321.2	0.0	–	401.0	0.0	–	401.6	0.0	–
`maven`	1053	0.0	–	–	0.0	–	–	0.0	–	–
`mina`	371	0.0	–	–	1.8	0.0	–	0.0	–	–
`mina-sshd`	1790	1.4	0.0	–	1.4	0.0	–	1.6	0.0	–
`opennlp`	791	0.0	–	–	0.0	–	–	0.0	–	–
`pdfbox`	1849	0.0	–	–	0.0	–	–	0.0	–	–
`ranger`	552	1.6	0.0	–	48.4	0.0	–	47.0	0.0	–
`ratis`	444	16.2	0.0	–	107.4	0.4	0.0	106.8	0.4	0.0
`rocketmq`	372	5.4	0.0	–	16.0	0.0	–	18.4	0.0	–
`shiro`	856	0.0	–	–	0.0	–	–	0.0	–	–
`strata`	16277	0.0	–	–	0.0	–	–	0.0	–	–
`soul`	1081	0.0	–	–	5.6	0.0	–	6.6	0.0	–
`wicket`	2699	0.0	–	–	22.0	0.0	–	26.8	0.0	–
Σ		**402.6**	**0.0**	–	**879.6**	**8.8**	**0.0**	**731.2**	**2.8**	**0.0**

results. This column shows the average number of test failures observed across runs of the corresponding test suite of a project in a given parallel execution configuration. We found that in 11 of the 25 projects (=44%) all three configurations produce failures. We highlighted in gray color the name of these projects on the table. Results suggest that it is not the case that there is one parallel configuration for every project that would make all tests pass. It is worth noting that the configuration `classes` (Section II-A) is the one that could execute the test suite with no failures more often (row Σ, column "#1"). Recall that the configuration `classes`, in contrast to the other two configurations, runs the test methods from each test class sequentially. To sum up:

Summary: In 44% of the projects we analyzed, no parallel configurations enabled a clean execution, i.e. an execution without test failures. This result shows that searching for the parallel configuration that makes test outputs consistent with those of a sequential execution is infeasible in general.

B. Answering RQ2: Is it practical to use a test dependency analyzer to partition test sets as to enable sound parallel execution?

The previous research question showed that parallel execution of tests often do not work "out of the box". Tests can be executed in different orderings breaking test dependencies and test can access shared data that has not been protected for concurrent access. Both circumstances can result in test failures during parallel executions.

Conceptually, one can circumvent failures due to broken test dependencies using the following approach: (1) Run tests in parallel, (2) Identify failed tests, and (3) Topologically sort the dependency subgraph associated with the transitive closure of test dependencies from the failed tests, and (4) Rerun tests sequentially, according to that ordering. If that alternative is acceptable, it would be unnecessary to run PASTE. That approach is sound, in contrast to PASTE, which cannot provide the guarantee that the test output produced will be identical with those obtained in sequential runs. We implemented that strategy to empirically confirm soundness and it was indeed able to circumvent failures. For that, we used PRADET, which is the state-of-the-art technique to compute test dependencies [22], [26].

It remained to evaluate how efficient the approach described above was. PRADET computes dependencies with a lightweight dynamic analysis. Unfortunately, we found that the cost of PRADET to support parallelization is prohibitive. The analysis of PRADET consists of three stages: (1) Run tests sequentially to obtain the original order of execution of tests, (2) Run a dynamic dataflow analysis to collect data dependencies from tests, (3) Run an iterative dependency refinement algorithm, which filters out all the *non-problematic* [22] data dependencies. A problematic test dependency is one which, if broken, toggles the verdict of the *dependent* test case. All other dependencies are non-problematic.

Table III shows the running times of PRADET broken down by each one of PRADET's steps. Step 1 is a proxy for the cost of a sequential execution. Assuming that only steps 2 and 3 need to re-executed on every test run to

Table III: PRADET's execution time (in seconds). The symbol "E" denotes an error.

project	#Deps.	Step 1	Step 2	Step 3
accumulo	5512	246	66	1904
atlas	0	190	3138	1358
avro **(E)**	-	-	-	-
biojava	184	574	550	7896
chronicle-queue	251	127	134	165
commons-collections	2	74	55	744
commons-io	0	194	3	22
cayenne	1092	134	161	2709
datasketches-java **(E)**	-	-	-	-
dubbo	0	85	53	92
httpcomponents-client	465	73	26	249
iotdb	0	195	61	40
kylin **(E)**	-	-	-	-
maven	0	33	110	182
mina	90	72	66	80
mina-sshd	1805	2052	477	1562
opennlp	41	72	61	1563
pdfbox	0	166	29	114
ranger	1529	411	331	10461
ratis	158	1750	1482	876
rocketmq	110	175	52	645
shiro	16	58	53	105
soul	32	65	34	222
strata	0	285	30	327
wicket	17	183	83	381

obtain the updated set of test dependencies,[3] the sum of costs of these two steps far exceeds the cost of Step 1, which fairly approximates the cost of a sequential execution of the test suite. To sum up, the smaller observed runtime of an execution in PRADET is higher than the execution time of a sequential test suite execution, rendering the sound approach described above to run tests impractical. Consider projects chronicle-queue and commons-collection, for illustration. chronicle-queue is a project manifesting a high number of test dependencies (251) through a low number of test cases (338), and commons-collection is a project with a very low number of test dependencies (2) through a very high number of test cases (16,923). The cost of running Step 3 alone is higher than the cost of running Step 1. It is important to note that we were unable to run PRADET on projects avro, datasketches-java, and kylin. The suffix "(E)" on the project name indicates that PRADET generated an error message and stopped execution before reporting dependencies.

Summary: The cost of running PRADET, which is the state-of-the-art tool to compute test dependencies, is substantially higher than the cost of running tests sequentially. It is therefore not practical to use PRADET to support test parallelization.

C. Answering RQ3: How reliable is PASTE?

This question evaluates whether it is possible to run a test suite with PASTE and consistently obtain the same results of a sequential test execution.

Table II shows the number of failures observed at each given stage of PASTE averaged across 5 runs. A "–" on the table indicates that PASTE was interrupted prior to that stage as some previous stage detected no failures. For illustration, let us consider the case where PASTE is executed with the configuration methods on project atlas. A total of 147.2 failures, on average, are observed at the end of the first stage. Most of these failures occur because of the contention created by PASTE to execute multiple test methods concurrently, leading to data races. To address that, PASTE sequentially executes the tests that failed on the first stage. At the end of the second stage, 6 failures are still observed, on average. These failures are associated to the six tests that consistently fail in all of the 5 runs. The test searchByALLTagAndIndexSysFiltersToTestLimit is among these six cases. We found that that test fails *when it is executed before* the test searchByALLTag. Listing 2 shows these two test cases. Note that the final statement of the test searchByALLTag updates the variable totalClassifiedEntities and that that variable is later read on the other dependent test. There is a read-write test order dependency between these tests manifested through this test field. When searchByALLTagAndIndexSysFiltersToTestLimit executes before searchByALLTag, the call to the method setLimit (line 20) uses -2 as argument, which is wrong. As result, the object params will store the incorrect value and the assertion at line 25 fails.

```
 1 public class ClassificationSearchProcessorTest ... {
 2  private AtlasGraph graph;
 3  private int totalClassifiedEntities = 0; ...
 4
 5  @Test(priority = -1)
 6  public void searchByALLTag() throws AtlasBaseException {
 7   ...
 8   params.setLimit(20);
 9   SearchContext context =
10     new SearchContext(params, graph, ...);
11   ClassificationSearchProcessor processor = new
        ClassificationSearchProcessor(context);
12   List<AtlasVertex> vertices = processor.execute();
13   Assert.assertTrue(CollectionUtils.isNotEmpty(vertices));
14   totalClassifiedEntities = vertices.size();
15  } ...
16
17  @Test
18  public void searchByALLTagAndIndexSysFiltersToTestLimit()
        throws AtlasBaseException {
19   ...
20   params.setLimit(totalClassifiedEntities - 2);
21   SearchContext context =
22     new SearchContext(params, graph, ...);
23   ClassificationSearchProcessor processor = new
        ClassificationSearchProcessor(context);
24   List<AtlasVertex> vertices = processor.execute();
25   Assert.assertTrue(CollectionUtils.isNotEmpty(vertices));
26  }...}
```

Listing 2: Test Order Dependency on Atlas [27].

It is worth noting that developers used the parameter priority of annotation @Test to make it explicit that the test searchByALLTag should execute before any other test declared at the same class. PASTE circumvented the provoked breakage of a test order dependency in its final stage by respecting the order of tests declared in the test class.

Note that the number of failures observed on the first

[3]This would be possible when no new test cases are added or removed.

stage with configurations `methods` and `classesMethods` are much higher (879.6 and 731.2 respectively) compared to the number of first-stage failures observed with configuration `classes` (402.6). That occurs because configuration `classes` is not as aggressive in using the CPU resources as the other configurations. Overall, results indicate that executing individual test cases sequentially significantly reduces the number of observed failures does to data races (columns "#1" and "#2", row Σ). Also, note that executing test classes sequentially is effective to circumvent failures caused by broken test dependencies (columns "#2" and "#3", row Σ). To sum up:

Summary: The strategies adopted by PASTE of executing test cases and test classes sequentially are effective to circumvent the test flakiness provoked by test parallelization. Considering all projects and configurations, there were no cases of provoked failure that "survived" the third stage of PASTE.

D. Answering RQ4: What are the speedups obtained with PASTE*?*

Finally, this question investigates the speedups obtained with the various parallelization strategies used by PASTE. It is worth noting that PASTE does *not* implement parallelization features, such as forking JVMs, spawning threads, or scheduling the execution of test classes and test cases to each of these computation units. Those features are already provided by the build system and testing frameworks. Instead, the focus of PASTE is to facilitate the use of those features in projects that were not prepared to be executed in parallel. It is also worth noting that the speedups observed with parallelization depend on a number of factors, including the prevalence of CPU-intensive tasks and the number of available cores in the machine that executes the tests. The rationale for this question are that (1) PASTE would not be useful without observing speedups and (2) it is important to observe the overhead associated with the additional test runs required by PASTE.

Table IV shows the 13 of the 25 projects (=52%) for which we observed speedups in at least one of PASTE's configuration. Column "Time" shows the time to run the test suite sequentially, column "Speedup" shows the ratio s/p, where s denotes the sequential execution time in seconds and p denotes the parallel execution time in seconds. As can be noted, PASTE (and test parallelization) cannot guarantee faster execution of test suites. However, there are several projects where it can be beneficial. Results clearly indicate that the configuration `classes` was the one that offered the best trade offs. The median speedup obtained with that configuration was 1.59x. Figures 3 and 4 (respectively) show, respectively, the distribution of speedups and execution times of PASTE on the same projects listed on Table IV. Although there were specific scenarios –across different configurations– where PASTE did not yield end-to-end acceleration, there were important cases where acceleration was observed.

Table IV: Projects that achieved speedup for at least one configuration for parallel test-execution.

project	Time (sequential)	Speedup `classes`	`methods`	`classesMethods`
`accumulo`	2m50s	1.23	0.92	0.93
`atlas`	1h22m12s	0.99	0.89	1.08
`biojava`	5m02s	1.59	1.43	1.49
`chronicle-queue`	1m20s	1.99	1.48	1.69
`commons-collections`	34s	1.10	1.06	1.09
`commons-io`	2m24s	1.00	1.03	1.02
`datasketches-java`	26s	1.05	0.82	1.02
`dubbo`	11m27s	0.93	0.84	1.03
`kylin`	10m38s	1.63	1.13	1.32
`maven`	59s	2.28	2.25	2.24
`mina`	49s	1.84	1.47	1.83
`ratis`	17m50s	1.87	1.20	1.15
`wicket`	3m19s	1.59	1.49	1.36
Average		**1.47**	1.23	1.33
Median		**1.59**	1.13	1.15

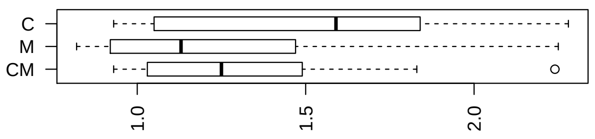

Figure 3: Distribution of speedups for each configuration (Classes, Methods, ClassesMethods), associated with Table IV.

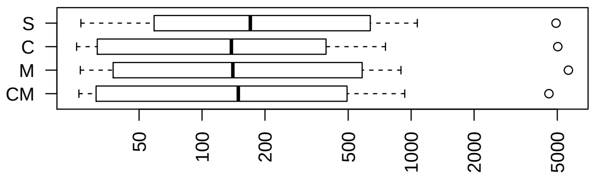

Figure 4: Distribution of PASTE running times (seconds) for (Sequential) and each configuration (Classes, Methods, ClassesMethods), associated with Table IV.

Summary: Results indicate that not all projects can benefit from test parallelization. Considering the projects we selected and our current implementation, we observed speedups in 52% of the projects. The configuration `classes` yielded higher efficiency overall, with average and median speedups of 1.47x and 1.59x, respectively.

VII. Threats to Validity

This section discusses the limitation of our approach and their corresponding threats to validity.

a) External validity: The benefits of PASTE may not generalize beyond the observations we made with our data set of programs. To mitigate this threat, we established a rigorous selection criteria and included a reasonably high number of projects. We did observe during the inspection of `pom.xml` files that projects differ in characteristics. For example, test suites where most tests are IO intensive are hard to parallelize as most tests depend on a limited resourced (e.g., the network, the disk). We did find that some of the projects have these characteristics—PASTE could accelerate nearly half of the test suites in our data set. For those cases, parallelization within a machine is unlikely to help. Cloud parallelization is a route to deal with this limitation, but support on Maven is still limited [28]. Another threat to external validity is the choices of parallel configurations we used to evaluate PASTE. We used a representative set of options that explores the resources available within each process. Although we can explore forking, there are no reasons to believe results would invalidate our findings.

b) Internal validity: In principle, there could be bugs in the implementation of PASTE that we missed. To mitigate this threat, the authors cross-verified the implementation scripts and their results before reaching consensus. The authors also debugged specific cases of broken dependencies—discussed as examples on this paper—to ascertain the credibility of results.

c) Construct validity: Construct validity refers to the degree to which inferences can be made from the proposed approach and its measures. The key metrics we used to assess PASTE were (i) the ability to consistently report the same results in parallel and sequential executions and (ii) the speedup obtained with the parallelization obtained with PASTE. The rationale for (i) is that it is important that developers trust results and the rationale for (ii) is that parallelization is only relevant if it can accelerate test execution. We believe these are the most important metrics that developers care when considering test parallelization.

VIII. Key Observations and Implications

The following are the main observations from our study:

- Not all projects can benefit from test parallelization. Considering the projects we selected, nearly half of them benefited from it and only few reported a significant speedup. For example, some projects are IO intensive and some projects have short-running test suites, for which the overhead of parallel runs does not pay off. It is important to note that projects with short-running test-suites are unlikely to be considered for parallelization. However, they can still be utilized for evaluation.
- Rerunning test cases (to avoid data races) and rerunning test classes (to avoid broken test dependencies) after a parallel execution of a test suite is a lightweight approach to speed up the execution of test suites with parallelization.

The following are the main implications of this work:

- Developers can use PASTE as a tool to find and debug *manifested test dependencies*, i.e. test dependencies that manifest a difference during the execution of the test suite with PASTE. After all such dependencies have been eliminated, the developer could use the original configuration options provided by the build system to execute tests in parallel.
- Developers that prefer not to address manifested test dependencies, can opt to use PASTE as is and still observe speedups.

IX. Related Work

In this section, we qualitatively compare and contrast PASTE with the relevant related work.

A. Test Selection (Minimization, and Prioritization)

Regression testing has been an active research topic [8], [29] in the domain of software engineering, both in industry and academia. There have been a plethora of techniques that address the problems of test-suite selection, prioritization, reduction. Largely, the soundness, i.e., the reliability of the proposed solutions comes without a guarantee. This means that the test cases exercising the subject may be uncover some underlying faults. Ekstazi [6], [30] is a sound regression test selection technique that dynamically detects all file dependencies. The core idea is to maintain a dependency matrix relating files and tests. Upon changes in files, which can be efficiently detected by reading file checksums, Ekstazi uses the matrix to determine which tests require execution. Ekstazi uses file coverage information to build and update the dependency matrix. Consequently, there is a small overhead imposed in the execution of tests, but results indicate that that overhead pays off.

B. Test Parallelization

Parallelization of software testing is complementary to selection, prioritization, and reduction but had remained unexplored until recently despite the fact that parallelization options existing in most build systems and testing frameworks. Candido et al. [7] studied the impact of the adoption of parallelization in Java projects. They showed (1) that the cost of executing a test suite in parallel mode can be often reduced, (2) that reduction in runtime cost depends on a number of factors, including the characteristic of the project (e.g., CPU intensive, IO intensive, etc.) and the characteristic of the tests (e.g., several tests, long-running tests, etc.) and (3) that the running test suites in parallel without changing code and tests can introduce flakiness.

The use of the Single Instruction Multiple Data (SIMD) design has been previously explored in research to accelerate test execution [31]–[36]. The SIMD architecture, as implemented in modern GPUs, for instance, allows the execution of a given instruction simultaneously against multiple data. For that reason, in principle, one test could be ran simultaneously against multiple inputs provided that multiple test inputs exist associated to that one test. Recent work [34], [36] explored that idea to speedup test execution of embedded software using graphic cards. Although benchmarks indicate superior performance compared to traditional multicore CPUs, the use of the technology in broader settings is limited. For example, execution of more general programs can violate the SIMD's lock-step assumption on the control-flow of threads. This violation would affect negatively performance. Furthermore, handling complex data is challenging in SIMD [31], [37]. The approach is promising when multiple input vectors exist for each test and the testing code heavily manipulates scalar data types. The datasets used in those papers satisfied those constraints.

Mahtab [38] and Hansie [39] are two approaches that introduce test-parallelization windows to perform multi-core acceleration at isolated process-level parallelism with independent unit test cases. Although isolated address space per process helps circumvent the issues with test-order dependencies and shared polluted state, this approach is limited to the nature of the subjects used in the study [38], [39], and does not generalize externally. PASTE being dependency-agnostic does not suffer from this limitation as the remedy comes in the form of test re-executions in subsequent stages is any test case dependency is violated. Currently, PASTE involves test-parallelization While Mahtab and Hansie combine test-selection, prioritization, and parallelization. Yet another recent tool for test-parallelization of C projects is due to Schwahn et al. [40]. Unlike Mahtab and Hansie, this tool performs static analysis to compute test case dependencies on files and globally shared variables. Specific emphasis is given on determining parts of a test-suite that have sound parallel execution with `make`. Comparatively, the pipeline of PASTE operates on the basis of a dynamic analysis that employs unrestricted parallelism devoid of dependency-preservation. Our empirical evaluation shows that this approach is soundy, i.e., without a soundness guarantee but practically reasonable.

C. *Dependency Tracking*

ElectricTest [10] is a tool for efficiently detecting data dependencies across test cases. Dependency tracking is important as to avoid test flakiness when parallelizing test suites. ElectricTest observes reads and writes on global resources made by tests to identify these dependencies at low cost. PRADET [22] is an improvement over ElectricTest. Unfortunately, results indicate that the cost of computing test dependencies is typically higher than the cost of running the tests sequentially, showing that the cost of using PRADET to determine which tests need to be rerun is prohibitive.

TEDD [41] is an NLP-based test-dependency detection tool targeting web applications which pose different challenges as program states are manipulated through client-server network operations which are very different form desktop applications performing in-memory or disk accesses. We remain to investigate how PRADET can be adapted to specific domains.

ForkScript [19] shows that Apache `maven` preferably circumvents the issue of test case dependency during parallelization by executing each test cases as a separate process, by forking JVMs. However, generic inter-process communications due to the underlying build system turns out to be a performance bottleneck along side the overhead of forking during parallelization. The solution deployed—patched `maven-surefire-plugin v3.0.0-M5`—optimizes forking by dynamically generating relevant test-execution code rather than a generic one. Comparatively, PASTE currently does not perform this optimization but there is no reason why it cannot. The consequences of this change may be observed as an improvement in the end-to-end speedup due to parallelization or reduction in the number of failures manifested due to broken test-dependencies. We remain to investigate how PASTE works with this patch in place.

D. *Continuous Integration*

Google [1], [2] and Microsoft [42] have been creating distributed infrastructures to efficiently build massive amounts of code and run massive amounts of tests. Those scenarios bring different and challenging problems such as deciding when to trigger the build under multiple file updates [43]. Although such distributed systems are targeted to extremely large scale code and test bases, the same ideas can be applied to handle the build process of large, albeit not as large, projects. For example, Gambi et al. [44] recently proposed CUT, a tool to automatically parallelize JUnit tests on the cloud. The tool allows the developer to control resource allocation and deal with the project specific test dependencies. Note that test suite parallelization is complementary to these high-level parallelism schemes.

Continuous Integration (CI) services, such as Travis CI [45], are becoming widely used in the open-source community [46], [47]. Accelerating time to run tests in CI is important as to reduce the period between test report updates. Module-level regression testing [48], for example, can be helpful in that setting. It is important to note that test failures are more common in CI compared to an overnight run or a local run, for instance. This can happen because of semantic merge conflicts [49], for instance. As such effect can impact developer's perception and tolerance towards failures, we are curious to know if developers would be willing to receive more frequent test reports at the expense of potentially increasing failure rates due to flakiness caused by parallelism.

X. Conclusions and Future Work

This paper presents PASTE, a lightweight approach to parallelize execution of test suites through the sequential re-execution of test cases (to avoid data races) and the sequential re-execution of test classes (to avoid broken test dependencies). We evaluated PASTE on 25 projects mined from GitHub using an objective selection criteria. Results show that (1) PASTE could circumvent flakiness introduced with parallelization in all projects that manifested them and that (2) the best parallel execution mode of PASTE was faster than the sequential execution mode by a factor of 1.59x. Overall, our findings showed that PASTE could be used to guide developers in finding manifested test dependencies and to automatically parallelize execution of test suites. The artefacts of this study are publicly available on GitHub at the following URL: https://github.com/STAR-RG/paste.

Acknowledgments

This work is partially supported by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, APQ-0570-1.03/14, CAPES grant 88887.136410/2017-00, and CNPq (grants 465614/2014-0, 309032/2019-9, 406308/2016-0, 409335/2016-9).

References

[1] J. Micco, "Tools for continuous integration at google," October 2010, google Tech Talks. http://www.youtube.com/watch?v=b52aXZ2yi08.

[2] Google Engineering Tools, "Testing at the speed and scale of google," June 2011, http://google-engtools.blogspot.com.br/2011/06/testing-at-speed-and-scale-of-google.html.

[3] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "Cloudbuild: Microsoft's distributed and caching build service," ser. ICSE 2016, p. 11–20. [Online]. Available: https://doi.org/10.1145/2889160.2889222

[4] M. McGarr and D. Marsh, "Towards true continuous integration: distributed repositories and dependencies," 2017, netflix TechBlog. [Online]. Available: https://bit.ly/3dZOL7Q

[5] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," ser. ICSE-SEIP 2017, pp. 233–242.

[6] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," ser. ISSTA 2015, pp. 211–222.

[7] J. Candido, L. Melo, and M. d'Amorim, "Test suite parallelization in open-source projects: A study on its usage and impact," ser. ASE 2017, pp. 838–848.

[8] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[9] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection (website)," 2017, http://ekstazi.org/research.html.

[10] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," ser. ESEC/FSE 2015, pp. 770–781.

[11] Maven, 2021, https://maven.apache.org/.

[12] JUnit, "Junit web site," 2021, http://junit.org.

[13] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," ser. ISSTA 2014, Jul., pp. 385–396.

[14] R. Guru, "Singleton," 2021. [Online]. Available: https://refactoring.guru/design-patterns/singleton

[15] TestNG, "Testng web site," 2021, http://testng.org.

[16] NUnit, "Nunit web site," 2021, http://www.nunit.org.

[17] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," ser. FSE 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[18] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," ser. ISSTA 2015, p. 223–233. [Online]. Available: https://doi.org/10.1145/2771783.2771793

[19] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the Performance of Maven's Test Isolation: Experience Report," ser. ISSTA 2020, p. 249–259. [Online]. Available: https://doi.org/10.1145/3395363.3397381

[20] Maven Surefire Plugin, 2021, http://maven.apache.org/surefire/maven-surefire-plugin/.

[21] MasterReplicationCoordinatorTest, "Accumulo masterreplicationcoordinatortest.java," 2021. [Online]. Available: https://bit.ly/3eHH7yf

[22] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," ser. ICST 2018. IEEE, pp. 1–11.

[23] "Accumulo repository," 2021. [Online]. Available: https://github.com/apache/accumulo/

[24] EasyMock Library, "Easymock," 2021. [Online]. Available: https://easymock.org/

[25] Apache, "Apache projects written in java," 2021. [Online]. Available: https://projects.apache.org/projects.html?language=Java

[26] "Pradet repository," 2021. [Online]. Available: https://github.com/gmu-swe/pradet-replication

[27] ClassificationSearchProcessorTest, "Atlas classificationsearchprocessortest.java," 2021. [Online]. Available: https://bit.ly/3eGGR2w

[28] M. team, "Multi-machine parallelization with maven," 2021. [Online]. Available: https://github.com/apache/maven-surefire/pull/240

[29] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: An empirical evaluation," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 1990–2032, Oct. 2016.

[30] A. Çelik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across JVM boundaries," ser. ESEC/FSE 2017, pp. 809–820. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106297

[31] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 597–613, Sep. 2008.

[32] C. H. P. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," ser. ISSRE 2012, pp. 221–230.

[33] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," ser. ICSE 2014, pp. 907–918.

[34] A. Rajan, S. Sharma, P. Schrammel, and D. Kroening, "Accelerated test execution using gpus," ser. ASE 2014, pp. 97–102. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642957

[35] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," ser. ESEC/FSE 2015, pp. 842–853.

[36] V. Yaneva, A. Rajan, and C. Dubach, "Compiler-assisted test acceleration on gpus for embedded software," ser. ISSTA 2017, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092720

[37] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," ser. ISSTA 2007, pp. 50–60.

[38] S. Mondal and R. Nasre, "Mahtab: Phase-wise acceleration of regression testing for C," *Journal of Systems and Software*, vol. 158, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301773

[39] ——, "Hansie: Hybrid and consensus regression test prioritization," *Journal of Systems and Software*, vol. 172, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302405

[40] O. Schwahn, N. Coppik, S. Winter, and N. Suri, "Assessing the state and improving the art of parallel testing for c," ser. ISSTA 2019. Association for Computing Machinery, p. 123–133. [Online]. Available: https://doi.org/10.1145/3293882.3330573

[41] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web Test Dependency Detection," ser. ESEC/FSE 2019, p. 154–164. [Online]. Available: https://doi.org/10.1145/3338906.3338948

[42] C. Prasad and W. Schulte, "Taking control of your engineering tools," *Computer*, vol. 46, no. 11, pp. 63–66, 2013.

[43] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," ser. ICSE-SEIP 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[44] A. Gambi, S. Kappler, J. Lampel, and A. Zeller, "Cut: Automatic unit testing in the cloud," ser. ISSTA 2017, pp. 364–367. [Online]. Available: http://doi.acm.org/10.1145/3092703.3098222

[45] "Travis ci," https://travis-ci.org/, 2021.

[46] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," ser. ASE 2016, pp. 426–437.

[47] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," ser. ESEC/FSE 2015, pp. 805–816.

[48] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .net," ser. ESEC/FSE 2017, pp. 848–853. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117763

[49] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," ser. ESEC/FSE 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139