Batch Execution of Microbenchmarks for Efficient Performance Testing

Mostafa Jangali*, Kundi Yao*, Yiming Tang†, Diego Elias Costa‡, and Weiyi Shang*
*Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
†Department of Software Engineering, Rochester Institute of Technology, USA
‡Computer Science and Software Engineering, Concordia University, Montreal, Canada
*{mjangali, kundi.yao, wshang}@uwaterloo.ca, †yxtvse@rit.edu, ‡diego.costa@concordia.ca

***Abstract*—Performance microbenchmarking is essential for ensuring software quality by providing granular insights into code efficiency. While automated performance microbenchmark generation tools (e.g., ju2jmh) are proposed to alleviate practitioners from manually curating microbenchmarks, the high volume of generated benchmarks can lead to protracted benchmarking execution time, as many of the generated benchmarks are too short in nature to be valuable for evaluating performance. In this paper, we present a novel approach that optimizes microbenchmark execution through a batching strategy, i.e., grouping benchmarks with similar code coverage and treating them as a single unit to 1) reduce execution overhead and 2) reduce the bias from microbenchmarks that are too short. We evaluate the effectiveness of this enhancement across various Java projects, comparing the execution times of clustered and individual microbenchmarks. Our findings demonstrate substantial improvements in execution efficiency, reducing execution time by up to 89.81% while preserving high microbenchmark stability.**

***Index Terms*—Software Performance, Performance Testing, Performance Microbenchmarking**

I. Introduction

In today's fast-paced digital world, where responsiveness and efficiency are paramount, ensuring optimal application performance is no longer a luxury but a necessity. Performance microbenchmarks are widely used to measure a software's performance at a granular level. However, due to their high granularity, the benchmarks are often short, leading to high overhead relative to their runtime. Especially in large projects, the sheer volume of microbenchmarks can create execution bottlenecks, delaying development cycles and increasing costs. These challenges are observed across many organizations and software projects, where lengthy performance test suites are a known issue [1].

Existing work has made a significant effort to ease the developers' workload in creating performance microbenchmarks, such as *ju2jmh* [2], which can automatically generate microbenchmarks from JUnit test suites. The introduction of *ju2jmh* fosters microbenchmarking in Java applications by incorporating the Java Microbenchmark Harness (JMH), which provides features to create and run performance microbenchmarks and enables developers to accurately measure the execution time of small units of code [3]–[5]. Creating and maintaining JMH benchmarks can be time-consuming and requires a deep understanding of both the system under test and the nuances of the JVM [6]. *ju2jmh* [2] mitigates this issue, but from effective test creation to execution, developers still need a robust solution to streamline the process.

To address this gap, we propose a batch execution-based strategy to enhance the efficiency of performance microbenchmarking. Our approach clusters benchmarks with similar code coverage to make batch execution feasible. This not only reduces execution overhead but also mitigates the inefficiency caused by excessively short benchmarks. Additionally, clustering benchmarks may provide deeper insights by exposing interactions among related code components that individual benchmarks could miss. Through this optimization, we aim to make performance testing faster, more reliable, and more accessible, enabling developers to integrate it seamlessly into their workflows.

To evaluate the effectiveness of our proposed approach, we compare it with individual benchmark execution across three large open-source projects: *RxJava*, *Eclipse-collections* and *ZipKin*. Our preliminary results show that batch execution could indeed reduce benchmarks' execution time, with savings ranging from 80.33% to 89.81%. While achieving high efficiency, batch execution does not significantly reduce the benchmark's stability, with only a slightly lower number of stable benchmarks compared to individual execution. This indicates that batch-executed microbenchmarks yield consistent results across repeated runs, thereby ensuring the reliability and robustness of performance evaluation. Our preliminary results demonstrate the promising feasibility of batch execution for microbenchmarks for efficient performance testing and could inspire developers to adopt batch execution to facilitate performance testing in real-world scenarios.

II. Study Overview

This section provides an overview of our study for batching microbenchmarks.

Figure 1 illustrates the overall workflow of our methodology. The process begins with preparing the microbenchmarking environment and providing microbenchmarks ready for batch execution from existing JMH microbenchmarks and conversions of JUnit test suites, with the latter implemented by *ju2jmh*.

The implementation of batch execution for microbenchmarks is based on the functional similarity identified between hand-crafted JMH microbenchmarks and those generated by

979-8-3315-0814-2/25 © 2025 IEEE

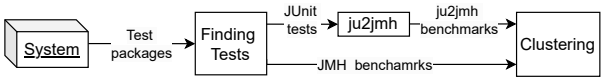

Fig. 1. Overview of our study methodology.

ju2jmh. Similar *ju2jmh* microbenchmarks are grouped into clusters, which are subsequently treated as a single, indivisible composite microbenchmark for performance testing.

A. Motivation

Performance testing is essential for identifying bottlenecks and optimizing software systems but often remains time-consuming and resource-intensive. Large performance test suites, common in industry projects, involve executing numerous microbenchmarks individually. While small microbenchmarks are precise and efficient for evaluating targeted functionality, their limited scope often fails to detect broader performance trends or anomalies [2]. Additionally, running small, isolated microbenchmarks incurs inefficiencies due to repeated overhead from initialization, execution, and result analysis. This repetitive processing consumes significant resources, making the testing process impractical, especially for large-scale microbenchmarking suites.

Existing tools like *ju2jmh* reduce developers' effort in generating microbenchmarks but do not address these inefficiencies or the broader limitations of small microbenchmarks. Our research proposes enhancing *ju2jmh* microbenchmarking efficiency by batching functionally similar microbenchmarks into clusters for performance testing. This clustering approach offers several benefits:

- **Broader Analysis:** Enhances the collective utility of microbenchmarks, enabling them to contribute to identifying broader performance trends.
- **Efficiency Gains:** Reduces redundant overhead, optimizing the use of computational and testing resources.
- **Improved Scalability:** Makes the performance testing process more practical and scalable, particularly for large-scale projects.

Leveraging batch execution for microbenchmarks can significantly enhance the efficiency of the execution process. Moreover, since batch execution does not directly alter the microbenchmarks, this approach minimizes the risks associated with improving test execution efficiency, which makes it possible to enhance microbenchmarking efficiency while maintaining the original accuracy. Ultimately, our goal is to streamline performance testing workflows, enabling faster feedback loops, reduced computational burdens, and more accessible performance testing for developers.

B. Batch Execution Construction

To enable batch-executing microbenchmarks, we leverage code coverage information to cluster functional similar microbenchmarks. Code coverage measures how different microbenchmarks exercise overlapping code regions. It is commonly used as an indicator of functional similarity in prior studies for test selection and reduction [7]–[9]. The *JaCoCo agent* [10], a library for calculating code coverage in Java projects, is employed to measure the percentage of overlapping covered code lines. The similarity score ranges from 0% (no overlap) to 100% (complete overlap where the smaller test's coverage is a subset of the larger one). While a 100% similarity score reflects substantial alignment in covered code segments, it does not necessarily imply identical functionality. Instead, it indicates a shared scope that allows for meaningful grouping of related benchmarks. This nuanced understanding of coverage ensures that clusters are cohesive and functionally relevant. Furthermore, batch execution of microbenchmarks maintains the same code coverage as individual runs, preserving the collective coverage achieved by each microbenchmark within the cluster.

C. Methodology for Batch Execution

Algorithm 1 illustrates our methodology for batch execution of microbenchmarks. It consists of two steps: 1) The first step involves ranking the most similar *ju2jmh* microbenchmarks for each hand-crafted JMH microbenchmark based on code coverage similarly. The more similar a microbenchmark is, the higher the chance it will be clustered into the same cluster. 2) For each hand-crafted JMH microbenchmark, we group the top-K most similar *ju2jmh* microbenchmarks to enable batch execution. The value of K is determined by incrementally summing the sizes of individual microbenchmarks until their combined execution time reaches an empirically observed threshold sufficient for detecting performance bugs (i.e., 5 μs) [2].

Algorithm 1 Benchmark Batching

```
1: Input: Handcrafted JMH Microbenchmarks, ju2jmh Microbenchmarks, Cluster Size K
2: Output: Clusters of Microbenchmarks
3: // Similarity Scoring
4: for each JMH Handcrafted Microbenchmarks do
5:     for each ju2jmh Microbenchmarks do
6:         Compute and store similarity(JMH, ju2jmh)
7:     end for
8:     Rank ju2jmh by similarity for JMH
9: end for
10: // Cluster Formation
11: for each JMH Hand-crafted Microbenchmarks do
12:     Create cluster;
13:     Add top ju2jmh until size(cluster)=K
14:     If size(cluster) ≤ K then Continue
15:     Add cluster to clusters
16: end for
17: Return: Clusters
```

III. Evaluation Setup

In this section, we evaluate our proposed batch execution strategy by comparing it with individually executed ones in terms of microbenchmarking efficiency and stability.

A. Approach

The following two metrics are used to evaluate the efficiency and stability of the batch execution strategy.

Microbenchmarking Execution Time: This metric is used to measure the efficiency of microbenchmarking for batch execution. By using individually executed microbenchmarks

as a baseline for comparison, we can examine how batch execution speeds up microbenchmarking.

Microbenchmarking Stability: To determine the reliability and consistency of the microbenchmarks in detecting performance-related issues, we assess the stability in microbenchmarking results across multiple executions of both individual and batch-executed microbenchmarks. Stability serves a critical metric that helps developers observe whether a program behaves consistently under performance testing and is useful for real-world performance testing enhancement. We analyze the stability of microbenchmarks using the *Relative Standard Deviation (RSD)* of their execution time, where a lower RSD value indicates higher stability and vice versa. For each benchmark and cluster, we calculate RSD across 30 iterations and measure its stability. In particular,

- A $\leq$1% RSD indicates a stable result where clustering retains accuracy.
- A >5% RSD suggests an unstable result where clustering may compromise accuracy [2].
- For RSD between 1% and 5%, we evaluated whether the cluster's RSD was lower than most individual benchmarks in that cluster to assess sufficiency.

B. Study Subjects

In this section, we introduce the three study subjects involved in this study. We experimented our clustering strategy deployed on *ju2jmh* microbenchmarks, on three open-source Java projects, `Rxjava`, `Eclipse-collections`, and `Zipkin`, which have readily available JUnit test cases and JMH microbenchmarks. The selected three open-source projects are widely recognized, well-maintained, and extensively studied in prior studies on performance microbenchmarking [2], [11]–[16].

Table I provides detailed information about our study subjects: the studied version (column **Version**, i.e., the latest version at the commencement of this study), extracted metadata from Github including the numbers of stars (column **Stars**) and the number of contributors (column **Contr.**), the number of the source lines of code (column **SLOC**), the total number of JMH benchmarks (column **#JMH**) and selected JUnit test cases (column **#JUnit**).

TABLE I
OVERVIEW OF THE STUDY SUBJECTS.

	Version	Stars	Contr.	SLOC	#JMH	#JUnit
RxJava	3	44.7K	277	311,975	1,217	9,825
Eclipse-collections	10.4.0	1.7K	88	135,017	986	24,758
ZipKin	2.7	14.4K	145	7,467	59	501
Total		60.8K	510	454,459	2,262	35,084

C. Experiment Settings

We select 14,117 out of 35,084 *ju2jmh* microbenchmarks with execution time below 2μs, as these microbenchmarks tend to have limited utility, due to their high variance and limited scope [2] that can benefit from batch execution. The 14,117 *ju2jmh* microbenchmarks are grouped into 1,723 clusters. Each cluster is represented as a JMH microbenchmark that sequentially invokes the individual microbenchmarks within its payload.

D. Execution environment

To perform the microbenchmarking procedure, we took advantage of cloud computing resources to simulate a real-world environment. To have a consistent measurement process, for all the experiments in this study, we deployed t2.xlarge (4 vCPUs, 16 GB memory) instances provided by *Amazon Web Services*[1]. Instances are run on Amazon Linux 2023 and OpenJDK 1.8. In total, our experiment consists of 15,840 measurements, each containing 30 data points measured, i.e., each data point is the throughput of one JMH microbenchmark case (annotated by `@benchmark` in testing code) measured in one second. The experiments in this study took approximately 264 machine hours to complete.

IV. PRELIMINARY EVALUATION RESULTS

To evaluate our proposed batch-executed strategy, our preliminary evaluation compares the efficiency and stability between individually and batch-executed microbenchmarks.

A. Execution Time Savings

Table II highlights the efficiency of the batch execution strategy across the study subjects. The results demonstrate significant reductions in execution time, with savings ranging from 80.33% to 89.81%.

Scalability: The clustering approach effectively groups benchmarks, with cluster sizes averaging 5 to 10 benchmarks. Larger projects like Eclipse-collections see the highest percentage of time saved (89.81%) due to greater opportunities for redundancy reduction.

Impact: The observations from Table II indicate the significant efficiency improvement for microbenchmarks brought by batch execution. Since batch execution does not involve code changes, this strategy will not affect code coverage, which minimizes the risks of reducing test reliability. The batch-executed strategy scales well across projects of different sizes, ensuring faster feedback loops and reduced costs in performance testing workflows.

TABLE II
TIME SAVED USING BATCH-EXECUTED STRATEGY

	RxJava	Eclipse-collections	ZipKin
# of *ju2jmh* benchmarks	2,896	11,109	112
# of clusters	570	1,132	21
Avg. size of clusters	5.08	9.81	5.42
Total time for individuals (hours)	48.3	185.15	1.86
Total time for clusters (hours)	9.50	18.87	0.35
% of time saved (%)	80.33	89.81	81.20

B. Benchmarks' Stability

Table III evaluates the stability of clustered and individual benchmarks by comparing the Relative Standard Deviation (RSD) of their execution time and the percentages of stable ($\leq$1% RSD) and unstable ($\geq$5% RSD) benchmarks across the study subjects.

RSD Comparison: The average RSD for clusters, across three study subjects, is comparable to individual benchmarks (e.g.,

[1]https://aws.amazon.com/

0.51% vs. 0.51% for RxJava and 0.35% vs. 0.75% for Eclipse-collections), with a maximum increment of 0.20%, reflecting the slightly better stability from batching microbenchmarks.

Stability: The percentage of stable benchmarks is consistently high for clusters (91.8%-100%) and slightly better than individual benchmarks in Eclipse-collections and Zipkin. Unstable benchmarks remain rare for both clusters ($\leq$0.79%) and individuals ($\leq$0.89%).

Impact: These results show that batch execution retains sufficient stability. This ensures that clustering does not compromise the ability to detect performance regressions effectively.

TABLE III
RSD OF CLUSTERS AND INDIVIDUALS, THE PERCENTAGE OF STABLE ($\leq$1%) AND UNSTABLE ($\geq$5%) MICROBENCHMARKS

	RxJava	Eclipse-collections	ZipKin
Average RSD of individuals (%)	0.51	0.75	0.50
Average RSD of clusters (%)	0.51	0.35	0.70
RSD difference	0.00	-0.40	0.20
Stable individuals (%)	95.9	79.8	97.3
Stable clusters (%)	91.8	97.8	100.0
Stability difference (%)	-4.1	18.0	2.7
Unstable individuals (%)	0.38	0.03	0.89
Unstable clusters (%)	0.00	0.79	0.00
Instability difference (%)	-0.38	0.76	-0.89

To conclude, the results demonstrate that the proposed batch execution strategy significantly improves the efficiency of performance microbenchmarking. Table II highlights substantial time savings, with batch execution reducing execution times by over 80% across all three Java projects, making performance testing more scalable and cost-effective. Table III confirms that batch execution maintains sufficient stability, with the variability (measured as RSD) remaining negligible. The batch-executed microbenchmarks exhibit overall high stability. Therefore, these results indicate that batch execution achieves significant reductions in execution time without compromising the stability needed for effective performance testing, offering a practical solution to streamline performance test suites in diverse software projects.

V. THREATS TO VALIDITY

In this paper, we leverage the code coverage as an indicator to cluster functionally similar microbenchmarks. While other methods such as static code analysis, dynamic call graphs, or semantic code embeddings could be used for similarity measurement, such approaches usually require complex code analysis frameworks or face challenges of dynamic language features. Therefore, we choose this lightweight yet effective approach to capture code with similar execution paths without deep analysis overhead while maintaining a high correlation with performance characteristics. When estimating execution time savings, we exclude the time to compute benchmark batching, which takes an average of 13.7 to 48.3 seconds per cluster in our study subjects, as this calculation is performed once and can be reused in subsequence executions.

VI. RELATED WORK

Benchmarking, especially microbenchmarking, requires significant computational resources to run and generate results [17]. Running benchmarks sequentially on limited resources increases overall execution time [17]. Redundant and ineffective microbenchmarks are an additional concern. Laaber and Leitner [12] pointed out the prevalence of benchmarks with limited or overlapping coverage, which contribute little value to performance testing processes.

Enhancing the efficiency of performance testing has been explored through various methodologies and case studies [2], [12], [13], [15], [18]. Laaber *et al.* [13] developed methods to dynamically halt microbenchmark executions once result stability is achieved, thereby reducing execution time without sacrificing result quality. Techniques like prioritizing performance regression tests based on risk analysis have also been explored by Huang *et al.* [19] demonstrating the effectiveness of targeted testing in reducing unnecessary test executions. Automated solutions, such as those proposed by AlGhamdi *et al.* [20], have further streamlined performance testing by introducing stopping criteria based on result stability.

The concept of enhancing the efficiency and effectiveness of performance microbenchmarks through batch execution is inspired by studies that have successfully implemented batch execution strategies for testing. Fallahzadeh *et al.* [21] introduced techniques for test batching and parallel execution in continuous integration systems, emphasizing the impact of resource allocation and feedback time on execution efficiency. Similarly, AlGhamdi *et al.* [22] explored reducing execution time in performance testing by grouping related test cases into clusters and executing them in parallel, ensuring effective resource utilization without compromising test quality [23].

Despite various advancements in performance testing, the application of batch execution to optimize microbenchmarking remains underexplored. Our study fills this gap by proposing a tailored batch execution strategy for microbenchmarks, the preliminary results demonstrate improved efficiency in microbenchmark execution.

VII. CONCLUSION AND FUTURE PLANS

Performance microbenchmarking benefits from automated microbenchmark generation but still faces challenges with long execution time, especially for large-scale software systems or extensive test suites. Our study proposes a batch-executed strategy to improve performance testing efficiency based on microbenchmarks' functional similarity. The evaluation of three subject projects indicates that batch execution largely reduces the execution time (80.33% to 89.81%) without compromising the benchmark's stability. In the future, we plan to investigate the effectiveness of the batch execution approach in detecting performance bugs, and explore the characteristics of the batch-executed benchmarks that may enhance their performance bug identification capability. Our future research shall provide practitioners with insights and guidelines for optimizing their performance testing strategies, potentially leading to more robust software systems.

Data Availability: The data and scripts used in this study are publicly available at https://github.com/senseuwaterloo/Batch-execution-ju2jmh-benchmarks.

References

[1] C. Laaber, H. C. Gall, and P. Leitner, "Applying test case prioritization to software microbenchmarks," *Empirical Software Engineering*, vol. 26, no. 6, p. 133, 2021.

[2] M. Jangali, Y. Tang, N. Alexandersson, *et al.*, "Automated generation and evaluation of jmh microbenchmark suites from unit tests," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1704–1725, 2022.

[3] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 international symposium on memory management*, 2013, pp. 63–74.

[4] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, 2007.

[5] D. Delaet, H. Vandierendonck, and K. De Bosschere, "A quantitative study of jvm execution characteristics using java grande benchmarks," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 7, pp. 555–577, 2004.

[6] H. Palikareva, M. O. Myreen, and G. Lowe, "Mechanising and verifying java jit optimisations in isabelle/hol," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, Springer, 2016, pp. 476–495.

[7] C. Coviello, S. Romano, G. Scanniello, *et al.*, "Clustering support for inadequate test suite reduction," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 95–105. DOI: 10.1109/SANER.2018.8330200.

[8] A. Khalilian and S. Parsa, "Bi-criteria test suite reduction by cluster analysis of execution profiles," in *Advances in Software Engineering Techniques - 4th IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 12-14, 2009. Revised Selected Papers*, T. Szmuc, M. Szpyrka, and J. Zendulka, Eds., ser. Lecture Notes in Computer Science, vol. 7054, Springer, 2009, pp. 243–256. DOI: 10.1007/978-3-642-28038-2_19. [Online]. Available: https://doi.org/10.1007/978-3-642-28038-2%5C_19.

[9] G. Nayak and M. Ray, "Modified condition decision coverage criteria for test suite prioritization using particle swarm optimization," *International Journal of Intelligent Computing and Cybernetics*, vol. 12, no. 4, pp. 425–443, 2019.

[10] *Jacoco: Java code coverage library*, https://www.jacoco.org, Accessed: 2024-11-25.

[11] P. Delgado-Pérez, A. B. Sánchez, S. Segura, and I. Medina-Bulo, "Performance mutation testing," *Software Testing, Verification and Reliability*, vol. 31, no. 5, e1728, 2021.

[12] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 119–130.

[13] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 989–1001.

[14] D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak, "What's wrong with my benchmark results? studying bad practices in jmh benchmarks," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1452–1467, 2019.

[15] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2469–2508, 2019.

[16] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 373–384.

[17] T. Schirmer, T. Pfandzelter, and D. Bermbach, "Elastibench: Scalable continuous benchmarking on cloud faas platforms," *arXiv preprint arXiv:2405.13528*, 2024.

[18] S. Pargaonkar, "A comprehensive review of performance testing methodologies and best practices: Software quality engineering," *International Journal of Science and Research (IJSR)*, vol. 12, no. 8, pp. 2008–2014, 2023.

[19] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 60–71.

[20] H. M. AlGhamdi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, IEEE Computer Society, 2016, pp. 279–289. DOI: 10.1109/ICSME.2016.46. [Online]. Available: https://doi.org/10.1109/ICSME.2016.46.

[21] E. Fallahzadeh, A. H. Bavand, and P. C. Rigby, "Accelerating continuous integration with parallel batch testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 55–67.

[22] H. M. AlGhamdi, C.-P. Bezemer, W. Shang, *et al.*, "Towards reducing the time needed for load testing," *Journal of Software: Evolution and Process*, vol. 35, no. 3, e2276, 2023.

[23] H. M. AlGhamdi, "Automated approaches for reducing the execution time of performance tests," M.S. thesis, Queen's University (Canada), 2017.