Software Mining – Investigating Correlation between Source Code Features and Michrobenchmark's Steady State

Amirmahdi Khosravi Tabrizi
akhosravitabriz@brocku.ca
Brock University
St. Catharines, ON, Canada

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

ABSTRACT

Microbenchmarking is a widely used method for evaluating the performance of a piece of code. However, the results of microbenchmarks for applications that utilize the Java Virtual Machine (JVM) are often unstable during the initial phase of execution, known as the warmup phase. This is due to the JVM's use of just-in-time compiler optimization, which is to identify and compile a "hot set" of important code regions. In this study we examine the static features of 586 microbenchmarks from 30 Java applications. To do so, we first extract static source code features of the benchmarks and then employ manual and descriptive data mining methods to identify meaningful correlations between these static features and the benchmarks' ability to reach a steady state. Our findings indicate that the number of function calls and lines of code have a considerable influence on whether or not the microbenchmarks reach a steady state.

CCS CONCEPTS

• **Software and its engineering → Software performance**; • **Information systems → Data mining**.

KEYWORDS

Microbenchmarking, Source code features, Data mining, Performance testing, Java, JMH

ACM Reference Format:
Amirmahdi Khosravi Tabrizi and Naser Ezzati-Jivan. 2023. Software Mining – Investigating Correlation between Source Code Features and Michrobenchmark's Steady State. In *Companion of the 2023 ACM/SPEC International Conference onPerformance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra,Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3584695

1 INTRODUCTION

Microbenchmarking is a lightweight performance testing technique for Java applications that involves measuring the time it takes for a specific piece of code to execute. It is less demanding compared to other testing techniques and is primarily used to test code performance. [11]

Microbenchmarking in Java software is challenging due to the JVM's *dynamic recompilation* method using the just-in-time JIT compiler to locate important code regions. This can cause performance fluctuations and unstable results. To address this, practitioners use two phases for microbenchmarking: the *warmup phase* and *Steady State*. The former is when the JIT compiler decides which code to compile dynamically, causing fluctuations. The latter is when the benchmark reaches a stable performance state, although not all microbenchmarks reach this phase. [1, 4, 11]

To obtain reliable microbenchmarking results, it is important to estimate the steady state start time accurately and understand factors that affect reaching steady state. Previous research has shown that developer estimations of warmup phase and corresponding benchmark configurations are often inaccurate and inefficient [11]. To address this issue, prior studies have proposed solutions such as Laaber et al.'s dynamic reconfiguration approach as an alternative to static configurations [6]. Additionally, Costa et al. [2] examined common pitfalls in utilizing the Java Microbenchmark Harness (JMH) [7], a widely used microbenchmarking framework in the Java community. Significantly, Laaber et al. [5] proposed a machine learning model that uses source code features to predict unstable benchmarks, suggesting a potential correlation between these features and the steady state of benchmarks.

Our study utilizes a dataset consisting of 586 microbenchmark results from 30 well-known Java open-source projects, including RxJava, Log4J2, and Apache Hive, which cover a range of project areas like application servers, libraries, and databases [11]. Our goal is to examine the correlation between the benchmarks' static source code features and their steady state attainment using the dataset, building on previous research [5]. To achieve this, we first enhance the dataset by including the static source code features of the benchmarks. Subsequently, we apply manual and descriptive data mining methods to identify potential correlations.

The remainder of this paper is organized as follows. Section 2 provides a succinct background on the descriptive data mining method utilized in this study. Section 3 details the dataset, methodology, and findings of our investigation. Finally, in Section 4, we conclude our findings and discuss potential avenues for future research.

2 BACKGROUND

2.1 Java Microbenchmark Harness (JMH)

JMH is a popular tool for measuring Java code performance, offering features such as automatic warmup, precise timing, and multithreading support. It allows for customization of benchmark configuration and running benchmarks multiple times with different parameters to provide comprehensive results. While primarily geared towards microbenchmarking, JMH is a powerful tool widely adopted by

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

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584695

developers and researchers. However, using JMH requires a deep understanding of Java and the JVM. [2, 7, 11]

2.2 Apriori Framework

Apriori is a data mining method used for discovering association rules [3, 8, 10]. It involves two steps: identifying itemsets with sufficient support and generating association rules by combining frequent itemsets. The resulting output is a list of discovered associations and their corresponding confidence levels, computed as the ratio of occurrence of the association to the occurrence of the antecedent.

$$Confidence(itemA \rightarrow itemB) = \frac{Records\ that\ Both\ A\ and\ B\ occurred}{Records\ that\ just\ A\ Occurred} \quad (1)$$

3 METHODOLOGY

In this section, we present a detailed explanation of our methodology which is broken down into three steps to address the research question:

Research Question: Is there a correlation between the ability of benchmarks to reach steady state and their static features from source code? If so, what are these features?

To investigate the potential correlation between the ability of benchmarks to reach steady state and their static features from source code, we first extract the static source code features of the microbenchmarks under examination. Next, we use manual mining techniques to identify any potential correlations or patterns between the extracted static features and the benchmarks' steady state. To further validate our findings, we apply a descriptive data mining method.

The following subsections will detail these three steps:

- **Step 1:** We extract the static source code features of the microbenchmarks under examination.
- **Step 2:** We use manual mining techniques to identify any potential correlations or patterns between the extracted static features and the benchmarks' steady state.
- **Step 3:** We confirm our findings from step 2 by utilizing a descriptive data mining method.

3.1 Feature Extraction (Step 1)

The *primary dataset*[1] we are utilizing comprises 586 microbenchmarking results from 30 Java open-source projects such as RxJava, Log4J2, and Apache Hive, representing various project domains like application servers, libraries, and databases [11]. Each project in the dataset has at least one Java benchFile that contains methods whose performance is being tested. After submitting these benchFiles to the Java Microbenchmark Harness (JMH), a commonly used microbenchmarking framework in the Java community [7, 11], it produces timeseries data for 10 different *forks* related to each method under test (Fig. 1). However, the existing dataset only comprises the benchmarks' results, including information pertaining to their time series and steady state status. It does not contain information regarding the benchmarks' static source code features, which are the focal point of our investigation. To rectify this, we employ

Figure 1: Illustration of the dataset structure

source code exploration tools such as srcML [9] and Lizard[2] to extract 11 static source code features (Table 1). These additional features are integrated into the primary dataset to form a new dataset that we will analyze, referred to as the *extended dataset*. There are 27 records not included in the extended dataset, representing a 3% reduction. This exclusion was a result of errors encountered during the extraction of features utilizing our tools. The new dataset, source codes, and other results are accessible from our GitHub repository[3].

As shown in Table 1, in the new dataset we have categorized the features into two main groups: *static* and *dynamic*. The static features are derived from the source code of the benchmarks, while the dynamic features provide information on the steady state situation of the benchmarks. Furthermore, the static category is divided into two subcategories: benchMethod and benchFile features. The benchMethod features pertain to the source code characteristics of the benchmark's method under test (benchMethod), while the benchFile features pertain to the source code characteristics of the benchmark's file (benchFile). It is essential to note that the static features provide insight into the structure and complexity of the code being tested, while the dynamic features provide information about the benchmarks' behavior during execution and are critical in understanding the performance characteristics of the code.

The dynamic features are extracted directly from the benchmarks' time series in the primary dataset. Among the three dynamic features presented in Table 1, DNNSF is particularly significant as it determines if a benchmark has reached steady state. A benchmark is considered steady when the number of its non-steady forks (forks that did not reach steady state in the last 500 iterations) is zero [11]. The remaining two features, DANCP and DANITS, provide information on the fluctuation of the benchmark's time series and the average number of iterations taken for the benchmark to reach steady state.

[1] https://github.com/SEALABQualityGroup/icpe-data-challenge-jmh

[2] https://pypi.org/project/lizard/

[3] https://github.com/amirmahdiKhosravi/ACM-SPEC-ICPE-2023-DataTrack

Static	**benchMethod Features**	• *BMLOC: Number of Lines of Code* • *BMCCN: Cyclomatic Complexity Number* • *BMNTC: Number of tryCatch* • *BMNSC: Number of switchCase* • *BMNFL: Number of forLoops* • *BMNWL: Number of whileLoops* • *BMNFC: Number of functionCalls*
	benchFile Features	• *BFLOC: Number of Lines of Code* • *BFCCN: Cyclomatic Complexity Number* • *BFNP: Number of Packages* • *BFLN: benchFile's Name-Length*
Dynamic	• *DNNSF: Number of no_steady Forks* • *DANCP: Average Number of Changing Points of Forks* • *DANITS: Average Number of Iterations to Steady State among Forks*	

Table 1: An overview of features

3.2 Manual Mining (Step 2)

Having prepared the extended dataset, it is now feasible to investigate the potential associations between the features, specifically between the static and dynamic features. We conduct our investigation at two different levels: 1) the benchMethod level, where we compare the static benchMethod features with the dynamic features, and 2) the benchFile level, where we compare the benchFile features with the dynamic ones. This approach allows us to gain a comprehensive understanding of the relationship between the source code characteristics and the performance characteristics of the benchmarks. Additionally, examining the associations at both the benchMethod and benchFile levels enables us to identify patterns and correlations that may not be apparent at one level alone.

benchMethod Level We classify benchMethods into three categories for comparison to gain insight into potential associations. These categories are: 1) *Steady benchMethod*: All forks are steady (in steady state for the last 500 iterations). 2) *Inconsistent benchMethod*: At least one fork is non-steady (did not reach steady state in the last 500 iterations). 3) *Non-steady benchMethod*: Benchmarks with the highest number of non-steady forks, which is 9.

Table 2 illustrates the three benchMethod categories, which we introduced earlier. Since each category might include more than one record, we have computed the average of the features of each category (look at first three rows of the Table 2). The last two rows of the table display the comparison results of two chosen categories from the first three rows.

To determine which two categories to compare, we selected those that would yield meaningful results for our goal. The first two categories we examined were non-steady and steady benchMethods, which are opposite to each other. The comparison of their differences (fourth row of Table 2) illustrates that BMLOC and BMNFC have the lowest values among the static features, with -2.46 and -1.65 respectively, indicating that on average, steady benchMethods have more lines of code and function calls compared to non-steady benchMethods. These two features also show a lower increase in value when moving from *diff(non-steady, steady)* to *diff(non-steady, inconsistent)*. Additionally, the dynamic feature results in the last two rows of the table show that non-steady benchMethods have a higher number of changing points in their time series, indicating that they are more unstable than the other two categories.

benchFile Level Similar to benchMethod Level, we introduce three categories for benchFiles to aid in understanding potential associations. These categories are: 1) Steady benchFile: All benchMethods within it are steady (no non-steady forks). 2) Non-steady benchFile: More than half of the benchMethods within it are non-steady (at least one non-steady fork). 3) Inconsistent benchFile: Neither steady nor non-steady.

Table 3 illustrates the three benchFile categories and the average of their features, as each category might include multiple records. The last two rows of this table depict the comparison of the established categories.

To decide which categories to compare for differences (last two rows of Table 3), we selected non-steady, steady and inconsistent, similar to our decision for benchMethod categories. The comparison shows that BFLOC and BFCCN do not show a consistent pattern, fluctuating between negative and positive values. BFNL does not show a significant difference, with values close to 0.00 in both comparisons. BFNP is the only feature with a considerable difference in *diff(non-steady, steady)* and *diff(non-steady, inconsistent)*, with a negative value indicating that on average, non-steady benchFiles have more packages than steady and inconsistent benchFiles. The dynamic features in the last two rows of the table also show that non-steady benchFiles have greater DANITS than the other two categories, indicating that it takes more iterations for benchmarks inside them to reach a steady state.

Our manual mining results suggest potential associations between three static features (BMLOC, BMNFC and BFNP) and the most significant dynamic feature (DNNSF). Here are our three assumptions about these associations:

(1) *BMNFC and DNNSF:* BenchMethods with more function calls are more likely to reach steady state.
(2) *BMLOC and DNNSF:* BenchMethods with more lines of code are more likely to reach steady state
(3) *BFNP and DNNSF:* BenchMethods in benchFiles with more packages are more likely to reach steady state.

3.3 Descriptive Mining (Step 3)

In this section, we employ the Apriori algorithm [3, 8, 10] to validate our assumptions. The Apriori algorithm is a framework that allows for investigating potential associations between various combinations of items by providing metrics such as confidence.

To utilize the Apriori framework, we defined a set of items based on our assumptions about potential associations. For two of our nominated static features (BMLOC and BFNP), we have defined two related items, HIGH and LOW, representing if the number related to the feature is higher or lower than the average, respectively. For BMNFC, another nominated static feature, we have defined three items: HIGH, AVERAGE, and ZERO, indicating if the number related to BMNFC is higher or equal to average and if it is equal to zero. We also considered a set of three related items for our dynamic feature under study (DNNSF): *ZERO* ($DNNSF = 0$), *LOW* ($0 < DNNSF < 6$), and *HIGH* ($DNNSF \geq 6$) for the Apriori algorithm to investigate their associations.

Table 4 presents the associations identified by the Apriori algorithm. As observed in the Table, there is no association related to BFNP and DNNSF items, indicating that there is no meaningful

	Number of Records	Static benchMethod Features							Dynamic Features	
		BMLOC	BMCCN	BMNTC	BMNSC	BMNFL	BMNWL	BMNFC	DANCP	DANITS
non-steady benchMethod	2	5.00	1.00	0.00	0.00	0.00	0.00	3.00	23.30	72.50
Inconsistence benchMethod	236	6.70	1.59	0.09	0.00	0.25	0.16	3.91	9.29	484.72
Steady benchMethod	321	7.46	1.68	0.11	0.00	0.21	0.14	4.65	7.37	221.21
diff (non-steady, Steady)		-2.46	-0.68	-0.11	0.00	-0.21	-0.14	-1.65	15.92	-148.71
diff (non-steady, Inconsistence)		-1.70	-0.59	-0.09	0.00	-0.25	-0.16	-0.91	14.00	-412.22

Table 2: The average of static benchMethod features and dynamic features for each of three benchMethod category, and their comparison.

	Number of Records	Static benchFile Features				Dynamic Features	
		BFLOC	BFCCN	BFNP	BFLN	DANCP	DANITS
Steady benchFile	96	119.50	1.79	18.65	27.40	7.51	221.24
Inconsistence benchFile	17	132.95	1.36	21.08	27.83	8.29	236.93
no_steady benchFile	94	127.45	1.61	16.12	27.89	8.69	448.60
diff (non-steady, Steady)		7.94	-0.17	-2.52	0.48	1.18	227.36
diff (non-steady, Inconsistence)		-5.50	0.25	-4.95	0.05	0.40	211.67

Table 3: The average of static benchFile features and dynamic features for each of three benchFile category, and their comparison.

	Confidence
(BMNFC_HIGH -> DNNSF_ZERO)	0.63
(BMNFC_AVERAGE -> DNNSF_LOW)	0.39
(BMLOC_HIGH -> DNNSF_ZERO)	0.61
(BMLOC_LOW -> DNNSF_LOW)	0.40

Table 4: List of associations and their confidence, found by Apriori.

association between these two features, thus negating our third assumption outlined in Section 3.2. However, the Table reveals associations among the remaining items. Two of these associations exhibit a confidence level greater than 0.50, which will be discussed in further detail in the following paragraphs.

Based on the first row of the Table, we can infer that there is an association between benchMethods with a high number of function calls and having zero number of non-steady forks. This correlation has a confidence level of 0.63. The confidence level indicates that there is a likelihood of 63% for a benchmark to be steady (with zero number of non-steady forks) when the number of function calls for its benchMethod is high. This supports our first assumption in 3.2.

As per the third row of Table 4, there is an association between high number of lines of code in benchMethods and zero number of non-steady forks. The confidence level of this association indicates that a benchmark with a high number of lines of code is likely to be steady (zero non-steady forks) with a likelihood of 61%. This supports our second assumption of Section 3.2.

Although the associations in the second and last rows of Table 4 have a confidence level lower than 0.5, they provide supplementary evidence to our findings by highlighting a correlation between BMNFC, BMLOC with DNNSF.

Findings Our study reveals a significant correlation between certain static features of the benchmark's source code and the benchmark's ability to reach steady state. These features are BMNFC (number of function calls in the benchMethod) and BMLOC (number of lines of code in the benchMethod). The results indicate that the higher the number of function calls and LOC a benchMethod has, the more likely it is to reach steady state.

4 CONCLUSION AND FUTURE WORKS

The aim of this study was to investigate the potential correlation between static features of a microbenchmark's source code and the microbenchmark's ability to reach steady state. To achieve this, we first expanded our primary dataset by extracting static features and incorporating them into that. We then utilized manual and descriptive data mining methods to identify and establish potential correlations. Our findings indicate that there is an association between the microbenchmark's ability to reach steady state and two static features extracted from its source code (out of the 11 extracted), namely the number of function calls (BMNFC) and lines of code (BMLOC). The greater values of these features, the greater likelihood of reaching steady state. In future work, it may be beneficial to investigate additional features representing different types of function calls (e.g., system calls, threading, locks, etc.) to further evaluate their impact on steady state.

5 ACKNOWLEDGEMENT

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Mitacs, Ciena and Bornea Dynamics for funding this research.

REFERENCES

[1] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S., and Tratt, L. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–27.

[2] Costa, D., Bezemer, C.-P., Leitner, P., and Andrzejak, A. What's wrong with my benchmark results? studying bad practices in jmh benchmarks. *IEEE Transactions on Software Engineering 47*, 7 (2019), 1452–1467.

[3] Dongre, J., Prajapati, G. L., and Tokekar, S. The role of apriori algorithm for finding the association rules in data mining. In *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)* (2014), IEEE, pp. 657–660.

[4] Gu, D., and Verbrugge, C. Phase-based adaptive recompilation in a jvm. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), pp. 24–34.

[5] Laaber, C., Basmaci, M., and Salza, P. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering 26*, 6 (2021), 1–53.

[6] Laaber, C., Würsten, S., Gall, H. C., and Leitner, P. 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.

[7] Leitner, P., and Bezemer, C.-P. 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.

[8] Liu, Y. Study on application of apriori algorithm in data mining. In *2010 Second international conference on computer modeling and simulation* (2010), vol. 3, IEEE, pp. 111–114.

[9] Maletic, J. I., and Collard, M. L. Exploration, analysis, and manipulation of source code using srcml. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 2, IEEE, pp. 951–952.

[10] Ranjan, R., and Sharma, A. Evaluation of frequent itemset mining platforms using apriori and fp-growth algorithm. *arXiv preprint arXiv:1902.10999* (2019).

[11] 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 Software Engineering 28*, 1 (2023), 1–57.