PhD Student in Software Performance Engineering – Take-Home Task

Task 1:

Provide your own thoughts about the scientific methodology and findings of the study. What aspects are well designed, and in what ways could the study be improved? Which results are particularly surprising, and why?

The research paper [1] offers a comprehensive study on Java's steady-state performance assessment, with a well-structured flow between sections, making it highly accessible to newcomers in the field. I appreciate the fact that this is the first study to evaluate the performance of 586 benchmarks across 30 open-source Java systems and provides access to the dataset. Such research is essential, as it not only provides valuable insights but also encourages further exploration of the topic by other researchers. The analyses conducted are extensive, as it is not only limited to steady state performance assessment but also quantifies inefficiencies caused by incorrect warm-up estimations. The research questions are well-formulated, and the methodology is advanced [2], utilizing one of the most effective techniques to determine steady-state execution in Java.

While it may be outside the current scope, a potential improvement could be the addition of a case study or example demonstrating how this study could be applied in a real-world industrial scenario. This would enhance the paper's impact by showcasing its practical relevance. Additionally, a comparative analysis using various steady-state detection techniques, rather than relying on a single method, could further strengthen the study. While the technique used in this paper is advanced and widely respected in the field, exploring alternative methods might reveal scenarios where other techniques offer better performance. This would provide a more comprehensive evaluation and highlight the versatility of different approaches to steady-state detection in benchmarking.

A surprising result for me was from RQ4, where dynamic reconfiguration was assessed for its effectiveness in estimating Java's steady state performance. I expected it to provide optimal warmup estimates, at least in majority of the cases, but the study revealed non-trivial errors, falling short of expectations. This was an interesting finding.

Discuss potential extensions of this study. Be concrete and specific and discuss what practical problems you would need to solve when working on these extensions.

As future work or potential extensions, the authors could explore dynamic reconfiguration techniques using large language models (LLMs). While current techniques outperform developer configurations, they still fall short of providing optimal estimates. Future efforts could focus on training LLMs on a large dataset of past benchmarks to predict the number of warmup iterations needed and when the steady state will be reached. This approach could dynamically configure parameters more effectively. Additionally, a comparative study could be conducted using different LLMs to evaluate which performs best. However, this proposal faces practical challenges. First, constructing a comprehensive dataset that includes benchmarks from a wide range of systems would require considerable time, effort, and manpower. Such a dataset must be inclusive, covering various types of systems to ensure the model's predictions are generalizable across different environments. Second, while LLMs are a cutting-edge development in software engineering, they are also known for issues like hallucinations[3] or generating irrelevant outputs, which could impact the reliability of their predictions. Addressing these concerns and ensuring the accuracy of LLM-generated configurations would be crucial for the success of this approach.

Task 2:

Explore the data (e.g., calculate and/or visualize basic descriptive statistics) and find out what the different fields in the data mean

I explored the JSON files located in the timeseries folder provided in the paper's GitHub repository. To analyze the data, I created a Jupyter notebook where I conducted an in-depth exploration of the JSON data. In this notebook, I calculated various descriptive statistics such as the mean, median, mode, and standard deviation, which helped in understanding the performance trends and variations across the different iterations and forks. To further visualize the data, I generated a series of graphs that illustrate the performance across iterations and forks. One of the key graphs I plotted shows the values above and below the average execution time, helping to visualize the level of consistency among the different forks. For this analysis, I selected a random JSON file from the dataset available in the repository to perform the calculations and visualizations. I have explained each code block in the Jupyter notebook thoroughly, providing clarity on the steps taken for data exploration, statistical calculations, and plotting.

Select one small finding in the original study and replicate it using your own scripts and code.

For this analysis, I focused on the findings from RQ1, which highlights that the two-phase assumption (i.e., a warmup phase followed by a steady-state phase) does not always hold true. To replicate this observation, I created a small Maven project from scratch, following the detailed instructions provided on the OpenJDK JMH GitHub repository. As suggested by the repository, I used the terminal on my MacBook Air M1 to set up the project, rather than relying on an IDE. After setting up the Maven project, I implemented a simple benchmark function in a file named `MyBenchmark.java`. This function performs a calculation of the square root of numbers up to 100000. Once the benchmark function was in place, I executed the benchmark to collect measurement data. I passed the following configuration through CLI arguments:

java -jar target/benchmarks.jar -wi 15 -i 15 -f 5 -tu ns -w 500ms -r 500ms -bm avgt

- -wi – warmup iterations set to 15
- -i – measure iterations set to 15
- -f – Number of forks set to 5
- -tu ns – Measurement unit of average execution time set to nanoseconds
- -w 500ms, -r 500ms – each iteration set to run for 500ms each
- -bm avgt – Benchmark mode set to average execution time

After running the iterations, I got the following result:

Result "org.sample.MyBenchmark.testMethod":
0.563 ±(99.9%) 0.008 ns/op [Average]
(min, avg, max) = (0.559, 0.563, 0.657), stdev = 0.019
CI (99.9%): [0.556, 0.571] (assumes normal distribution)

Benchmark	*Mode*	*Cnt*	*Score*	*Error*	*Units*
MyBenchmark.testMethod	*avgt*	*75*	*0.563*	*± 0.008*	*ns/ops*

The result of the benchmark showed that the average execution time was **0.563 ns** with a negligible error margin of **0.008 ns**. At first glance, this suggests that the two-phase assumption held, as the benchmark seemed to have reached a steady state after the warmup iterations, and the execution times appeared consistent. However, upon a deeper examination of the individual iteration results within each fork, I observed that even after the warmup

phase ended, the execution times did not remain completely stable. In some iterations, the values fluctuated — occasionally higher than expected, suggesting that the execution was not perfectly steady. The values would rise and fall across different iterations, indicating that even after the warmup period, the system's performance was not entirely stable. These observations support the conclusion that benchmarks do not always reach a steady state as predicted by the two-phase assumption. This finding underscores the importance of carefully examining the individual iteration results, even for smaller functions, to detect potential anomalies or fluctuations in performance. In larger or more complex experiments, these inconsistencies could be more pronounced, further challenging the assumption of steady-state performance in benchmarking as discussed in the paper.

In addition to the initial setup, I performed a trial-and-error process to determine the optimal number of warmup iterations. I began with a small number of warmup iterations and gradually increased this value, aiming to observe when the benchmark would reach a steady state, where the execution times for the operations would remain consistent across the last few iterations. However, I found that no matter how many warmup iterations I set, the consistency would eventually break down at some point during the subsequent iterations. This further supports the discussion in the paper that the two-phase assumption does not always hold, as even when attempting to adjust the warmup period, I observed fluctuations in execution times.

I also extended the measurement phase to run for 50 iterations to gather more data. Interestingly, during this extended phase, the error slightly increased to **0.011 ns**, compared to the initial value of **0.008 ns**. This increase in error suggests that, over a larger number of iterations, the instability in execution times became more pronounced. This further emphasizes the inconsistency observed in the benchmark results and adds weight to the conclusion that the two-phase assumption does not always guarantee steady performance.

Conduct a small analysis of your own that's not yet discussed in the original paper. Present your results, and discuss why you have selected this analysis.

For a small-scale analysis, I randomly selected a JSON file from the timeseries folder and loaded its contents into a variable. I then computed the mean, standard deviation, and kurtosis for each of the 10 forks. In statistical terms, kurtosis [4] measures the degree of tailedness in a data distribution. A high kurtosis value indicates that the distribution has significant outliers or heavy tails. To calculate the kurtosis, I used the kurtosis function from the scipy.stats library. I have provided the details of the code and methodology in the Jupyter notebook. Below are the results of the analysis:

Fork	Mean	Stdev	Kurtosis
1	0.565354	0.03156	32.67
2	0.56526	0.033367	38.94
3	0.555414	0.032173	36.85
4	0.558636	0.035215	29.45
5	0.554446	0.031582	33.23
6	0.55938	0.031863	35.2
7	0.558306	0.030115	41.24
8	0.560759	0.030553	49.28
9	0.565595	0.032818	37.44
10	0.565069	0.034351	38.83

As evident from the table, the standard deviation values for all the forks are relatively low compared to the mean, indicating that there is minimal variation between the data points. At first glance, this might suggest that the execution is consistent across iterations. However, when we examine the kurtosis values, a different picture emerges. The kurtosis values are significantly high, exceeding 30, which points to a high degree of tailedness, or

in other words, the presence of extreme outliers. This sharp contrast between the low standard deviation and high kurtosis reveals that simply relying on standard deviation to assess the stability of the iterations can be misleading. The high kurtosis suggests that the data is skewed, with occasional extreme values that are not reflected in the standard deviation alone. This analysis highlights the importance of considering multiple descriptive statistics, such as both standard deviation and kurtosis, when evaluating the performance of benchmarks. Relying solely on basic metrics like the mean and standard deviation might give an incomplete or inaccurate picture of the data's behavior.

References:

[1] Traini, Luca, et al. "Towards effective assessment of steady state performance in Java software: Are we there yet?." *Empirical Software Engineering* 28.1 (2023): 13.
[2] Barrett, Edd, et al. "Virtual machine warmup blows hot and cold." *Proceedings of the ACM on Programming Languages*1.OOPSLA (2017): 1-27.
[3] Yao, Jia-Yu, et al. "Llm lies: Hallucinations are not bugs, but features as adversarial examples." *arXiv preprint arXiv:2310.01469* (2023).
[4] https://www.sciencedirect.com/topics/social-sciences/kurtosis