# Uncovering Steady State Executions in Java Microbenchmarking with Call Graph Analysis

Madeline Janecek
mj17th@brocku.ca
Brock University
St. Catharines, ON, Canada

Sneh Patel
wsp18oo@brocku.ca
Brock University
St. Catharines, ON, Canada

Naser Ezzati-Jivan
nezzati@brocku.ca
Brock University
St. Catharines, ON, Canada

## ABSTRACT

Developers often use microbenchmarking tools to evaluate the performance of a Java program. These tools run a small section of code multiple times and measure its performance. However, this process can be problematic as Java execution is traditionally divided into two stages: a warmup stage where the JVM's JIT compiler optimizes frequently used code and a steady stage where performance is stable. Measuring performance before reaching the steady stage can provide an inaccurate representation of the program's efficiency. The challenge comes from determining when a program should be considered as in a steady state. In this paper, we propose that call stack sampling data should be considered when conducting steady state performance evaluations. By analyzing this data, we can generate call graphs for individual microbenchmark executions. Our proposed method of using call stack sampling data and visualizing call graphs intuitively empowers developers to effectively distinguish between warmup and steady state executions. Additionally, by utilizing machine learning classification techniques this method can automate the steady state detection, working towards a more accurate and efficient performance evaluation process.

## CCS CONCEPTS

• **Software and its engineering → Software performance**; **Just-in-time compilers**.

## KEYWORDS

ICPE Data Challenge, Call Graph Analysis, Time Series Analysis, Machine Learning, Benchmarking, Steady State, Warmup

**ACM Reference Format:**
Madeline Janecek, Sneh Patel, and Naser Ezzati-Jivan. 2023. Uncovering Steady State Executions in Java Microbenchmarking with Call Graph Analysis. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3584689


*ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal*

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584689

## 1 INTRODUCTION

Microbenchmarking, which is the practice of monitoring the performance of a repeatedly executed small unit of code, is a common method used in Java software performance testing [9]. However, to obtain accurate results, one must take into account how the Java Virtual Machine (JVM) executes code. The JVM uses Just-In-Time (JIT) compilation to convert key parts of the Java code into more efficient machine code [7], which can lead to initial performance inconsistencies.

To account for this, microbenchmarking tools often require users to define an initial warmup phase during which performance metrics are not collected, and only consider executions that follow this phase as being in a steady state. However, determining when a benchmark has reached a steady state is a difficult task. Current practices rely almost exclusively on the user's estimation, and some benchmarks may never reach a steady state [2].

Consequently, there is a need for more advanced techniques to determine if and when a benchmark reaches a steady performance state. In this work, we propose that call stack sampling allows for a more in depth analysis of microbenchmarking that is not seen with existing practices. The benefits of using call stack samples are twofold. Firstly, call stack sampling can provide a detailed picture of internal benchmark execution with low overhead and instrumentation. As such, it provides more visibility than performance metrics like execution duration. Secondly, our results indicate that executions recorded during and after a warmup phase have clear structural differences. As such, this data is well suited for various analyses of steady state performance. We present how the data may be used to visualize the differences between executions taken from warmup and steady state phases. Furthermore, we investigate how machine learning classification may be used to automate call stack-based steady state detection.

## 2 BACKGROUND & RELATED WORK

In this section, we provide an explanation of the JMH microbenchmarking tool that we utilized to gather the data for our graphs, as well as methods for analyzing the flame graphs that were produced. Additionally, we highlight other works in the field that served as inspiration for our research.

### 2.1 JMH

Java Microbenchmark Harness (JMH) is a toolkit to measure the performance of Java programs. To ensure that accurate results are produced the executions are executed in multiple forks (default being 5). JMH can measure the performance of Java in different benchmark modes depending on the goal of the analysis.

Each mode splits the executions it measures into the warmup and steady state. The idea is the JVM will detect frequent executions (such as loops), and then it will dynamically compile the code into optimized machine code. This results in the starting executions having unsteady execution times producing inaccurate analyses. These unstable executions are labelled as warmup executions and should not be used to measure the performance of the applications. The later executions usually have more stable run times and as such are labelled as steady state executions.

Despite the criticality of discarding warmup exeuctions, JMH does not provide a method of detecting when a program reaches a steady state. Users initialize the number of warmup executions at the start of the benchmark (default being 5). For example, if a developer states that the benchmark should have 10 warmup executions, each execution after that would be considered a steady state execution. This might work for some programs, however, it is difficult to know exactly how many warmup states a Java program requires for it to produce accurate steady state executions. If the number of warmup executions is wrong JMH would label some executions that should be warmup as steady state, resulting in inaccurate data. To further complicate matters, recent research shows that some programs never reaches a steady state [9]. At this point, most accurate way to detect when a steady state is reached is to apply change point algorithms upon the execution runtimes. By conducting call graph informed steady state detection, we automate the process while providing more visibility than other runtime based methods.

## 2.2 Flame Graph

Generally, the reports that CPU profiles create are long and time-consuming to analyze manually. An alternative to such reports are flame graphs, which are call stack visualizations first introduced by Brendan Gregg [4]. Using flame graphs we can understand the performance of programs, and discover function-level bottlenecks that it may have. In a flame graph, each function is a box stacked on top of the function that calls it. The width of the graph determines the time spent on that function.

## 2.3 Related Work

Traini et al. [9] address several questions related to microbenchmark performance, including the question of whether or not microbenchmarks always reach a steady state. To answer this question, they examine the runtime of each execution and employ a change point algorithm to determine the steady state, as it is theorized to have a significantly reduced time compared to the warmup state. To determine whether or not the benchmark has reached this state, they analyze the partition created by the change point algorithm and evaluate whether a specific portion has similar timings. If so, the benchmark is considered to have reached a steady state. In our own research, we utilize this method to label graphs as exhibiting warmup or steady state behaviour. Using these results as ground truths allows us to see if a call graph informed classification can produce similar results to current state-of-the-art approaches.

Change point algorithms are typically designed to operate on low-dimensional data, and identifying change points within higher-dimensional data remains a challenge [5]. However, He et al. present a method for utilizing change point algorithms on higher-dimensional data by converting the data into vectors and training them on a classification model [6]. Zambon et al. take a similar approach where a sequence of graphs is mapped onto vectors to perform change point detection [10]. In our research, we convert the call graphs into vectors and use various classification models. This allows for the use of a broad range of change point detection techniques [1].

In their work, Bauer et al. [3] investigate the automation of the process of identifying change points in time series analyses using machine learning classification models. They employ various models to discover change points in the data, and then evaluate the accuracy of these change points by using preset labels to validate their dataset. Subsequently, they apply their model to unlabelled data in order to compare the results. In our research, we performed an analysis of the execution durations of multiple benchmark results to investigate the performance of the program. Through this analysis, we were able to confirm that the executions identified as being within a steady state displayed lower and more consistent runtimes compared to those found in the warmup states. This was determined by comparing the runtime values of each execution and observing a clear distinction between the two stages of execution. Our analysis on call graph

# 3 METHODOLOGY

In this section we present our approach for collecting call stack samples from JMH microbenchmarks, their visualization and analysis, as well as a steady state classification method. These proposed methods aim to provide an accurate and efficient way to uncover the steady states of a Java program visually and intuitively.

## 3.1 Data Collection

We collect series of call stack samples using Perf throughout the entire JMH benchmarking process. Perf is a profiling tool that can be used to collect performance data within Linux-based systems. One of its many features is CPU stack trace sampling, which is where we record which functions are actively running at regular sampling intervals. Perf's overhead is directly correlated to the frequency of sampling, however it is still capable of collecting hundreds of samples every second with negligible performance impact. Perf is well suited for our purposes as it is capable of collecting data from multiple levels. As such, we can see kernel code execution and its impact on performance. Additionally, perf has JIT symbol support, making it possible to interpret the call stack samples collected from the JVM.

For each JMH benchmark execution, we also record timestamped logs indicating the start and end of an execution, which allows us to break up the data to make execution-to-execution comparisons. Each execution's samples may then be combined to make a call graph, which is what we use to analyze each execution separately.

## 3.2 Visualization

After generating a call graph for each of the executions, we turn to flame graphs to highlight the structural differences between the warmup and steady states phases. Flame graphs are compact visual representations of call stack samples that can assist in software performance analysis. Each box in the graph (known as a stack

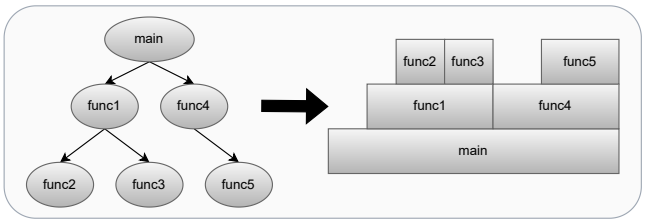


**Figure 1: Call graphs (left) compared to flame graphs (right).**

frame) is representative of a function call, where the stack frames below the topmost stack frame show the call path. The width of each frame is indicative of the total time that function with that particular call path spent consuming CPU resources. Wider boxes may be the result of the function having long execution times, however it might also result from a high call frequency.

Take, for example, the simple flame graph depicted in Figure 1. Using the flame graph we can easily see the functions that make up the application's execution, each function's calling function, and comparatively how much time each function took throughout the entire execution. This can be extremely useful for a variety of performance based analyses, including bottleneck detection, anomaly root cause identification, and so forth. For our purposes, comparing the flame graphs of each execution shows that there is a clear visual difference between those that are warming up and those that have reached a steady state.

Take the example shown in Figure 2. Executions that are in steady states (the second row) appear to have more consistent internal behavior, as demonstrated by their flame graph, when compared to those in the warm-up phase (the first row). By analyzing the graphs in detail, it becomes evident that the warm-up executions exhibit a greater number of interpret functions, indicating that the JVM is operating in interpret mode and resulting in higher fluctuations in performance. In contrast, the steady state shows fewer interpret functions, likely indicating that certain parts of the code have been compiled and optimized by the JVM, resulting in more consistent internal runtime behavior and steady execution time.

## 3.3 Classification

In addition to visualizing the microbenchmark executions, we set out to automate steady state detection using structural call graph indicators. To do this we defined this task as a binary classification problem where executions are either in a warmup or steady state.

*3.3.1 Vectorization.* Many machine learning classification algorithms operate on fixed-length vectors, however, the call graphs we collect for each execution have varying structures and numbers of nodes. To overcome this challenge, we must transform each graph into a vector that effectively captures its content as well as its topological characteristics. To accomplish this task, we employ the graph embedding technique Graph2Vec [8]. Graph2Vec utilizes a process of generating vectors with initially random values, which are then refined to represent the nodes, edges and features of the graphs over several iterations. This allows us to effectively use these transformed vectors as input for the machine learning classification algorithms.

**Table 1: Steady state classification performance**

| | Accuracy (%) | Precision (%) | Recall (%) |
|---|---|---|---|
| MLP | 88.5 | 31.6 | 12.5 |
| RF | 90.4 | 75.0 | 6 |
| DT | 82.8 | 19.3 | 22.5 |
| KNN | 89.8 | 45 | 9 |
| SVC | 90 | 50 | 1 |

*3.3.2 Data Labeling.* Training and testing our classification models requires a set of labeled data. To generate these labels we utilize the state-of-the-art steady state detection technique described in [9]. This consisted of recording the runtimes of each execution and then applying the PELT change point detection algorithm to identify any shifts into a steady performance state.

*3.3.3 Classifier Training.* Once the data has been vectorized and labeled, it was used to train machine learning classification algorithms to establish the relationship between the features of an execution's call graph and its stage. In this work, we trained and evaluated the performance of five distinct machine learning classification algorithms for steady state identification. The classifiers we examined include Multi-Layer Perceptron (MLP), Random Forest (RF), Decision Tree (DT), K-nearest neighbour (KNN), and C-Support Vector Classifier (SVC) for which the results will be provided in the following section.

# 4 EXPERIMENTAL SETUP AND DISCUSSION

The data used for our experimental evaluation was collected in a Linux Ubuntu 22.04 LTS environment with the 64-bit kernel 5.15.0-56. Perf version 5.15.64 was used to perform the call stack sampling, while JMH measured the performance of a simple binary search. Apache Log4j 2.11.2 was chosen to record the start and end of each execution as it provides high throughput while imposing minimal overhead. 10000 executions collected across 10 forks of the microbenchmarking were collected an processed following the procedure discussed in Section 3.1. The experimental results and scripts used to generate them have been make publicly available [1].

## 4.1 Visualization

As discussed in Section 3.3.1, generating flame graphs for initial executions and those identified by the PELT algorithm as having reached a steady state of performance shows a clear visual difference (see Figure 2). The executions recorded during the warmup stage have a much larger and less uniform stack depth. This is primarily because the functions have yet to be compiled, and as such they are still being executed by the JVM's interpreter function. Once the functions are compiled and can run natively, the flame graphs' structure becomes much more stable.

## 4.2 Classification

In order to evaluate the performance of the classification methods discussed in Section 3.3.3, we computed several metrics such as True Positives (TP), True Negatives (TN), False Positives (FP) and

[1] https://github.com/sneh2001patel/Uncovering-Steady-State-Executions-in-Java-Microbenchmarking-with-Callgraph-Analysis

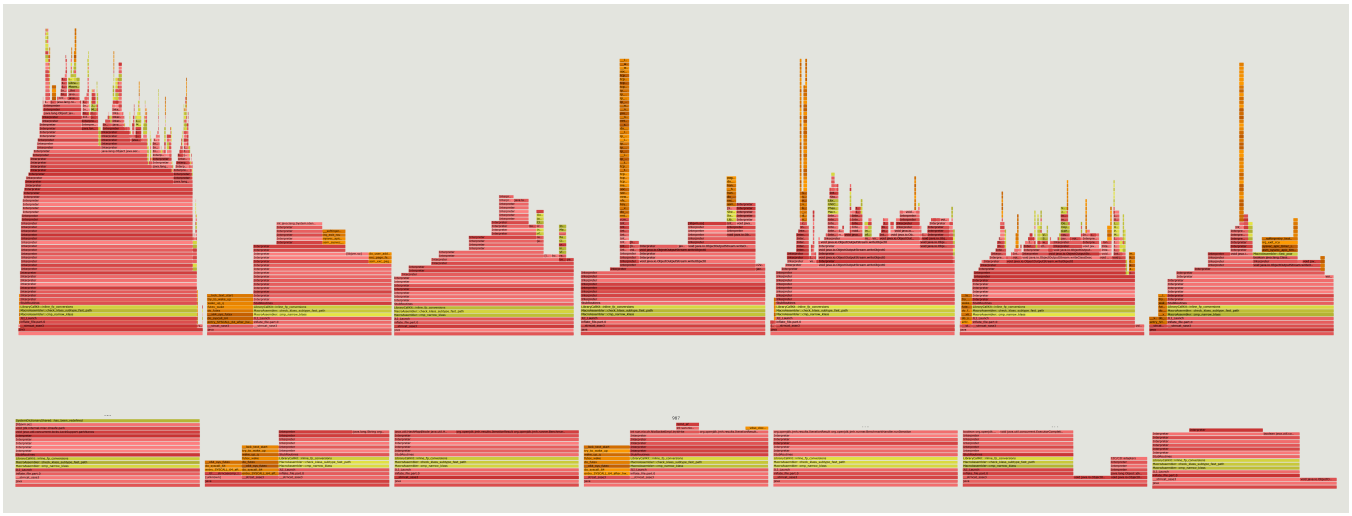

**Figure 2: Visual differences in processing between executions in warmup (top row) and steady (bottom row) states.**

False Negatives (FN). These values were then used to calculate the accuracy (1), precision (2), and recall (3) of each classifier algorithm.

$$Accuracy = \frac{(TP + TN)}{(TP + FP + TN + FN)} \tag{1}$$

$$Precision = \frac{TP}{(TP + FP)} \tag{2}$$

$$Recall = \frac{TP}{(TP + FN)} \tag{3}$$

The results of classification performance analysis are displayed in Table 1. The Random Forest (RF) method showed the best performance with the highest accuracy and precision scores. Despite the clear distinction in the visual representations of the call graphs of warmup and steady state executions, the recall scores for all models were relatively low. The accuracy of the model is high, but the low recall indicates that there is room for improvement in our classification methods. One possible reason for this low recall score is the limited size of the dataset and therefore potential inaccuracies in the labels of each graph.

The labels (of executions and the corresponding call graphs) were generated using the same method as described in [9] in which the authors applied a PELT algorithm to detect change points in the execution durations, and then determined whether 500 consecutive executions had similar duration. If so, the execution was labeled as steady state, otherwise as warm-up. Our dataset, however, was not as large as the one used in [9]. Therefore, instead of checking for 500 consecutive executions, we checked if 5% of the executions (50) after the change point had similar duration. This could lead to potential inaccuracies in the labeling, as checking a smaller number of consecutive executions may not be sufficient to ensure the accurate labeling.

Another reason for the low recall score could be the size (number of items) of each vector in the resulting vectors set of the Graph2Vec method. A smaller size may not contain enough information to accurately represent each call graph. Larger vector sizes can store more information about the graphs, being more representative of their strucutre and content, however, this also increases the time required to create the model.

In this study, we used vectors of a moderate size. Nonetheless, in our future work, we plan to investigate alternative vectorization technique and the use of larger vector sizes in order to improve the classification results. We also plan to use a larger dataset by running more benchmarks and increasing the number of forks and iterations for each benchmark. This will allow us to gather more data and potentially improve the recall score of our models.

## 5 CONCLUSIONS

In this paper, we presented a technique for distinguishing between steady state and warmup state executions using visualized execution call graphs and flame graphs. We utilized graphical representation to demonstrate and intuitively compare the internal executions of warmup and steady states. Our method showed a clear distinction between the shape, content, number of internal JVM functions (such as interpret function) and topological structures of the executions of the warmup and steady states. We then trained the classification methods to learn the internal execution behaviour of steady and warmup executions. Our method showed the Random Forest (RF) method has the best performance with the highest accuracy and precision scores.

In future research, we plan to improve the classification results by utilizing a larger dataset and experimenting with different vectorization methods with possibly more representative and different-size vectors. We will also explore the use of unsupervised learning to remove the need for labeled training data.

## ACKNOWLEDGMENT

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), MITACS, Ciena, and Bornea Dynamics Limited is gratefully acknowledged.

# REFERENCES


[1] Aminikhanghahi, S., and Cook, D. A survey of methods for time series change point detection. *Knowledge and Information Systems 51* (05 2017).

[2] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S., and Tratt, L. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang. 1*, OOPSLA (oct 2017).

[3] Bauer, A., Straesser, M., Beierlieb, L., Meissner, M., and Kounev, S. Automated triage of performance change points using time series analysis and machine learning.

[4] Gregg, B. Visualizing performance with flame graphs.

[5] Grundy, T., Killick, R., and Mihaylov, G. High-dimensional changepoint detection via a geometrically inspired mapping. *Statistics and Computing 30*, 4 (mar 2020), 1155–1166.

[6] He, Y., Burghardt, K. A., and Lerman, K. Leveraging change point detection to discover natural experiments in data. *EPJ Data Science 11*, 1 (2022), 49.

[7] Horký, V., Libič, P., Steinhauser, A., and Tůma, P. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2015), ICPE '15, Association for Computing Machinery, p. 337–340.

[8] Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., and Jaiswal, S. graph2vec: Learning distributed representations of graphs. *CoRR abs/1707.05005* (2017).

[9] Traini, L., Cortellessa, V., Di Pompeo, D., and Tucci, M. Towards effective assessment of steady state performance in java software: Are we there yet? *Empirical Softw. Engg. 28*, 1 (jan 2023).

[10] Zambon, D., Alippi, C., and Livi, L. Change-point methods on a sequence of graphs. *IEEE Transactions on Signal Processing 67*, 24 (2019), 6327–6341.