Debugging the Performance of Maven's Test Isolation: Experience Report

Pengyu Nie
University of Texas at Austin, USA
pynie@utexas.edu

Ahmet Celik
Facebook, Inc., USA
celik@fb.com

Matthew Coley
George Mason University, USA
mcoley2@gmu.edu

Aleksandar Milicevic
Microsoft, USA
almili@microsoft.com

Jonathan Bell
George Mason University, USA
bellj@gmu.edu

Milos Gligoric
University of Texas at Austin, USA
gligoric@utexas.edu

ABSTRACT

Testing is the most common approach used in industry for checking software correctness. Developers frequently practice reliable testing—executing individual tests in isolation from each other—to avoid test failures caused by test-order dependencies and shared state pollution (e.g., when tests mutate static fields). A common way of doing this is by running each test as a separate process. Unfortunately, this is known to introduce substantial overhead. This experience report describes our efforts to better understand the sources of this overhead and to create a system to confirm the minimal overhead possible. We found that different build systems use different mechanisms for communicating between these multiple processes, and that because of this design decision, running tests with some build systems could be faster than with others. Through this inquiry we discovered a significant performance bug in Apache Maven's test running code, which slowed down test execution by on average 350 milliseconds *per-test* when compared to a competing build system, Ant. When used for testing real projects, this can result in a significant reduction in testing time. We submitted a patch for this bug which has been integrated into the Apache Maven build system, and describe our ongoing efforts to improve Maven's test execution tooling.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Build system, Maven, test isolation

ACM Reference Format:
Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the Performance of Maven's Test Isolation: Experience Report. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Los Angeles, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395363.3397381

ISSTA '20, July 18–22, 2020, Los Angeles, CA, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397381

1 INTRODUCTION

Previous research shows that tests in industry are riddled with flakiness [8, 19, 20, 28, 30, 32]. A common practice to combat flaky tests is to run them in isolation from each other. This aims to eliminate *test-order dependencies*, i.e., potential effects that one test execution may have on executions of other tests (e.g., by polluting some shared static state) [6, 7, 23, 42]. Depending on the level of isolation, however, the introduced overhead can be substantial.

Test isolation can be implemented to various degrees, ranging from full sandboxing (i.e., each test is run in its own freshly provisioned OS virtual machine) to sequential execution of all tests inside a single OS-level process. The former approach provides absolute isolation, though at a very high cost; the latter, in contrast, introduces minimal overhead but only guards against races caused by concurrency.

Speaking of Java projects, running all tests inside a single Java Virtual Machine (JVM) is the default mode for many build systems, including Ant, Gradle, and Maven. Although efficient, using this default mode may impact the correctness of test results and lead to intermittent test failures when there exist test-order dependencies.

The most commonly used middle ground is running each test in its own process. In case of Java projects, this entails *forking* a new JVM for every test. The overhead of this technique (compared to running all tests in a single JVM) is known to be significant. Prior work has shown that the slowdown can be as high as several orders of magnitudes [6]. The high cost of forking is commonly associated with the cost of spawning a large number of JVM processes.

Recent work introduced VmVm [6], an approach that defines test virtualization to isolate each test within a single JVM. Specifically, VmVm tracks accesses to static fields during a test execution and automatically re-initializes those fields prior to the execution of the subsequent test. VmVm brings the best from the two worlds: test isolation via lightweight containers and reuse of all loaded classes among tests by executing everything in a single JVM. Extensive evaluation of VmVm showed significant performance improvements over forking and low overhead compared to running all tests inside a single JVM.

Despite its powerful approach, we believe that VmVm faces two key obstacles on its way to a wider adoption. It is a complicated system, which involves dynamic bytecode instrumentation, and such instrumentation must be updated with each new version of Java. Perhaps rightfully so, many developers may be hesitant to be early adopters of research tools, when developers need tools that they can rely on, but researchers are pressured to keep inventing new

tools over maintaining older work. Instead, developers rely widely on open source frameworks and tools that are already adopted and maintained by the community.

Inspired by the initial VmVm evaluation and aforementioned challenges, we set out to seek a simple yet effective approach to optimize forking, with the goal of creating a solution that we could ensure would be more widely adopted. We began by performing an *exploratory study* to compare the performance overhead of test isolation in three different build systems (Ant, Maven and Gradle). Although all three of these build systems executed the exact same JUnit tests, each build system provides a separate implementation of the glue that actually *executes* those tests. In this study, we evaluated the overhead of executing a trivial test, which simply called `Thread.sleep(250)`, repeatedly, in new processes. Somewhat surprisingly, our results of profiling build system runs show that, for Maven in particular, a large portion of the overhead is unexplainable by OS-imposed overheads: on average each test took approximately 350 milliseconds longer to run with Maven than with Ant or Gradle.

To better understand and isolate the sources of overhead in test isolation, we created ForkScript, a highly-optimized JUnit test runner that runs each test in its own process. During a build, on-the-fly, ForkScript generates a single *specialized execution script* for running all tests and collecting test results. Our execution script utilizes process management provided by the operating system, as well as several utilities available on Unix systems. The build system, hence, needs to interact only with that execution script process, instead of all individual test processes.

Of course, ForkScript could suffer the same adoption limitations as VmVm: ForkScript needs to be tightly integrated with the build system and test framework and represents a maintenance challenge. Hence, developers may choose to continue to allow the build system to run their tests as it normally would, and again, we may not be successful in getting developers to adopt ForkScript. Instead, we used ForkScript as a baseline while we performed a deep-dive into Maven's test execution performance, and were able to reveal a bug in the inter-process communication (IPC) code that is used for test communication in Maven. This bug imposed a constant overhead of approximately 350 milliseconds per-test-class, resulting in a particularly enormous relative overhead in projects with very fast running tests. Our patch for this bug has been integrated into Maven, resulting in a significant speedup for test runs that require test isolation. Moreover, since our contribution is merged into Maven (and is not a standalone tool or Maven plugin), we can be confident that it will have practical impact.

This experience report describes our process discovering, debugging and patching this bug, as well as our experiences working with the community to continue to improve Maven's IPC code. Our results show that ForkScript can save up to 75% (50% on average) of test execution time compared to Maven forking when running a single JVM at a time. The benefits of ForkScript for runs with parallel processes slightly decrease, although savings remain high. Most importantly, our results also show that with our patch that has been merged into Maven, every developer can see similar performance to ForkScript. Reflecting on the experience of developing VmVm, ForkScript, and then patching Maven itself, it is clear to us that working with open source communities can be both extremely challenging and rewarding. Developing ForkScript was far easier than debugging and patching the underlying flaw in Maven. Although ForkScript ultimately was somewhat faster than our patched version of Maven, by creating and integrating a simple patch into Maven we have finally been able to have much broader impacts by getting our improvements in front of every developer who uses Maven.

The main contributions of this paper include:

- A closer look at the cost of forking (as implemented in the existing build systems) and a finding that substantial overhead behind it comes from generic inter process communication (IPC).
- A novel, simple, technique, dubbed ForkScript, to optimize forking by creating specialized test execution script on-the-fly.
- A deep-dive performance measurement of test execution in Ant, Maven and Gradle, resulting in a patch that has been merged into Maven.
- An extensive evaluation of our approach on 29 open-source projects, totaling 2 million lines of code.
- Impact on several design decisions for future releases of the Maven build system [3, 5].

Our experience report clearly shows that researchers should explore rigorous and systematic testing of build systems, including their performance, correctness, and usability. These topics have not received much attention by the testing community in the past.

2 MOTIVATING STUDY

While past work on test acceleration typically focuses on a *single* build system (e.g. all of the projects evaluated in the VmVm paper used the Ant build system [6]), we were particularly curious if different build systems' approach to test isolation could result in different performance. Hence, we conducted a simple experiment to compare the overhead that each build system adds when executing tests with isolation. To do so, we generated a test suite consisting of 100 JUnit test classes, each with a single JUnit test method that simply calls `Thread.sleep(250)` and then returns. These simple tests are not appropriate for benchmarking complex runtime systems like VmVm, but provide the perfect opportunity to measure overheads introduced by the build system, since each test should require a constant execution time. We measured the total time taken by each of Ant, Maven and Gradle to execute this entire test suite (with test isolation, and excluding the time taken by other parts of the build process). Then, we subtracted the actual time needed to run each test (250 msec) to identify the overhead per-test introduced by each build system.

Table 1 presents the results of this study. We found that the overhead introduced by each build system varied dramatically, from just 259 msec in the case of Ant to 596 msec in the case of Maven. This wide range makes us hopeful that there may be things that can be changed in Maven or Gradle to reduce the overhead of test isolation to at least that of Ant. We know that there will have to be *some* overhead to running tests in this method, since there is a non-zero overhead imposed by the operating system when creating a new process. However, *all* of the build systems will have to pay for that same overhead, and hence, any differences in overhead between build systems would have to be explained by

Table 1: Overhead Introduced by Each Build System for Isolating a Single Test in Milliseconds.

Build System	Overhead Per-Test (ms)
Ant 1.10.6	259
Gradle 5.6.1	412
Maven (Surefire 3.0.0-M3)	596

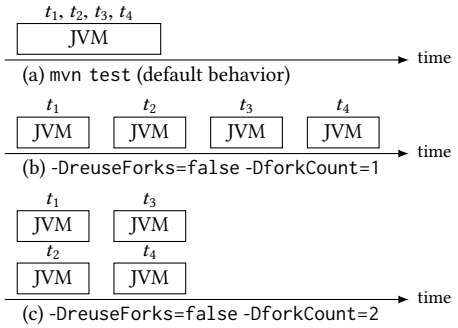

Figure 1: Illustrative example of three Surefire configurations when running four tests ({t1, t2, t3, t4}).

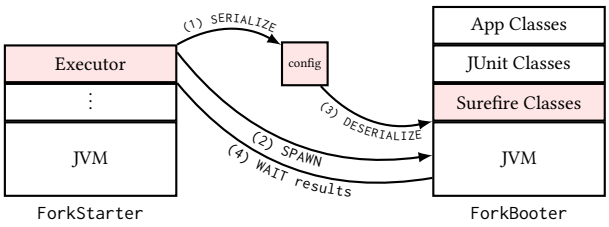

Figure 2: Surefire's workflow. Shaded boxes highlight the main sources of overhead introduced by Surefire.

different IPC mechanisms. Hence, if we can identify why Maven is so much slower than Ant, we could, ideally propose a change to Maven to improve its performance. Such a change could afford developers with some of the speedup that they might obtain from using a system like VmVm to isolate their tests, but with the added convenience of not requiring *any* per-project changes, by making a change to the build system directly. At the same time, while we can see from this data that Ant has the lowest overhead of these three options, we don't know for certain that it represents an ideal solution: perhaps an even more efficient system could be faster.

3 BACKGROUND

We provide a high level overview of Maven and describe the key steps taken by Maven when executing a set of tests.

Maven defines a build lifecycle—a well-defined sequence of *phases*, including compilation, code generation, test execution, etc. A user can specify a set of plugins to be executed with each build phase; a plugin is associated with a phase by its developers, and a user can only choose which plugins to include in the build run. An official plugin for running unit tests is Maven Surefire [3]. Surefire

```
Input: tests - set of tests
Input: config - build configuration
1:  function ForkStarter(tests, config)
2:      executor ← makeExecutor(config.forkCount)
3:      for τ in tests do:
4:          classpath ← getClasspath(τ, config)
5:          serializedConfig ← serialize(τ, classpath)
6:          task ← makeCommandLineTask(serializedConfig)
7:          submit(executor, task)
8:      done
9:      waitForAll(executor)
10: end function
```

Figure 3: Overview of the key steps executed in the `ForkStarter` class, which is a part of the Surefire plugin; this code is executed in the same Java Virtual Machine that is running the main Maven process.

implements complex logic for finding test classes in a project, filtering tests, etc. A rich set of options enables user to fine tune test execution [4].

To execute tests in a project, a developer can simply execute the following command: `mvn test`. This command starts Maven (which is written in Java and executes in its own JVM), goes through all phases until (and including) the test phase and runs user specified plugins; all the plugins are executed in the same JVM as the main Maven process, although the plugins can spawn other processes. The aforementioned command would trigger Surefire plugin to spawn *a single* new JVM process and execute *all tests* in the new JVM (Figure 1.a). If a user wants to isolate tests, by running each test in a separate JVM, she can use the following command: `mvn test -DreuseForks=false -DforkCount=1`(Figure 1.b). The first command-line option (`reuseForks`) specifies that JVM should not be reused after test execution is finished, and the second command-line option (`forkCount`) specifies the number of JVM instances that should run in parallel. Figure 1.c illustrates a configuration with two JVMs running in parallel.

Behind the scenes, the Surefire plugin finds the tests, extracts the build configuration needed to tune the test execution, and prepares options for a new JVM. We only consider execution *with* forking (without reusing JVMs) in the reminder of this section, i.e., configurations illustrated in Figure 1.b and Figure 1.c. When it comes to actual test execution, there are two key classes involved: `ForkStarter` and `ForkBooter`. Figure 2 visualizes their interaction. `ForkStarter` executes in the same JVM as Maven and spawns new JVMs. `ForkBooter` is the main class started in the new JVM process, which reads configuration prepared by `ForkStarter` and executes one test.

Figure 3 summarizes the key steps that are executed in the `ForkStarter` class. The input to `ForkStarter` is (1) the set of tests to execute, and (2) a parsed build configuration. Initially (line 2), `ForkStarter` creates an executor, i.e., an instance that maintains a thread pool [2], where each thread will be responsible for spawning new JVMs and triggering test executions. In the next step (lines 3-8), `ForkStarter` iterates over the set of tests to create one task for each test and submit the task to the executor. Each iteration of the loop makes a classpath for the current test, serializes the configuration

```
Input: serializedConfig - path to serialized configuration file
1: function ForkBooter(serializedConfig)
2:     config ← deserializeConfig(serializedConfig)
3:     setUpJVM(config)
4:     junit ← makeJUnit
5:     execute(junit, config.test)
6:     sendGoodbye()
7: end function
```

Figure 4: Overview of the key steps executed in the `ForkBooter` class, which is a part of the Surefire plugin; this code is executed in the new Java Virtual Machine.

for the new JVM to a temporary file, creates a task, and submits the task. Once all tasks are submitted, `ForkStarter` waits for all tasks to complete (line 9).

Figure 4 summarizes the key steps that are executed in the `ForkBooter` class, which is the main class in newly spawned JVM. In the first step (line 2) `ForkBooter` deserializes the configuration from the temporary file, sets up configuration for the current JVM (line 3), creates an instance of JUnit (line 4), and executes the test with the JUnit (line 5). Finally (line 6), it sends a goodbye signal (via standard output) to the `ForkStarter`.

4 BEST-CASE PERFORMANCE: NO IPC

When isolating test case executions in their own process, the test running infrastructure must have some form of *interprocess communication* (IPC) to coordinate between the two processes. We speculated that this must be where the increased overhead that we saw from Maven and Gradle in comparison to Ant. Based on our initial profiling of Maven runs and considering the steps taken, in `ForkStarter` and `ForkBooter`, to execute a set of tests, we highlight, in Figure 2 the steps that introduce overhead *in addition* to the cost of spawning new JVM processes. Specifically, (1) using thread pool and executors to manage processes requires additional class loading and adds substantial complexity especially for those test runs when only a single JVM is run at a time (i.e., there is a single thread in the thread pool), (2) exchanging configuration with new JVMs via serialization/deserialization to/from files requires costly IO operations, (3) class loading of Surefire's classes (e.g., `ForkBooter`) in each new JVM adds on top of already costly class loading of classes under test [33], and (4) "pumping" input/output between the original JVM and newly created JVMs requires extra threads, synchronization, etc.

Although some of the extra steps taken by Surefire may be necessary for certain build configurations, we believe that the overly generic design adds substantial overhead for simple configurations, which are common for many small and medium sized open-source projects, as those used in our evaluation. Thus we set to design and develop a simple but effective approach to remove overly generic steps, taken by Surefire, whenever such approach is feasible.

To reduce the IPC between a build system and the processes it spawns, we present ForkScript. During a build, on-the-fly, ForkScript generates a single specialized execution script for running all configured tests and collecting test results; the tests may be run sequentially or in parallel. Our execution script utilizes process

```
Input: tests - set of tests
Input: config - build configuration
Output: S - specialized script that executes tests
1: function ForkScript(tests, config)
2:     count ← 0
3:     for τ in tests do:
4:         classpath ← getClasspath(τ, config)
5:         S ←+ $timeout ‘config.timeout \$
6:         S ←+ $java -cp ‘classpath
7:             forkscript.JUnitRunner ‘τ ‘config &$
8:         if (count + 1) % config.forkCount == 0 then:
9:             S ←+ $wait$
10:        fi
11:        count++
12:    done
13: end function
```

Figure 5: A simplified version of a multi-staged program used by ForkScript to generate a specialized script; this program replaces `ForkStarter` code in Figure 3.

management provided by the operating system, as well as several popular utilities available on Unix systems. The build system, hence, needs to interact only with that shell script process, instead of all individual test processes.

ForkScript replaces `ForkStarter` and completely removes `ForkBooter`. At the same time, other parts of Maven and Surefire need not be changed at all. Moreover, we rely on Surefire to discover tests, perform filtering, extract build configuration from configuration files, etc. The changes introduced by ForkScript happen when it comes to concrete test execution steps.

Figure 5 shows a simplified multi-staged program [37], which replaces code in Figure 3 and Figure 4, used by ForkScript to generate a specialized script for the given set of tests and build configuration. We use standard notation from code generation community to represent code fragments and holes [26]. $\overset{+}{\leftarrow}$ concatenates the strings and appends a newline character.

The input to the ForkScript function is the same as for `ForkStarter`: a set of tests and the build configuration. The ForkScript function, similar to `ForkStarter`, iterates over each test in the set of tests, but rather than adding each test as a task to an executor, ForkScript extracts configuration (e.g., classpath for the test) and appends a shell command that will execute the test with the appropriate setup. Depending on the number of JVMs that should run in parallel (`forkCount`), ForkScript splits the set of tests in one or more *buckets*. Note that tests within each bucket are executed in parallel, as scheduled by the OS, but each test runs in its own JVM. In our implementation, we use the `split` command (available on Unix) to split tests into buckets.

To execute each test, ForkScript uses a custom `JUnitRunner`, which, unlike the default JUnit runner (`org.junit.runner.JUnitCore`), accepts several command line arguments to fine tune the output of the runs. Clearly, to be able to support other testing frameworks (e.g., TestNG) or a different version of the same framework, we would have to implement one custom runner for each testing framework and each version.

Because the specialized script is generated in each test run, ForkScript does not introduce any burden on the developers to maintain those scripts. In other words, changes in the set of tests (if tests are added or deleted), as well as changes in the classpath, are automatically reflected in the subsequent test runs. This transparency for users enabled an easy transition to ForkScript.

We believe that a large number of projects with a simple build configuration, can substantially benefit from ForkScript. This is also evidenced by the results of our evaluation (see Section 6). Considering the frequency of test runs, especially for those projects that use Continuous Integration [15, 25, 39], e.g., Travis CI, ForkScript can have substantial impact on developers' productivity and software reliability.

5 PERFORMANCE PROFILING MAVEN

ForkScript provides a barebones, stripped down mechanism to run JUnit test classes in their own process, but may not suit all developers' needs. Hence, we *also* carefully profiled Maven to identify the specific source of the ~350msec/test performance overhead (in comparison to Ant). To start, we refined our overhead measurement experiment described in Section 2 to break down the time measurement into three components. In particular, we wanted to attribute the overhead into time needed to launch a new process, for that process to run the test, and then for that process to terminate. Figure 6 shows an abstract timeline of the execution of a single JUnit test class which is representative for Ant, Maven and Gradle. We profiled: T_1, the duration between when the build system begins running a test until the child process (the test runner) starts; T_2, the duration between when the child process starts until the child process terminates; and T_3, the duration between when the child process terminates until when the build system determines that the test has completed.

To conduct this profiling, we implemented a simple load-time Java bytecode instrumentation agent that modified the appropriate methods of each build system to record the system time of each of the four events shown in Figure 6. Again, we ran our profiler on 100 tests that each simply call `Thread.sleep(250)`, and report the average duration of each of the steps to execute each test in Table 2. We were surprised to find that the majority of Maven's overhead (compared to Ant) came from T_3, the duration between when the test ends (in the `ForkedBooter` child process) to when the parent process (the build system, aka the `ForkStarter`) finds that the test has completed.

With this clue in hand, we attached a Java debugger to the child process (`ForkedBooter`) and set a breakpoint at the point in the execution that represents the start of T_3 in Figure 6 — when the test completes. We examined the program behavior carefully, and found that the program was stalling while a thread was reading from the standard input stream, `<stdin>`. Furthermore, we found that it was this thread blocking on `<stdin>` that was causing a roughly 350 millisecond delay for each test execution.

Upon further investigation, we found that this is not a problem unique to Maven. Normally, when a Java thread is reading from an `InputStream`, it can be interrupted by another thread (for instance, if the JVM is shutting down). However, when reading from `<stdin>`, the thread can *not* be interrupted until they reach a JVM "safepoint", which occurs every 300 milliseconds [38].

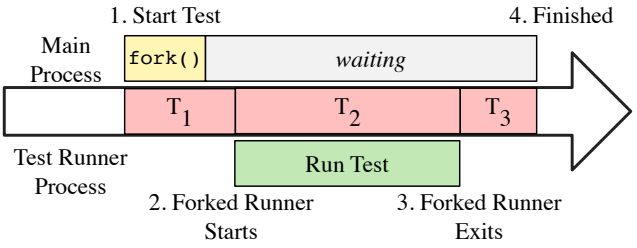

Figure 6: Critical regions in test execution to profile. We focus on three regions: 1) between when the build system decides to execute a test and when the forked process launches, 2) between when the test starts and when it finishes, and 3) between when the test finishes and when the build system considers the test complete.

Table 2: Profiling Results that Break Down the Time to Run Each Test Using the Durations Described in Figure 6. Our Patch Significantly Reduces the T_3 Measurement.

Build System	T_1[ms]	T_2[ms]	T_3[ms]
Ant 1.10.6	250	253	9
Gradle 5.6.1	395	253	17
Maven (Surefire 3.0.0-M3)	244	253	352
Maven (With our patch)	217	252	17

To resolve this bug, the fix is simple: when the `ForkedBooter` (child JVM) determines that it is time to shut down, do not continue to read from `<stdin>`. We proposed this three-line patch (plus a one-line change to an integration test) to the Maven Surefire developers, who gladly accepted and merged it into their master branch [14]. With this change, we shaved approximately 350 milliseconds off of *each* test class execution time for Maven, bringing its performance much closer in line with Ant. Since the change was integrated directly into the project, every user of Apache Maven stands to benefit from this improvement. We are interested in applying a similarly detailed performance analysis to Gradle's performance in the future.

Interesting future work would also consider the implications of using network sockets over standard input for test running communication. In addition to avoiding OS peculiarities regarding blocking on `<stdin>` versus other streams, migrating communication to sockets would allow Maven to run tests on separate physical machines than the machine invoking Maven — a powerful feature. We began working with the Apache Maven community on developing such a feature, but development is still underway: the test running code in Maven is extremely old, and it was not designed with the goal of making such drastic IPC changes easy.

6 STUDY

To assess the benefits of our contributions, we answer the following research questions:

RQ1: What are the performance improvements obtained by ForkScript compared to the default Maven forking?

RQ2: How does the improvement scale as the number of concurrent processes increase?

RQ3: How does the patched Maven compare to ForkScript?

We run all experiments on a 4-core Intel Core i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 18.04. We used Oracle Java HotSpot(TM) 64-Bit Server (1.8.0_181). Furthermore, we used Maven 3.5.3.

We first describe the subjects used in our study, as well as experiment setup, then we answer our research questions.

6.1 Subjects

To select the subjects for our study, we mostly searched through recent work on regression testing and test isolation [6, 21, 23, 27]. Additionally, we set a couple of requirements for each project: the project must (1) be buildable with the Maven build system, (2) have non-trivial number of tests, (3) have tests whose execution time is non-negligible, and (4) successfully build at its latest revision. We use the latest revision of each project available at the time of (the latest run of) our experiments rather than revisions used in prior studies; this makes our experiments feasible as building old revisions can be challenging.

Table 3 shows the list of 29 projects used in our experiments. All the selected projects are open-source and available on GitHub. For each project, we show the size of the project in terms of the number of lines of code (LOC); number of Maven modules (#Modules); number of source files in the repository (#Files); number of test classes, i.e., tests (Classes); number of test methods (Methods); location of the project on GitHub (URL); and the revision used in our experiments (SHA). We used cloc [13] to collect LOC, our scripts to collect number of files, and Maven default output to report the number of tests and test methods.

Figure 7 shows the distribution of test execution time, per project, for all tests; the x-axis is in the log_{10} scale.

Although projects in Table 3 may not be representative of all Java projects (or even Java projects that build with Maven), we believe that the set of chosen projects covers a broad range of application domains. Additionally, we can see that the projects vary in LOC, number of files, number of tests, and number of test methods. The last two rows in Table 3 (Avg. and $\sum$) show the average and total values (where applicable).

6.2 Experiment Setup

We briefly describe our experiment setup. The main goal is to compare test execution time if one uses Maven forking vs. ForkScript. To that end, we perform the following changes and steps.

We modify recent version of Surefire to output the exact time taken for the entire test execution. Note that Surefire by default prints only time for *each test* and the total execution time for the *entire build*, but it does not print the total time for test execution as defined by the `ForkStarter` function in Figure 3. Thus, we modified `ForkStarter` to output time at the beginning and at the end; this is the only change in Surefire for our experiments. We did the same change for the ForkScript function in Figure 5. As ForkScript has no impact on other phases of the Maven build life cycle, we believe that measuring only test execution time provides a fair comparison and avoids any potential noise from other phases in the build.

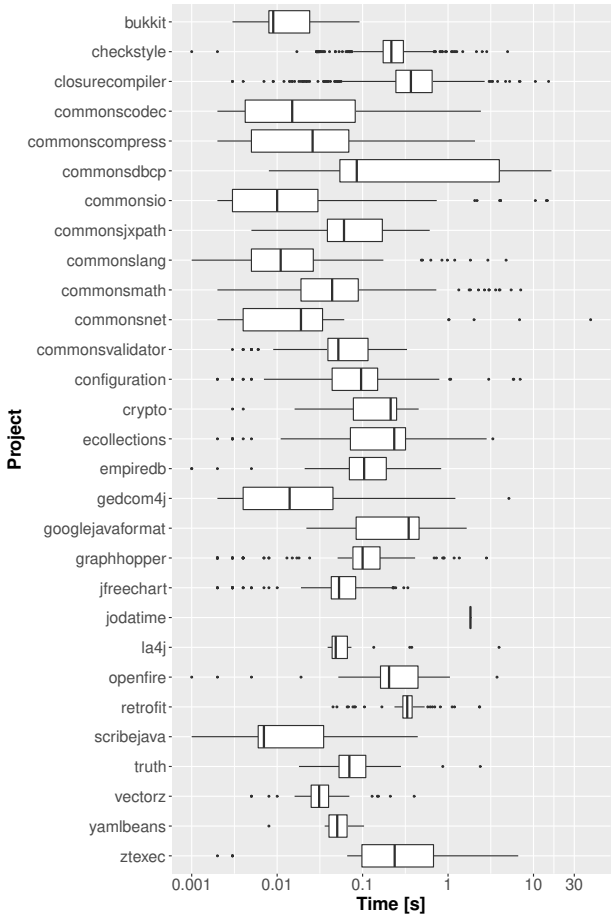

Figure 7: Distribution of test execution times in log_{10} scale.

For each project in Table 3 we execute the following steps:

a) Clone the project from GitHub and checkout the revision shown in Table 3 (SHA),
b) Install the project (by executing `mvn install`), which will execute all the build phases and download necessary dependencies (i.e., third party libraries),
c) Run tests using Maven in the offline mode to avoid unnecessary connection to the Internet, and
d) Run tests using ForkScript in the offline mode.

We store the logs for the last two steps to be able to extract the number of tests executed in each run, as well as the test execution time. We use the former to check the correctness of our runs, i.e., each ForkScript run should execute the same number of tests as the Maven run. We manually confirmed that all build outcomes are the same. We use the latter to compute the savings of ForkScript compared to default Maven implementation. Namely, we extract total test execution time for each Maven run (denoted with T^{mvn}) and ForkScript run (denoted with T^{FS}) and compute savings as $RT = \frac{T^{\text{mvn}} - T^{\text{FS}}}{T^{\text{mvn}}} \times 100\%$. This saving is the key metric used in our evaluation. We use the same metric for both sequential runs and parallel runs.

Table 3: List of Subjects Used in our Study.

Project	LOC	#Modules	#Files	#Tests		URL	SHA
				Classes	Methods	https://github.com/	
bukkit	32,560	1	731	38	903	Bukkit/Bukkit	f210234
checkstyle	206,965	1	2,001	297	2,993	checkstyle/checkstyle	f7626ac
closurecompiler	357,610	3	1,196	354	13,713	google/closure-compiler	5cae9b7
commonscodec	20,400	1	127	54	869	apache/commons-codec	1406889
commonscompress	44,711	1	354	137	1,057	apache/commons-compress	e127b13
commonsdbcp	20,654	1	106	30	569	apache/commons-dbcp	5226462
commonsio	30,455	1	248	105	1,354	apache/commons-io	9e5475f
commonsjxpath	24,910	1	232	38	409	apache/commons-jxpath	3138e7a
commonslang	76,203	1	325	148	4,084	apache/commons-lang	58a8f12
commonsmath	153,695	1	1,177	367	4,158	apache/commons-math	b05b3b9
commonsnet	28,207	1	272	43	268	apache/commons-net	cc819eb
commonsvalidator	16,082	1	149	70	541	apache/commons-validator	a377131
configuration	66,847	1	457	169	2,786	apache/commons-configuration	bc69f94
crypto	5,964	1	87	25	111	apache/commons-crypto	d69ef95
eclipsecollections	298,762	16	2,578	3,196	161,417	eclipse/eclipse-collections	f2543f2
empiredb	52,700	7	472	25	108	apache/empire-db	f816837
gedcom4j	47,204	1	503	243	1,300	frizbog/gedcom4j	be310f2
googlejavaformat	13,951	1	69	17	1,037	google/google-java-format	579772a
graphhopper	59,435	7	565	137	1,460	graphhopper/graphhopper	b2db5c2
jfreechart	133,847	1	987	341	2,176	jfree/jfreechart	520a4be
jodatime	86,138	1	330	1	4,222	JodaOrg/joda-time	c9f2764
la4j	13,581	1	117	20	835	vkostyukov/la4j	2826a4a
openfire	203,070	8	1,589	31	235	igniterealtime/Openfire	6c32335
retrofit	19,339	5	220	65	595	square/retrofit	5c2f505
scribejava	11,160	6	238	25	109	scribejava/scribejava	09364b6
truth	28,805	2	179	59	1,447	google/truth	14f72f7
vectorz	53,487	1	397	72	456	mikera/vectorz	a05c69d
yamlbeans	6,716	1	51	8	89	EsotericSoftware/yamlbeans	f561099
ztexec	3,163	1	76	20	93	zeroturnaround/zt-exec	6c3b93b
Avg.	72,986	2	545	211	7,220	N/A	N/A
Σ	2,116,621	75	15,833	6,135	209,394	N/A	N/A

6.3 Improvements For Sequential Runs

RQ1: What are the performance improvements obtained by ForkScript *compared to the default Maven forking?*

Overall, we find that savings obtained by ForkScript for sequential runs are 50% on average and up to 75%. We discuss the details below.

To answer this research question, we followed the experiment setup from the previous section and executed tests with the following command: `mvn test -DreuseForks=false -DforkCount=1`. Recall that the first command-line option will enforce that each test is executed in its own JVM, and the second option specifies that there is only one forked JVM at a time. In other words, we are executing tests sequentially one JVM at a time. Recall (Section 4) that ForkScript too respects the `forkCount` parameter, so in this case ForkScript too will run a single test at a time in its own JVM. We use Fork1 to denote this run configuration (for both Maven and ForkScript); we do not encode `reuseForks` in the name of the configuration, because *all* our experiments use this option, i.e., enforce full test isolation.

Table 4 shows the results. The first column lists names of subjects, and columns 6 and 7 show test execution time in seconds for Maven and ForkScript, respectively. Column 8 shows savings (RT) of ForkScript compared to Maven. Interestingly, we can observe that ForkScript, for Fork1, speeds up test execution for all subjects. Minimum speedup is 12% for commonsdbcp and maximum speedup is 75% for bukkit. The last two rows show the average and total values for all projects.

We took an extra step to check what projects benefit the most from ForkScript. We mostly reasoned about the obtained speedup, number of tests (as we already reported in Table 3), and test execution time per test (as we already showed in Figure 7). We observed that projects with low median time per test benefit the most; median is shown as the vertical line for each boxplot in Figure 7. This is not surprising, as the overhead of forking is more observable for tests that run shorter. For example, ForkScript was more beneficial for

Table 4: Test Time in Seconds of Maven and ForkScript, with NoFork and Fork with 1 and 2 Parallel Processes. RT is the Time Savings of ForkScript Compared to Maven.

Project	NoFork				Fork 1				Fork 2			
	T^{mvn}[s]	T^{FS}[s]	RT[%]	T^{new}[s]	T^{mvn}[s]	T^{FS}[s]	RT[%]	T^{new}[s]	T^{mvn}[s]	T^{FS}[s]	RT[%]	T^{new}[s]
bukkit	0.98	0.51	47	0.79	19.07	4.62	75	6.79	7.01	2.56	63	4.18
checkstyle	28.48	27.86	2	28.82	231.64	129.27	44	146.17	114.62	82.03	28	90.45
closurecompiler	68.07	74.12	-8	71.11	408.43	289.74	29	301.97	233.35	209.39	10	210.08
commonscodec	6.86	6.40	6	6.62	34.65	13.11	62	16.77	14.44	9.27	35	9.75
commonscompress	11.21	10.14	9	11.11	82.75	29.40	64	40.93	33.56	17.11	49	22.23
commonsdbcp	70.53	70.28	0	73.49	90.01	78.59	12	80.43	49.91	42.15	15	47.45
commonsio	58.45	57.80	1	58.13	107.79	69.70	35	75.18	52.56	45.94	12	38.60
commonsjxpath	1.88	1.57	16	1.64	21.14	8.48	59	9.33	8.74	4.75	45	5.30
commonslang	13.49	13.38	0	13.50	88.15	32.11	63	41.51	34.56	22.29	35	22.81
commonsmath	53.12	53.18	0	53.49	243.92	109.10	55	243.62	103.24	61.21	40	103.28
commonsnet	57.51	58.47	-1	57.93	78.93	63.16	19	65.77	48.46	52.02	-7	47.82
commonsvalidator	1.86	1.60	13	1.74	37.25	12.27	67	15.86	13.80	7.76	43	9.25
configuration	24.85	24.14	2	24.14	127.49	61.50	51	70.60	59.38	34.89	41	40.10
crypto	1.86	1.50	19	1.59	15.48	7.17	53	8.45	7.15	4.40	38	5.04
eclipsecollections	76.81	71.00	7	76.13	2,233.72	1,157.09	48	1,307.81	1,014.09	670.55	33	737.34
empiredb	2.36	1.68	28	2.06	14.48	6.38	55	6.70	6.29	4.17	33	4.36
gedcom4j	17.05	9.35	45	18.44	149.74	46.24	69	77.05	64.77	28.65	55	46.01
googlejavaformat	4.29	3.81	11	4.18	15.67	9.90	36	11.46	8.05	6.84	15	7.28
graphhopper	20.18	14.46	28	19.46	91.34	41.33	54	50.71	43.33	27.98	35	31.50
jfreechart	2.10	1.40	33	2.14	174.61	59.41	65	77.70	62.74	32.32	48	43.85
jodatime	2.79	2.26	18	2.68	2.99	2.28	23	2.79	2.92	2.48	15	2.73
la4j	5.77	5.15	10	5.62	15.25	7.70	49	9.35	7.48	5.51	26	6.39
openfire	8.61	7.80	9	7.66	29.73	16.77	43	16.71	13.85	12.02	13	10.87
retrofit	21.15	17.31	18	17.35	53.24	35.67	33	36.22	30.83	26.41	14	24.89
scribejava	4.19	2.46	41	2.85	13.61	4.23	68	5.88	6.65	3.35	49	4.05
truth	5.65	4.93	12	5.05	35.99	14.57	59	17.39	16.37	9.15	44	10.76
vectorz	1.49	1.01	32	1.21	37.92	11.00	70	14.87	14.58	6.21	57	8.50
yamlbeans	0.70	0.32	54	0.43	4.24	1.20	71	1.66	1.69	0.73	56	1.09
ztexec	15.58	15.46	0	15.94	25.93	19.61	24	20.86	14.39	14.56	-1	12.10
Avg.	20.27	19.28	15	20.18	154.66	80.74	50	95.88	72.02	49.88	32	55.45
Σ	587.87	559.35	N/A	585.30	4,485.16	2,341.60	N/A	2,780.54	2,088.81	1,446.70	N/A	1,608.06

bukkit than for closurecompiler (75% vs. 29%). As another example, ForkScript was more beneficial for commonscodec than for commonsdbcp (62% vs. 12%). An exception to this rule is commonsnet, which has low median value and relatively low saving (19%). With a closer look, we found that a few tests in commonsnet dominate the entire test execution time and the number of tests in this project is small, thus the cumulative overhead is small and savings are low.

For completeness, we also report test execution time without test isolation. Namely, we simply execute: `mvn test` with Maven and ForkScript. The results are shown in Table 4 in columns 2-4 (NoFork). Although ForkScript provides savings even in this case, we note that the absolute difference in test execution time is rather small, and this configuration was not the motivation for our work.

6.4 Improvements For Parallel Runs

RQ2: How does the improvement scale as the number of concurrent processes increase?

Overall, our results show that ForkScript outperforms Maven for parallel runs. Concretely, ForkScript saves, on average, 32% of test execution time when running 2 JVMs in parallel. Next, we discuss the results in details.

To execute tests in parallel we execute the following command: `mvn test -DreuseForks=false -DforkCount=2`. We have also obtained results with `−DforkCount = 4`, which showed similar results, but we do not report the details in this paper. As our machine has 4 cores, we have not tried running experiments with larger number of parallel processes.

Table 4 shows the results for parallel runs. Columns 10-12 (Fork2) show execution time and savings when running (up to) two JVMs in parallel. We can see that savings remain substantial. On average, for runs with two parallel processes, ForkScript saves 32%. For runs with two parallel processes, minimum saving (slowdown in this case) is -7% while maximum is 63%. The reduction in savings, when running multiple JVMs in parallel, was expected as the total execution time approaches theoretical maximum, i.e., time to

execute the longest test in the project. Note that we expect that runs for jodatime are the same for all forking options because this project has only one test (although it has many test methods).

6.5 Comparison With Patched Maven

RQ3: How does the patched Maven compare to ForkScript?

Finally, we compared ForkScript with the patched Maven. Clearly, we expected that ForkScript would outperform the patched Maven, but we expected to see much smaller differences compared to the non-patched Maven version.

Table 4 shows the results. In columns titled "T^{new}" we report time for executing tests with the patched Maven. We can observe that savings of patched Maven are substantial over the non-patched version. Additionally, ForkScript slightly outperforms patched Maven, as was expected (Section 4), because ForkScript supports only a subset of features.

7 IMPLICATIONS AND LESSONS LEARNED

Reflecting on our experience, we believe that there are several important lessons that we have learned which are applicable to practitioners and researchers:

Detect performance bugs through differential testing. To the best of our ability to judge, this performance bug has been in Maven since the introduction of the feature to isolate tests (beyond the history of the project's git repository). Developers everywhere have likely assumed that the performance bug that we found was simply the normal, expected behavior. Only by performing fine-grained differential testing of three different build systems (described in Section 2) did we even recognize that there was a flaw. Performance bugs are notoriously difficult to find, but when there are alternative systems that accomplish the same goal, differential testing can help to reveal them.

Find simple fixes that can be integrated today. While our fix solved the performance bug that we found, it could also be solved by a long-term effort that is rewriting the entire `ForkedBooter` and `ForkStarter` IPC system that relies on `<stdin>`. This architecture dates back to the creation of this project (over a decade ago), and unfortunately poses problems for projects that, themselves use `<stdin>` and `<stdout>`. Instead, there is a significant refactoring within the Maven community to replace the `<stdin>` communication with non-blocking socket channels [1]. However, this effort will take several months, and require the involvement of multiple Apache community members. When we first isolated the bug, we actually went down this route *first* as a fix, and almost prepared a pull request to make this change in Maven. While we made it work (and pass all tests), the changes required to the codebase were extremely invasive. We created the simple patch described in Section 5 only after realizing that it would take months to transition the project from `<stdin>` to sockets in a way that would best comply with the community practices.

For researchers: engage in the open source community. As software testing researchers, engaging with the open source community is an extremely valuable opportunity to both discover interesting problems and to have a broader impact on the community. Open source projects can have a diverse group of contributors, each of whom may have another software development job, with their own differing perspectives and challenges. Engagement can come in many rich forms: in the past we have reported bugs and made pull requests to Maven, but never before had we joined the Slack channel. We encourage other researchers to engage with the open source community not only by finding and reporting bugs in projects, but by patching them and integrating new features that the community wants. Compared to our past experiences contributing to the same project (Maven), we found that it is far easier to have a pull request merged when our change is either to fix a clear bug or to implement a feature that already is on developer's wishlists. In contrast, proposing pull requests that simply integrate our research prototypes or research ideas into the open source project are far less likely to be successful, since open source project maintainers might be wary to accept large new features that may not be necessary in the eyes of the maintainer, but nonetheless will become a maintenance burden for that person. Finding a balance between research novelty and practical impact is key, as always. In the meantime, we continue to work with the project maintainers to migrate the IPC from `<stdin>` to socket channels, and look forward to continue to contribute to Maven.

For researchers: perform systematic testing of build systems. There is a vast amount of literature related to build systems, but it mostly focuses on build system design (e.g., [24]) and testing correctness of build scripts (e.g., [9]). Researchers have invested limited effort in checking correctness of build systems, with an exception of several efforts to formally design and prove (a part of) build systems [12, 31]. This experience report clearly shows the need for more work in this direction.

8 THREATS TO VALIDITY

External. The set of projects used in our evaluation may not be representative of all Java projects. To mitigate this threat, we used a large number of open-source projects, which differ in size, application domain, and number of tests. Moreover, many of the projects used in our study have been used in prior work on (regression) testing [6, 11, 21, 27].

The reported results are obtained on a single machine. However, we obtained results for many projects across four different machines, and the savings were substantial in all cases. We do not report results from all machines as showing those numbers would not add new insights.

We used a single revision of each project in this paper, and the results could differ for other revisions. We simply chose the latest revision of each project available at the time of our experiments; in our initial experiments, not reported in this paper, we used earlier revisions, and we observed similar savings when using ForkScript.

Internal. Our code and scripts may contain bugs that could impact our conclusions. To increase our confidence in our implementation, we did many sanity checks. Specifically, we checked that the number of tests is the same regardless of the build configuration and we confirmed that each log file contains the format appropriate for the used configuration. In addition, we wrote tests for our code and did pair programming.

Construct. We have not compared our approach with other techniques that have been developed to speed up execution of tests, including regression test selection [35, 41], test case minimization [34],

and efficient class loading [33]. We believe that those other techniques are *orthogonal* to our approach and could provide additional benefits if used together. We discuss the details of other techniques in the Related Work section.

ForkScript and VmVm share the same overall goal (reducing time needed to run tests in isolation), but take radically different approaches. VmVm requires bytecode instrumentation to change each test to be self-isolated, and is quite brittle and can result in unexpected test failures. It was also not designed for Java 8 (but Java 7), let alone today's Java 14. Instead, ForkScript changes the build system to run each test more efficiently. We were faced with 1) using only very old, Java 7, and few projects for our evaluation, 2) use an apparently updated version of VmVm that is also fragile and unsupported (https://github.com/Programming-Systems-Lab/vmvm/issues/7), or 3) not report any unfair comparison in this paper. We thought that choice 3 was the fairest for everyone.

9 RELATED WORK

Test-order dependencies and test isolation. Muslu et al. [32] showed that running tests without isolation hides bugs. Zhang et al. [42] were among the first to demonstrate the impact of test order dependence on test outcome. They also proposed a technique to detect test order dependencies and showed the impact of detected dependencies on five test-case prioritization techniques. Bell and Kaiser [6] performed an extensive study to evaluate the cost of isolating tests via forking and presented a technique, named VmVm, for unit test virtualization. Later work by Bell et al. [7] presented ElectricTest, an approach for detecting dependencies among tests by utilizing JVM's garbage collection and profiling. By forcing garbage collection at the end of each test, ElectricTest captures if objects modified by one test are read by the subsequent test(s). Gyori et al. [23] introduced PolDet to detect tests that pollute shared state. PolDet captures the state of the heap at the beginning and at the end of each test and reports if the captured states differ. Unlike prior work, we focused on identifying hidden overhead in forked runs, and we proposed a technique to remove costly IPC for those build configurations when such communication is not necessary.

Flaky tests. Test-order dependency is only one source of flaky tests, i.e., tests that non-deterministically pass or fail for the same input without any change to the program [19, 20, 28, 30]. Other sources of flakiness include non-determinism due to thread scheduling, network access, IO, etc. Initial work on flaky tests mostly focused on creating taxonomy of flaky tests and avoiding flakiness. Recent work focuses on automatically detecting flaky tests [8]. Our work is on avoiding (rather than detecting) flaky tests due to test dependencies.

Build systems. A build system is in charge of orchestrating the execution of build tasks [16, 24, 29, 36]. Although most build systems support a handful of built-in in process tasks, in practice most build tasks involve spawning a process and waiting for it to complete. Such orchestration, no matter how simple it may be, inevitably introduces some overhead. Depending on the number of advanced features the build system may support (e.g., incrementality, shared caches, process sand-boxing) [12, 17, 22], the introduced overhead may be significant. For example, to implement a simple timestamp-based incrementality, the build system has to record all input and output files of a build task and check their timestamps before it determines if the task needs to be rerun; to support fetching build outputs from a shared cache, a fingerprint of the build task must be determined, which (among other things) includes computing the checksums of all input files; finally, if the goal is to ensure that a build task doesn't read/write any files other than those it declares, the task must be run in a sandbox or otherwise monitored for file accesses it makes.

All those advanced features carry certain benefits at the cost of adding more overhead in the worst-case scenario (e.g., clean builds with cold caches). It is up to the developers of a project to decide what is the best trade-off for building their project. From the build system's point of view, ForkScript replaces a number of fine-grained build tasks (individual test executions) with a single build task (which executes all tests). Making the granularity of build tasks coarser is likely to undermine the benefits of the aforementioned advanced features (incrementality/cacheability/sandboxing). However, many projects are not set up to use those features in the first place (or their build system simply doesn't support them), in which case there is no penalty to pay for using ForkScript. Using ForkScript is also appropriate in continuous integration loops which are typically configured to run clean builds only [10, 25, 39].

Other techniques to speed up JVM runs. Nikolov et al. [33] proposed recoverable class loaders, an approach to snapshot class loaders and associated class objects. Xu and Rountev [40] presented an approach for identifying inappropriate use of data structures in Java programs. Nailgun [18], which is currently maintained by Facebook developers, provides a client, protocol, and server for running Java programs without paying the startup cost. Our work is orthogonal to prior efforts and optimizes test isolation by avoiding overly generic implementation of inter process communication.

10 CONCLUSION

This experience report captures our efforts to demystify and better understand why test isolation is computationally expensive. The result of our efforts — a patch integrated into the popular Apache Maven build system — will result in a significant reduction in test execution time for many developers. Based on these results, we also created a research prototype, ForkScript. ForkScript is based on our finding that a large portion of test execution time, if test isolation is needed, goes into overly generic infrastructure of build systems, e.g., inter process communication. To reduce this time, ForkScript creates a specialized script for a given build configuration and maintains the script as the build configuration changes. ForkScript is publicly available. Our approach of using differential testing of multiple build systems to detect performance bugs could be useful for other researchers or practitioners. We continue to engage with the open source community to further improve the inter process communication components in Maven.

ACKNOWLEDGMENTS

The authors thank Karl Palmskog, Chenguang Zhu, and the anonymous reviewers for their comments and feedback. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517.

REFERENCES

[1] Apache. 2018. *Test XML file is not valid when rerun "fails" with an assumption.* https://issues.apache.org/jira/projects/SUREFIRE/issues/SUREFIRE-1556.

[2] Apache. 2018. *Thread Pool in Maven Surefire Code.* https://github.com/apache/maven-surefire.

[3] Apache. 2019. *Maven Surefire Plugin.* https://maven.apache.org/surefire/maven-surefire-plugin/.

[4] Apache. 2019. *Maven Surefire Plugin — surefire:test.* https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html.

[5] Apache. 2019. *Should Surefire specialize test runner when test isolation (i.e., fork) is needed?* https://issues.apache.org/jira/browse/SUREFIRE-1516.

[6] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *International Conference on Software Engineering*. 550–561.

[7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *International Symposium on Foundations of Software Engineering*. 770–781.

[8] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *International Conference on Software Engineering*. 433–444.

[9] Cor-Paul Bezemer, Shane Mcintosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Softw. Engg.* 22, 6 (2017), 3117–3148.

[10] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build System with Lazy Retrieval for Java Projects. In *International Symposium on Foundations of Software Engineering*. 643–654.

[11] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *International Symposium on Foundations of Software Engineering*. 809–820.

[12] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *International Symposium on Formal Methods*. 643–657.

[13] Al Danial. 2020. *Cloc.* https://github.com/AlDanial/cloc.

[14] Tibor Digana. 2019. [SUREFIRE-1516] Poor performance in reuseForks=false. https://github.com/apache/maven-surefire/commit/5148b02ba552cd79ac212b869dec10d01ba4d2e6.

[15] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *International Symposium on Foundations of Software Engineering*. 235–245.

[16] Sebastian Erdweg, Moritz Lichter, and Weiel Manuel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Object-Oriented Programming, Systems, Languages & Applications*. 89–106.

[17] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *International Conference on Software Engineering, Software Engineering in Practice*. 11–20.

[18] Facebook. 2020. *Nailgun.* https://github.com/facebook/nailgun.

[19] Martin Fowler. 2018. *Eradicating Non-Determinism in Tests.* http://martinfowler.com/articles/nonDeterminism.html.

[20] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should We Control?. In *International Conference on Software Engineering*. 55–65.

[21] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *International Symposium on Software Testing and Analysis*. 211–222.

[22] Google. 2020. *Bazel.* https://bazel.build/.

[23] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *International Symposium on Software Testing and Analysis*. 223–233.

[24] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. 2002. *The Vesta Software Configuration Management System.* Research Report. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-177.pdf.

[25] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Automated Software Engineering*. 426–437.

[26] Sam Kamin, Lars Clausen, and Ava Jarvis. 2003. Jumbo: Run-time Code Generation for Java and Its Applications. In *International Symposium on Code Generation and Optimization*. 48–56.

[27] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *International Symposium on Foundations of Software Engineering*. 583–594.

[28] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653.

[29] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. 2012. The Evolution of Java Build Systems. *Empirical Software Engineering* 17, 4–5 (2012), 578–608.

[30] Atif M. Memon and Myra B. Cohen. 2013. Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness. In *International Conference on Software Engineering*. 1479–1480.

[31] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proc. ACM Program. Lang.* 2, International Conference on Functional Programming (2018).

[32] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *International Symposium on Foundations of Software Engineering*. 496–499.

[33] Vladimir Nikolov, Rüdiger Kapitza, and Franz J Hauck. 2009. Recoverable Class Loaders for a Fast Restart of Java Applications. *Mobile Networks and Applications* 14, 1 (2009), 53–64.

[34] Voas JM. Offutt J, Pan J. 1995. Procedures for Reducing the Size of Coverage-based Test Sets. In *International Conference on Testing Computer Software*. 111–123.

[35] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *Transactions on Software Engineering* 22, 8 (1996), 529–551.

[36] Peter Smith. 2011. *Software Build Systems: Principles and Experience.* Addison-Wesley Professional.

[37] Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming.* Springer Berlin Heidelberg, 30–50.

[38] tevemadar. 2018. *Blocking on stdin makes Java process take 350ms more to exit.* https://stackoverflow.com/a/48979347.

[39] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *International Symposium on Foundations of Software Engineering*. 805–816.

[40] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Conference on Programming Language Design and Implementation*. 160–173.

[41] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[42] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *International Symposium on Software Testing and Analysis*. 385–396.