On the Impact of Hitting System Resource Limits on Test Flakiness

Fabian Leinen
Technical University of Munich
Munich, Germany
fabian.leinen@tum.de

Alexander Perathoner
Technical University of Munich
Munich, Germany
alexander.perathoner@tum.de

Alexander Pretschner
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

ABSTRACT

Regression testing aims to determine whether a change to a system introduces new bugs or can be merged safely. Flaky tests, which are tests that fail non-deterministically and unrelated to the change, can undermine this effort. In research and practice alike, it is often assumed that limited test execution resources can lead to test flakiness. We hypothesize that hitting resource limits during test execution can lead to changed timing behavior, which in turn promotes timing-related flakiness. However, there is no empirical evidence indicating whether hitting these resource limits increases the likelihood of test flakiness. To shed light on this, we created a dataset of 20 open-source projects for macOS, which contains a total of 232 UI test cases, with 23 of them being flaky. For all tests, we measured the CPU usage continuously during test execution. We discovered that executions of flaky tests spend significantly more time at the CPU limit than executions of non-flaky tests. Contrary to our expectations, we found that failing runs of flaky tests spend less time at the CPU limit than passing runs.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**.

KEYWORDS

Flaky Tests, Regression Testing, Software Testing, Continuous Integration

ACM Reference Format:
Fabian Leinen, Alexander Perathoner, and Alexander Pretschner. 2024. On the Impact of Hitting System Resource Limits on Test Flakiness. In *2024 International Flaky Tests Workshop 2024 (FTW '24), April 14, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3643656.3643898

1 INTRODUCTION

Regression testing aims to find newly introduced bugs in changes to the system [1, 2]. Flaky tests, which are tests that fail non-deterministically and independently of changes to the test or the system under test (SuT) [3, 4, 5], are a major challenge in regression testing [6, 7] faced by many tech companies [8, 9, 10, 11, 12]. Studies in a variety of programming languages and domains such as JavaScript [13, 14], Android apps [15], user interface (UI) tests [16], and language and domain-independent environments [5], have shown that timing-related root causes, like async wait and concurrency, are the most prevalent causes of flaky tests. Timing-related flakiness stems from simultaneous concurrent tasks, which can introduce non-deterministic event sequencing and potentially result in flaky test outcomes. For instance, a race condition occurs when access to a shared variable is improperly synchronized between threads, making the final state dependent on the thread that accesses the variable first.

The underlying hypothesis of our work is that reaching CPU limits during test execution promotes timing-related flakiness. Empirical studies [17] and developer surveys [18, 19], where professionals reported that insufficient computational power such as CPU and RAM were major causes of flaky tests, support this hypothesis. Accordingly, researchers have suggested approaches that use this perceived characteristic to detect flaky tests and determine their root causes. For example, Silva et al. stress the CPU and RAM during test execution, thus increasing the likelihood of detecting flaky tests [20]. Similarly, Terragni et al. propose utilizing resource limitations to identify the root cause of flaky tests [21]. They claim that if a test passes without limited network bandwidth, but fails when the network bandwidth is limited, for instance, the root cause can be attributed to the network.

While previous studies have applied artificial resource limits to provoke test flakiness, to our knowledge no works exist that investigate whether actually hitting (natural) resource limits impacts test flakiness. To fill this gap, we conduct a study on a total of 232 test cases from 20 open-source projects. Out of these 232 test cases, 23 show multiple test verdicts and are hence considered flaky and further used for our study. We limited our selection to projects for macOS, because the cost of a macOS build server is usually higher than Linux or Windows[1], thereby prompting increased use of weaker and cheaper hardware. We further limit ourselves to UI tests since they have been found to be particularly susceptible to flakiness [16]. Taking into account 30 executions of each test case, we (1) identify which tests show flakiness and (2) measure their CPU usage to answer our research questions:

RQ1: To what extent do executions of flaky tests reach the CPU limit compared to executions of non-flaky tests?
To answer this question, we contrast executions of flaky tests with executions of non-flaky tests in terms of how much time they spend in CPU limits, that is when the CPU usage is at 100%. In our dataset, executions of flaky tests spend significantly more time in CPU limits compared to their total runtime than executions of non-flaky tests.

FTW '24, April 14, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0558-8/24/04…$15.00
https://doi.org/10.1145/3643656.3643898

[1]Price per minute on GitHub: Linux (4 vCPUs): $0.016; Windows (2 vCPUs): $0.016; macOS (3 or 4 vCPUs): $0.080; source https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions#per-minute-rates

RQ2: To what extent do failing executions of flaky tests reach the CPU limit compared to passing executions?
Considering only executions of flaky tests, we examine the proportion of time failing executions spend in CPU limits compared to passing executions. Contrary to expectations, we find that failed executions of flaky tests spend relatively less time in CPU limits than passed ones.

In summary, the contributions of this paper are as follows:

- **Evaluation:** Our study is the first to assess the impact of reaching CPU limits on test flakiness.
- **Dataset:** We present the first flaky test dataset[2] containing resource usage information, along with the first dataset of flaky tests for macOS. The dataset comprises 232 test cases (with 23 of them identified as flaky) from 20 open-source projects.
- **Software:** We provide open-source access to our GitHub action[3] designed to record resource usage and instructions on how to expand the presented dataset[4]. This tool can aid researchers and practitioners in studying the influence of resource limits on test flakiness in their systems.

2 RELATED WORK

Evidence that resource constraints cause flaky tests. According to a survey conducted by Parry et al., flaky test failures can occur due to resource exhaustion [18]. Respondents reported that unrelated system load, including antivirus scans, can also lead to flaky test failures. In another survey conducted by Habchi et al., developers pointed out that resource limits, specifically related to CPU and RAM, can cause tests to become flaky [19]. Presler-Marshall et al. conducted a test suite of approximately 500 Selenium tests under various configurations [22]. The study revealed that a more powerful CPU can reduce the chance of a test failing inconsistently between 55% and 70% compared to a weaker CPU.

Approaches actively limiting resources. A study by Silva et al. actively limited CPU, RAM, disk, and network [17]. The study concluded that resource limits affected 47% of all the flaky tests included in the study. Specifically, the study revealed that tests written in Java and JavaScript are more susceptible to CPU limits, while tests in Python are more sensitive to RAM limits. In their work, Silva et al. proposed Shaker, a tool designed to increase the probability of detecting flaky tests by adding stressor tasks that target the CPU and RAM to produce asynchronous waiting and concurrency [20]. An empirical evaluation found this approach to significantly increase the efficacy of rerunning tests in detecting flaky tests. Parry et al. assessed the efficacy of test rerunning for detecting flaky tests and introduced four distinct noise types during test execution, two of which were resource-related: threads were arbitrarily deprioritized, which resulted in reduced CPU time, and network speed was randomly restricted [23]. Although the effectiveness of each type of noise was not reported, the study revealed a nearly doubled number of flaky tests with noise compared to those without. Terragni et al. suggested limiting various resources, including CPU, RAM, and network bandwidth, to identify the root cause of flaky tests [21]. However, they neither implemented this approach nor evaluated it.

Gap. Anecdotical evidence suggests that limited resources can potentially increase or cause test flakiness, and some approaches have successfully induced flakiness by actively limiting resources. However, previous research has not probed whether running into resource limits, for example reaching the CPU limit, in normal, unrestricted testing environments enhances a test's flakiness.

3 DATA COLLECTION

Recall that the underlying hypothesis of our work is that reaching the CPU limit during test execution promotes timing-related flakiness. Prior to empirically investigating this hypothesis in the next section (see Section 4), we first assemble a dataset suitable to answer our research questions. To create this dataset, we have implemented a GitHub action designed to record resource usage, including CPU usage during test execution, and subsequently applied it to a series of open-source projects.

In the following sections, we initially introduce the GitHub action we utilized to record resource usage in Section 3.1, then subsequently present the dataset we compiled in Section 3.2.

3.1 GitHub Action: Measuring Resource Usage

To emulate the "real-world" continuous integration (CI) processes as closely as possible, we opt for using GitHub's CI infrastructure. We, therefore, implemented our tool for measuring resource usage as a GitHub action, which is a single task that can be added to a GitHub workflow, which is GitHub's term for a CI pipeline. It builds on the existing GitHub action `telemetry-action`[5]. This action employs the node.js `systeminformation` package to record the usage of CPU, RAM, disk and network once per second. Upon completion of a workflow, the action generates a series of graphs that are appended to the workflow's summary. However, this existing action presents two fundamental drawbacks for our purposes: First, the sampling frequency is set to a constant value of once per second. A higher frequency is desirable to capture short-term spikes in resource usage. Second, the only output format is a visual series of graphs, which makes further analysis inefficient. We therefore forked this action and modified it to accommodate our needs by setting the sampling frequency to 10 times per second and storing the recorded measurements as a JSON file.

To evaluate the overhead introduced by our action, we ran it for 25 minutes in a GitHub workflow. We applied the default process status command (`ps`) to evaluate the CPU utilization of the GitHub action at one-second intervals. In our experiment, the mean CPU usage was 14.2%, with a minimum and maximum of 10.0% and 54.4% respectively. We expected a high CPU workload at the end of our measurement when the action saves all the data. However, we also detected short-duration spikes of high CPU load during the execution of the action. Even though these might sporadically influence the test execution, we contend that similar short-term CPU spikes may also occur on CI servers due to other processes.

[2]Dataset: doi.org/10.6084/m9.figshare.24639174.v1
[3]GitHub action: github.com/tum-i4/workflow-telemetry-action
[4]Evaluation scripts: github.com/tum-i4/On-the-Impact-of-Hitting-System-Resource-Limitations-on-Test-Flakiness

[5]https://github.com/catchpoint/workflow-telemetry-action

Table 1: Projects used in the study. The full project names and URLs are available in the dataset.

Name		Test Suite	
Project	Stars	Tests	Runtime [min]
Maccy	7735	128	5.30
Sparkle	6409	3	4.40
Pine	3170	8	1.00
phpmon	2706	1	1.33
tip	916	2	0.13
fluentui-apple	783	9	3.00
clocker	520	39	4.00
formvalidator-swift	499	1	0.53
SlimHUD	262	17	0.50
mac-ibm-notifications	261	5	0.53
macos-menubar-wireguard	193	4	0.13
Komet	160	62	19.83
Calendr	150	30	3.70
Json-Model-Generator	80	1	0.63
AboutThisApp	76	2	0.20
StatusItemController	63	2	0.20
Splitter	38	12	2.33
prose-app-macos	23	1	0.50
TelegramColorPicker	12	3	0.73
Timer	2	2	1.63

3.2 Executing Open-source Test Suites

To empirically study the impact of CPU usage on flakiness, we compile a dataset of flaky test cases and executions. Since the cost of macOS build servers is typically higher than for other operating systems, and macOS has not yet been studied in the context of flaky tests, we opt to create a dataset of UI tests in macOS projects.

We use the SourceGraph search engine[6] to find open-source macOS projects containing UI tests using the following query:

```
select:repo (type:file content:XCUIApplication lang:Swift)
            (repo:has.tag(macos) OR
             repo:has.tag(osx) OR
             repo:has.topic(macOS))
```

At the time of our search in May 2023, this query returned a sample of 236 repositories. We manually filter out repositories that do not contain UI tests, only contain default tests auto-generated by the integrated development environment (IDE), or those that have no commits post-2018 and fail to compile. This leaves us with the 20 projects listed in Table 1. All these projects are written in Swift, and contain at least one macOS-compatible target and a test suite based on the XCTest[7] framework.

Given the limited number of available projects and the consequently low expected number of flaky tests, we manually look for commit messages that address flaky tests in all projects containing 10 or more tests. For the two projects where we find such a commit (Maccy and clocker), we use the commit before the repairing commit as the version of the project, rather than the latest commit. For all other projects, we use the latest commit. The used commit hashes are available in the dataset accompanying this paper.

To obtain test results and measure CPU usage, we fork each repository and add a workflow that starts our GitHub action to record resource usage and then execute all UI tests in the repository 30 times. These test executions are performed on a GitHub hosted runner with 3 CPU cores. We then combine the files containing the logs and those containing the resource measurements into one aggregated file. For seven test cases, our dataset includes fewer than 30 executions, as some executions concluded before a measurement could be taken. From the total of 232 test cases, 23 show multiple test verdicts and are therefore considered flaky. These tests are from five projects, with project `SlimHUD` contributing the highest number of flaky tests, with eight such tests. We manually categorize these flaky tests according to the classifications of flaky UI tests provided by Romano et al. [16]. We identify *Environment* and *Test Runner API Issue* as the most common root causes. In our dataset, 10 flaky tests result from *Environment*, split between *Platform Issue* (6 tests) and *Layout Difference* (4 tests). An additional 13 flaky tests result from *Test Runner API Issue*, split between *DOM Selector Issue* (7 tests) and *Incorrect Test Runner Interaction* (6 tests).

[6] https://sourcegraph.com
[7] https://developer.apple.com/documentation/xctest

4 EVALUATION

Using the dataset obtained in Section 3, we examine the impact of reaching CPU limits on the flakiness of test executions. First, we present a formal framework for our evaluation in Section 4.1, and then we use it to address our research questions concerning the *executions of flaky vs. non-flaky tests* and *passes vs. failures of flaky tests* in Section 4.2 and Section 4.3, respectively.

4.1 Evaluation Framework

For our evaluation, a test execution t is a sample from a test case, modeled as a random variable T. Each test execution is a tuple $t = (r, l_{\text{CPU}})$, comprised of a test verdict $r \in \{P, F\}$ and the relative test execution time spent in CPU limits l_{CPU}. The relative time spent in CPU limits is calculated as the ratio of the time spent in CPU limits to the total execution time of the test. A value of $l_{\text{CPU}} = 0.5$, for example, indicates that the CPU limit was reached for half of the execution time. We consider the CPU limit to be reached when the CPU usage is at 100%. Following Kowalczyk et al., we use the entropy of the observed test verdicts $H_r(T)$ to measure a test's flakiness [10].

4.2 Executions of Flaky vs. Non-flaky Tests

The first research question aims to determine whether executions of flaky tests spend more time in CPU limits than executions of non-flaky tests. Using our framework from Section 4.1, we formalize our first research question. The expected value of the relative time spent in CPU limits of test case T is denoted by $\mathbb{E}_{l_{\text{CPU}}}[T]$.

$$\mathbb{E}_{l_{\text{CPU}}}[T] \overset{?}{>} \mathbb{E}_{l_{\text{CPU}}}[T'] \qquad \text{for } H_r(T) > 0 \wedge H_r(T') = 0$$

For each test execution t, we calculate the relative time spent in CPU limits, l_{CPU}. We then partition the executions into two groups: executions of flaky tests (with $H_r(T) > 0$) and executions of non-flaky tests (with $H_r(T) = 0$).

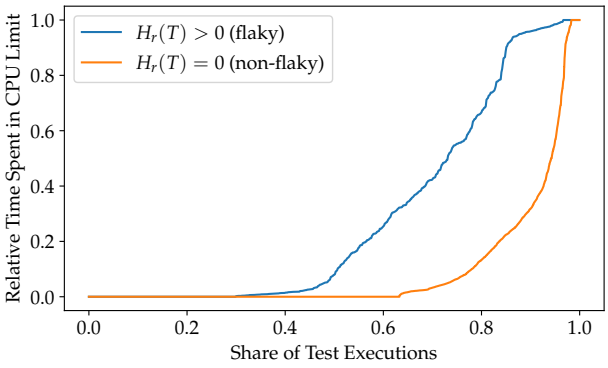

Figure 1: Flaky vs. non-flaky tests: cumulative distribution of the relative time spent in CPU limits.

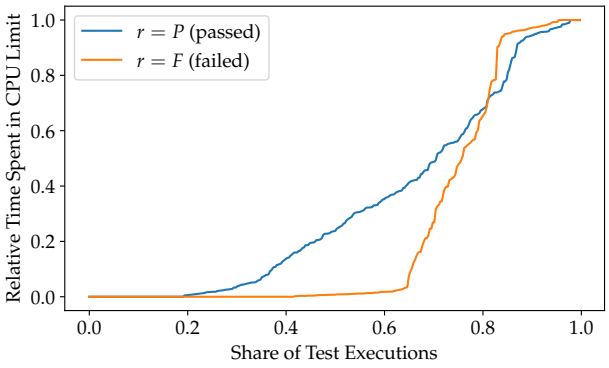

Figure 2: Passes vs. failures of flaky tests: cumulative distribution of the relative time spent in CPU limits

Figure 1 shows the cumulative distribution of the relative time spent in CPU limits for both groups. This plot can be interpreted as follows: for a given value x on the x-axis, the corresponding value y on the y-axis indicates that x% of the executions of the corresponding group spent less than y time in the CPU limit. For instance, 75% of the executions of non-flaky tests spent less than 7% of their execution time in CPU limits. For flaky tests, this value is 56%. Visually, the cumulative distribution of the relative time spent in CPU limits for flaky tests is higher than for non-flaky tests. This impression can be put into numbers by calculating the integral of the cumulative distribution function, intuitively speaking, the area under the curve. This is 0.29 for flaky tests and 0.09 for non-flaky tests.

The visual impression of flaky tests spending relatively more time in CPU limits than non-flaky tests is supported by the statistical measurements: For the cumulative function at hand, the integral corresponds to the mean of the relative time spent in CPU limits for the corresponding group. Thus, from our observations, the expected value of the time spent in CPU limits for a flaky test is its mean, 0.29, while it is 0.09 for a non-flaky test.

$$\mathbb{E}_{l_{\text{CPU}}}[T] > \mathbb{E}_{l_{\text{CPU}}}[T'] \qquad \text{for } H_r(T) > 0 \wedge H_r(T') = 0$$
$$0.29 > 0.09$$

We conduct a Mann-Whitney U test, given the non-parametric data, to assess statistical significance. For this test, the null hypothesis asserts no significant difference exists between the two dataset groups. The test yielded a u-statistic of $2.9 \cdot 10^6$ and a p-value of $5.2 \cdot 10^{-77}$, which is far below the commonly accepted threshold $\alpha = 0.05$. This allows us to confidently reject the null hypothesis, suggesting a significant difference in the relative time spent in CPU limits between flaky and non-flaky tests. Flaky tests spent more time in CPU limits than non-flaky tests in our dataset.

RQ1 (Executions of Flaky vs. Non-flaky Tests): In the dataset at hand, executions of flaky tests spent relative to their execution time significantly more time in CPU limits than executions of non-flaky tests.

4.3 Passes vs. Failures of Flaky Tests

The second research question explores the relation between passed and failed executions of flaky tests and reaching CPU limits. In this section, we only consider executions of flaky tests. Once again, we use our framework from Section 4.1 to formalize our second research question. The expected value of the relative time spent in CPU limits of test case T considering only failed executions is denoted by $\mathbb{E}_{l_{\text{CPU}}|r=F}[T]$. The expected value considering only passed executions is accordingly denoted by $\mathbb{E}_{l_{\text{CPU}}|r=P}[T]$.

$$\mathbb{E}_{l_{\text{CPU}}|r=F}[T] \overset{?}{>} \mathbb{E}_{l_{\text{CPU}}|r=P}[T] \qquad \text{for } H_r(T) > 0$$

We calculate the relative time spent in CPU limits l_{CPU} for each test execution t and split the results into two groups: passed and failed executions.

Figure 2 shows the cumulative distribution of the relative time spent in CPU limits for failed and passed runs of flaky tests. The integrals of the cumulative distribution functions for failed and passed runs are 0.25 and 0.34, respectively. Again, the integrals correspond to the means of the relative time spent in CPU limits. Thus, from our observations, the expected value of the time spent in CPU limits for a failed run of a flaky test is 0.25, while it is 0.34 for a passed run.

$$\mathbb{E}_{l_{\text{CPU}}|r=F}[T] \not> \mathbb{E}_{l_{\text{CPU}}|r=P}[T] \qquad \text{for } H_r(T) > 0$$
$$0.25 \not> 0.34$$

To test for significance, we once more employ a Mann-Whitney U test with the null hypothesis stating that there is no significant difference between the two groups in the dataset. The Mann-Whitney U test yielded a u-statistic of $7.5 \cdot 10^4$ and a p-value of $1.8 \cdot 10^{-9}$ indicating a significant difference between the two groups.

RQ2 (Passes vs. Failures of Flaky Tests): In our dataset, failed test executions of flaky tests spent relatively less time in CPU limits than passed ones.

4.4 Discussion

The underlying hypothesis of our study is that reaching CPU limits fosters timing-related flakiness. In our study, we find indications that both support and contradict this hypothesis. When addressing RQ1, we find that executions of flaky tests spent significantly more time in CPU limits than executions of non-flaky tests, thereby supporting our hypothesis. Conversely, in RQ2 considering only executions of flaky tests, we find that passed runs spent more time in CPU limits than failed runs.

The positive result of RQ1 (Executions of Flaky vs. Non-flaky Tests) might not result from a *causal* relationship between reaching CPU limits and flakiness. According to previous research [24], large tests are more likely to be flaky (where *large* might refer to binary size or RAM usage). It is reasonable to assume that large tests also spend more time in CPU limits. Therefore, the findings of RQ1 might result from a *correlation* between reaching CPU limits and the flakiness of the tests, mediated by the hidden state of the test's size.

The negative results of RQ2 (Passes vs. Failures of Flaky Tests) could potentially indicate that our hypothesis is incorrect. Indeed, we did not find quantitative explanations for this result. We initially assumed that some failed runs of flaky tests might fail directly after starting due to some unmet precondition. However, the data do not support this assumption.

Drawing from the findings of both RQs, we find evidence for both support and contradiction of our hypothesis. However, we consider the results of our study interesting and promising enough to warrant further research.

4.5 Threats to Validity

We identified several threats to the validity and generalizability of the findings from this study, and have taken steps to minimize these threats where possible. Firstly, our dataset is relatively small, consisting of 232 test cases, of which 23 are flaky. Although we work with test *executions* rather than test *cases*, giving us larger numbers from which we can draw more significant conclusions, adding more executions of the same test case does not necessarily increase the variance and generalizability of our findings. Like all empirical studies, our results are only valid for the projects we have selected. Secondly, we rely on the boolean decision of whether a test is flaky or not. This simplifies the problem, as tests might be flaky to a certain degree. Also, using rerunning to determine flakiness could potentially lead to some tests being misclassified as non-flaky [25]. Thirdly, our tool only records resource usage every 100ms, so we may miss spikes in resource usage. However, we have observed the CPU usage to be relatively stable over multiple iterations of our tool. Fourthly, our tool may affect the test execution and thereby influence the results. However, as numerous processes run on CI servers, we believe our tool merely adds noise to the system that could equally be caused by any other process running on CI servers. Lastly, the implementation of our tool or analysis scripts may have bugs. All crucial parts were implemented or reviewed by at least two authors of this study. We also publish our code alongside this paper, allowing it to be independently verified.

5 FUTURE RESEARCH DIRECTIONS

We presented our study evaluating the impact of reaching CPU limits on the flakiness of test executions, and we discussed the significance of our results. To further strengthen the conclusions, four concrete areas can be addressed in future research:

Study Size: To improve the statistical significance of our results, more projects from a wider range of domains can be included in the study. To improve the validity when differentiating flaky from non-flaky tests, the number of reruns per test should be increased.

Additional Resource Types: As reaching limits other than CPU limits can also impact test flakiness, future research should comprise additional resource types. Although RAM, disk and network usage are currently recorded and included in the dataset, these aspects have been excluded from our study. The reason for this exclusion was the absence of meaningful conclusions from the comparatively sparse data available. RAM usage (not reaching limits) showed a correlation with flakiness, however, we suspect this is again due to a correlation with the size of the tests.

Lighter Tooling: Although our tool's mean overhead of 14.2% should not significantly impact test execution, the overhead should still be reduced for further research. We hypothesize the overhead is primarily due to the node.js runtime. Transitioning the implementation language of the tool to a compiled language or a more lightweight runtime should significantly reduce the overhead.

Integration Into Actual CI Processes: To achieve the highest possible impact in practice, we strive to emulate "real-world" CI processes as closely as possible. We accomplish this by using the same infrastructure services as the projects in our study, namely GitHub actions. This could be further enhanced by measuring the resource usage within the actual CI processes. Even though our tool is implemented as a GitHub action, which makes this technically feasible, we anticipate difficulties convincing projects to integrate our tool into their CI processes. This research direction seems especially promising for industry research collaborations.

6 CONCLUSION

Flaky tests pose a considerable challenge in regression testing. Prior research suggests that timing-related issues are the predominant causes of flaky tests. We hypothesize that timing-related flakiness is exacerbated when test execution reaches resource limits. To examine this hypothesis, we execute a study involving a total of 232 test cases from 20 open-source projects. We ran each test case 30 times and logged the verdicts and resource usage for each execution. From the 232 cases, 23 exhibit multiple test verdicts and are, therefore, classified as flaky. Our findings reveal that flaky test executions spend more time in CPU limits than non-flaky test executions. However, only considering flaky test executions, we discover that failed test executions spend less time in resource limits than passed ones. These findings offer contradictory results towards our hypothesis that reaching CPU resource limits during test execution promotes timing-related flakiness.

In order to formulate clearer conclusions, we outline future research directions. In adherence to open research practices, we make our dataset and GitHub action to record resource usage publicly available to enable other researchers to conduct related investigations.

REFERENCES

[1] H.K.N. Leung and L. White. 1989. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance - 1989.* IEEE Comput. Soc. Press, Miami, FL, USA. DOI: 10.1109/ICSM.1989.65194.

[2] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering*. IEEE, Berlin, Germany, (Oct. 2019). DOI: 10.1109/ISSRE.2019.00031.

[3] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation*. (Sept. 2018). DOI: 10.1109/SCAM.2018.00009.

[4] Morena Barboni, Antonia Bertolino, and Guglielmo De Angelis. 2021. What We Talk About When We Talk About Software Test Flakiness. In *Quality of Information and Communications Technology*. Vol. 1439. Springer International Publishing, Cham. DOI: 10.1007/978-3-030-85347-1_3.

[5] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, (Nov. 2014). DOI: 10.1145/2635868.2635920.

[6] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. A Survey of Flaky Tests. *ACM Transactions on Software Engineering and Methodology*, 31, 1, (Jan. 2022). DOI: 10.1145/3476105.

[7] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, (June 2020). DOI: 10.1145/3377811.3381749.

[8] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Beijing China, (July 10, 2019). DOI: 10.1145/3293882.3330570.

[9] John Micco. The State of Continuous Integration Testing @Google. (2017). Retrieved Oct. 24, 2022 from https://storage.googleapis.com/pub-tools-public-publication-data/pdf/45880.pdf.

[10] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, Seoul South Korea, (June 27, 2020). DOI: 10.1145/3377813.3381370.

[11] John Ahlgren et al. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE, Madrid, ES, (May 2021). DOI: 10.1109/ICSE-SEIP52600.2021.00023.

[12] Fabian Leinen, Daniel Elsner, Alexander Pretschner, Andreas Stahlbauer, Michael Sailer, and Elmar Jürgens. 2024. Cost of Flaky Tests in Continuous Integration: An Industrial Case Study. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 327–338.

[13] N. Hashemi, A. Tahir, and S. Rasheed. 2022. An Empirical Study of Flaky Tests in JavaScript. In *2022 IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Los Alamitos, CA, USA, (Oct. 2022). DOI: 10.1109/ICSME55016.2022.00011.

[14] Yu Pei, Sarra Habchi, Renaud Rwemalika, Jeongju Sohn, and Mike Papadakis. [n. d.] An empirical study of async wait flakiness in front-end testing.

[15] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *2018 IEEE International Conference on Software Maintenance and Evolution*. (Sept. 2018). DOI: 10.1109/ICSME.2018.00062.

[16] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, (May 2021). DOI: 10.1109/ICSE43902.2021.00141.

[17] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. 2023. The Effects of Computational Resources on Flaky Tests. (Oct. 18, 2023). arXiv: 2310.12132 [cs]. Retrieved Oct. 24, 2023 from. preprint.

[18] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice*. DOI: 10.1145/3510457.3513037.

[19] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2022. A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests. In *2022 IEEE Conference on Software Testing, Verification and Validation*. IEEE, Valencia, Spain, (Apr. 2022). DOI: 10.1109/ICST53961.2022.00034.

[20] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Adelaide, Australia, (Sept. 2020). DOI: 10.1109/ICSME46990.2020.00037.

[21] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. 2020. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, Seoul South Korea, (June 27, 2020). DOI: 10.1145/3377816.3381742.

[22] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn Stolee. 2019. Wait, Wait. No, Tell Me. Analyzing Selenium Configuration Effects on Test Flakiness. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test*. IEEE, (May 2019). DOI: 10.1109/AST.2019.000-1.

[23] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil Mcminn. 2022. What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection? *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*. DOI: 10.1145/3524481.3527227.

[24] John Micco. 2016. Flaky Tests at Google and How We Mitigate Them. (May 2016). Retrieved Dec. 6, 2023 from.

[25] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, Madrid, ES, (May 2021). DOI: 10.1109/ICSE43902.2021.00140.