2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)

Wait Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness.

Kai Presler-Marshall, Eric Horton, Sarah Heckman, Kathryn T. Stolee
Department of Computer Science
North Carolina State University
Raleigh, NC
{kpresle, ewhorton, sarah_heckman, ktstolee}@ncsu.edu

***Abstract*—Flaky tests are a source of frustration and uncertainty for developers. In an educational environment, flaky tests can create doubts related to software behavior and student grades, especially when the grades depend on tests passing. NC State University's junior-level software engineering course models industrial practice through team-based development and testing of new features on a large electronic health record (EHR) system, iTrust2. Students are expected to maintain and supplement an extensive suite of UI tests using Selenium WebDriver. Team builds are run on the course's continuous integration (CI) infrastructure. Students report, and we confirm, that tests that pass on one build will inexplicably fail on the next, impacting productivity and confidence in code quality and the CI system. The goal of this work is to find and fix the sources of flaky tests in iTrust2.**

We analyze configurations of Selenium using different underlying web browsers and timeout strategies (waits) for both test stability and runtime performance. We also consider underlying hardware and operating systems. Our results show that HtmlUnit with Thread waits provides the lowest number of test failures and best runtime on poor-performing hardware. When given more resources (e.g., more memory and a faster CPU), Google Chrome with Angular waits is less flaky and faster than HtmlUnit, especially if the browser instance is not restarted between tests. The outcomes of this research are a more stable and substantially faster teaching application and a recommendation on how to configure Selenium for applications similar to iTrust2 that run in a CI environment.

***Index Terms*—Software Testing, Selenium, WebDriver, GUI Tests, Flaky Tests**

I. Introduction

Selenium WebDriver is a tool designed to automate interactions with web browsers, and supports directly controlling many popular browsers through a modular architecture. Its primary use case is for developing repeatable suites of integration and regression tests for web-based user interfaces (UI).

Tests use Selenium to simulate user driving the UI, then verify expectations about the results of the actions performed. A common issue is that Selenium tests, like other automated tests with a broad scope, are often non-deterministic (flaky) [1]. Test flakiness is costly, as effort must be spent determining if the test failed due to an underlying bug [2], or if the failure is a result of the test environment. In an academic setting, flaky tests may impact student grades without fault of their own.

This work studies Selenium tests in the context of the undergraduate software engineering class project iTrust2 at NC State University with the goal of increasing stability experienced by students as measured by test flakiness and runtime performance. iTrust2 is open source and designed to provide students with an industry-like experience by exposing them to a large system and continuous integration (CI) [3]. Flaky tests are a source of frustration among students. While this may reflect actual industry experiences, for an educational context that introduces students to software engineering principles, ambiguous feedback from flaky tests can cause undue stress.

There are many potential sources of flakiness, but a typical situation involves a test case that attempts to verify the presence of a UI element before the browser has completely loaded, causing it to fail. To expose the source of flakiness in iTrust2, we start by analyzing the impact of the WebDriver and waiting strategy. We then test the best WebDrivers and wait strategies on three different hardware configurations and operating systems. Our results indicate that reusing a single Chrome instance across an entire test suite, when also paired with Angular waits, gives universally lower flakiness and runtime regardless of the hardware and software environment. The improvement is sufficient to allow the iTrust2 tests to run successfully on all hardware we tried. We refactored the iTrust2 test suite based on these findings, and it has already proved successful in class, giving a project that runs more reliably and more quickly for student projects.

In summary, our work provides the following contributions:

1) A more stable version of iTrust2 that is suitable for use as a teaching tool in undergraduate software engineering classes and which performs well on our CI environment.
2) A comparison of four methods of waiting for expected conditions on webpages when performing automated testing with Selenium.
3) A comparison of runtime and stability of iTrust2's Selenium tests on three different hardware and operating system configurations.
4) A recommendation on the optimal configuration of Selenium for applications similar to iTrust2.

II. Motivation

NC State University's undergraduate software engineering course uses iTrust2, a large Java EE medical records application, as one of its primary teaching tools. A successor to the original iTrust application that saw the course through ten

978-1-7281-2237-3/19/$31.00 ©2019 IEEE
DOI 10.1109/AST.2019.000-1

years [3], iTrust2 was introduced to students in Fall 2017 as part of a larger course redesign. iTrust2 uses the Spring and AngularJS frameworks used in many enterprise applications. It consists of about 30,000 lines of Java and JavaScript code tested with nearly 500 Selenium tests.

Each semester, twenty five groups of students, working in teams of four or five, push their work to GitHub, where it is then automatically built and tested using Jenkins CI [4]. When students develop far in advance of a deadline, Jenkins is able to return feedback quickly (within 15 minutes); however, as the deadline approaches and load on the system increases, feedback becomes less timely. This exacerbates the issue of test flakiness by giving students less time to respond to any failures and ascertain their cause.

Consider the method in Figure 1, `fillHospitalFields`, extracted from iTrust2. It is a subroutine in a test case which verifies that submitting the form results in a new hospital record being registered with the system. The method starts with the implicit assumption that the browser is currently on the page for creating a new hospital object. It then asks Selenium to find and fill the input fields for name (lines 2-5), address (lines 6-8), state (lines 10-12), and zipcode (lines 14-16). Finally, it instructs Selenium to click the submit button and trigger a form submission (line 18).

In this example, Selenium may fail because it cannot find the element it looks for (e.g., an element with an `id` called `address`). Such a failure could be caused by several mistakes in the test or application: the wrong page was specified by the test case; the locator used to select the element was faulty; or there was a bug in the underlying application being tested. However, it is also possible that the test and application are both fine, and the befoe the browser had finished loading the page, Selenium checked, and failed. Selenium has no way of knowing if the UI will ever be ready, so this leaves it to developers to tell Selenium when to check.

The motivation of this work is to proactively identify and remove test flakiness and improve performance in iTrust2 by finding and implementing optimal Selenium and system configurations. All tests are known to be capable of passing, but under the conditions of student computers and our CI environment they do not all pass consistently.

III. Background

A WebDriver class implements the Selenium interface `WebDriver`. Each driver provides an interface for Selenium to control a single web browser. All WebDrivers in this study are named for the web browser they control. For example, ChromeDriver is the WebDriver implementation for the Google Chrome browser. Because WebDrivers have a 1:1 mapping with their browsers, we refer to the driver and the browser interchangeably.

A. Drivers

We use the drivers for Chrome, Firefox, PhantomJS, and HtmlUnit in this study. The HtmlUnit driver represents a headless browser designed specifically for automation; PhantomJS

```
1  public void fillHospitalFields (String
     ↪ hospitalName) {
2      final WebElement name =
         ↪ driver.findElement( By.id( "name" )
         ↪ );
3      name.clear();
4      name.sendKeys( hospitalName );
5
6      final WebElement address =
         ↪ driver.findElement( By.id(
         ↪ "address" ) );
7      address.clear();
8      address.sendKeys( "121 Canada Road" );
9
10     final WebElement state =
         ↪ driver.findElement( By.id( "state"
         ↪ ) );
11     final Select dropdown = new Select(
         ↪ state );
12     dropdown.selectByVisibleText( "CA" );
13
14     final WebElement zip =
         ↪ driver.findElement( By.id( "zip" )
         ↪ );
15     zip.clear();
16     zip.sendKeys( "00912" );
17
18     driver.findElement( By.className( "btn"
         ↪ ) ).click();
19 }
```

Fig. 1. A method from one of the iTrust2 Selenium tests that automates filling in fields on a web page.

is a more capable browser designed for the same purpose; Chrome and Firefox are two popular and widely-used web browsers. The default configuration is used for HtmlUnit and PhantomJS, as both are already headless. Chrome and Firefox are run in headless mode (a requirement for our CI environment), with Chrome additionally specifying the options `window-size=1200x600` and `blink-settings=imagesEnabled=false`, both of which were selected to improve runtime. We omitted drivers for browsers not supported by every major OS, such as Apple Safari and Microsoft Edge.

B. Wait Strategies

Several waiting strategies are considered. The first, *No Wait* is the default behavior of immediately failing when an element is not found. *No Wait* informs a baseline against which to compare other wait strategies. *Thread Wait* calls Java's `Thread::sleep`, which pauses thread execution for a fixed period of time. *Explicit Wait* is a Selenium construct that tells the driver to wait for an explicit amount of time or until some condition has been satisfied (whichever occurs first). Explicit waits are supported by all drivers. We use explicit waits to verify the presence of elements. For example, the following will wait for an element with the `name` of `notes` to appear:

```
1 WebDriverWait wait = new WebDriverWait(
    ↪ driver, 2 );
2 wait.until( ExpectedConditions
    ↪ .visibilityOfElementLocated( By.name(
    ↪ "notes" ) ) );
```

Angular Wait is a Selenium construct that tells the driver to wait until the Angular web framework has completed all

requests. It is only supported by the Chrome driver, and can be used as follows.

```
new NgWebDriver( (ChromeDriver) driver
  ↪ ).waitForAngularRequestsToFinish();
```

Unlike Explicit Waits, Angular Waits do not wait for a specific element to appear: rather they wait until *all* dynamic requests are finished. This has the potential to work better if the locator that would be passed to an Explicit Wait is overly general and thus would locate an element before the page is truly ready.

C. Current Configuration

The existing test suite for iTrust2 uses the HtmlUnit driver with a combination of *No Wait* and *Explicit Wait* strategies to verify the correctness of elements. HtmlUnit has been used for its reasonable runtime performance on our Jenkins CI systems, with 59 *Explicit Waits* scattered through individual tests where the *No Wait* approach proved insufficient.

IV. Study

We explore the following research questions, measuring test flakiness and runtime performance as the dependent variables, in the context of iTrust2:

- **RQ1** What is the impact of the WebDriver?
- **RQ2** Which wait methods are the most stable?
- **RQ3** What is the impact of hardware (CPU and memory)?
- **RQ4** What effect does the host operating system have?
- **RQ5** What is the effect of restarting the browser between individual tests?

We refer to a single run of an application's entire test suite as a *build*. A group of multiple builds is referred to as an *evaluation*. All evaluations in this work contain 30 builds. An *evaluation* is used to determine runtime performance, measured in seconds (i.e., average test execution time over all the builds in an evaluation). Because we know that all tests can pass, any test failure is seen as indicative of test flakiness. Thus, test flakiness is defined as the sum of all test failures over an execution.

We refer to the choice of driver and the waiting strategy employed as *WebDriver configuration*. We refer to choice of CPU, memory, and operating system as *system configuration*. When considering a combination of WebDriver and system configurations, we say *Selenium configuration*, or just *configuration* if it is not ambiguous.

Our configuration options are summarized in Table I. Rows A and B correspond to WebDriver configuration and will guide RQ1 and RQ2 (Section V-A), rows C and D are hardware configuration and will guide RQ3, and row E guides RQ4 (Section V-B). The *RQ* columns identify the configurations used in the evaluation. For example, RQ1 has a * for row A, meaning all four options are considered. In row C, RQ1 uses column 2, representing 4GB of memory. Not all options are possible (e.g., HtmlUnit with Angular waits), but this provides a general outline for how each RQ was evaluated.

A. Setup

To address RQ1 and RQ2, we modified the current iTrust2 codebase and introduced a common superclass for all test classes. This superclass provides a WebDriver factory method and a method for performing waits. To obtain a list of flaky tests, we mined the Jenkins test logs from our undergraduate software engineering course in Spring 2018. Any Selenium test failure observed that appeared unconnected to the Git commit that triggered the build was considered a potentially flaky one and was included in our study. We manually analyzed 1,000 Jenkins test logs, and found 19 distinct locations in the source code where tests appeared to be flaky. Before every flaky location, we inserted a call to the waiting method in our superclass. We branched the codebase, implementing the WebDriver factory and waiting method for every valid combination of driver and waits (see Table I, rows A & B, yielding 13 valid configurations[1]).

To address RQ3 and RQ4, we procured two computers. First, an Acer netbook provisioned with Ubuntu 17.10, known hereafter as "NB-Linux". It was deliberately selected for its poor performance, as it represents the lower end of what students have been observed using. Second is an HP workstation system provisioned with Ubuntu 17.10 (known hereafter as "HP-Linux") and Windows 10 (known hereafter as "HP-Windows"). It was selected for giving performance similar to most student systems and to evaluate the impact of operating system when hardware is controlled for. Turbo Boost was disabled on both platforms, and Hyper-threading on the HP Z420 (the Acer's CPU does not support any form of SMT[2]), to result in a more consistent execution environment. Evaluations on HP-Linux were tested with 2, 4, 8, 16, and 32GB memory. Evaluations on HP-Windows were run at 8GB memory as a comparison against HP-Linux at 8GB.

To address RQ5 we reconfigured our WebDriver factory to not restart the browser between individual tests and retested on our three environments (NB-Linux, HP-Linux, and HP-Windows).

B. Execution

After each build, the execution time and list of failing test cases were recorded. At the end of an evaluation (30 builds), test flakiness was recorded as the sum of all failures seen in each build, and runtime was computed by averaging the runtime of each build within the evaluation.

We ran evaluations for each configuration on our procured systems, using performance results on NB-Linux to inform our selection of run configurations on the HP workstation.

V. Results

The results from our study can be grouped into three broad categories: the impact of WebDriver configurations (RQ1 and

[1](Chrome + Firefox + HtmlUnit + PhantomJS) * (No Wait + Thread.sleep + Explicit) = 12 + Chrome * AngularWait = 13

[2]Simultaneous Multithreading, a hardware technique allowing the simultaneous execution of two or more logical threads per physical CPU core. Hyper-threading is Intel's implementation of the technology.

TABLE I
SELENIUM & SYSTEM CONFIGURATION OPTIONS

		Options									
	Factor	1	2	3	4	5	RQ1	RQ2	RQ3	RQ4	RQ5
A	Wait	No Wait	Explicit	Thread.sleep	Angular	-	*	*	2,4	2,4	2,4
B	WebDriver	HtmlUnit	Chrome	Firefox	PhantomJS	-	*	*	1,2	1,2	1,2
C	Memory	2GB	4GB	8GB	16GB	32GB	2	2	*	3	3
D	Processor	AMD C60 (NB)	Intel E5-1620 (HP)	-	-	-	1	1	1,2	2	1,2
E	OS	Windows 10	Linux 4.13	-	-	-	2	2	2	1,2	1,2

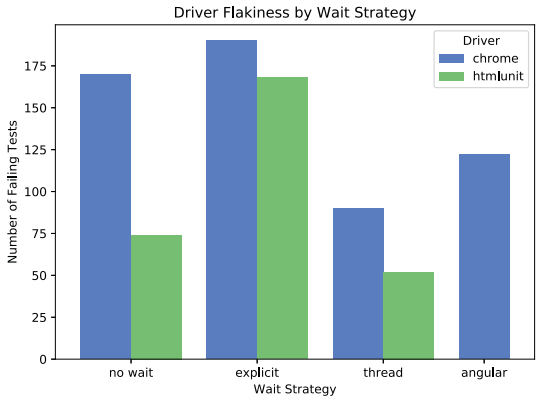

Fig. 2. Total failing test cases for supported waiting strategies with Chrome and HtmlUnit. Each bar represents one evaluation. Evaluations were run on NB-Linux.

RQ2), the impact of system configuration (RQ3 and RQ4), and further optimization (RQ5)[3].

A. WebDriver Configuration

RQ1 addresses the impact of WebDriver choice on test stability and runtime. This corresponds to varying rows A and B of Table I. Even on our fast HP-Linux system, running the test suite with Firefox or PhantomJS took well over an hour a build, making them unsuitable for use in a CI environment.

HtmlUnit, the least featured and least resource-intensive WebDriver used, experienced fewer flaky tests overall than Chrome on NB-Linux. This was consistent across all wait strategies on this system; not once did Chrome deliver a more stable testing experience. These results are shown in Figure 2; the Y-axis reports the total number of tests that failed across each evaluation. Because each test was run thirty times, a test that failed in multiple builds will increase the count on each observed failure. Runtime differences between the WebDrivers were minor; an average build time of 31 minutes for HtmlUnit and 35 minutes for Chrome.

> ***RQ1:*** *HtmlUnit yields fewer flaky tests than Chrome on NB-Linux, regardless of wait strategy. HtmlUnit is approximately 10% faster than Chrome in terms of test runtime.*

[3]Our updated version of iTrust2 and evaluation scripts are located at https://github.com/ncsu-csc326/iTrust2-SeleniumAnalysis

RQ2 addresses the impact of the specific waiting strategy on test flakiness. We evaluated waiting strategy for Chrome and HtmlUnit by running both browsers with all supported waits on NB-Linux. Figure 2 presents the total number of test flakes seen in each evaluation.

We see that Thread Waits perform the best for both HtmlUnit and Chrome, while Explicit Waits performed the worst. The performance of Thread Waits was unsurprising – by suspending test execution for a relatively long (5 second) period of time, we give the browser hopefully ample time to catch up to where the test expects it to be. Surprisingly, No Wait manages to perform better than Explicit Wait, particularly so for HtmlUnit, where it resulted in under half of the flakes seen with Explicit Waits. Explicit Waits used the exact same timeout (5 seconds) as Thread Waits, so we expected to see similar results for both. While waiting approach had a sizable impact upon stability, its impact upon runtime was minor on both browsers: no more than an 8%[4] difference was observed between the fastest (no waits) and slowest (explicit waits and thread waits) regardless of browser.

We also acknowledge that RQ1 and RQ2 were evaluated on poor hardware. As we show in the next section, better hardware leads to better performance and less flakiness for Chrome. At the same time, using a slower system for evaluation is valuable as it mimics some students' situations.

We note here that while Thread Waits provide the lowest flakiness score, the number of unique tests impacted by the failures is actually the highest. That is, the Thread Wait failures are more insidious, being both more likely to occur for any test and less predictable. On the other hand, the Explicit Waits and Angular Waits are more predictable; a test failing with an Explicit or Angular Wait is more likely to fail again within the evaluation. Since we want to suggest a wait strategy that has predictable behavior, we move forward with Angular and Explicit Waits when evaluating system configurations.

> ***RQ2:*** *Thread waits give the lowest flakiness for both HtmlUnit and Chrome, with Explicit Waits giving the highest.*

B. System Configuration

Anecdotally, we expected that slower hardware results in more unstable and slower builds. We consider three types of system configuration: the CPU, the amount of RAM, and the

[4]Calculated as $(T_{slow} - T_{fast})/T_{fast}$, where T_{fast} is the runtime of the faster build, and T_{slow} is the runtime of the slower build

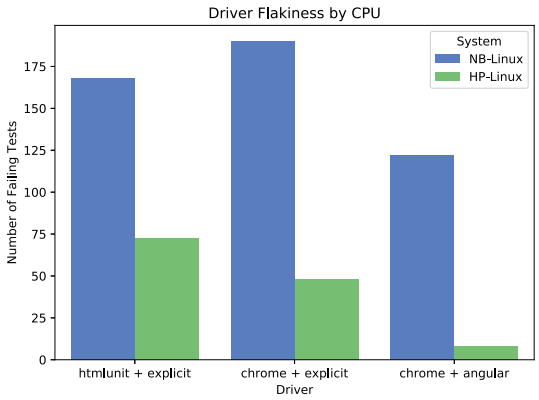

Fig. 3. Number of failing tests for HtmlUnit with explicit waits and Chrome with explicit and Angular waits for NB-Linux and HP-Linux systems (fixed 4GB memory, increasing CPU performance).

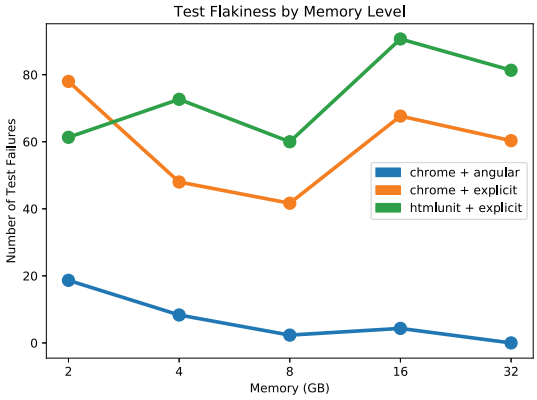

Fig. 4. Total number of test case failures for all builds in evaluations of Chrome with explicit and Angular waits and HtmlUnit with explicit. All evaluations were run on the HP-Linux platform (D2/E2).

OS managing test processes. We prune our search space by focusing our testing on Angular Waits and Explicit Waits.

1) Increasing CPU: Evaluations of flaky tests in Chrome and HtmlUnit on NB-Linux and HP-Linux (at 4GB RAM) are presented in Figure 3. Note that there are no results for HtmlUnit and Angular Waits, as Chrome is the only browser to support them. For both browsers, moving to HP-Linux resulted in substantially fewer flaky tests across the evaluations, regardless of wait strategy. The impact was particularly pronounced for Chrome, with test flakiness falling by over 70%[5] in both configurations, but HtmlUnit still saw a very respectable improvement of 55%.

The average build time over all evaluations was reduced by 80-84% when moving to faster hardware (i.e., D1 to D2). With Chrome, the average build times were reduced from 35 minutes to 7 minutes; with HtmlUnit, they dropped from 31 minutes to 5 minutes. Thus, not only does Chrome give far more stable build results on fast hardware, it does so quickly as well. Fast build times are imperative for CI environments, and better hardware is an easy way to achieve this.

RQ3a: *A faster CPU results in a substantially faster build on both HtmlUnit and Chrome.*

2) Increasing Memory: Figure 4 shows the impact of manipulating the amount of available memory for each of three configurations of driver and wait method. The Y-axis shows the sum total of test failures over all 30 builds in an evaluation. The X-axis shows the memory configuration. Each evaluation was run three times (totaling 90 builds of iTrust2 and its test suite), and each data point on the graph is an average over the three evaluations. No significant outliers were observed in any configuration. For example, with *HtmlUnit + explicit*, there were an average of 61 test failures per evaluation with 2GB of memory. This is an average of two per build.

[5]Calculated as $(F_{high} - F_{low})/F_{high}$, where F_{high} is test flakiness from the flakier build, and F_{low} is the flakiness from the less flaky build

There is no clear trend on flakiness for *chrome + explicit* or for *htmlunit + explicit*, but the trend for *chrome + angular* is decreasing flakiness as memory is increased. If we had cut off the evaluation at 8GB, we would have concluded that *chrome + explicit* has fewer flakes with more memory, but the behavior at and above 16GB is not clear. Further exploration is needed.

HtmlUnit had more flakes, on average, with extra memory. It is the most lightweight browser tested and does not appear require significant resources. However, our evaluations do show the number of test failures varied wildly, suggesting that HtmlUnit may not give consistent results, even with sufficient hardware. This is supported by the data for RQ2, where the naive No Wait approach outperformed the Explicit Wait.

RQ3b: *More memory results in tests that fail less regularly for Chrome + Angular, but not for the other configurations.*

3) Host Operating System: RQ3 specifically considers hardware. However, all results are presented for systems running Linux. Next, we turn to RQ4 to generalize past Linux. While the primary goal was to improve the experience on the CI server used for automated feedback and grading projects[6], a secondary goal was to give students a more stable testing environment for their local development.

Section V-B1 indicates that Chrome on a system with sufficient CPU and memory was the most stable configuration for Linux. However, an attempt to replicate this on Windows was unsuccessful. Our evaluations found 1923 test failures (across a 30-build evaluation) on Windows (E1) vs 12 on Linux (E2) with other factors held constant (A4/B2/C3/D2). The Windows build time was much higher, averaging 44 minutes versus six for Linux. Other browsers and waits also performed poorly on Windows, giving either high runtime, flakiness, or both. For instance, HtmlUnit was fast, but flaky, and Firefox and PhantomJS were still too slow. We turn now to a solution that helped all platforms, but particularly Windows.

[6]Our CI environment runs CentOS Linux 7.5

Fig. 5. Average build time, in seconds, of Chrome with Angular waits on each of the three systems under test. Evaluations **Without Restart** used a single instance of a WebDriver across all tests in a build. Evaluations **With Restart** created a new WebDriver instance before every test.

RQ4: *Windows gives worse performance than Linux with respect to flaky tests and runtime given comparable hardware.*

C. Restarts

Attempting to generalize our results to Windows worked poorly, with all configurations resulting in high test failures, unacceptable runtime, or both. We consider now the impact of restarting the browser between each test versus a configuration that does not do so.

By default, each test starts by launching a fresh WebDriver and logging in as a user of the appropriate type. Here, we modified the test suite to share a single Chrome instance across all tests rather than launching the browser between every test. To ensure a consistent starting environment, we introduced a method that would log out of the iTrust2 web portal if a user was logged in and ensured it was run before every test.

Figure 5 shows average test execution times when running with and without restarts on each of our systems (NB-Linux, HP-Linux, HP-Windows). Our results show a decrease in build time for every system under test. Even HP-Linux, which already saw the best runtime, saw the tests run 49% faster. Similar to HP-Linux, on NB-Linux we saw a 46% reduction in test execution time. However, HP-Windows saw the biggest proportional improvement, from 44 to five minutes, an improvement of 89%. In addition to faster speeds, none of the platforms had any flaky tests in this configuration. Results here did not generalize to other WebDrivers, where test flakiness and runtime both remained high. For instance, HtmlUnit on Windows remained flaky (giving over a thousand test failures across a single evaluation), while Firefox still was slow (with build time over an hour) on all platform configurations.

RQ5: *By using Angular waits and not restarting the Chrome browser between tests, we get substantially faster performance and no test failures.*

VI. Discussion and Future Work

Our work provides the first investigation of which we are aware into the effect of Selenium configuration on both reliability and runtime. We end our investigation with an optimal configuration for our use case of running a large Selenium test suite in a CI environment for an undergraduate software engineering class: Chrome + Angular waits + no browser restarts + Linux + fast processor. We have implemented this configuration in our current version of iTrust2 and reconfigured our CI environment accordingly. Knowledge of this configuration has already proven a valuable resource for the class by providing us with a substantially more stable teaching application that performs much better on our CI environment. However, there is still much left for future work.

Explicit Waits: In Section V-A, Explicit Waits give worse performance in terms of total test failures than any other approach. We presumed that test flakiness resulted from the browser having insufficient time to load a page before the test started interacting with it, so any waiting approach should perform better than none at all. We struggle to explain why performance here was so poor, particularly in relation to the Thread Wait approach, so further investigation is needed.

Hardware and Operating System: We see in Figure 4 that the number of test failures increases when going from 8GB to 16GB. Every other memory increase for Chrome, and all but one for HtmlUnit, resulted in a decrease in number of failing tests. While the three evaluations performed at each configuration suggests this is not due to random variation, we cannot be certain without further exploration.

WebDrivers: We limited our search to WebDriver implementations which run on all major operating systems. It remains open as to whether an untested driver performs better on any of the operating systems included in the study. If this is the case, future applications may benefit by detecting and using the best driver for the platform on which they are running.

VII. Threats to Validity

The threats to validity of our experiments are as follows:

Conclusion Validity: Due to the time taken to run experiments, most of the conclusions on test flakiness in this paper were drawn from a single 30-build evaluation (the conclusions to RQ3b were drawn from an average across three evaluations). This may not be sufficient to observe all tests that could be flaky. However, it does represent a number of builds similar to the number of teams in the software engineering course each semester. Still, extending the number of builds per evaluation may lead to different conclusions.

Internal Validity: In our experiments, we were careful to vary only a single factor at a time (WebDriver, wait strategy, CPU, memory, operating system, or browser restarts). We also controlled for other factors, such as software versions, memory and disk performance, and internet connection. All testing was performed in an automated and repeatable manner. We thus believe that the differences we observed come from the experimental factor that was varied.

Construct Validity: We assume that test flakiness occurs because the test attempts to interact with an element that has not yet appeared on the page; other factors, such as test order, may also impact correctness. However, we eliminated all observed test case flakiness solely by changing Selenium settings, and without changing the order in which the tests are run, which suggests that our tests do not face this issue.

External Validity: Our study has focused on a single artifact, iTrust2. While the Spring and AngularJS frameworks used in iTrust2 are widely used, we have not attempted to replicate our results past iTrust2. We have, however, performed some generalization, applying our optimal configuration to two expanded versions of iTrust2 that were developed in parallel to our work. Our solution, when applied, eliminated all test flakiness and improved runtime. Replicating our work on projects other than iTrust2 would see if the results generalize further.

VIII. Related Work

Automated testing frameworks for web applications are necessary [5], and research has focused on generating robust element locators [6] and repairing test cases when localization fails [7]. Closest to our work, Kuutila et al. benchmark the effect of programming language and WebDriver choice on overall performance [8], finding that the choice of WebDriver and language bindings have an impact on mean execution time of Selenium tests as well as test results. Their results agree broadly with ours that Chrome is both the fastest and the most stable of the drivers that we consider.

Flakiness has been found to be particularly troublesome in large continuous integration environments [2]. Bell et al. present DeFlaker, an application that automatically detects flaky tests in Java applications [9]. Their work focuses not on our goal of fixing flaky tests, but finding them in the first place. Additionally, their implementation relies heavily on Java AST analysis, which makes it inapplicable to applications that place functionality in a Javascript front-end, such as iTrust2.

Luo et al. analyze what causes automated tests in open-source projects to be flaky, revealing that asynchronous behavior is the leading cause of flakes [10]. This agrees with our hypothesis that Selenium tests are flaky because of the time for dynamic elements to appear on a web page.

Denzler and Gruntz [11] and Luukkainen et al. [12] discuss using the Spring framework for teaching applications in their courses. However, they make no mention of testing these applications nor any flakiness experienced with them.

No study we are aware of focuses on removing test flakiness in Selenium. Most studies instead focusing on regression tests and assuming that failures happen because of changes to a codebase (e.g., DeFlaker integrates with version control to watch for changes [9]). Others, such as Kuutila et al. [8], use different combinations of waits for each browser under test in an attempt to guarantee correctness. Doing so prevents their results from generalizing to the consideration of the effect of configuration options on test stability.

IX. Conclusions

Our work has implications for automated testing of web applications using Selenium. We evaluated four WebDrivers and four different approaches for waiting on elements to appear on a page. We demonstrated that there are differences between the studied WebDrivers: HtmlUnit driver performs best where system resources are heavily constrained and the browsers must be run in their default configuration, while Chrome works best on faster systems or where configuration can be optimized. We demonstrated that hardware has a significant impact upon the runtime and reliability of a test suite and that a faster CI environment makes a meaningful difference. Our work has already contributed a substantially more stable teaching application to our undergraduate software engineering course, giving students confidence in the tests they write and the results they see.

X. Acknowledgements

This work was supported in part by the NC State DELTA Course Redesign Grant and NSF SHF grants #1714699, #1749936, and #1645136. We would also like to thank T. Dickerson, E. Gilbert, D. Grochmal, C. Harris, A. Hayes, R. Jaouhari, N. Landsberg, M. Lemons, A. Phelps, D. Rao, A. Shaikh, and A. Wright for their assistance developing iTrust2. Finally, we'd like to thank B. Thorne for feedback throughout the project.

References

[1] M. Fowler. (2011) Eradicating non-determinism in tests. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[2] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *International Conference on Software Engineering*, vol. 2, May 2015, pp. 39–48.

[3] S. Heckman, K. T. Stolee, and C. Parnin, "10+ years of teaching software engineering with itrust: The good, the bad, and the ugly," in *International Conference on Software Engineering: Software Engineering Education and Training*, ser. ICSE-SEET '18. ACM, 2018, pp. 1–4. [Online]. Available: http://doi.acm.org/10.1145/3183377.3183393

[4] S. Heckman and J. King, "Developing software engineering skills using real tools for automated grading," in *Technical Symposium on Computer Science Education*, ser. SIGCSE '18. ACM, 2018, pp. 794–799. [Online]. Available: http://doi.acm.org/10.1145/3159450.3159595

[5] E. Vila, G. Novakova, and D. Todorova, "Automation testing framework for web applications with selenium webdriver: Opportunities and threats," in *International Conference on Advances in Image Processing*, ser. ICAIP 2017, 2017, pp. 144–150. [Online]. Available: http://doi.acm.org/10.1145/3133264.3133300

[6] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Reducing web test cases aging by means of robust xpath locators," in *Software Reliability Engineering Workshops*, Nov 2014, pp. 449–454.

[7] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Repairing selenium test cases: An industrial case study about web page element localization," in *Int'l Conf. on Software Testing, Verification and Validation*, March 2013, pp. 487–488.

[8] M. Kuutila, M. Mäntylä, and P. Raulamo-Jurvanen, "Benchmarking web-testing-selenium versus watir and the choice of programming language and browser," *arXiv preprint arXiv:1611.00578*, 2016.

[9] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *International Conference on Software Engineering*, 2018.

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[11] C. Denzler and D. Gruntz, "Design patterns: Between programming and software design," in *International Conference on Software Engineering*, ser. ICSE '08. ACM, 2008, pp. 801–804. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368202

[12] T. V. Matti Luukkainen, Arto Vihavainen, "Three years of design-based research to reform a software engineering curriculum," 2012.