Evaluating Benchmark Quality: a Mutation-Testing-Based Methodology

Federico Lochbaum[1], Guillermo Polito[1]

[1]University of Lille, Inria, Centrale Lille, France

Abstract

Performance benchmarking is a crucial tool for evaluating software efficiency. Unlike behavioral tests, where Mutation testing and Test Coverage propose metrics to measure test quality, there are no methodologies for evaluating the quality of benchmarks. Coverage provides insights into execution but does not necessarily correlate with performance bugs. In this paper, we propose to assess the effectiveness of benchmarks by measuring their capacity to find performance issues. We explore a methodology that evaluates the quality of benchmarks based on mutation testing, where artificial performance bugs are introduced into programs and the benchmark's ability to detect them is measured. We present a series of experiments where we measure the sensitivity of benchmarks to catch artificially introduced bugs, providing a systematic approach to validate their effectiveness in finding performance issues. We introduce a deeper understanding of benchmark quality and offer insights into improving benchmark measuring.

Keywords

Test Cases, Benchmark Quality, Performance Evaluation, Code Coverage, Mutation Testing

1. Introduction

Performance evaluation is crucial to understand resource consumption both in academic and industrial settings [1, 2, 3, 4, 5]. Such evaluations are often made by means of benchmarking, i.e., systematic measurement and analysis [1, 6, 7, 8]. Benchmarking is commonly related to speed benchmarking, to analyse the time taken to perform a task. However, benchmarking techniques can be used to evaluate other metrics such as energy consumption [9] or memory usage [10, 11]. Benchmarks are typically written as collections of programs called *benchmark suites* that exercise the application under evaluation to detect performance variations.

Unlike functional testing, which has well-established methodologies for evaluating test quality, benchmarking lacks systematic methodologies to assess its effectiveness. A benchmark provides performance metrics, but how well it detects performance issues remains unclear. Existing work proposes methodologies to design and select benchmark programs [12, 13] or statistical frameworks to get precise results [14]. However, to the best of our knowledge, no works have explored the idea of evaluating benchmark quality.

Traditionally, test quality metrics such as test coverage and mutation testing have been used in software testing to evaluate test behavior. Test coverage measures the extent to which a test suite exercises a program [15]. On the other side, mutation testing assesses test effectiveness by introducing controlled modifications and observing whether tests detect them [16]. However, these techniques primarily focus on correctness rather than performance, leading to the question of whether similar methodologies are adapted to assess performance benchmark quality.

In this paper, we introduce a general methodology to measure benchmark performance quality. For this purpose, we explore a mutation-testing-based approach to evaluate benchmark effectiveness. Traditional mutation testing introduces controlled bugs in a program, and leverages the *autovalidating* property of tests to assess if a mutation was detected or not. Our approach adapts mutation testing to performance benchmarks with two ideas. First, we introduce performance mutation operators that

IWST 2025: International Workshop on Smalltalk Technologies, July 1–4, 2025, Gdansk, Poland

*Corresponding author.

†These authors contributed equally.

introduce artificial performance degradations. Second, we introduce a performance-specific oracle. and measure a benchmark's sensitivity to these changes. On top of these, leveraging mutation testing principles, we define a measure to quantify how well benchmarks detect performance regressions.

2. Motivation

Although any program can be used as a benchmark program, writing *effective* benchmarks remains complex under the current state of the art.

High-costs incurred by benchmarking. Sound benchmarking is challenging because it requires the selection of representative benchmarks, the study of varying workloads, and statistical rigourosity to cope with non-determinism [17, 18, 19]. This requires a combination of application-specific knowledge, statistical knowledge, and system design knowledge [20] and is often found as a barrier to wider adoption [3, 4]. On the one hand, benchmarks should be representative of the execution of the application in normal conditions. On the other hand, benchmark results should consider the noise that is inherent in existing hardware and software systems. There is an agreement in the community on the need for tooling to reason about performance and automate the detection of regressions [7, 5].

Non-representative benchmark programs waste resources. The choice of benchmark programs is crucial in performance analysis because they guide, and may misguide, optimisation decisions. To avoid wasting resources, benchmark programs should be *representative* of normal application executions. This is particularly a problem of microbenchmarks [21, 4] and synthetic benchmarks [22, 23]. Obtaining **representative benchmark programs requires a deep understanding of the application, the programming language, its implementation and the underlying hardware**. We find an example in the programming language community. The DaCapo benchmarks [18] proposed in 2006 a suite of programs representing typical Java applications. This suite inspired others for Scala [24], Javascript, and WebAssembly [25, 26]. Although currently in use by language implementation researchers, these suites are still *not considered representative* of realistic executions, producing biases in research results. The representativity problem affects even modern just-in-time compilers and production-ready garbage collectors supported by strong companies such as Google and Oracle. State-of-the-art Virtual Machines such as V8 deprecated the usage of synthetic benchmarks in favour of real applications [27]. Recently, JVM's garbage collectors in production since 2018 (Shenandoah and ZGC) have been observed to suffer performance losses on realistic workloads [28].

Benchmark datasets count as much as benchmark programs. The representativity of benchmarks depends not only on the chosen programs: application behaviour and performance vary depending on their workload *i.e.,* the size and shape of their inputs. Regarding their size, small datasets put less pressure on memory accesses while large datasets execute for longer times and unveil more profile-guided optimisation opportunities to modern JIT compilers. In addition, data that varies a lot in the found data types may prevent optimisations such as procedure inlinings. Benchmark programs thus need to execute on different dataset configurations to detect these variations, since performance regressions (and improvements) may be present on certain workloads and not on others.

Non-determinisms demand complex methodologies. Benchmarks also suffer from non-determinisms arising from hardware, operating systems, and programming language implementations [29, 30, 21, 31, 32, 33, 34], commonly referred to as *noise*. This problem gets worsened because benchmark results are mostly based on unstable metrics: metrics that are subject to noise and vary from one measure to another, such as *e.g.,* wall-clock time. Two wall-clock time measurements for the same program will vary depending on factors such as garbage collection, thread/process scheduling and even CPU temperature. This raises major issues with the reproducibility and statistical significance of the results. Nowadays, such instabilities are approached by methodological means [35, 36, 17, 19].

Notably, Georges et al. [37] proposed in 2007 a statistically rigorous methodology based on (a) improving statistical significance with repeated executions and (b) automatically determining when a benchmark reaches a steady state by analyzing the variation of measurements. Recently [38, 39] the language implementation community discovered that such methodologies were based on the *false assumption* that benchmarks always stabilize [40].

Goal: How can we measure benchmark quality? We aim at identifying the most suitable benchmarks for a given problem while minimising exploration time and maximising reachability. The absence of methodologies for evaluating benchmark quality presents a significant challenge to assessing benchmark effectiveness. Without a structured approach to measuring benchmark effectiveness, performance testing remains ad hoc and potentially unreliable. Supposing we introduce a well-known bug that significantly reduces the performance of a system, we can't guarantee that our ad hoc test cases will find it. Having a methodology for evaluating benchmarks, we can correlate them with their capacity to detect potential performance issues.

In this paper, we propose a methodology for evaluating the effectiveness of benchmarks in detecting performance bugs. Instead of focusing on isolated case studies or specific performance scenarios, we aim to develop a structured evaluation framework that is applied across different benchmarks and software test suites. Moreover, we intend this methodology to be malleable enough to be adapted for any set of benchmarks.

We contribute to a more rigorous understanding of benchmarking quality and provide insights into improving benchmarking methodologies for performance software.

3. Challenges Defining a Benchmark Evaluation Methodology

This section presents the main challenges to establishing a benchmark evaluation methodology. The first of it is to define what benchmark quality is and select which benchmark properties we want to assess. But also, we need to define an oracle that says when a benchmark detects or does not detect a performance issue.

3.1. Mutation Testing and Test Quality

Mutation testing [41] is a technique for evaluating the quality of a test suite. It introduces mutations to a target program and validates whether the existing tests can detect these mutants. If a test suite fails when run against a mutant, the mutant is said to be killed, otherwise, it survives. The goal is to simulate common programming errors and assess if the test suite is robust enough to catch them.

In the past forty years, mutation testing has been widely adopted in industry because of the advances in computing performance. This increased the interest of the research community that studied new methods to improve mutation performance, its applications to a large number of programming languages and frameworks, and even its usage for topics like security [42].

3.2. Performance Mutation Testing

Our proposed methodology is illustrated in Figure 1. It is structured as a mechanism where a benchmark is executed twice per evaluation. The first one, with the base application and this execution, is going ot be used for the oracle as the assert baseline. The second one will be forced to run over a mutated application where it should catch a performance perturbation. The oracle will use both execution to define how good the benchmark behaves.

Benchmark quality. We propose to treat each benchmark as a test case, where it is evaluated against a target application. A benchmark usually provides performance metrics as a result, as it is *execution time*, but there is no linear relationship between those metrics and performance issues. The nature of benchmarks does not fit with finding bugs but rather with measuring performance. This is because,

even if the benchmark can identify a performance problem, it cannot tell which part of the code it comes from. We propose to use the elapsed time of test case execution concerning a defined baseline as our metric.

Performance bug introduction. We augment *mutation testing* by introducing artificial performance bugs and validating how many of these mutants are detected. Mainly, we want to use a benchmark as a black-box program, accepting any benchmark for our methodology. Then, we modify the benchmark program input by introducing controlled mutants that degrade significantly the program's performance [43]. This means that if the benchmark is capable of catching the performance bug, running it with a performance-mutated input must have an output measure of time worse than without the mutation.

Oracle. To verify whether the current benchmark successfully detected a performance perturbation, we suggest using an oracle. An oracle knows how to compare the execution time result of the benchmark run with a defined baseline. The considered baseline that we propose to use is the result of executing the same benchmark with a non-perturbed input. Besides the baseline, the oracle establishes a reliability threshold to define when, given the benchmark result, it considers the benchmark detects the mutant.

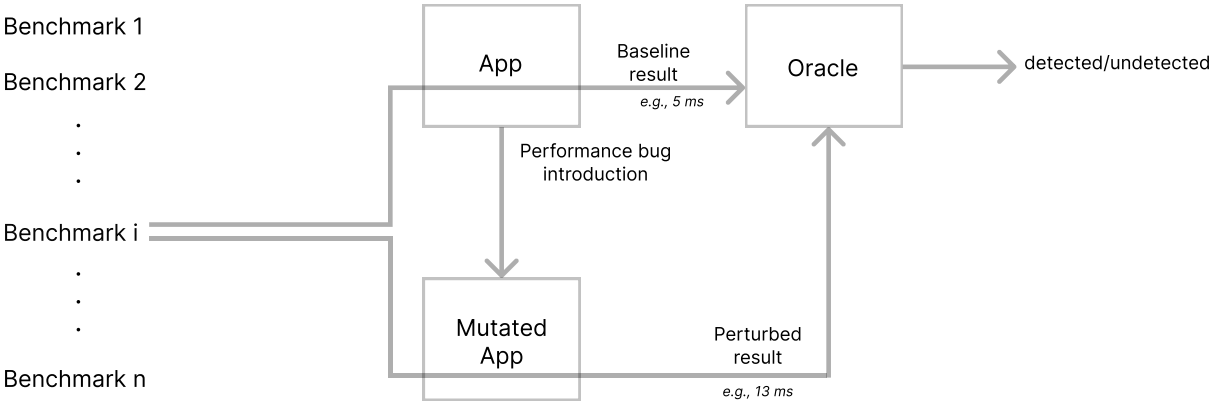

Figure 1: Proposed benchmark evaluation methodology.

3.3. Methodology Requirements

To guide our methodology in our experiments and to assess the effectiveness of benchmarks in detecting performance bugs, we seek to explore the following research questions:

RQ1: Representative performance issues. What represents a potential performance issue or bug?

RQ2: Artificial bugs. Can we introduce artificial performance issues to assess benchmark quality?

RQ3: Measure sensibility. What is the degree of confidence in these measures?

4. Experimentation

Our objective is to provide an example and instantiate this methodology with a real use case. In this section, we instantiate our methodology that we have presented before using mutants that introduce performance perturbations with sleep statements, and an oracle that determines performance variations by comparing run-time averages. To evaluate this concrete setup, we analysed a set of autogenerated benchmarks for the Pharo regular expressions library.

4.1. Experimentation details

Use case: Regexp. We selected as a use case the validation of a set of benchmarks for regular expressions. This stresses the implementation of the *matches:* method. Benchmarks are generated using a fuzzer strategy.

Benchmarks generation with Monte Carlo tree search. We use a Monte Carlo tree search grammar fuzzer over a regular expression grammar to generate benchmark tests. We implemented a fuzzer to generate regexes [44]. We guide the fuzzer by the execution time taken for the generated regex to perform the *matches:* method with the minimal acceptable input.

4.2. Instantiating methodology

Mutants: sleep statements. We use Mutalk [1], a *Mutation Testing framework* for Pharo to introduce artificial performance bugs. We define a series of performance mutation operators that introduce sleep statements in every program sequence node *e.g.,* `n milliseconds wait`. Each operator generates a mutant at the beginning of each sequence node. This way, not only do methods contain performance perturbations, but also control flow statements, and more specifically, loops.

Oracle: Comparing Runtime Means. Our oracle uses a baseline to compare each benchmark result, once for each applied mutation. For each benchmark, we compare its run time with and without mutant perturbations. To cope with the noise and non-determinism of performance measurements, we compute baseline values by running a benchmark multiple times and considering the average and standard deviation. For these experiments, we compute the baseline with 30 iterations. We consider that a mutant is killed (*i.e.,* that the performance perturbation is detected) if the run time of the perturbed benchmark lies within the average and one standard deviation of the baseline.

4.3. Results

As a first approach using this methodology, we executed **100** benchmarks. Each benchmark is configured to run *n* iterations, where *n* forces the baseline to run at least 300 ms, thus minimising noise. We introduce **62** mutants per mutant operator, and we execute every benchmark once per mutant.

Figure 2 presents the preliminary results of these experiments using a *dispersion chart.* The dispersion chart allows us to understand the mutation score detection distribution per benchmark and operator. Figure 3 presents the stacked values between the average and the standard deviation for each benchmark baseline. Table 1 shows the ratio between the stdevs and averages of every benchmark baseline.

Statistics	Value
Average	0.4284428074
Max	1.0785776
Min	0.2107902808
Q1	0.3633576572
Q2	0.3861501
Q3	0.4055622512

Table 1
Ratio between the standard deviation and the average

[1] https://github.com/pharo-contributions/mutalk

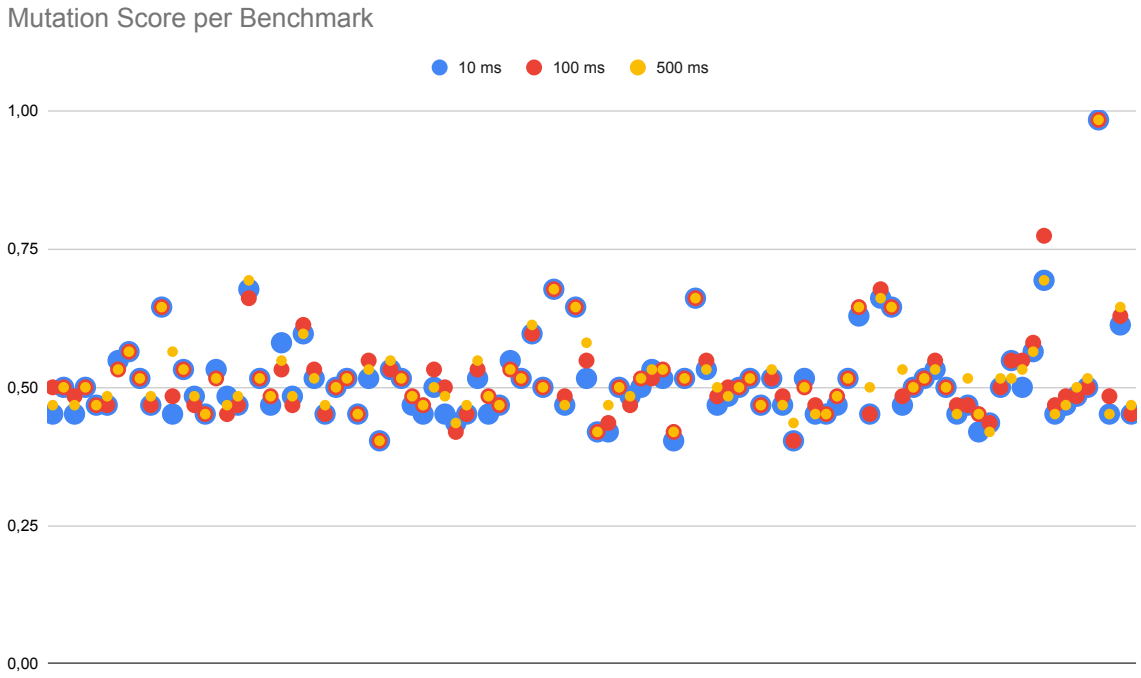

Figure 2: Dispersion chart of Benchmark effectiveness.

4.4. Analysis

At first glance, we can observe in Figure 2 that there is no huge difference between the points of different operators. This led us to think that small perturbations work almost as well as huge perturbations do. In only 23 benchmarks, 500 ms perturbation reaches better results with an average difference of 2,6% compared to 10 ms perturbation. We also observe that on average, the mutation score per benchmark is above 50% having highlights with high mutation scores, getting almost 100%.

Moreover, we observe in Figure 3 and Table 1 that the standard deviations of the benchmark baseline, on average, do not exceed 50% of the average, having a maximum ratio of 107% of the average and a minimum of 21%. However, we observe that even in the third quarter, benchmarks do not exceed 40% of the baseline average, which makes the sample reliable.

5. Lessons Learned and Future Work

Initially, we expected to see a large difference in the mutation score for mutant operators with huge perturbations. But we noted that small perturbations work as well as huge perturbations.

Test variance. We observed that some benchmark test execution times have a big standard deviation. Therefore, we cannot use them for stable measurements, and we need to filter them in a previous step. In this sense, we need to study better benchmark selection techniques that allow us to reduce the number of benchmarks that introduce noise in our experiment samples.

Small Benchmarks. Having very low test execution times makes them too sensitive to external disturbances. For this reason, it is necessary to consider a minimum noise tolerance to perform tests with very short execution times.

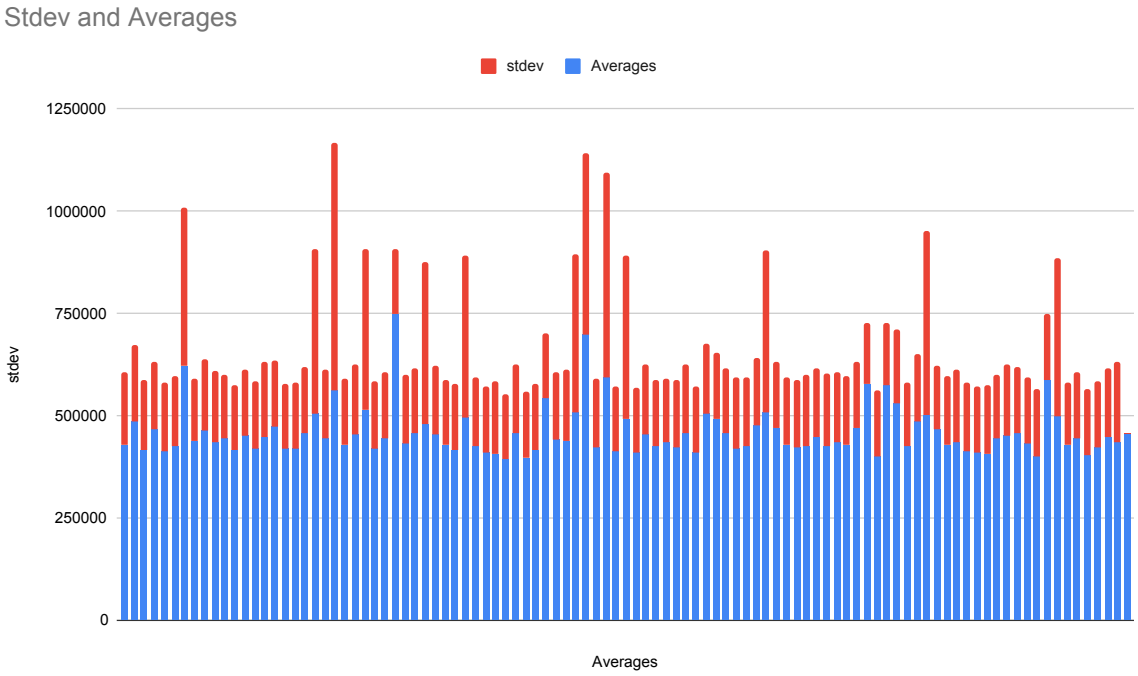

Figure 3: Stacked column chart between stdevs and averages.

Runtime perturbation, Garbage Collector and JIT Compilation. A recognisable noise cause is the Garbage Collector executions. The garbage collector is not a disableable feature. This means we need to guarantee somehow that the garbage collector won't introduce performance noise during our experimentations. For example, we need to study techniques that ensure the Virtual Machine guarantees executing benchmarks under the same conditions minimizing external noise.

Elapsed time as quality metric. Using the wall-clock time as a performance metric is very unstable and requires so many executions to stabilise it. This leads us to carry out executions that take a long time and are therefore expensive. Following this, we need to study alternative metric techniques, as time estimation based on static metrics, like the number of messages sent or memory accesses.

6. Conclusion

In this paper, we propose a systematic methodology to evaluate the effectiveness of performance benchmarks. We propose a method to introduce artificial performance bugs by extending mutation testing. We use an oracle to assess the effectiveness of a benchmark using a baseline generated. Finally, we define a benchmark quality and we instantiated the framework in a real setting. We present some preliminary results of the experiment, and we perform an analysis to suggest future improvements.

The results demonstrate that the proposed methodology provides enough information to compare benchmark effectiveness. However, we find limitations regarding the measure used to define a mutant kill. Either we need to stabilise the baseline by running each benchmark a higher number of times, or we need to approach the omission of false negatives/positives from performance noise.

Another interesting approach could be to study other metrics that allow us to measure the effectiveness of a benchmark with higher precision, or to propose another mutant kill detection technique.

Acknowledgments

This project is financed by the ANR JCJC project convention ANR-25-CE25-0002-01.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] S. Zaman, B. Adams, A. E. Hassan, A qualitative study on performance bugs, in: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12, IEEE Press, 2012, pp. 199–208.

[2] Q. Jiang, X. Peng, H. Wang, Z. Xing, W. Zhao, Summarizing evolutionary trajectory by grouping and aggregating relevant code changes, in: International Conference on Software Analysis, Evolution, and Reengineering, 2015.

[3] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, F. Willnecker, How is performance addressed in devops?, in: ACM/SPEC International Conference on Performance Engineering, ICPE '19, Association for Computing Machinery, 2019. URL: https://doi.org/10.1145/3297663.3309672. doi:10.1145/3297663.3309672.

[4] P. Leitner, C.-P. Bezemer, An exploratory study of the state of practice of performance testing in java-based open source projects, in: International Conference on Performance Engineering, ICPE '17, 2017. URL: https://doi.org/10.1145/3030207.3030213. doi:10.1145/3030207.3030213.

[5] P. Stefan, V. Horky, L. Bulej, P. Tuma, Unit testing performance in java projects: Are we there yet?, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 401–412. URL: https://doi.org/10.1145/3030207.3030226. doi:10.1145/3030207.3030226.

[6] M.-H. Lim, J.-G. Lou, H. Zhang, Q. Fu, A. B. J. Teoh, Q. Lin, R. Ding, D. Zhang, Identifying recurrent and unknown performance issues, in: 2014 IEEE International Conference on Data Mining, 2014, pp. 320–329. doi:10.1109/ICDM.2014.96.

[7] A. Nistor, T. Jiang, L. Tan, Discovering, reporting, and fixing performance bugs, in: 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 237–246. doi:10.1109/MSR.2013.6624035.

[8] V. Horký, P. Libič, L. Marek, A. Steinhauser, P. Tůma, Utilizing performance unit tests to increase performance awareness, in: International Conference on Performance Engineering, ICPE '15, 2015. URL: https://doi.org/10.1145/2668930.2688051. doi:10.1145/2668930.2688051.

[9] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat, Evaluating the Impact of Java Virtual Machines on Energy Consumption, in: 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Bari, Italy, 2021. URL: https://hal.inria.fr/hal-03275286.

[10] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, G. Portokalidis, Nibbler: Debloating binary shared libraries, in: Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 70–83. URL: https://doi.org/10.1145/3359789.3359823. doi:10.1145/3359789.3359823.

[11] G. Polito, L. Fabresse, N. Bouraqadi, S. Ducasse, Run-fail-grow: Creating tailored object-oriented runtimes, The Journal of Object Technology 16 (2017) 2:1–36. URL: https://hal.archives-ouvertes.fr/hal-01609295. doi:10.5381/jot.2017.16.3.a2.

[12] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al., The dacapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, 2006, pp. 169–190.

[13] S. Marr, B. Daloze, H. Mössenböck, Cross-language compiler benchmarking: are we fast yet?, ACM SIGPLAN Notices 52 (2016) 120–131.
[14] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, ACM SIGPLAN Notices 42 (2007) 57–76.
[15] M. Böhme, L. Szekeres, J. Metzman, On the reliability of coverage-based fuzzer benchmarking, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1621–1633.
[16] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (1978) 34–41.
[17] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, B. Wiedermann, Wake up and smell the coffee: Evaluation methodology for the 21st century, Commun. ACM 51 (2008). URL: https://doi.org/10.1145/1378704.1378723. doi:10.1145/1378704.1378723.
[18] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo Benchmarks: Java Benchmarking Development and Analysis, in: Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '06, Association for Computing Machinery, New York, NY, USA, 2006, pp. 169–190. URL: https://doi.org/10.1145/1167473.1167488. doi:10.1145/1167473.1167488.
[19] E. Weyuker, F. Vokolos, Experience with performance testing of software systems: issues, an approach, and case study, IEEE Transactions on Software Engineering 26 (2000) 1147–1156. doi:10.1109/32.888628.
[20] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, P. Cao, How to build a benchmark, in: ICPE'15, ACM, 2015. URL: https://doi.org/10.1145/2668930.2688819. doi:10.1145/2668930.2688819.
[21] C. Laaber, P. Leitner, An evaluation of open-source software microbenchmark suites for continuous performance assessment, in: International Conference on Mining Software Repositories, MSR '18, 2018. URL: https://doi.org/10.1145/3196398.3196407. doi:10.1145/3196398.3196407.
[22] A. Sarimbekov, L. Stadler, L. Bulej, A. Sewe, A. Podzimek, Y. Zheng, W. Binder, Workload characterization of jvm languages, Software: Practice and Experience 46 (2016) 1053–1089. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2337. doi:https://doi.org/10.1002/spe.2337. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2337.
[23] T. Ogasawara, Workload characterization of server-side javascript, in: 2014 IEEE International Symposium on Workload Characterization (IISWC), 2014, pp. 13–21. doi:10.1109/IISWC.2014.6983035.
[24] A. Sewe, M. Mezini, A. Sarimbekov, W. Binder, Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine, in: Object Oriented Programming Systems Languages and Applications, OOPSLA '11, 2011. URL: https://doi.org/10.1145/2048066.2048118. doi:10.1145/2048066.2048118.
[25] F. Pizlo, Jetstream benchmark suite, ???? URL: https://browserbench.org/JetStream/, retrieved June 07 2022.
[26] S. Cazzulani, Octane: The javascript benchmark suite for the modern web, ???? URL: https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html, retrieved June 07 2022.
[27] V8ByeOctane, Retiring octane - https://v8.dev/blog/retiring-octane, ???? URL: https://v8.dev/blog/retiring-octane, retrieved June 07 2022.
[28] Z. Cai, S. M. Blackburn, M. D. Bond, M. Maas, Distilling the real cost of production garbage collectors, CoRR abs/2112.07880 (2021). URL: https://arxiv.org/abs/2112.07880. arXiv:2112.07880.
[29] D. Costa, C.-P. Bezemer, P. Leitner, A. Andrzejak, What's wrong with my benchmark results? studying bad practices in jmh benchmarks, IEEE Transactions on Software Engineering (2021). doi:10.1109/TSE.2019.2925345.
[30] C. Laaber, J. Scheuner, P. Leitner, Software microbenchmarking in the cloud. how bad is it really?,

Empirical Software Engineering 24 (2019) 2469–2508. URL: https://doi.org/10.1007/s10664-019-09681-1. doi:10.1007/s10664-019-09681-1.

[31] P. E. Nogueira, R. Matias, E. Vicente, An experimental study on execution time variation in computer experiments, in: ACM Symposium on Applied Computing, SAC '14, 2014. URL: https://doi.org/10.1145/2554850.2555022. doi:10.1145/2554850.2555022.

[32] D. Gu, C. Verbrugge, E. Gagnon, Code layout as a source of noise in jvm performance., Stud. Inform. Univ. 4 (2005) 83–99.

[33] A. Oliveira, J.-C. Petkovich, T. Reidemeister, S. Fischmeister, Datamill: Rigorous performance evaluation made easy, in: Proc. of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE), Prague, Czech Republic, 2013, pp. 137–149.

[34] J. Y. Gil, K. Lenz, Y. Shimron, A microbenchmark case study and lessons learned, in: Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, VMIL'11, SPLASH '11 Workshops, 2011, p. 297–308. URL: https://doi.org/10.1145/2095050.2095100. doi:10.1145/2095050.2095100.

[35] V. Horký, P. Libič, A. Steinhauser, P. Tůma, Dos and dont's of conducting performance measurements in java, in: ICPE'15, ACM, 2015, pp. 337–340. URL: https://doi.org/10.1145/2668930.2688820. doi:10.1145/2668930.2688820.

[36] C. Curtsinger, E. D. Berger, Stabilizer: Statistically sound performance evaluation, in: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 219–228. URL: https://doi.org/10.1145/2451116.2451141. doi:10.1145/2451116.2451141.

[37] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 57–76. URL: https://doi.org/10.1145/1297027.1297033. doi:10.1145/1297027.1297033.

[38] L. Traini, V. Cortellessa, D. Di Pompeo, M. Tucci, Towards effective assessment of steady state performance in java software: Are we there yet?, 2022. URL: https://arxiv.org/abs/2209.15369. doi:10.48550/ARXIV.2209.15369.

[39] E. Barrett, C. F. Bolz-Tereick, R. Killick, V. Knight, S. Mount, L. Tratt, Virtual machine warmup blows hot and cold, in: International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'17), ACM, 2017. doi:https://doi.org/10.1145/3133876.

[40] T. Kalibera, R. Jones, Rigorous benchmarking in reasonable time, in: International Symposium on Memory Management, ISMM '13, Association for Computing Machinery, 2013. URL: https://doi.org/10.1145/2491894.2464160. doi:10.1145/2491894.2464160.

[41] T. A. Budd, R. J. Lipton, R. A. DeMillo, F. G. Sayward, Mutation analysis, Yale University. Department of Computer Science, 1979.

[42] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, M. Harman, Chapter six - mutation testing advances: An analysis and survey, volume 112 of *Advances in Computers*, Elsevier, 2019, pp. 275–378. URL: https://www.sciencedirect.com/science/article/pii/S0065245818300305. doi:https://doi.org/10.1016/bs.adcom.2018.03.015.

[43] J. P. Sandoval Alcocer, A. Bergel, M. T. Valente, Learning from source code history to identify performance failures, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 37–48. URL: https://doi.org/10.1145/2851553.2851571. doi:10.1145/2851553.2851571.

[44] Z. Alsaeed, M. Young, Finding short slow inputs faster with grammar-based search, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 1068–1079.