Università degli Studi di Milano Bicocca

Università della Svizzera Italiana

Double Degree Program

Master Degree in Computer Science

Analysing Performance Variability in Applications running on the Java Virtual Machine

Advisor: Rosà Andrea

Advisor: Denaro Giovanni

Co-advisor: Bulej Lubomir

Relazione della prova finale di:

Mattia Biancini

Matricola 865966

Anno Accademico 2024-2025

"Se ancora non vuoi rinunciare al tuo desiderio,
se ancora quel castello ti appare più brillante di qualsiasi altra cosa,
prosegui sulla tua strada."
Kentaro Miura

Abstract

Performance variability is a well-known detrimental phenomenon, which is considered a long-standing, important and open research problem in both academia and industry. Variability has been observed to induce severe and unanticipated performance degradation in diverse instances of the same application, with no readily discernible explanations. It is important to note that a significant proportion of programs are susceptible to variability, even in the context of single-threaded execution, where the input remains constant between executions and the environment is stable. Consequently, this phenomenon cannot be disregarded.

The present work addresses this issue by seeking to comprehend and mitigate variability, with a particular emphasis on applications operating within managed runtimes. It is acknowledged that a considerable number of studies have been conducted on this phenomenon. However, it is notable that none of these studies have addressed a scenario where all potential forms of variability have been excluded.

We use two state-of-the-art benchmark suites (*DaCapo* and *Renaissance*) to perform the analysis of variability and the categorisation of patterns presented by distinct workloads. The utilisation of ProfDiff and JitWatch facilitates the comprehension of the underlying causes of such phenomena from the perspective of the compiler. The removal of the C1 layer, in conjunction with the eventual augmentation of the inlining size, was found to be an effective strategy for the mitigation of variability across a range of workloads.

Contents

Chapter 1

Introduction

Performance Variability (hereafter referred to as simply *variability*) in computing systems has emerged as a critical challenge in modern informatics, particularly as systems grow in complexity and scale. Variability manifests itself in unpredictable execution times, inconsistent throughput, and fluctuating resource utilization, even under seemingly identical conditions and on the same hardware [29]. This phenomenon is observed across diverse computing environments, including High-Performance Computing (HPC) clusters, cloud infrastructures, and edge computing systems. While some degree of variability is inherent due to non-deterministic factors such as hardware heterogeneity, operating system scheduling, and network latency, it remains an open research problem in both academia and industry, necessitating further investigation to understand and mitigate its effects.

In modern computing environments, programs increasingly exhibit non-deterministic behaviour [25] due to frequent software updates [4] and the pervasive use of shared hardware resources. Software updates, while essential for security and functionality, can introduce variability by altering system libraries, dependencies, or runtime environments. Simultaneously, the competition for shared hardware resources—such as CPU cores, memory bandwidth, and I/O channels—creates race conditions that further exacerbate performance unpredictability. This non-determinism poses significant challenges for operators of HPC resources, as it becomes difficult to discern whether observed performance improvements or degradations are the result of intentional optimisations or merely artifacts of underlying variability [49]. Consequently, operators must rely on sophisticated monitoring and analysis tools to isolate the effects of optimisations from the noise introduced by these dynamic factors.

The study of variability is particularly relevant in the context of modern programming languages and runtime environments, such as the Java Virtual Machine (JVM) and GraalVM. Java, as one of the most widely used programming languages, powers a vast array of applications, from enterprise systems to big data processing frameworks. However, the performance of Java applications can vary significantly depending on the runtime environment, Just-In-Time (JIT) compiler optimisations, and garbage collection strategies. GraalVM, with its advanced polyglot capabilities and optimising compiler, introduces additional layers of complexity and potential variability. Understanding how these factors interact to influence performance is crucial for developers and system operators alike.

This thesis focuses on identifying patterns of performance variability using the DaCapo-Chopin 23.11-MR1 [10] and Renaissance 0.16 [37] benchmarks, which are widely recognized for their ability to simulate real-world workloads in Java applications. By analysing these benchmarks this research aims to uncover the root causes of variability and provide actionable insights for optimising performance. The findings will contribute to a deeper understanding of how modern runtime environments and compilers influence performance predictability, enabling more efficient resource utilization and improved system reliability. My aim is not focusing on environments where variability is driven by external factors like network latency, dynamic compute-node technologies, or I/O operations, I concentrate on a tightly controlled experimental setup. In this setup, I eliminate external known sources of variability [39, 7, 12] to study a specific type of performance variability: variability that is not intrinsic to the workload, algorithm, or data source. Specifically, I target applications that exhibit variability even when, across different executions, they perform the same work on the same input, without engaging in any storage or network operations, and with no other CPU-intensive applications running concurrently on the system. To investigate this phenomenon, I conduct experiments using Java [23] and GraalVM [18] versions 23 and 24, running the DaCapo [8] and Renaissance [36] state-of-the-art benchmark suites. These benchmarks are chosen for their diverse workloads, which allow us to observe variability across a range of computational patterns. My work has three primary objectives:

1. Quantify the existence of variability in a controlled environment where external factors are excluded.
2. Recognize and categorize patterns of variability that emerge during benchmark execution.
3. Investigate the root causes of these patterns, focusing on JVM and GraalVM internals such as Just-In-Time (JIT) compilation, garbage collection, and thread scheduling.

A significant challenge in this work was instrumentation. Using the default profdiff tool [17] introduced additional latency, which could obscure or distort the observed variability patterns. This limitation necessitated the development of alternative instrumentation strategies for JVM-based benchmarks. Furthermore, the use of this tool is limited to the Graal suite and consequently it was not possible to instrument the code for the JVM as well.
To summarise, my work makes the following contributions:

- Using DaCapo and Renaissance, I analyse variability on different workloads and categorize them into patterns (Chapter 4).
- Through some optimisation, I explain how variability can be mitigated (Chapter 5).

To complement my work, I present:

- The state-of-the-Art (Chapter 2). I introduce related work on variability mitigation at a higher level, as well as other studies on variability that highlight the importance of this phenomenon from academic and industrial perspectives.
- Background information (Chapter 3). I explain the structure of the compiler, what it does and the non-determinism of its optimisations.
- Further Discussion (Chapter 6). I discuss the limitations of the work carried out, including the choices that were not made and the issues that remain unresolved. Finally, I also outline future work.

- My final remarks (Chapter 7).

Chapter 2

State-of-the-Art

Performance variability in Java Virtual Machine (JVM) and GraalVM environments represents one of the most challenging aspects of modern software performance engineering. Conversely, managed runtime environments exhibit a higher degree of complexity, which can lead to substantial and frequently unpredictable performance variations. This is in contrast to conventional native applications, where performance characteristics tend to be more predictable. This variability poses significant challenges when conducting rigorous performance evaluations, optimising applications, or ensuring consistent service-level agreements in production environments [31].

This chapter provides a comprehensive survey of the state-of-the-art in understanding, measuring, and managing performance variability in JVM and GraalVM environments. A close examination of closely related works that have addressed various aspects of this challenge is undertaken, with a particular focus on the intrinsic variability within managed runtime systems. In the ensuing sections, an exploration is conducted of both localized and holistic approaches to analysing and mitigating performance variability, with a focus on key methodologies and findings from recent research.

2.1 Variability in Managed Runtime Environment

2.1.1 Measurement Bias and Experimental Rigour

Measurement bias represents a significant and prevalent issue in the context of computer system evaluation. During their studies, Georges et al. [15] found that bias occurs in all architectures used, concluding that this phenomenon cannot be entirely avoided. In order to reduce the time required of researchers, two techniques were developed for the avoidance and detection of this bias:

- *Experimental setup randomization*, which involves running experiments in multiple different setups to eliminate or reduce bias through the distribution of observations.
- *Causal analysis*, which allows researchers to detect and validate the outcomes of performance analysis.

These techniques provide a framework for ensuring the reliability of experimental results in performance studies.

Barrett et al. [6] pioneered a fully automatic statistical approach based on change-point analysis. This methodology was developed for the purpose of determining whether a steady state has been reached and, if so, whether it represents peak performance.

Their findings revealed that only a small portion of deterministic processes (30.0% - 43.5%) reach a steady state and peak performance, which poses a problem for many real-world VM benchmarking practices that assume steady-state performance is always achieved.

2.1.2 Minor Performance Variation Studies

Improving the details of a benchmark remains a critical task, as real-world programs must evolve over time. These changes inevitably lead to performance fluctuations. Sandoval et al. [4] conducted a comprehensive study of performance evolution, analysing how source code changes most significantly impact program performance during software evolution. Their focus on the Pharo ecosystem [34] was driven by its tooling, hooks, and expressive reflective API, which enabled detailed measurement of benchmark performance across hundreds of software versions. Their findings include:

- Only 2% of application revisions have commit messages related to performance (indicating developers' lack of awareness of performance fluctuations they introduce).
- Approximately 1 out of 3 revisions introduces performance variations.
- The most prominent cause of performance regression is the composition of collection operations.

VMs employ techniques to manage dynamically allocated objects, but even when allocation is efficient, it incurs overhead. Escape analysis can determine whether an object needs to be allocated at all and whether its lock might be contended. In many cases, an object escapes due to a single unlucky branch, preventing optimisations. Stadler et al. [41] proposed a flow-sensitive Escape Analysis (*Partial Escape Analysis* (PCA)) that optimises objects that do not escape and ensures their existence in heap branches where they do escape. Integrating PCA into the Graal compiler reduced memory allocation by up to 58.5%, improving performance by up to 33%.

Container technology is undergoing rapid development and is being widely adopted in large-scale production environments. Containers facilitate the creation of virtual, isolated spaces within which users can run their applications. They offer functionality analogous to that of virtual machines (VMs), yet are characterised by a more streamlined operational profile. Ye and Ji [50] investigate the performance implications of utilising Docker Containers in conjunction with Spark applications. It has been determined that the configuration parameters of the containers have a substantial impact on the performance of the application that is executed on them. In order to achieve optimal performance, a performance prediction model was developed. This model was based on SVR (Support Vector Regression), and resulted in a prediction error of less than 10% for all typical Spark applications.

2.1.3 Rigorous Statistical Approaches

While certain researchers concentrate on particular sources of variability, others adopt a more comprehensive approach to understanding the phenomenon. There is a consensus that multiple runs across different environments are necessary, and that a more rigorous statistical approach is essential. Kalibera and Jones [25] introduced a novel mathematical framework that has been termed a "cookbook" by virtue of its function as a compendium of methods for determining the number of repetitions required to obtain reliable results.

The researchers then compared their method, which was based on effect sizes and confidence intervals, to heuristic-based practices. The latter often led to an insufficient or excessive number of experiments.

As demonstrated by Georges et al. [16], the execution of software varies significantly depending on the compilation plan employed. The researchers proposed the use of matched-pair comparisons for the purpose of analysing performance data from replay compilations, utilising multiple plans. The findings of the present study suggest that, for a given experimental time budget, the consideration of a greater number of compilation plans rather than an increase in the number of runs per plan is more beneficial.

Gu et al. [19] adopted a different approach, focusing on critical factors such as instruction workload, hash code location, code positioning, and data location. The researchers provided quantitative and comparative analyses of potential confounding factors in JVM performance assessments. The culmination of their efforts was an optimisation case study on a novel garbage collection optimisation, which resulted in enhanced GC performance in both JIT and interpreter environments.

In a world where machine learning is playing an increasingly important role, Laaber et al. [26], using machine learning algorithms, predict benchmarks stability before executing the code. In their study they used more than 4000 benchmarks executions with 11 different algorithms with different threshold to classify stability and different number of iteration. They find that increasing iteration leads to better prediction. The same happens for threshold size. Another stunning fact is that to achieve such a results they only 7 features are individually important for good predictions. Furthermore, the features concerning the called source code are collectively the most important. Although, source code is not the only reason of variability, these features can be effectively used to predict whether a benchmark is stable or not.

2.2 Performance Variability in different Environment

Performance variability is not only a phenomenon related to Managed Runtime Environment, indeed is more documented and studied on systems where some form of variability is expected.

2.2.1 Variability in High-Performance Computing (HPC) systems

HPC systems are widely used nowadays. Variability has an high impact on this systems: the longer the task takes, the more time it takes to get usable results for analysis. This has a slippery slope effect on HPC application because a single task can slow down the entire system. This problem impacts both users and developers. The first one is affected by unpredictable performance therefore measuring and comparing performances of different programs is more difficult. The second may not see the benefit of an optimisation hidden by the background performance variance.
Another significant finding in comprehending the causes of variability in HPC systems has been made by Bhatele et al. [7]. Utilising the pF3D application, it was determined that jobs exceeding the system's designed size can result in variability reaching up to 27.8% in the presence of other large jobs. However, this is not attributable to the manner in which the job is divided, but rather to the interference caused by other jobs. It is therefore imperative to select the most suitable platform for the task at hand.

Skinner and Kramer [39] examinate variability in HPC workloads. Variability is divided

in two macro-causes: architectural and application causes. They detect, measure and address the causes of performance variability on specific workloads, underling that generalizing possible solution for specific workloads lose meaning.

Maricq et al. [**maricq_taming_nodate**] use a different approach to classify variability. They consider two types:

- variability of the same physical system under repeated experiments
- variability between different physical systems that are supposed to be identical.

The causes of variability that are considered are manifold, but the most salient are: the temperature of the hardware, variations in timings and ordering, remapping of storage blocks or memory cells, variance in manufacture and "fail-slow" hardware.
They develop CONFIRM, a tool designed to estimate the number of iterations required for a run to assist experimenters in gathering statistically significant results. This instrument facilitates the design of experiments that are more efficacious in determining the representativeness of servers, and subsequently enables the execution of measurement analysis upon the conclusion of experiments.

Wright et al. [49] investigate if how often variations in runtime occur and look at the distributions of runtimes obtained. They also examine if there is a correlation between certain set of nodes and same performance impediment.

Tang et al. [42] propose vSensor, a novel approach for light-weight and online performance variance detection. It can sense performance variance at runtime without introducing additional overhead. Leveraging vSensor inside programs to detect performance variance brings three benefits:

- No performance model is required
- Low overhead and interference
- v-sensors make it easier to locate the root causes of performance variance

Another machine learning approach is developed by Tuncer et al. [45], who notice that all the previous form of variability measurement and mitigation still rely on human operations, so they propose a framework to automatically detect the compute nodes in HPC system where the phenomenon is observed. Their framework uses machine learning algorithms on resource usage and performance metrics, succeeding in 98% of the time while leading to 0.08% of false positives.

Finally, Dorier et al [12] focus on a specific variability cause, I/O operations. A study was conducted on three distinct coordination strategies: interfering, serialising and interrupting. It was observed that each of these strategies is suboptimal in different contexts. Consequently, the CALCioM (Cross-Application Layer for Coordinated I/O Management) framework was developed. This innovative framework has the capacity to select the most appropriate one for achieving targeted machine-wide efficiency. The CALCioM framework has the capacity to suspend I/O activity in order to facilitate the execution of another application. Furthermore, it is capable of awaiting the completion of I/O activity by another application, or alternatively, it can continue to access the file system despite the presence of another application that is also accessing the same file system. This strategy is predicated on a holistic perspective of the set of running applications and their respective I/O activity, as perceived from each level of the I/O stack. The objective is to optimise a specified metric of machine-wide efficiency.

2.2.2 Variability in Cloud Computing Environments

Cloud Computing introduces a new cause of variability, network latency, due to traffic ingestion most likely.

Iosup et al. [22] analyse causes and effects of long-term variability in cloud computing environments. They find out that causes are several: system size, workload variability, virtualization overhead and resource time sharing. Analysing, over the years, time-dependant characteristics they find that there are yearly and daily patterns of variability and even periods of stable performance. Secondly, they find that the impact of the performance varies greatly across application types.

To add an order to forms of variability, Leitner and Cito [27] categorize variability in patterns not only related to date and time under 15 different hypotheses based on the previous published peer-reviewed literature that they validate. The hypotheses are based on temporal time (hour of the day and day of the week), CPU and I/O hardware and application bounding, region of server location and scalability. They conclude their work asserting that different cloud providers grant different performance and their tendency to keep improving and excelling on others requires updates more frequently than others areas.

Dean and Barroso [43] conducted a comprehensive investigation into the various causes of variability present within the context of large-scale web services. These causes were methodically categorised into two overarching macro-categories: namely, hardware causes and software causes. The objective of this research is to develop a system in which the user perceives all processes as fluid, with response times of less than 100 milliseconds. To achieve such a result they develop fault-tolerant and tail-tolerant techniques yielding latency improvement.

Analysing previous work, Abedi and Brecht [1] discover that the methodologies falsely report that for two identical system the performance vary up to 38% with 95% of confidence level. The absence of meticulousness in the comparison of two or more competing alternatives is a point of contention. The phenomena that influence variability can also be unpredictable, such as the impact of shared resources and the work of other applications on the same physical device. The utilisation of a randomised multiple interleaved trials methodology is proposed as a means of attaining repeatable performance metrics. This approach is posited as a foundation for the equitable comparison of competing alternatives, with the temporal arrangement of trials being arbitrarily assigned to ensure equitable comparison.

Another significant challenge associated with cloud benchmarking is the issue of costs. It is evident that Brute-forcing tests can incur a significant financial expense. Consequently, a design-based strategy can be employed to reduce the financial burden on environments. Wang et al. [48] developed a smart test oracle, namely CAPT (Cloud Application Performance Tester), which consists of three major components. Firstly, a test-based approach is employed for the purpose of characterising the per-resource performance distribution of a cloud. Secondly, a cost-effective approach to test case execution is proposed. Thirdly, a smart test oracle is required to estimate the application's performance on the target cloud using the results from the first two components. The findings demonstrate that CAPT is capable of accurately estimating the performance of various applications, with an average error rate of 4.9%. Furthermore, it has been shown to reduce testing costs by 66.9% on average.

Laaber et al. explore the effects of cloud environments on the variability of performance

test results and to what extent slowdowns can still be reliably detected even in a public cloud, focusing on software microbenchmarking. It is evident that the result variability undergoes a transition from a coefficient of variation of 0.03% to more than 100% when the same experiments are repeated on multiple occasions. This phenomenon can be attributed to the specific benchmark and the environment in which it is executed, leading to its categorization as follows:

- Variability inherent to the benchmark
- Variability between trials
- Variability between instances

The study was concluded with the assertion that only 77%–83% of benchmark-environment combinations are capable of reliably detecting slowdowns using trial-based sampling.

2.3 Variability in Parallel or Multi-Tenant Computer Systems

Co-processing is becoming increasingly prevalent in modern architectures. However, due to the non-deterministic nature of thread scheduling, a new cause of variability arises: different thread scheduling can significantly alter the performance of a benchmark, leading to misleading results.

In the context of concurrent or multi-threaded applications, two divergent approaches have been adopted. In their study, Lepak et al. [28] sought to eradicate the sources of non-determinism in multi-threaded programs, with the objective of attaining determinism. In order to obtain such a result, the performance analysis is based on the number of instructions per cycle (IPC), a deterministic value that is stable across different runs. In contrast, Heirman et al. [20] acknowledge the prevalence of non-determinism, recognising its manifestation in sequential benchmarks. In order to ensure the rigour of the analysis of benchmark results, reliance is placed on statistics in order to characterise and average out variability. The prevailing sentiment is that a solution remains elusive, yet the utilisation of low-level metrics is regarded as a potential pivotal element, given their relative resistance to variability.
This decision is made in recognition of the fact that a solution has yet to be found and that the scientific method requires the reproducibility of experiments. Hoefler and Belli [21] therefore opt to employ a less stringent form of reproducibility that is better able to account for the results obtained by different researchers. These results are not replicable due to variability and a lack of rigour in the description of the methodology. The concept of *interpretability* is introduced, and it is stated that *an experiment is interpretable if it provides enough information to allow scientists to understand the experiment, draw their own conclusions, assess their certainty and possibly generalise results.*
As we have seen, variability is a well-known phenomenon in real systems, but unfortunately it is often ignored in simulation experiments. This leads to incorrect architectural conclusions when applying the results to the real world. Alameldeen and Wood [3] propose a methodology for introducing variability into simulations by introducing pseudo-random perturbations on different simulations and standard statistical techniques. Through their work, it is possible to reliably determine whether it is safe to draw conclusions from simulation experiments.

In multi-tenant environments, the problem of variability observed in parallel benchmarks is mirrored by the fact that resources are shared and different jobs can be performed on the same machine by different users. The variable workload and the resources dedicated

to a tenant, which are not always constant, lead to further fluctuations in performance. The absence of performance isolation thus necessitates a more meticulous approach to performance analysis.
Shue et al. [38] identify two main issues:

- Multi-tenant interference and unfairness
- Variable and unpredictable performance

In order to enhance predictability, the focus is on achieving global max-min fairness with high utilisation. This ensures that no tenant is able to gain an unfair advantage over the others during system load periods. However, when certain tenants utilise less than their allocated resources, these resources can be redistributed to other tenants. The implementation of Pisces (Predictable Shared Cloud Storage) facilitates the establishment of an architecture that is characterised by its fairness, which is determined by users' needs and the weight, representing the resources required. These weights are subject to constant updating. The outcome is realised with reduced overhead (less than 3%) and without significant throughput degradation.
In a departure from previous approaches, Angel et al. [5] adopt a novel strategy to achieve a form of isolation by creating an abstraction of a dedicated virtual data centre (VDC). The term 'VDCs' is used to denote virtual data centres, which encapsulate end-to-end throughput guarantees that hold across distributed appliances and the intervening network. Pulsar, a tool developed by them, offers tenants their own VDC. Pulsar is constituted by a logically centralised controller that employs novel mechanisms to estimate tenants' demands and appliance capacities. The controller then allocates data centre resources based on flexible policies.

2.4 Contribution

The present thesis adopts a different focus to the aforementioned works. The focus of this study is not on variability that is intrinsic to the workload, the algorithm, or the data sources used. The objective of this study is to identify and examine those applications that demonstrate variability, even in instances where they execute the same work on the same input. It is crucial to note that these applications do not engage in any storage or network operations, and there are no other CPU-intensive applications in execution on the system. In other words, the focus is on scenarios where external and well-known causes of variability have been excluded.

Chapter 3

Background

In this chapter, the discussion focuses on the structural composition of the JIT compiler Section 3.1, with a particular emphasis on the concept of non-determinism. In the following section, the two tools employed to enhance the evaluation of variability in the experimental setting will be presented: ProfDiff Section 3.2 and JitWatch Section 3.3.

3.1 Compilers and Non-Determinism

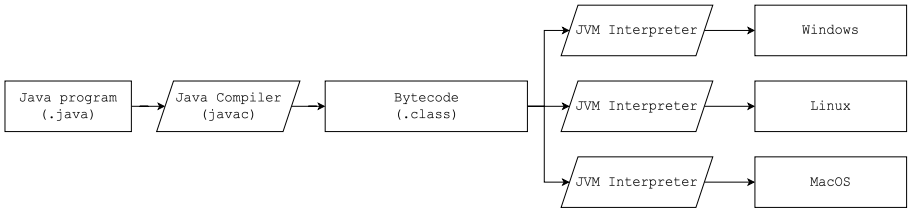


Figure 3.1: Java Compilation Process

Java is a programming language that is both compiled and interpreted [23]. The source code written in Java needs to be compiled before it can be executed on a machine (Figure 3.1). Using `javac` in command line a `.class` machine-independent file (called *bytecode*) is produced. It contains the specification of a Java class. This facilitates the portability of the code across disparate machines, obviating the requirement for the original source code to be shared.
The bytecode, when need to be executed, is initially interpreted by a HotSpot VM like JVM. In the Listing 3.1 it is possible to see what basically a JVM interpreter does.

```
do {
    byte opcode = fetch an opcode;
    switch(opcode) {
        case opCode1:
            fetch operands for opCode1;
            execute action for opCode1;
            break;
        case opCode2:
            fetch operands for opCode2;
            execute action for opCode2;
            break;
        ...
    }
} while(more to do);
```

Listing 3.1: Java Pseudocode of JVM interpreter

The advantages of this approach are:

- Easy to generate virtual machine code
- The code is architectural independent
- Bytecode can be more compact (macro operations)

The disadvantages, instead are:

- Every instruction is considered in isolation
- Poor performance due to interpretative overhead (tipically 5-20x slower than native code)
- Needs bytecode verification to detect illegal programs

3.1.1 Just-In-Time (JIT) Compilers

In order to improve performance, JIT compilers interact with the JVM at runtime and compile appropriate bytecode sequences into native machine code. Typically, a JIT compiler takes a block of code, optimises the code inside the block and then translates it into optimised machine code. A code block can consist of:

- Whole method
- Region of code
- Instruction trace

The potential advantages are:

- Targets specific architectural details
- Observes program properties only possible at runtime
- Efficiently allocates optimisation time towards important methods

Two competitive concerns are identified: the time taken for compilation and the amount of memory required, in contrast to the quality of the code. To maximize this efficiency only hot parts of the code are compiled:

- Methods that are frequently invoked
- Loops with many iterations

HotSpot ships with two JIT Compilers [24]:

- *C1* (Client Compiler) - It compiles methods quickly but emits machine code that is less optimised than the server compiler. This complier is used for quick startup. Also, in this compiler, the smaller memory footprint is more important than steady-state performance.
- *C2* (Server Compiler) - The compiler often takes more time (and memory) to compile the same methods. However, it generates better optimised machine code than the code generated by the client compiler. It provides better runtime performance after the application reaches the steady state.

Tiered Compilation

Despite the fact that the JVM functions with one interpreter and a pair of JIT compilers, there exist five potential levels of compilation. The rationale behind this is that the C1

compiler is capable of operating on three distinct levels. The distinction between these three levels lies in the extent of profiling conducted.

- *Level 0*: Interpreted Code - All the code is interpreted
- *Level 1*: C1 without Profiling - Code compiling for methods that are considered trivial
- *Level 2*: C1 with Basic Profiling - This level is used when C2 queue is full. Subsequently, the JVM undertakes a recompilation of the code on level 3, utilising full profiling. Subsequently, when the C2 queue is less busy, the JVM recompiles it on level 4.
- *Level 3*: C1 with Full Profiling - Level 3 is part of the default compilation path. Thus, the JVM uses it in all cases except for trivial methods or when compiler queues are full. The most common scenario in JIT compilation is that the interpreted code jumps directly from level 0 to level 3.
- *Level 4*: C2 - Code compiled to achieve maximum long-term performance. Generally speaking, the execution efficiency of C2 code is more than 30% higher than that of C1 code [40].

The most common pattern of compiling are the one shown in the Figure 3.2.

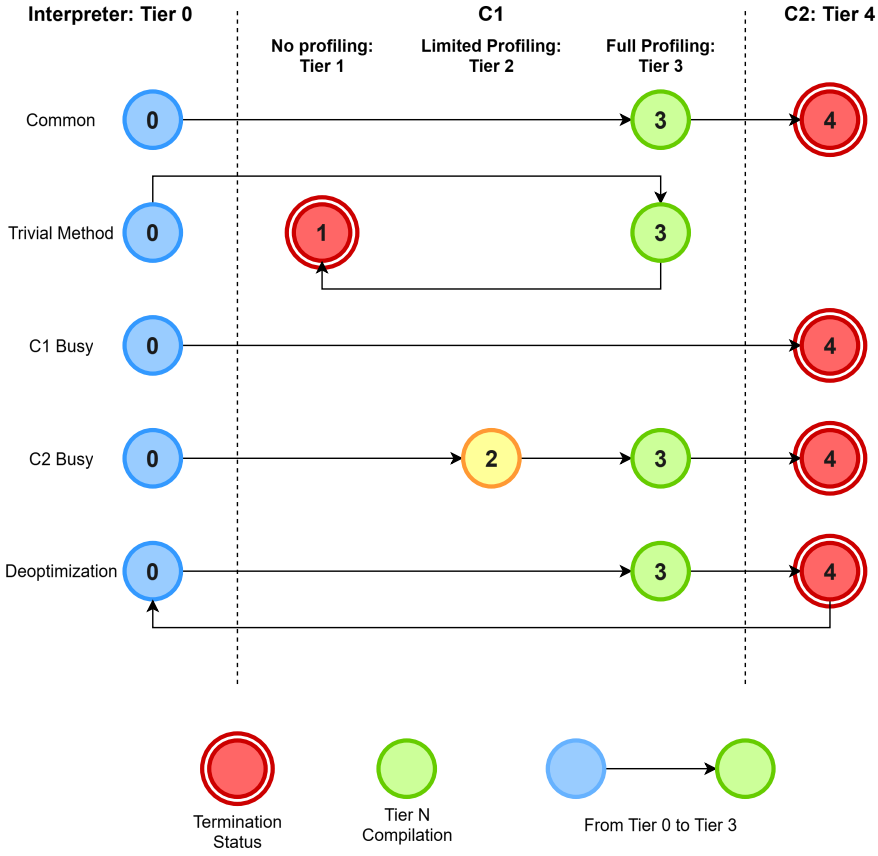


Figure 3.2: Common Compilation Paths - *This image is translated from* [40]

Graal JIT Compiler

Graal [35, 14, 18] constitutes a dynamic optimising compiler for the HotSpot JVM, engineered to supersede the default C2 compiler in specific contexts. Like C2, Graal employs feedback-driven optimisations that leverage runtime data—including receiver type profiles

and branch probabilities gathered during interpreter execution—to generate speculative code. These optimisations rely on behavioral assumptions during compilation, such as method receiver types or control flow paths. Crucially, the validity of these assumptions remains inherently uncertain. When subsequent evidence invalidates an assumption, Graal triggers deoptimisation: reverting to interpreted execution and scheduling method recompilation.

This compiler emphasizes intraprocedural analysis, optimising individual methods while enabling further optimisations through callsite inlining. Graal's speculative paradigm utilizes assumptions vulnerable to invalidation via class loading or runtime modifications (e.g., incorrect static receiver type predictions for virtual calls [13]). Deoptimisation functions not merely as a fallback (Figure 3.3), but as a critical mechanism reconstructing the interpreter's state (local variables, operand stacks) from compiled code contexts. This ensures consistency between optimised and interpreted states, permitting dynamic JVM adaptation while preserving system integrity. Graal thus balances aggressive code optimisation with runtime adaptability to achieve high-performance execution.

3.1.2 Optimisation and Deoptimisation: Comparing C2 and Graal

Just-In-Time (JIT) compilers optimise code by leveraging runtime execution profiles and speculative techniques. A critical divergence emerges in their deoptimisation handling when underlying assumptions are violated. Graal [18] immediately reverts to interpreter mode upon assumption failure, whereas C2 employs a conservative strategy requiring repeated deoptimisation events before reverting.

Zheng et al. [51] characterized recurrent deoptimisation patterns in Graal, proposing two mitigation strategies:

- A *conservative* approach delaying compiled code invalidation until observing sufficient deoptimisations (aligning with C2's philosophy)
- An *adaptive* approach dynamically selecting deoptimisation actions using precise runtime profiles

Empirical results demonstrated that the adaptive strategy reduces compilation overhead and yields statistically significant performance gains compared to conservative methods. This highlights a fundamental trade-off: Graal's default immediacy enables rapid response to changing runtime conditions, while C2's conservatism prioritizes compilation stability at potential performance cost during transitional phases. Both compilers utilize speculative feedback-driven optimisation, but their deoptimisation granularity and recovery mechanisms represent distinct design philosophies within the HotSpot ecosystem.

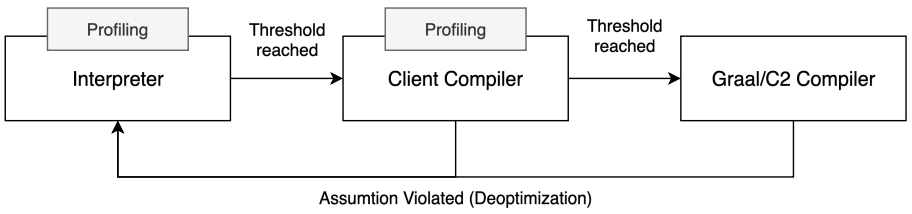


Figure 3.3: Deoptimisation Fallback - Image taken from [17]

3.1.3 Non-determinism in JIT Compilers

It is evident that JIT compilation offers numerous advantages with regard to average case performance in comparison with bytecode interpretation. In a real-time environment, the

primary disadvantage associated with this approach is its status as a novel source of non-determinism [9]. In the most intricate systems, a specific function may be compiled at unpredictable intervals. It is an inevitable consequence of the function being called for the first time (*cold* miss) that it will be compiled at that point. However, there are two further possibilities for its subsequent compilation.

1. In the event that the run-time system determines that an elevated level of optimisation would be advantageous.
2. In the event that the function has been evicted from the cache since the previous execution. This phenomenon may be attributed to either a capacity miss, signifying the eviction of a function due to an overloaded code cache, or a conflict miss, indicating the loading of a function with an overlapping address range since the previous function execution.

3.2 ProfDiff

ProfDiff [17, 32] is a tool that runs only on Graal compiler [18] and extends it with an option to collect and store optimisation logs.
For each compilation unit, the compiler generates an optimisation tree and an inlining tree. The utilisation of these data by ProfDiff facilitates the association of optimisation decisions with their inlining context, thereby leveraging the optimisation-context tree.
A superior compiler, such as Graal, will only compile methods for which the execution counter exceeds predefined thresholds. In the context of JIT, these methods are designated as hot. The ProfDiff approach is predicated on the observation of suboptimal optimisation decisions, which result in an augmentation of the execution time of the compilation unit. The consequences of these decisions are particularly pronounced in methods categorised as *hot*.
The identification of the most time-consuming compilation units is achieved through the utilisation of ProfDiff, which employs the proftool [30] to analyse the execution program, a profiler based on the perf tool [33]. Proftool is a tool that analyses the execution time of a program, both in the virtual machine and in the generated code. ProfDiff designates a configurable number of compilation units with the highest execution shares as hot.

3.2.1 Executing and Comparing a run

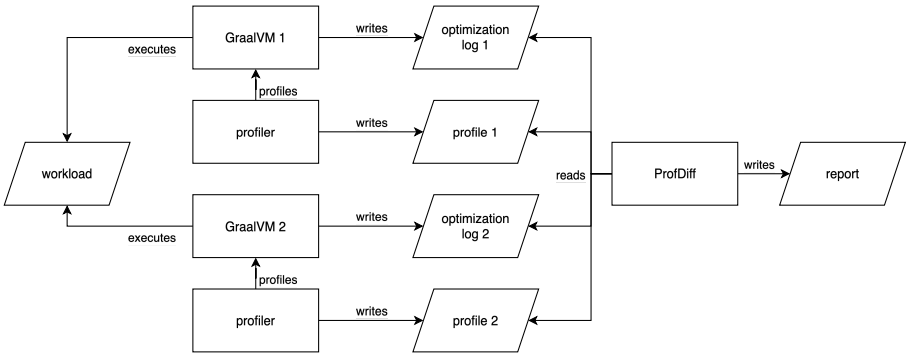


Figure 3.4: Executing and Comparing two Experiments - Image taken from [17]

As demonstrated in Figure 3.4, the illustration presents a scenario in which two instances of the same workload are executed and subsequently analysed through the utilisation of the ProfDiff tool. As is evident from the divergent nomenclature employed by the Graal

compilers, a comparison of runs across different versions of the compilers is indeed feasible. The comparison is restricted to hot compilation units in order to avoid reporting likely unimportant differences.

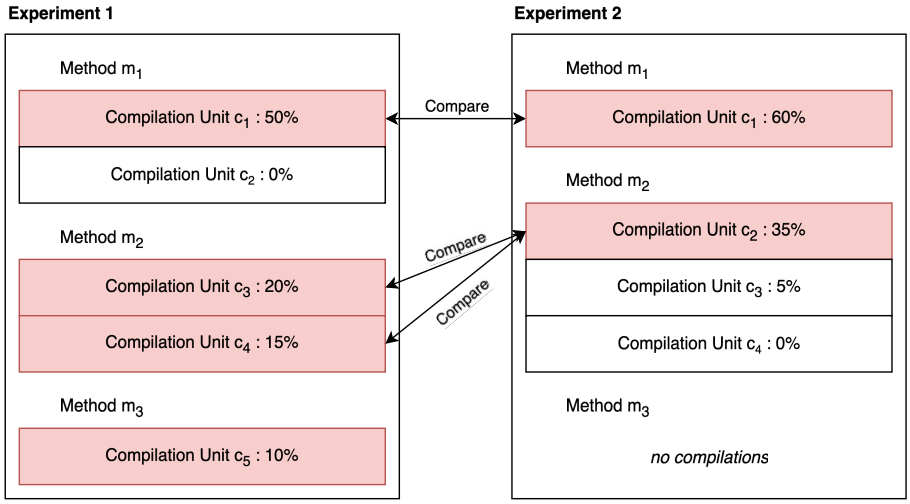


Figure 3.5: Comparing Hot Compilation Units - Image taken from [17]

The result of the comparison of inlining, optimisation and/or optimisation-context trees, is another tree with highlight which optimisations the compilations have in common and which optimisations are different.
As you can see from Figure 3.5, the comparison is run only between comparison units that are considered hot (in the figure they are highlighted in red). Several compilation units may be rooted in the same method due to speculative assumptions and consequent recompilations.
In the presence of inlining, comparing hust pairs of compilation units is insufficient. For example, suppose that method m_3 from Figure 3.5 has been inlined in compilation c_1 in experiment 2. The problem would be that in the comparison between experiment 1 c_5 is not taken into account. Using *compilation fragments*, which basically are the sections of optimisation tree containing the inlined method, it is possible to solve the problem.
Every inlinee is a potential compilation fragment, but creating fragments for all inlining is infeasible. Throgh the following algorithm Listing 3.2 it is possible to assume that a certain compilation fragment need to be created.

```
1    FOR EACH hot_compilation_unit:
2        FOR EACH node IN inlining_tree:
3            IF method IS hot:
4                CREATE compilation_fragment
```

Listing 3.2: Pseudocode of the algorithm to create a compilation fragment

What happen is that in the call tree when a compilation fragment is created, a node is added to the tree as if the method inlined has been called. This allow to see no differences from outside between an inlined method and an optimised one, as you can see in Figure 3.6.

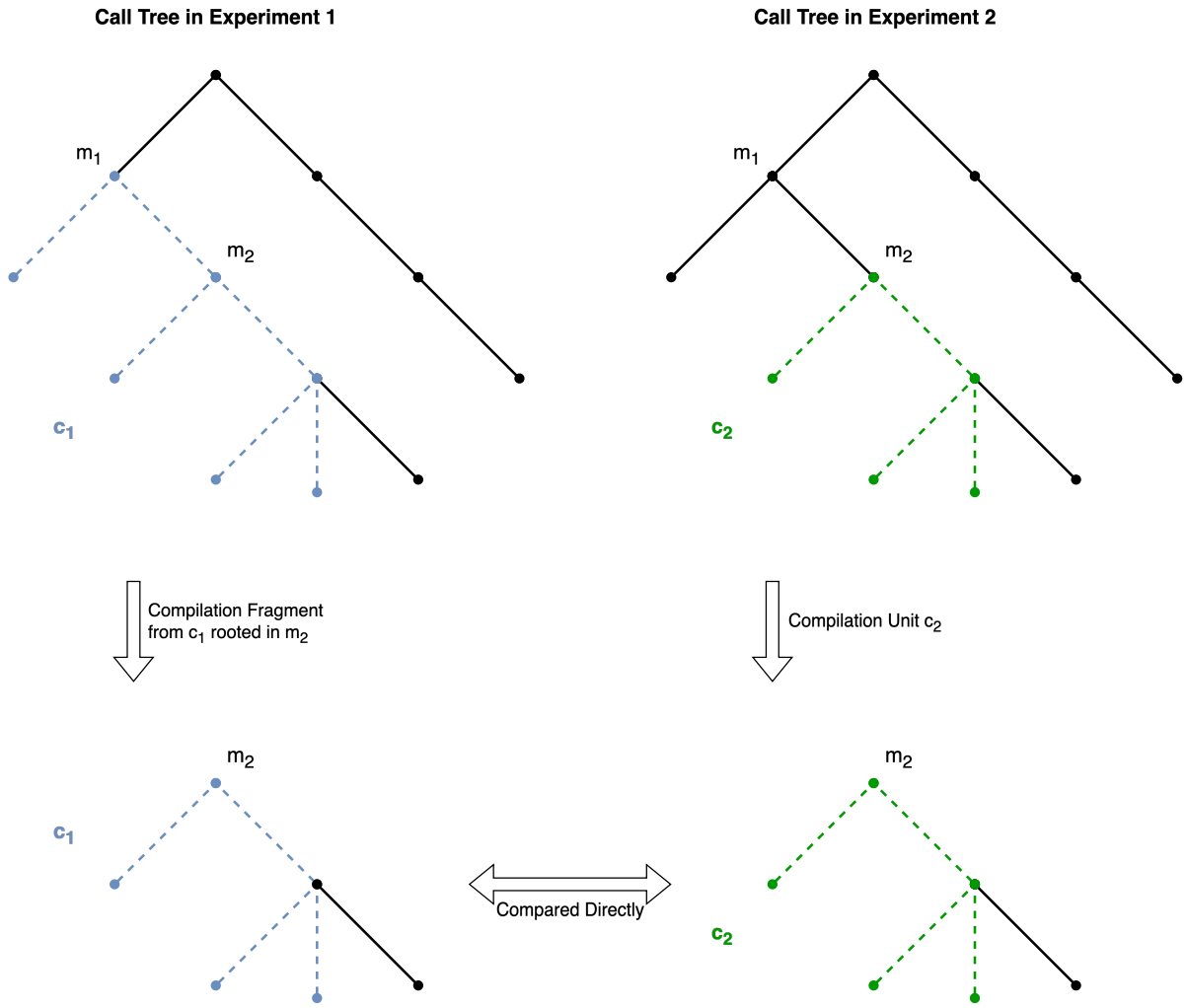


Figure 3.6: The compilation fragment created from c_1 (compilation unit in blue) is directly comparable with the compilation unit c_2 - Image taken from [17]

3.3 JitWatch

Using these flags [47] it is possible to create a `log` from a java execution:

- `-XX:+UnlockDiagnosticVMOptions` - Enable normal processing of flags relating to field diagnostics
- `-XX:+LogCompilation` - Log compilation activity in detail to LogFile
- `-XX:LogFile=file_name` - If `LogVMOutput` or `LogCompilation` is on, save VM output to this file `[default:./hotspot_pid%p.log]` (%p replaced with pid)

JitWatch [2] is a software capable of displaying the information provided by the log in a human-readable way. This is achieved through the log generated by compilation and the source code.
The software facilitates the visualisation of global information over time, including the number of JIT compilations. Additionally, it enables the exploration of more specific data, such as the process of source code compilation into bytecode, the identification of inlining failures, and the analysis of their underlying causes.

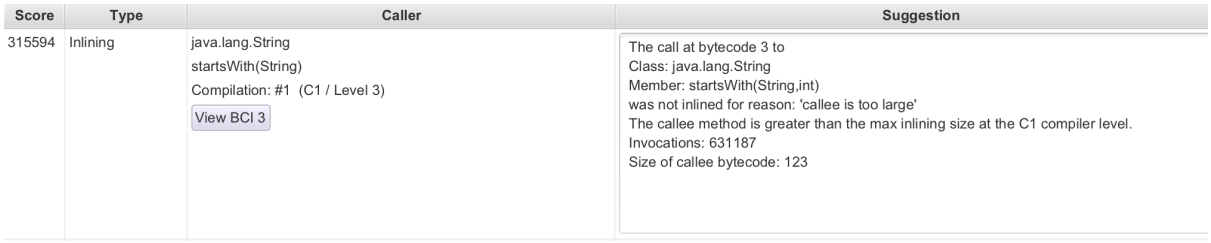


Figure 3.7: Suggest from JitWatch to increase `MaxInlineSize` due to inlining failure - Image taken from [2]

From Figure 3.7 is it possible to see an additional feature of JitWatch. Analysing the log information it is able to make suggestion to improve performances.

Chapter 4

Analysis of Performance Variability

In this chapter, we first present the evaluation settings (Section 4.1). After, we present the evaluation methodology (Section 4.2). Finally, we analyse workloads' pattern (Section 4.3) and we draw our conclusions (Section 4.5).

4.1 Settings

All the experiments run on a server equipped with 16-core Intel(R) Xeon(R) Gold 6326 (2.90GHz) with 256 GB of RAM and one NUMA node. The server run Linux Ubuntu Jammy (v. 22.04 LTS [46]).
We perform our experiments on JVM and GraalVM versions 23 and 24, representing the most recent stable releases at the beginning of the studies. However, we find that often the performance are similar between same VMs, and hence we do show measurements on versions 23 only when they differ between the 24.

4.2 Methodology

The objective of this work is to identify applications that demonstrate variability, even in instances where they execute the same work on the same input, without performing any storage or network operations, and in the absence of other CPU-intensive applications running on the system. In other words, the focus is on scenarios in which external and well-known causes of variability have been excluded [39, 7, 12].
The DaCapo-Chopin [8, 10] and Renaissance [36, 37] benchmark suites are utilised for the execution of experimental procedures. It is evident that both of these suites are contemporary and technologically advanced, serving as a benchmark for the Java Virtual Machine. The execution of a single *run* of a benchmark involves the execution of the same workload for a number of *iterations*, which can be categorised as follows:

- *Warm-up* - Compilation is still ongoing and GC ergonomics have yet to stabilize
- *Steady State* - No more compilation is needed and performance are theoretically stable

All benchmarks use the same input across all iterations and runs (without using random or pseudo-random data), which is loaded into the JVM memory before the start of the first iteration. Input data and temporary scratch folders are stored on *tmpfs* [44] (mounted in the RAM) and there is no network access by benchmark design; hence, local storage and the network have no impact on variability.

We execute a benchmark for 30 runs with a total of 50 iterations for DaCapo Suite and 100 for the Renaissance Suite. The last 10 iterations are considered as steady state. The discrepancy in the number of warm-up iterations is determined by the number of iterations recommended by the two suites (DaCapo [8] and Renaissance [11]).

4.2.1 DaCapo Chopin Suite

We use version 23.11-chopin-MR2 of the DaCapo benchmark suite, the most recent release of the suite at the time of writing. The suite is comprised of more than 20 workloads, mainly design to evaluate the Graal compiler against the HotSpot C2 compiler [8].

4.2.2 Renaissance Suite

It is evident that the version of Renaissance Suite is 0.15, despite the subsequent release of version 0.16. The decision not to update the version is based on the fact that the updates do not affect workloads that present visible variability. Renaissance incorporates workloads that utilize parallelism i.e. *akka-uct*, *fj-kmeans*, and *reactors*, along with various frameworks such as Apache Spark and Scala.

4.3 Results

The most significant metric that is taken into account is the execution time, measured in nanoseconds. A range of additional metrics are also given consideration, including branch instructions, branch misses, cache accesses, cache misses and the number of cycle iterations.
During the preliminary experiment, which we will call *baseline*, we did not make any changes to the default settings of the suites, except for setting the heap size to a fixed 12GB.
What we expect from baseline is the absence of variability at least on sequential workloads since the input is fixed as we mentioned before in Section 4.2. The non-determinism of VMs should not be visible since the execution path and the optimizations should be always the same. This do not happen for many of the workloads.
We used violin plots to better compare trends in different runs. On x-axis we have the run number while on the y-axis the time value measured in ms. In the following graphs, four colours can be distinguished, representing different phases of the run, respectively:

- *Blue* - First iteration
- *Light Blue* - Second iteration
- *Yellow* - Warm-up iterations ($[3, 40]$ for DaCapo workloads and $[3, 90]$ for Renaissance workloads)
- *Black* - Steady state (last 10 iterations)

Whilst, to have a better view of the trend in a single run we use line charts. Here, differently from violin plots, we have on x-axis the iteration number, while the y-axis remains the same. In these charts we have highlighted mean and median for the steady state, respectively in red and green.

4.4 Baseline

In this section, we present the variability patterns observed across all workloads. To better analyse these patterns, workloads are grouped alphabetically by their associated benchmarking suite. Each workload is labelled with its corresponding benchmark suite using the acronyms `[DC]` for DaCapo and `[R]` for Renaissance. For workloads exhibiting variability, only the platforms where such variability occurs are highlighted.

4.4.1 No Significant Variability Workloads

Of the 40 workloads, 15 exhibit no significant variability ($\sim 37.5\%$), meeting both criteria: variability inter-run must be under 5% of range from mean in correspondent iterations and intra-run the trend must be decreasing or constant.
In Figure 4.1, we observe how an experiment (e.g., als from Renaissance) with no variability appears. The figure illustrates that high execution time variation typically occurs during warm-up iterations (black dots), which should instead be concentrated near the bottom of the graph, representing the steady-state performance.

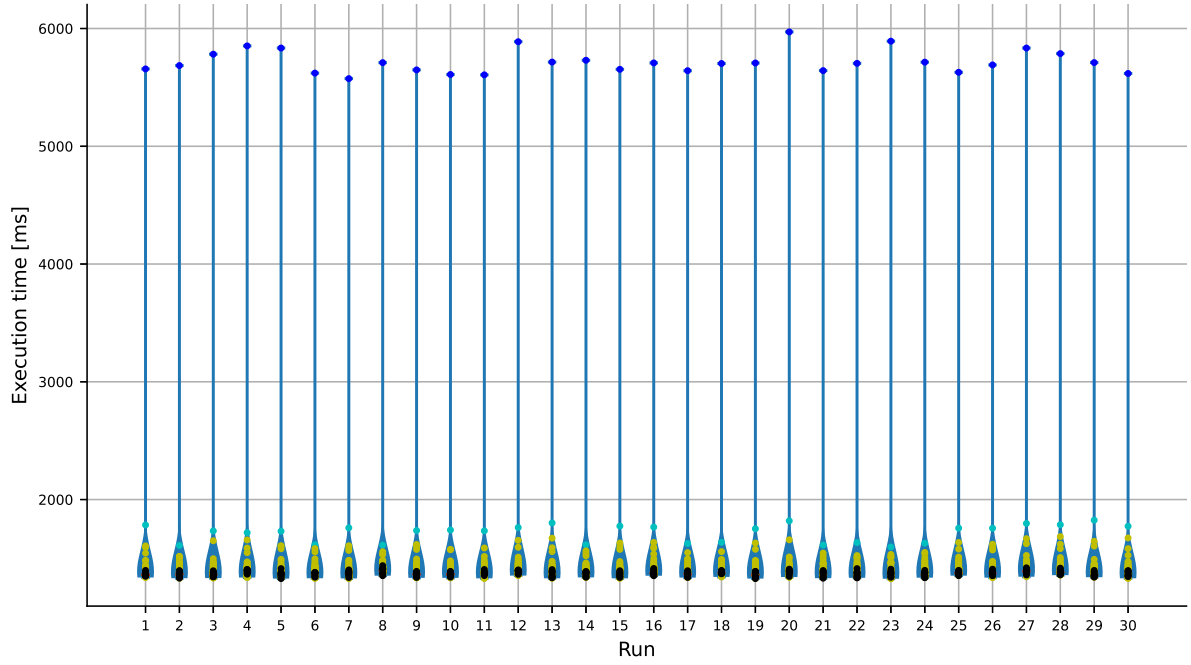


Figure 4.1: Violin plot on baseline experiment for als - Example of an experiment with no significant variability

In Figure 4.2, run 1 on GraalVM 23 of als is shown. For workloads with no variability, the execution time trend demonstrates a gradual decrease followed by stabilization in the last 10 iterations (steady state), indicating consistent performance after the warm-up phase.

4.4.2 Variability in the First Iteration

It has been established that certain workload parameters exhibit variability between runs. The initial run (which is always fully interpreted) exhibits significant variation between runs. As a case in point, we present one of the most substantial workloads that exhibit this pattern: scala-kmeans on GraalVM 24 (Figure 4.3), which demonstrates a coefficient of variation of 14.6%.

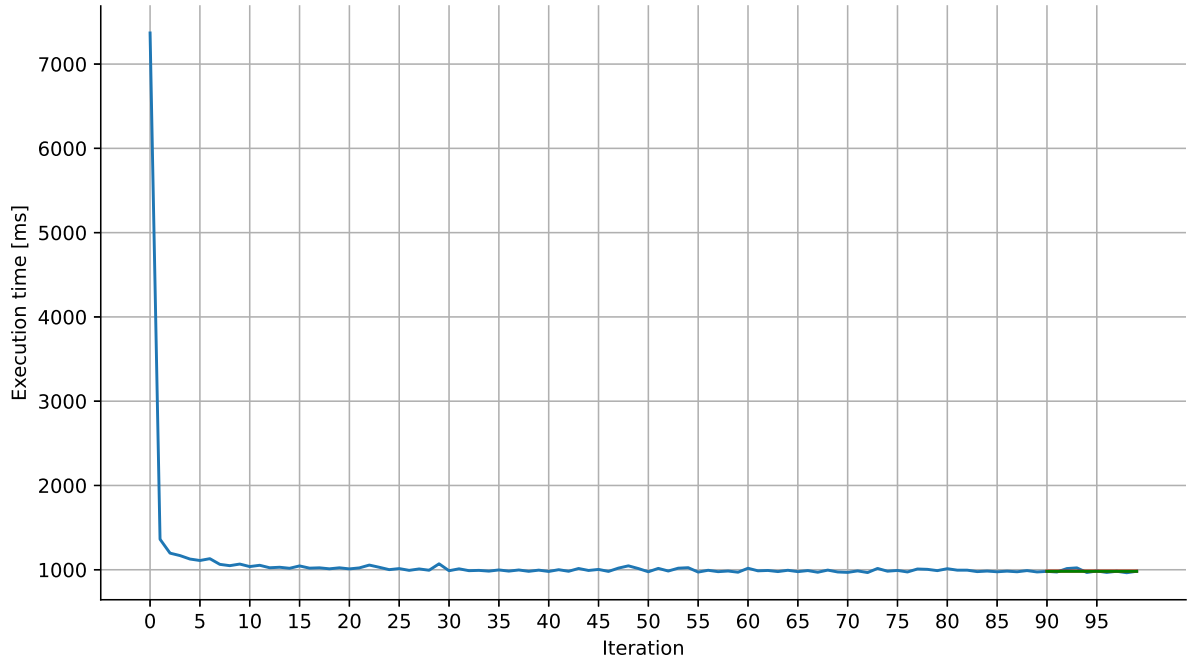


Figure 4.2: Run 1 on GraalVM 23 for als workload - Example of run with no significant variability

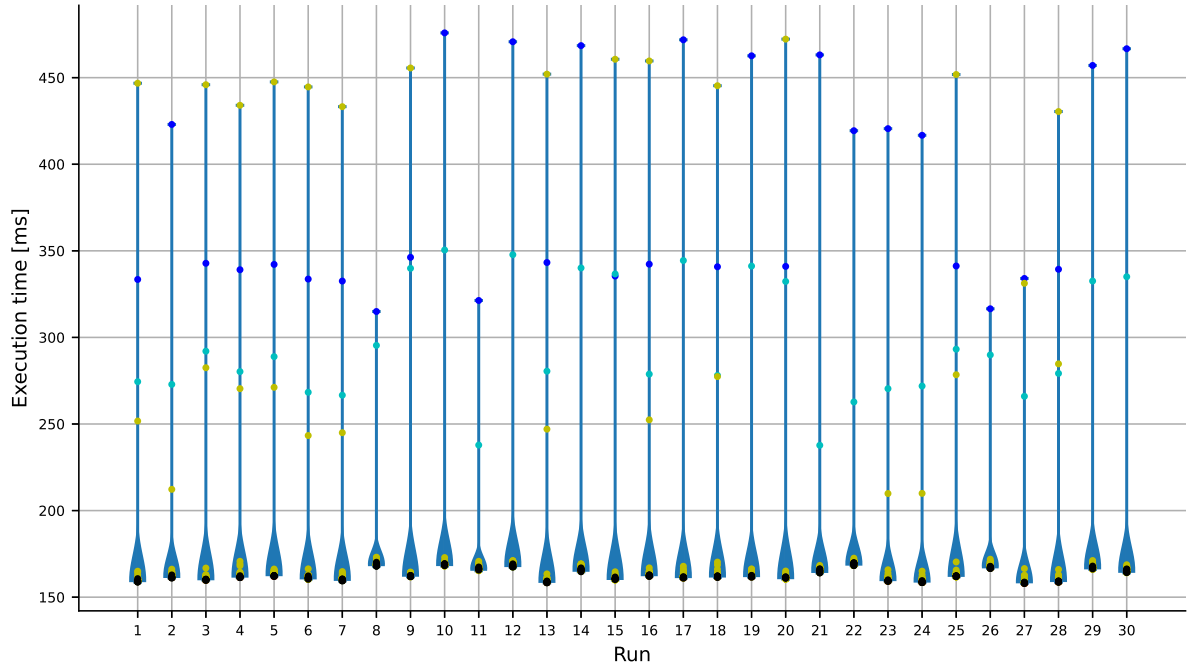


Figure 4.3: Violin Plot for Scala-kmeans on GraalVM 24

Table 4.1 provides a concise overview of the percentage change in performance, measured as the ratio of the results obtained from the initial execution of a workload across different platforms. N/A indicates a variability value of less than 5%. It is noteworthy that no recurring pattern has been identified, suggesting that the observed variations are not attributable to any specific predominance of Graal over Java or version 23 over version 24.
The coefficient of variation cv is calculated as follows:

$$cv = \frac{\sigma}{\mu}$$

	Java 23	Java 24	Graal 23	Graal 24
akka-uct	$cv = 2.5\%$ $\mu = 7172.165$ $\sigma = 182.146$	$cv = 2.4\%$ $\mu = 7107.084$ $\sigma = 171.624$	$cv = 3.4\%$ $\mu = 8855.728$ $\sigma = 302.449$	$cv = 2.9\%$ $\mu = 7795.725$ $\sigma = 226.995$
batik	$cv = 1.3\%$ $\mu = 2682.891$ $\sigma = 35.501$	$cv = 3.0\%$ $\mu = 2691.198$ $\sigma = 80.256$	$cv = 0.6\%$ $\mu = 2891.382$ $\sigma = 17.991$	$cv = 4.9\%$ $\mu = 2773.824$ $\sigma = 134.963$
chi-square	$cv = 1.3\%$ $\mu = 2389.212$ $\sigma = 31.074$	$cv = 1.1\%$ $\mu = 2521.898$ $\sigma = 27.692$	$cv = 2.8\%$ $\mu = 2480.844$ $\sigma = 68.598$	$cv = 13.7\%$ $\mu = 2976.624$ $\sigma = 406.968$
fj-kmeans	$cv = 1.1\%$ $\mu = 1489.975$ $\sigma = 15.950$	$cv = 1.4\%$ $\mu = 1490.641$ $\sigma = 20.495$	$cv = 3.5\%$ $\mu = 1550.936$ $\sigma = 53.727$	$cv = 7.3\%$ $\mu = 2030.258$ $\sigma = 148.181$
graphchi	$cv = 3.2\%$ $\mu = 6468.713$ $\sigma = 208.005$	$cv = 3.1\%$ $\mu = 6578.603$ $\sigma = 204.733$	$cv = 2.7\%$ $\mu = 5652.720$ $\sigma = 151.145$	$cv = 3.0\%$ $\mu = 5862.173$ $\sigma = 174.469$
h2	$cv = 3.3\%$ $\mu = 3636.333$ $\sigma = 121.209$	$cv = 3.2\%$ $\mu = 3448.926$ $\sigma = 109.175$	$cv = 2.2\%$ $\mu = 3965.884$ $\sigma = 87.677$	$cv = 1.4\%$ $\mu = 3825.232$ $\sigma = 54.841$
h2o	$cv = 3.0\%$ $\mu = 4708.718$ $\sigma = 140.197$	$cv = 3.7\%$ $\mu = 4729.367$ $\sigma = 175.141$	$cv = 3.2\%$ $\mu = 4898.720$ $\sigma = 157.883$	$cv = 3.2\%$ $\mu = 4621.383$ $\sigma = 146.724$
page-rank	$cv = 1.5\%$ $\mu = 6352.565$ $\sigma = 97.305$	$cv = 1.8\%$ $\mu = 6284.493$ $\sigma = 115.246$	$cv = 3.4\%$ $\mu = 7549.495$ $\sigma = 255.328$	$cv = 3.5\%$ $\mu = 7873.981$ $\sigma = 275.307$
rx-scrabble	$cv = 5.2\%$ $\mu = 352.473$ $\sigma = 18.418$	$cv = 2.8\%$ $\mu = 333.980$ $\sigma = 9.503$	$cv = 2.3\%$ $\mu = 393.015$ $\sigma = 9.025$	$cv = 2.8\%$ $\mu = 360.521$ $\sigma = 10.043$
scala-doku	$cv = 13.9\%$ $\mu = 3162.719$ $\sigma = 438.762$	$cv = 5.7\%$ $\mu = 2837.611$ $\sigma = 160.363$	$cv = 12.1\%$ $\mu = 2124.226$ $\sigma = 257.855$	$cv = 3.2\%$ $\mu = 2390.143$ $\sigma = 75.651$
scala-kmeans	$cv = 3.2\%$ $\mu = 267.500$ $\sigma = 8.661$	$cv = 1.8\%$ $\mu = 267.775$ $\sigma = 4.833$	$cv = 14.6\%$ $\mu = 356.558$ $\sigma = 52.056$	$cv = 15.7\%$ $\mu = 381.879$ $\sigma = 59.927$
scala-stm-bench7	$cv = 5.0\%$ $\mu = 1660.499$ $\sigma = 82.808$	$cv = 2.2\%$ $\mu = 1623.144$ $\sigma = 36.229$	$cv = 3.1\%$ $\mu = 1909.110$ $\sigma = 59.673$	$cv = 2.3\%$ $\mu = 1686.362$ $\sigma = 38.777$
scrabble	$cv = 14.5\%$ $\mu = 278.335$ $\sigma = 40.329$	$cv = 7.4\%$ $\mu = 256.788$ $\sigma = 18.951$	$cv = 5.5\%$ $\mu = 589.443$ $\sigma = 32.236$	$cv = 9.1\%$ $\mu = 470.505$ $\sigma = 42.589$
sunflow	$cv = 2.3\%$ $\mu = 3644.329$ $\sigma = 83.083$	$cv = 3.1\%$ $\mu = 3700.536$ $\sigma = 113.697$	$cv = 5.2\%$ $\mu = 4524.740$ $\sigma = 233.975$	$cv = 5.4\%$ $\mu = 4480.630$ $\sigma = 243.552$
zxing	$cv = 8.9\%$ $\mu = 2939.307$ $\sigma = 262.964$	$cv = 7.2\%$ $\mu = 2886.436$ $\sigma = 208.810$	$cv = 16.2\%$ $\mu = 4375.674$ $\sigma = 707.906$	$cv = 7.9\%$ $\mu = 4103.542$ $\sigma = 323.910$

Table 4.1: Summary of percentage variation of first iteration in workloads that present first iteration variability pattern
Mean and Standard Deviation are measured in ms

4.4.3 Periodic Intra-run Variability

We define periodic variability as a pattern in which recurring optimization cycles are repeated multiple times within a single run. Ten distinct workloads are observed to exhibit this pattern ($\sim$ 23%), albeit with notable variations. Specifically, some of these workloads demonstrate a consistent repetition of the pattern throughout the majority of the execution, while others exhibit this pattern in specific sections or only during the final iterations. In Table 4.2, a synopsis of the workload is provided.

	Java 23	Java 24	Graal 23	Graal 24
batik	50.0%	33.3%	Absent	Absent
biojava	36.7%	10.0%	40.0%	60.0%
chi-square	All	All	All	All
dec-tree	All	All	All	All
gauss-mix	All	All	All	All
jython	All	All	All	All
movie-lens	30.0%	10.0%	26.7%	Absent
naive-bayes	All	All	Absent	Absent
rx-scrabble	All	All	All	All
scala-kmeans	6.7%	10.0%	Absent	Absent

Table 4.2: Summary of the percentage of run affected by periodic pattern for each workloads

Batik `[DC]`

Batik is a sequential workload from DaCapo suite. In each platform tested the periodic pattern is visibile, but not in every run and not for all the duration of the run. From Figure 4.4 is clearly visible that the pattern is visible only in the steady state iterations.

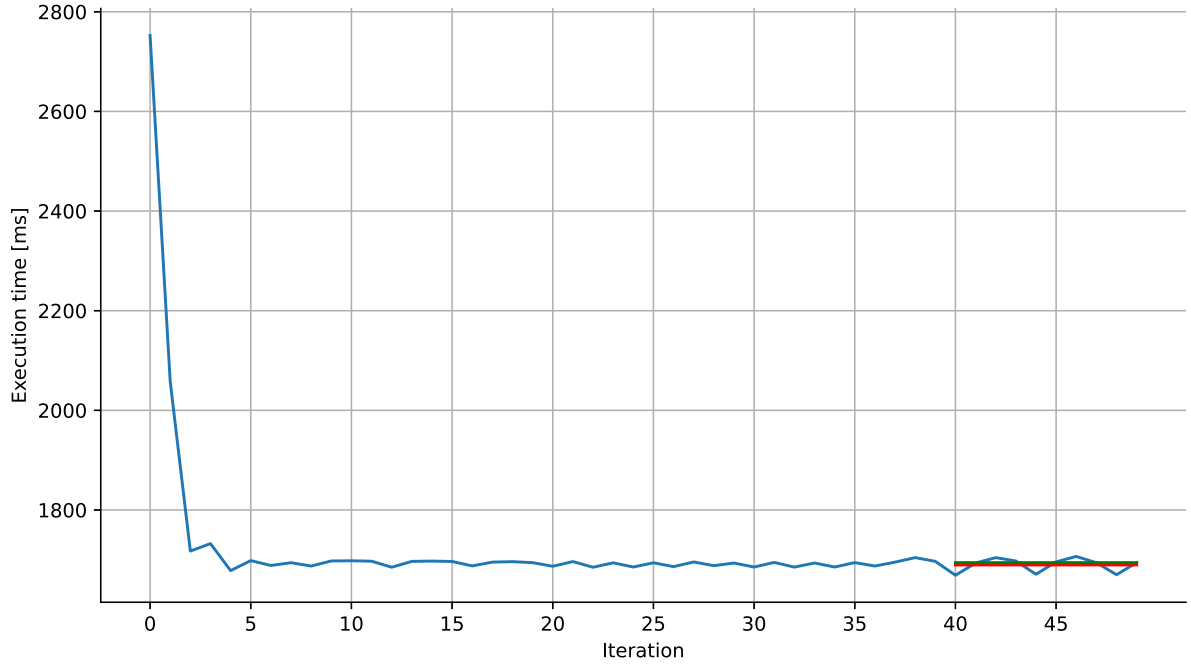


Figure 4.4: Run 7 of batik workload on GraalVM 24
It is possible to see the periodic pattern in steady state iterations.

Biojava `[DC]`

BioJava is a workload that exhibits intra-run variability, which we will discuss later. On the Graal platform, it is possible to observe two distinct patterns repeating in some runs (Figure 4.5): the first one before the jump at the iteration 40, and the second one after.

On the Java platform, the periodic pattern is mainly visible on version 23 rather than

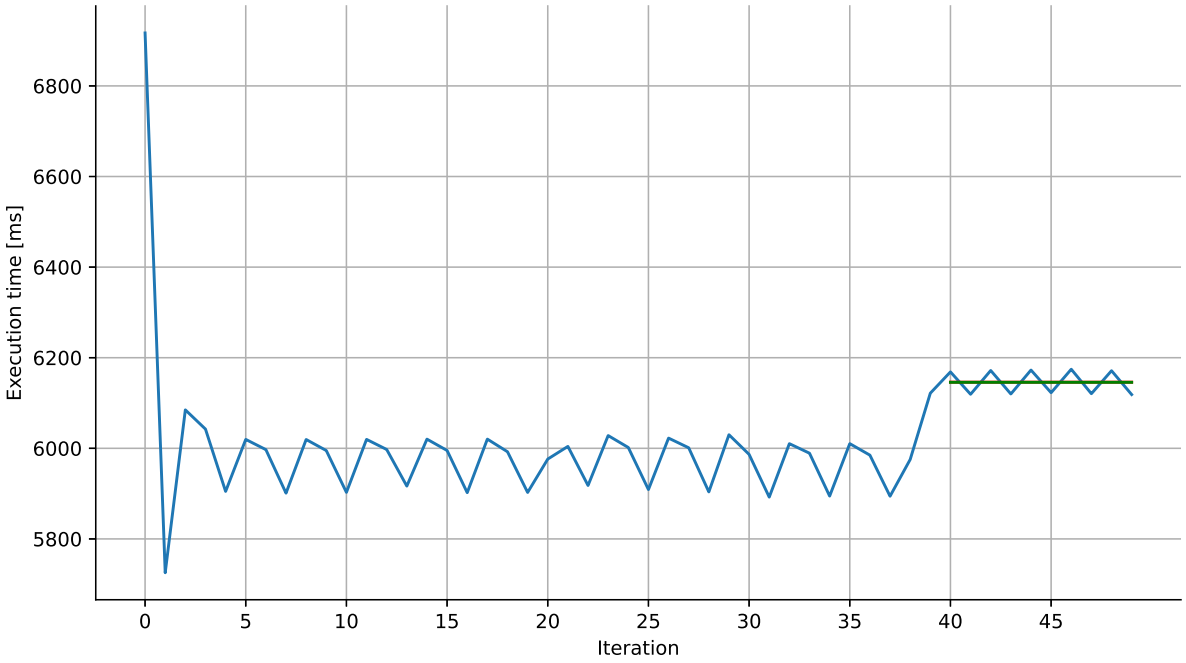


Figure 4.5: Run 10 of biojava workload on GraalVM 23
Two different patterns can be distinguished, one before iteration 40 and one after.

version 24. Here (Figure 4.6), there is only one repeating pattern since there is no jump before the iteration 40.

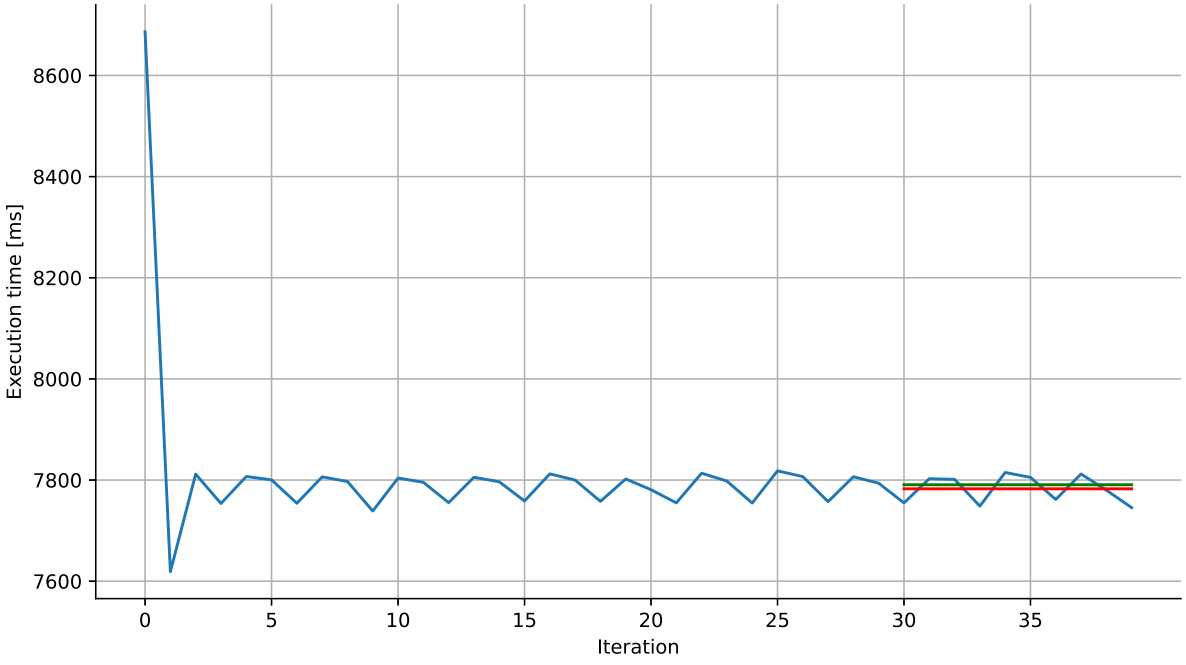


Figure 4.6: Run 3 of biojava workload on Java 23
It is possible to see the periodic pattern repeating between iterations periodically.

Chi-square `[R]`

Periodic variability is visible in all the platforms. From Figure 4.7 it is clearly visible the pattern typical of this workload. Chi-square is part of *apache-spark* and later we will see that other workloads from this group present this pattern.

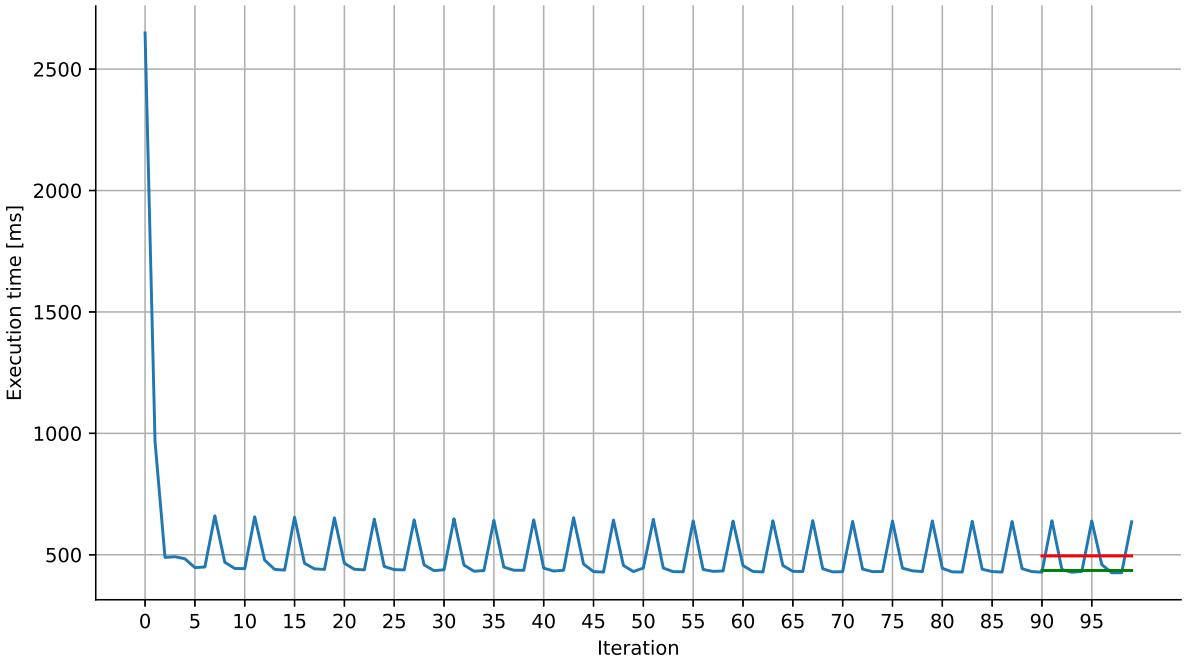


Figure 4.7: Run 13 of chi-square workload on GraalVM 24
It is possible to see the periodic pattern repeating between iterations periodically.

Dec-tree `[R]`

The Dec-Tree workload is similar to the Chi-Square workload and part of the same group from renaissance. The pattern constantly repeats itself during the run, as can be seen in Figure 4.8.

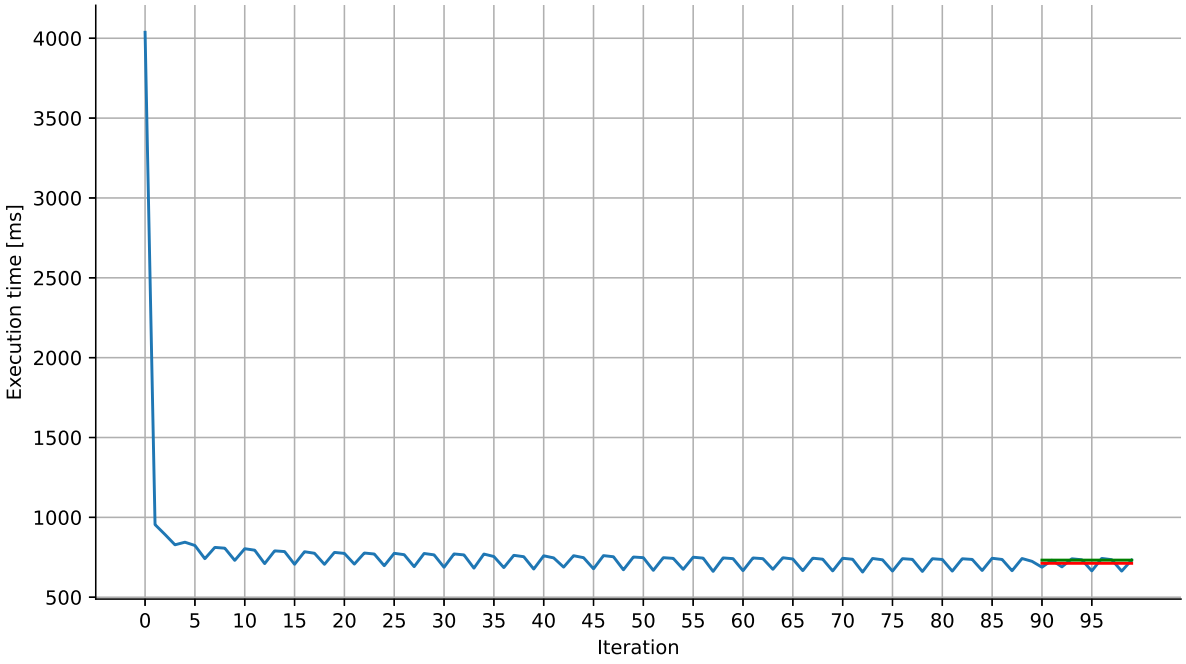


Figure 4.8: Run 6 of dec-tree workload on Java 24
It is possible to see the periodic pattern repeating between iterations periodically.

Gauss-mix [R]

As with the previous chi-square and dec-tree, gauss-mix is part of Renaissance's *apache-spark* group. It presents a similar pattern to them, repeating itself for the majority of the run.

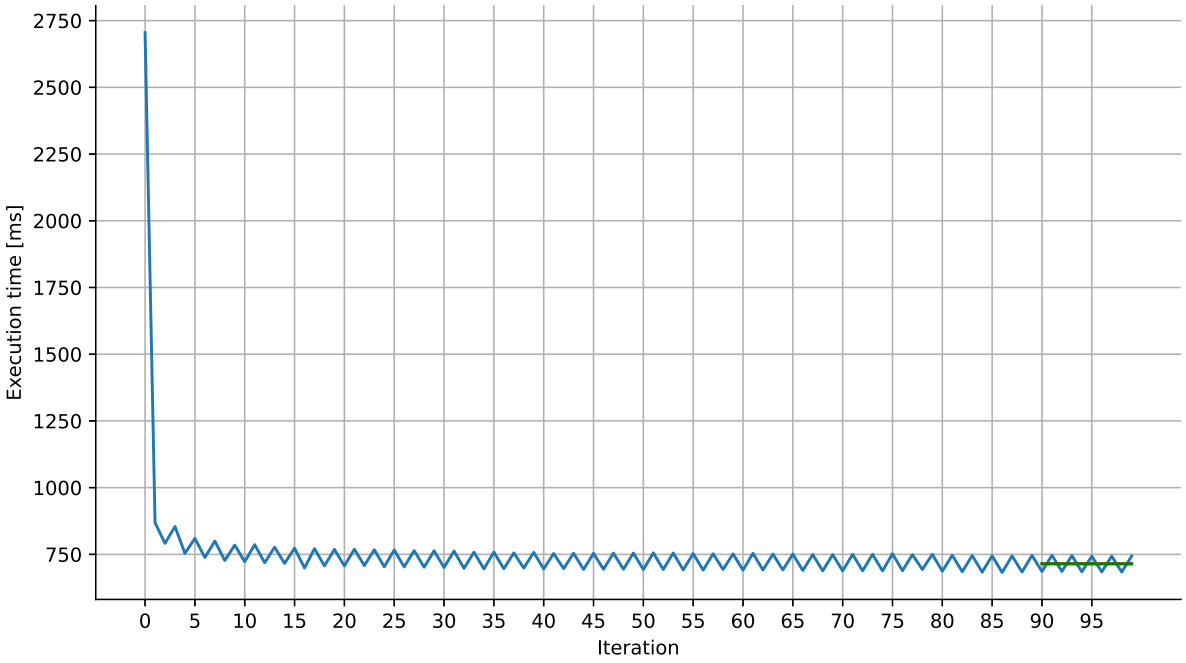


Figure 4.9: Run 4 of dec-tree workload on Java 23
It is possible to see the periodic pattern repeating between iterations periodically.

Jython [DC]

Jython is a workload from DaCapo benchmark suite. The pattern is present here in all the platform as in chi-square. Nonetheless, an antecedent phase of optimisation was observed, from which point, commencing from the 10th iteration, a periodic pattern emerged, as illustrated in Figure 4.10.

Movie-lens [R]

As for chi-square, movie-lens is a workload from Renaissance suite and in particular from *apache-spark* group. Contrary to the previous workload from the same group, movie-lens does not exhibit this pattern in every iteration; rather, it manifests in a subset of the iterations. However, the behaviour is consistent with that of the other apache-spark workloads (Figure 4.11).

Naive-bayes [R]

This is the another workload from *apache-spark* group of Renaissance. During the run not always the pattern is not constantly repeated, e.g. in between the peeks in Figure 4.12.

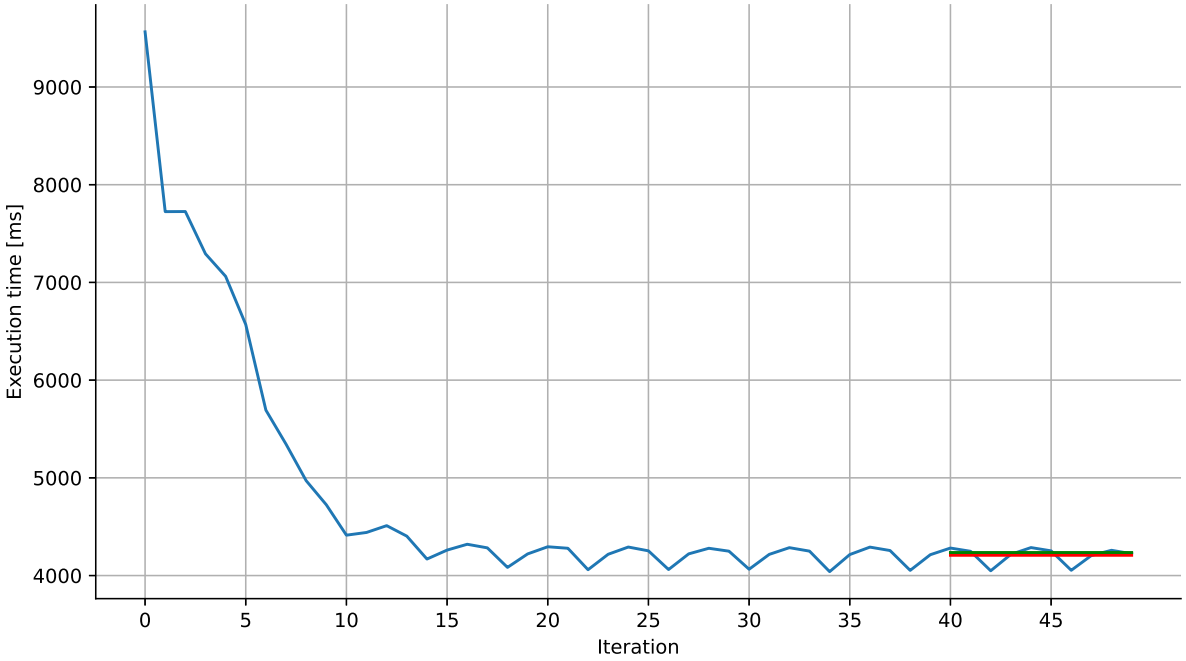


Figure 4.10: Run 12 of jython workload on Java 23
From the 10th iteration the pattern is visible

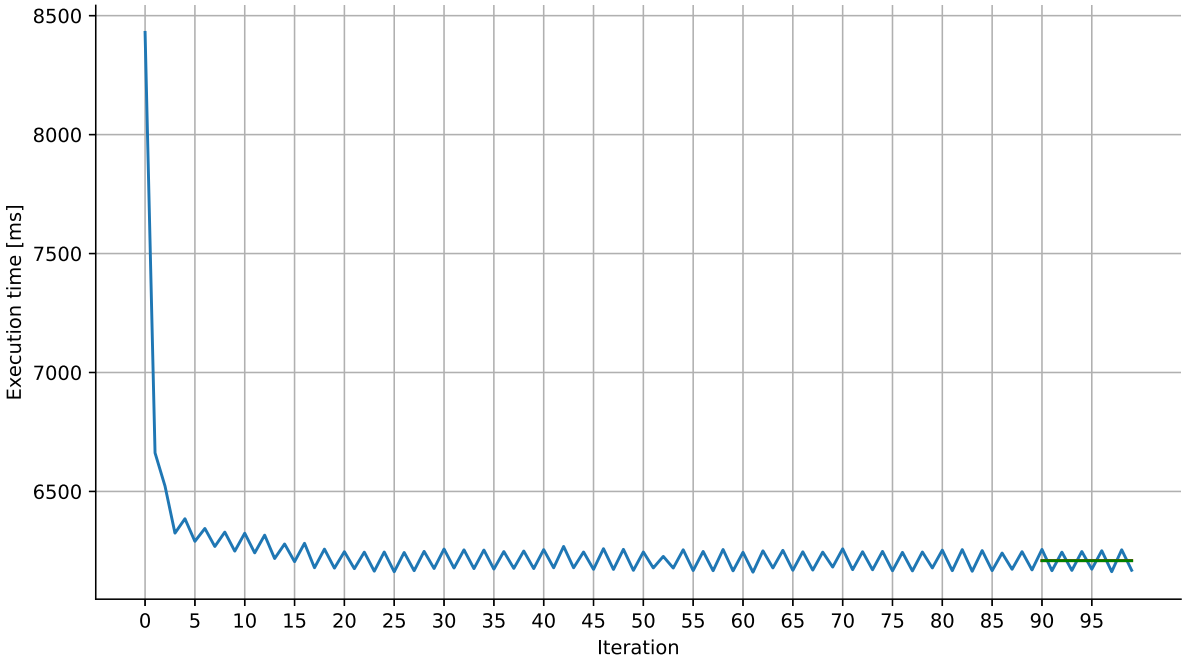


Figure 4.11: Run 6 of movie-lens on Java 24
It is possible to see the periodic pattern repeating between iterations periodically.

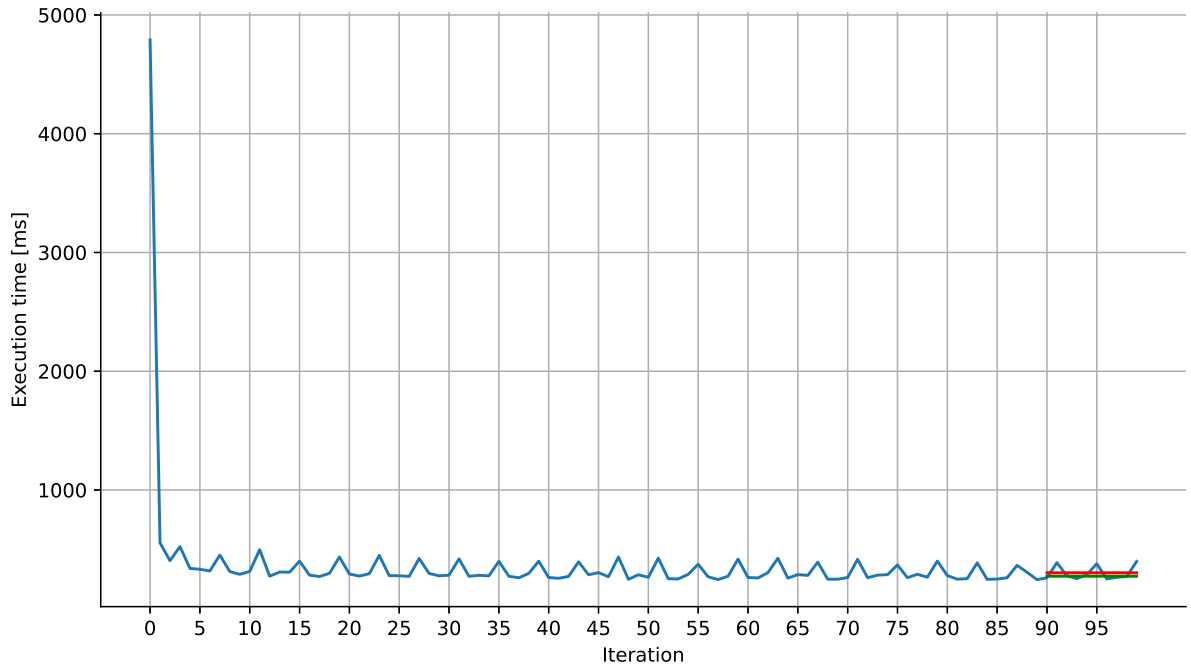


Figure 4.12: Run 21 of naive-bayes workload on Java 23

rx-scrabble `[R]`

Rx-scrabble is a constituent of the *functional* group known as Renaissance. The periodic pattern is observed to repeat with consistency throughout the duration of the run, subsequent to the initial few iterations, despite the minimal fluctuation recorded (approximately 50 milliseconds). The pattern is repeated regularly throughout the run (Figure 4.13).

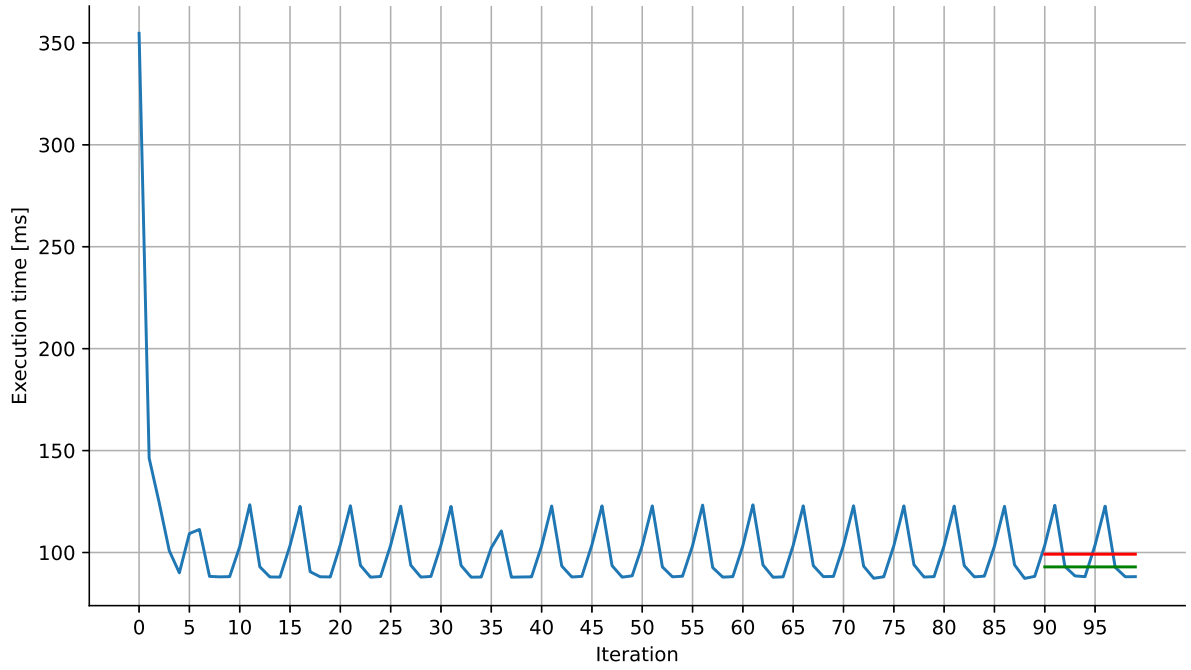


Figure 4.13: Run 5 of rx-scrabble workload on GraalVM 24

scala-kmeans `[R]`

Scala-kmeans is a workload that originates from the Scala group of Renaissance. The pattern is only evident on the Java platform, where it exhibits a distinct behaviour from the majority (Figure 4.14).

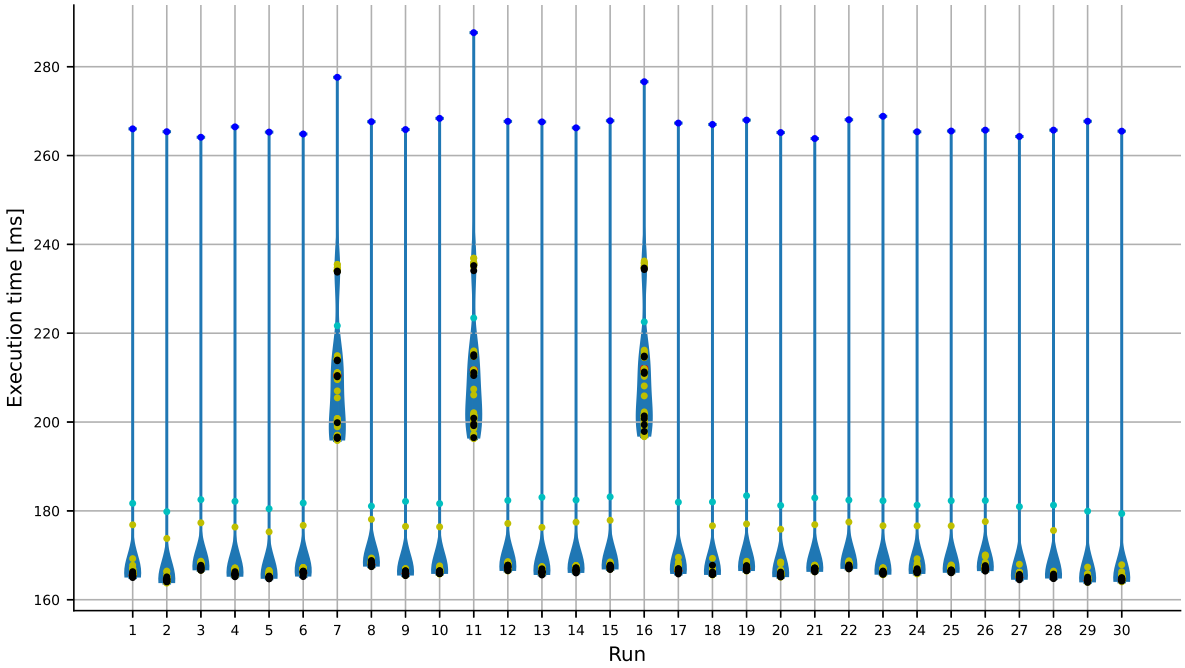


Figure 4.14: Scala-kmeans violin plot on Java 24 platforms

As illustrated in Figure 4.15, the fluctuations alternate between two values, with the pattern repeating on subsequent iterations. It is the only workload that exhibits this behaviour.

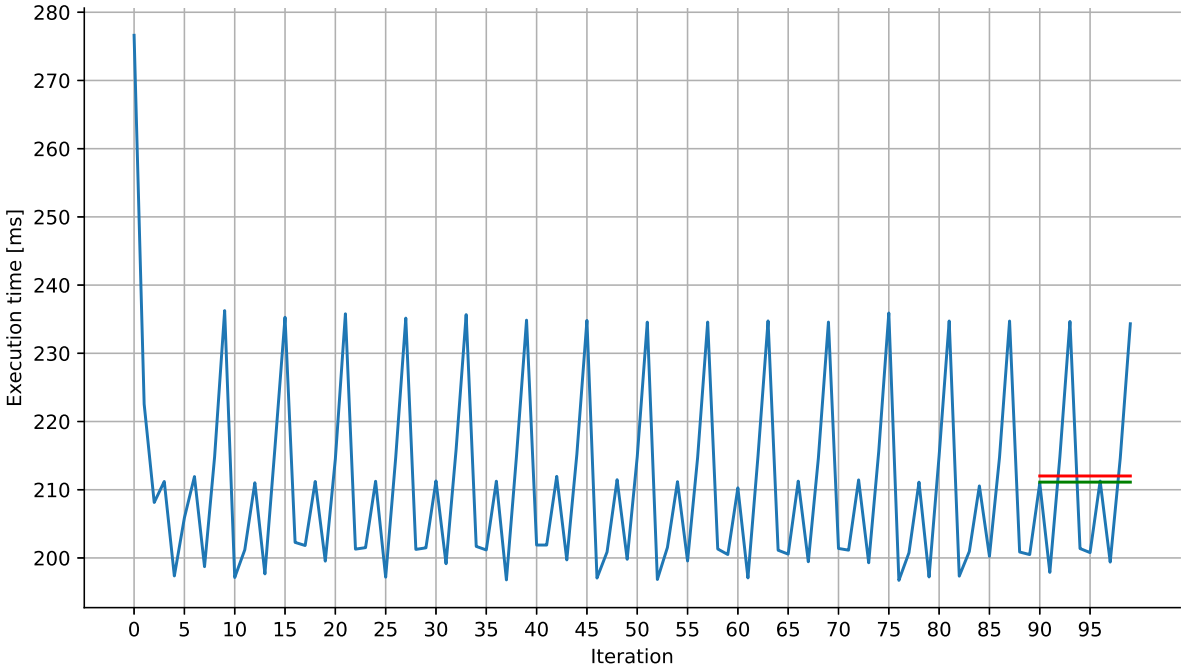


Figure 4.15: Run 16 of scala-kmeans workload on Java 24

4.4.4 Intra-run Performance Variability

A considerable proportion of workloads manifest this form of variability, whereby performance undergoes significant variation during execution. In certain instances, such as in the case of sunflow, the variance is observed to be present even between steady-state iterations. Conversely, in other scenarios, such as in the biojava context, the variance is found to be less pronounced between steady-state iterations.
As illustrated in Table 4.3, the range of fluctuation for the various workload parameters is documented. The fluctuation range is calculated using the coefficient of variation as metric only in steady state iterations:

$$cv = \frac{\sigma}{\mu}$$

	Java 23	Java 24	Graal 23	Graal 24
akka-uct	$cv = 2.8\%$ $\mu = 5974.797$ $\sigma = 169.269$	$cv = 2.9\%$ $\mu = 6024.168$ $\sigma = 176.754$	$cv = 3.0\%$ $\mu = 6030.576$ $\sigma = 178.590$	$cv = 2.5\%$ $\mu = 6131.937$ $\sigma = 152.949$
biojava	$cv = 0.8\%$ $\mu = 7801.960$ $\sigma = 61.346$	$cv = 1.0\%$ $\mu = 8128.017$ $\sigma = 84.133$	$cv = 0.8\%$ $\mu = 6169.950$ $\sigma = 52.408$	$cv = 0.8\%$ $\mu = 6138.695$ $\sigma = 48.353$
h2o	$cv = 1.9\%$ $\mu = 3084.261$ $\sigma = 59.102$	$cv = 1.9\%$ $\mu = 3118.210$ $\sigma = 60.714$	$cv = 1.8\%$ $\mu = 3013.454$ $\sigma = 55.341$	$cv = 1.8\%$ $\mu = 2945.530$ $\sigma = 54.452$
luindex	$cv = 1.1\%$ $\mu = 4917.181$ $\sigma = 53.892$	$cv = 1.1\%$ $\mu = 4887.258$ $\sigma = 52.420$	$cv = 1.1\%$ $\mu = 4093.923$ $\sigma = 43.590$	$cv = 1.1\%$ $\mu = 4072.915$ $\sigma = 45.219$
par-mnemonics	$cv = 2.4\%$ $\mu = 2077.882$ $\sigma = 49.488$	$cv = 2.3\%$ $\mu = 2084.003$ $\sigma = 47.812$	$cv = 12.4\%$ $\mu = 1338.268$ $\sigma = 165.372$	$cv = 12.3\%$ $\mu = 1303.169$ $\sigma = 160.669$
scala-stm-bench7	$cv = 6.0\%$ $\mu = 867.821$ $\sigma = 51.842$	$cv = 5.4\%$ $\mu = 851.076$ $\sigma = 45.974$	$cv = 7.6\%$ $\mu = 804.135$ $\sigma = 61.200$	$cv = 7.8\%$ $\mu = 822.090$ $\sigma = 63.934$
sunflow	$cv = 9.3\%$ $\mu = 3661.059$ $\sigma = 339.290$	$cv = 8.4\%$ $\mu = 3644.444$ $\sigma = 307.280$	$cv = 10.8\%$ $\mu = 3195.589$ $\sigma = 345.537$	$cv = 11.5\%$ $\mu = 3054.418$ $\sigma = 350.754$

Table 4.3: Summary of performance fluctuation intra-run
Mean and Standard Deviation are measured in ms

Akka-uct `[R]`

Akka-uct is a workload from the *concurrency* group of Renissance. It is imperative to observe that the fluctuations persist throughout the entirety of the execution, commencing from the second iteration. As demonstrated in Figure 4.16, there is no distinction between warm-up iterations and steady state.

Biojava `[DC]`

Biojava is a workload from the DaCapo suite. It has been observed that there is a downward trend in performance, commencing from the second iteration, in the majority of runs. In Graal, the jump is before last 10 iterations (Figure 4.5), while in Java, from the second to the third iteration.
A further distinction between bio-java and other DaCapo workloads is that the number of iterations in Java are 40, even when the workload is constrained to 50 iterations, as

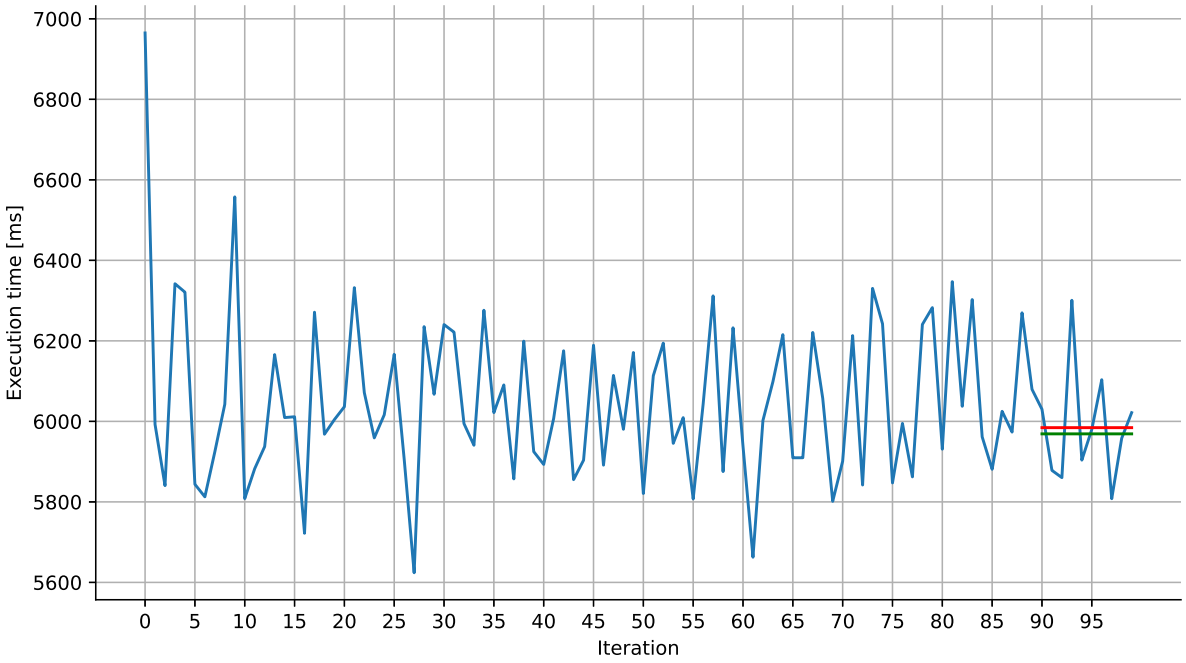


Figure 4.16: Run 3 of akka-uct workload on Java 23

illustrated in Figure 4.17.

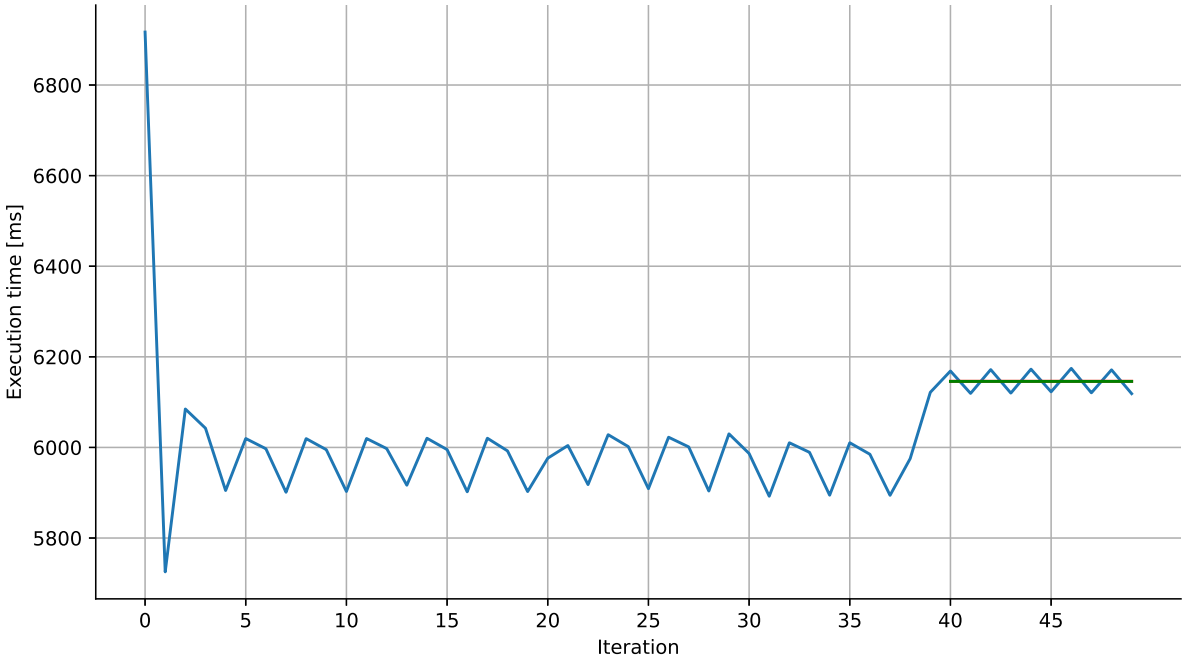


Figure 4.17: Run 10 of biojava workload on GraalVM 23

H2o [DC]

It is evident that there is variability in the h2o present during the course of the run. The range of performance is not significantly elevated for the Java platforms (approximately 400 milliseconds) or the Graal platforms (approximately 300 milliseconds). Warm-up iterations consistently demonstrate superior performance in comparison to steady-state iterations. A meticulous examination of a single run (e.g. Figure 4.18) reveals a recurring trend of steadily improving performance as the iterations progress.

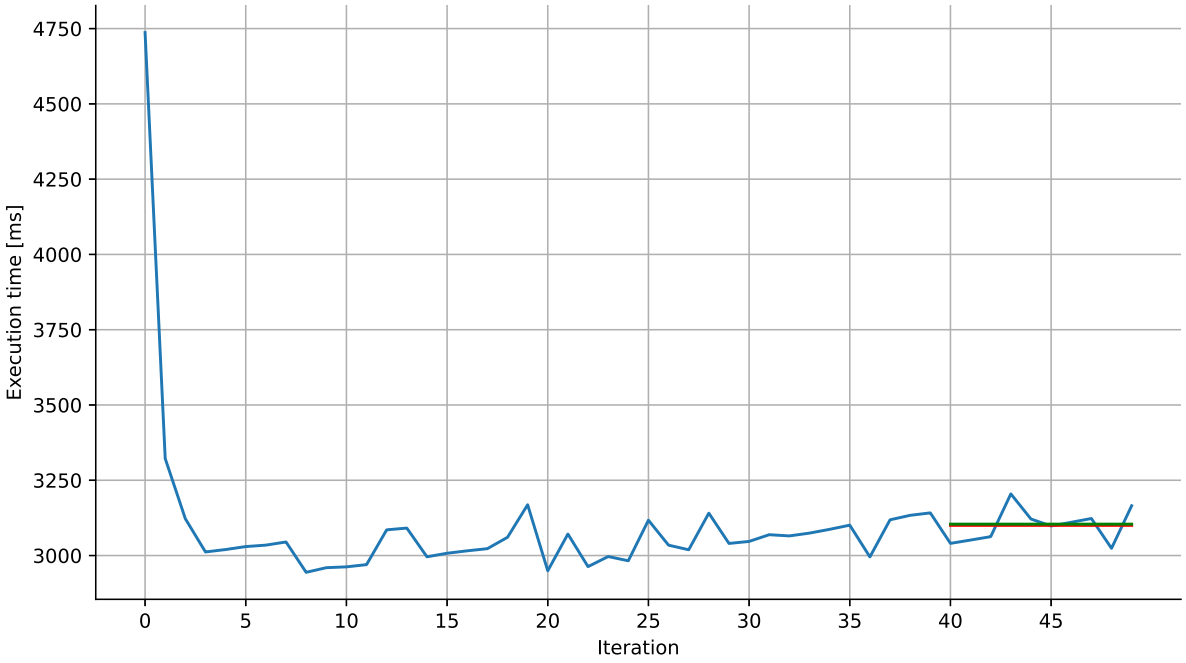


Figure 4.18: Run 24 of h2o workload on Java 24

Luindex [DC]

Luindex displays a comparable tendency to that exhibited by h2o. The variations in performance are confined to the run and the overall Java platforms (375 ms), with Graal demonstrating superior performance (275 ms). In this instance, too, the warm-up iterations invariably demonstrate superiority over steady-state iterations. In contrast to the case of h2o, where a discernible change in performance is only evident after the initial warm-up iterations , a clear distinction in performance is apparent in this instance throughout the entirety of the run.

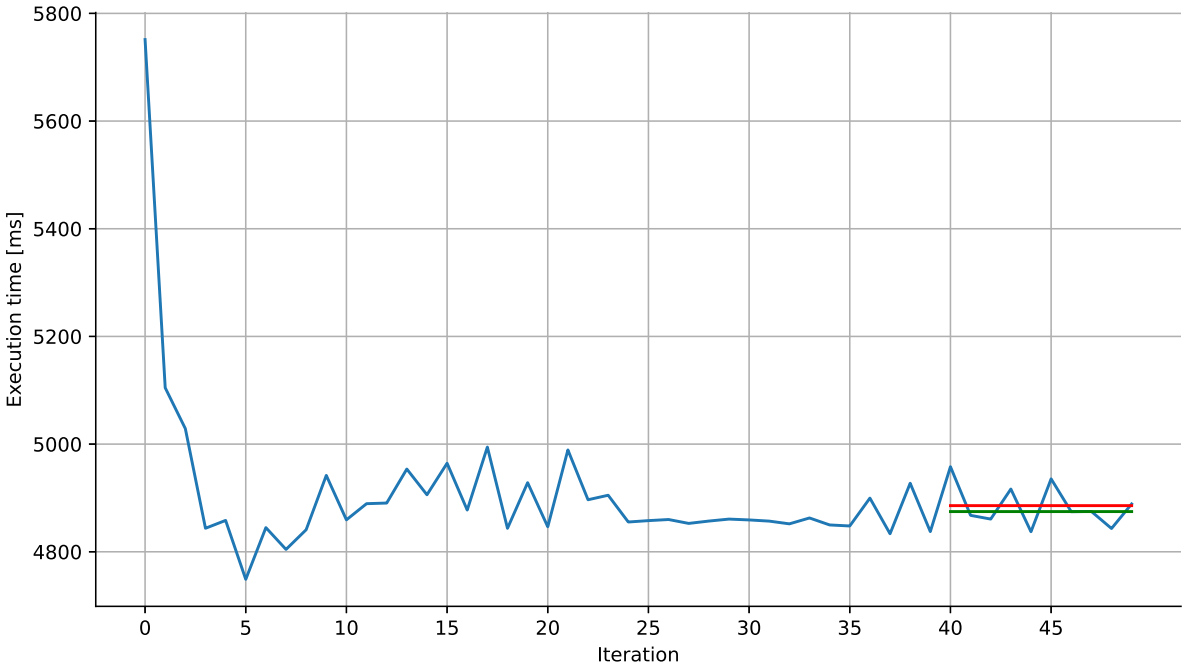


Figure 4.19: Run 21 of luindex workload on Java 24

Par-mnemonics `[R]`

Par-mnemonics is a workload of the concurrency group of Renaissance. It is evident that a considerable degree of intra-run variability is exhibited throughout the majority of the run. The variation is characterised by a state of greater complexity and heightened concentration as it progresses towards a steady state, a phenomenon that is clearly evident in Figure 4.20.

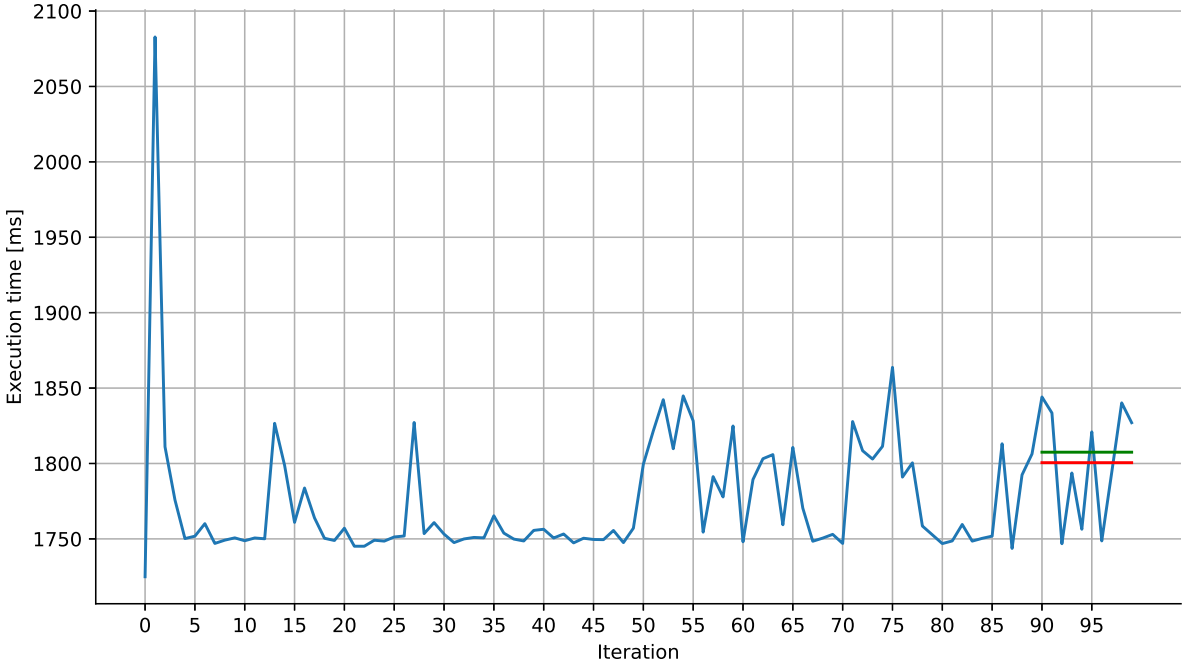


Figure 4.20: Bad run 4 of par-mnemonics workload on GraalVM 24

Scala-stm-bench7 `[R]`

Scala-stm-bench7 discrepancies emerge in the performance evolution between distinct execution instances. It is evident that variability in the workload may occur during the entirety of the run, akin to the variability observed in previous workloads. Alternatively, this variability may be confined to the warm-up phase. However, as illustrated in Figure 4.21, the performance during this phase is suboptimal.

Sunflow `[DC]`

Sunflow is a workload from the DaCapo suite that exhibits both high levels of variability within a single run and between different runs. The range within which the fluctuations vary is approximately 3 seconds across all platforms, given that the values vary from 2 to 5 seconds per iteration.
On Graal, the fluctuation range predominantly resides beneath the initial iteration. Conversely, within Java, the initial iteration invariably occupies a position midway through the fluctuation range (Figure 4.22).

4.4.5 Inter-run Performance Variability

The final category of variability pertains to instances wherein performance exhibits significant variation across different runs and/or platforms. For significant variation we mean a variation that exceeds 5% from the overall average performance and the trend between

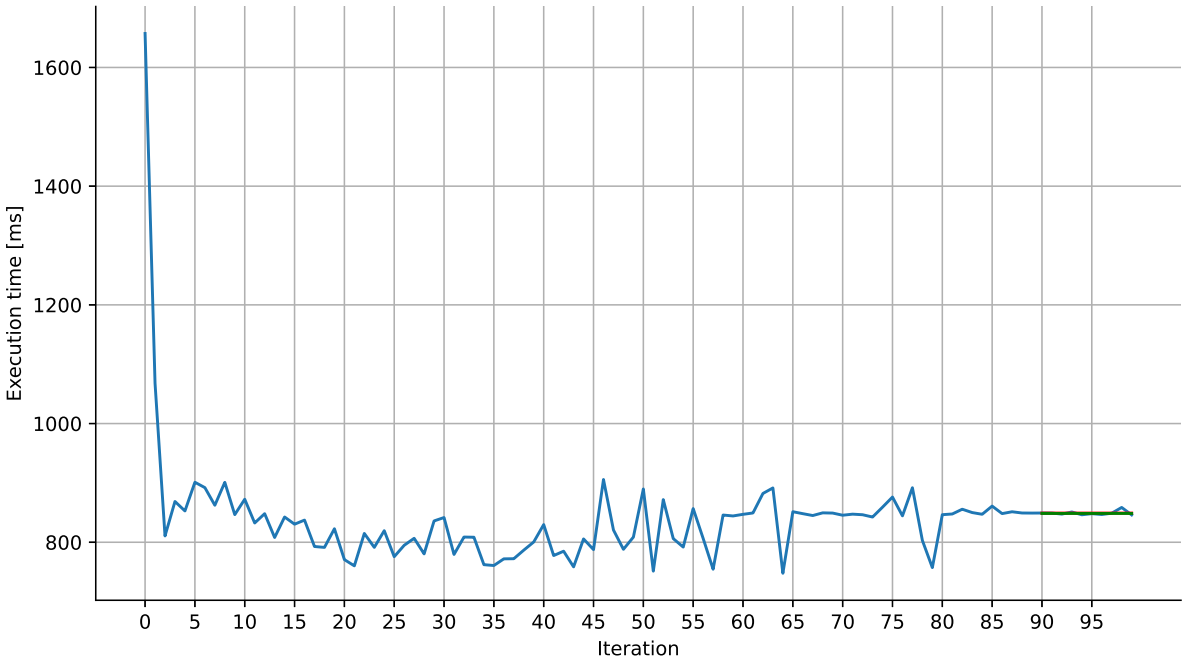


Figure 4.21: Run 30 of scala-stm-bench7 on GraalVM 24

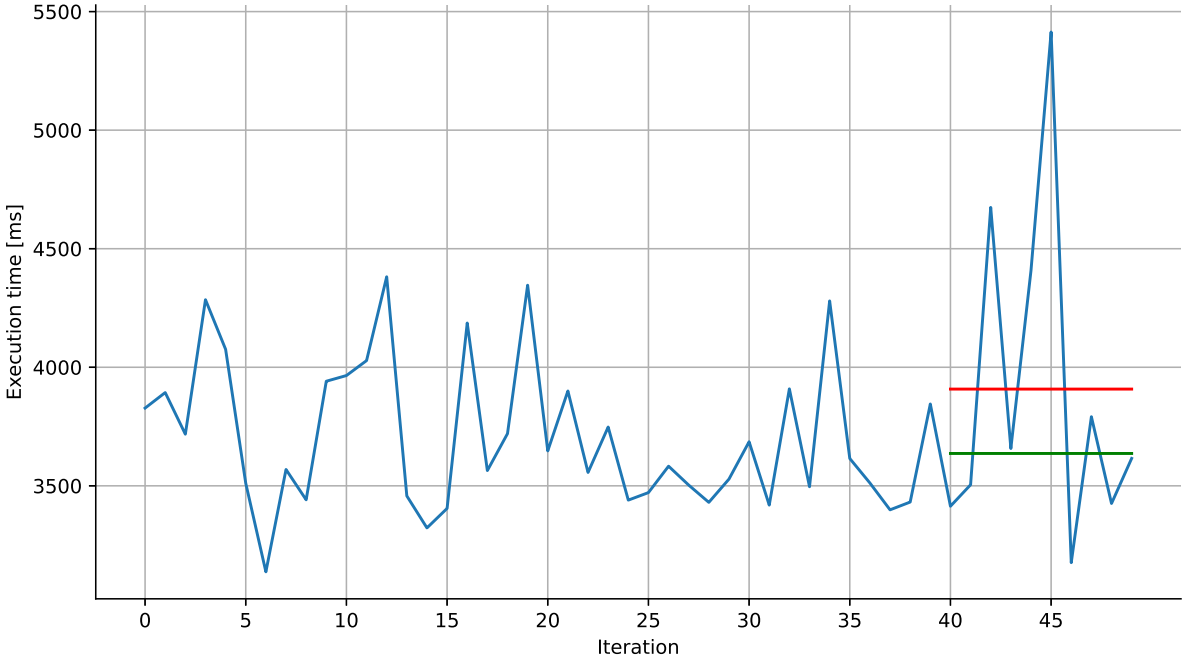


Figure 4.22: Run 7 of sunflow workload on Java 23 - The first iteration is not the slowest even though it is fully interpreted

runs must be different. In Table 4.4 we can find a summary of the pattern inside the workloads:

	Java 23	Java 24	Graal 23	Graal 24
chi-square	$cv = 12.4\%$ $\mu = 585.438$ $\sigma = 72.653$	$cv = 13.9\%$ $\mu = 592.726$ $\sigma = 82.468$	$cv = 16.2\%$ $\mu = 512.897$ $\sigma = 83.248$	$cv = 15.5\%$ $\mu = 472.855$ $\sigma = 73.161$
graphchi	$cv = 0.9\%$ $\mu = 4950.328$ $\sigma = 43.235$	$cv = 1.8\%$ $\mu = 5001.356$ $\sigma = 92.510$	$cv = 1.2\%$ $\mu = 3639.552$ $\sigma = 43.776$	$cv = 0.9\%$ $\mu = 3815.660$ $\sigma = 34.720$
page-rank	$cv = 2.0\%$ $\mu = 2803.210$ $\sigma = 56.521$	$cv = 2.2\%$ $\mu = 2717.471$ $\sigma = 60.221$	$cv = 4.1\%$ $\mu = 2584.378$ $\sigma = 106.172$	$cv = 3.8\%$ $\mu = 2688.759$ $\sigma = 100.914$
par-mnemonics	$cv = 2.4\%$ $\mu = 2077.691$ $\sigma = 49.664$	$cv = 2.3\%$ $\mu = 2085.510$ $\sigma = 48.889$	$cv = 12.8\%$ $\mu = 1341.451$ $\sigma = 171.248$	$cv = 12.3\%$ $\mu = 1303.528$ $\sigma = 160.923$
scala-kmeans	$cv = 15.4\%$ $\mu = 173.329$ $\sigma = 26.642$	$cv = 8.0\%$ $\mu = 170.806$ $\sigma = 13.666$	$cv = 1.8\%$ $\mu = 163.029$ $\sigma = 2.890$	$cv = 2.1\%$ $\mu = 163.233$ $\sigma = 3.410$
scala-stm-bench7	$cv = 6.2\%$ $\mu = 878.263$ $\sigma = 54.876$	$cv = 5.6\%$ $\mu = 869.509$ $\sigma = 48.816$	$cv = 7.8\%$ $\mu = 819.372$ $\sigma = 63.970$	$cv = 7.7\%$ $\mu = 836.587$ $\sigma = 64.621$
spring	$cv = 4.2\%$ $\mu = 2160.256$ $\sigma = 90.101$	$cv = 3.9\%$ $\mu = 1935.422$ $\sigma = 74.978$	$cv = 2.8\%$ $\mu = 1971.445$ $\sigma = 54.651$	$cv = 2.1\%$ $\mu = 1729.533$ $\sigma = 36.792$
sunflow	$cv = 8.9\%$ $\mu = 3653.175$ $\sigma = 325.502$	$cv = 8.6\%$ $\mu = 3654.043$ $\sigma = 312.795$	$cv = 10.8\%$ $\mu = 3177.118$ $\sigma = 344.068$	$cv = 11.8\%$ $\mu = 3054.171$ $\sigma = 361.715$

Table 4.4: Summary of performance fluctuation inter-run
Mean and Standard Deviation are obtained executing them singolarly on each run then meaning the results

It has been established that a number of these workloads exhibit intra-run variability. Consequently, any comparison between runs will be subject to significant variation. As was discussed in the preceding section, these workloads will not be pursued any further in this section.

Graphchi `[DC]`

In Graphchi, variability between runs is visible only on Java 24. As demonstrated in Figure 4.23, it is evident that within a single execution, there is an absence of variability in performance. However, when comparing different executions, there is a range of performance values from 4.8 seconds to 5.3 seconds.

Par-mnemonics `[R]`

In consideration of inter-run variability, it is observed that distinct behaviours are exhibited based on the utilisation platform. Specifically, on Graal, the outcomes of the runs are known to be distributed over two possible performance outcomes, as illustrated in Figure 4.24. Conversely, in the Java environment, the behaviour is characterised by greater variability, as demonstrated in Figure 4.25.

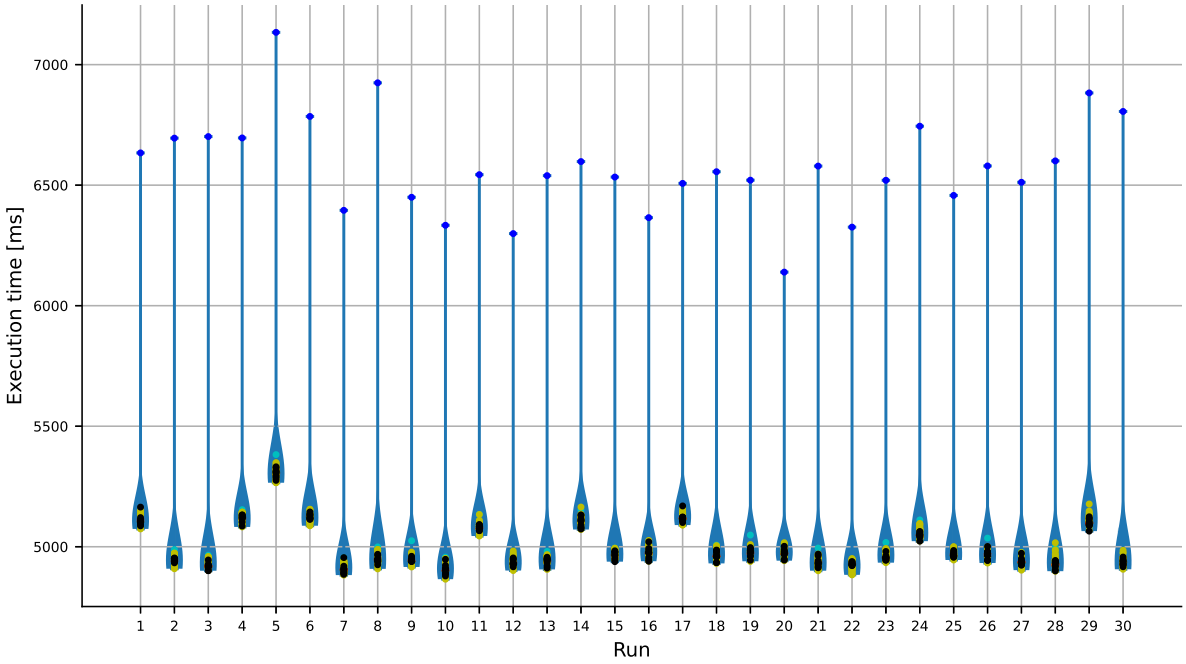


Figure 4.23: Graphchi workload on Java 24

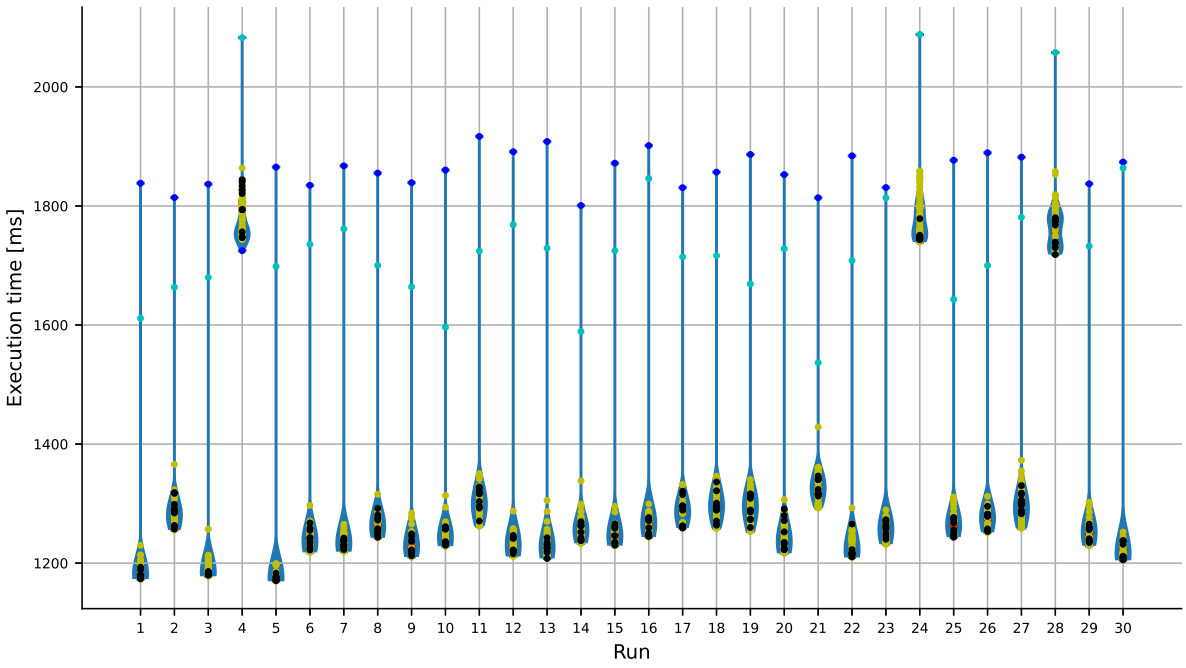


Figure 4.24: Violin plot for runs in par-mnemonics workload on GraalVM 24

4.4.6 Mnemonics [R]

Mnemonics exhibits a distinctive pattern of variability. It is evident that each platform exhibits distinct patterns; consequently, it was deemed appropriate to consider this workload as a discrete entity, rather than to group it with other similar workloads.
Mnemonics constitutes a component of the functional workloads by Renaissance, encompassing future-genetic, par-mnemonics, rx-scrabble and scrabble.

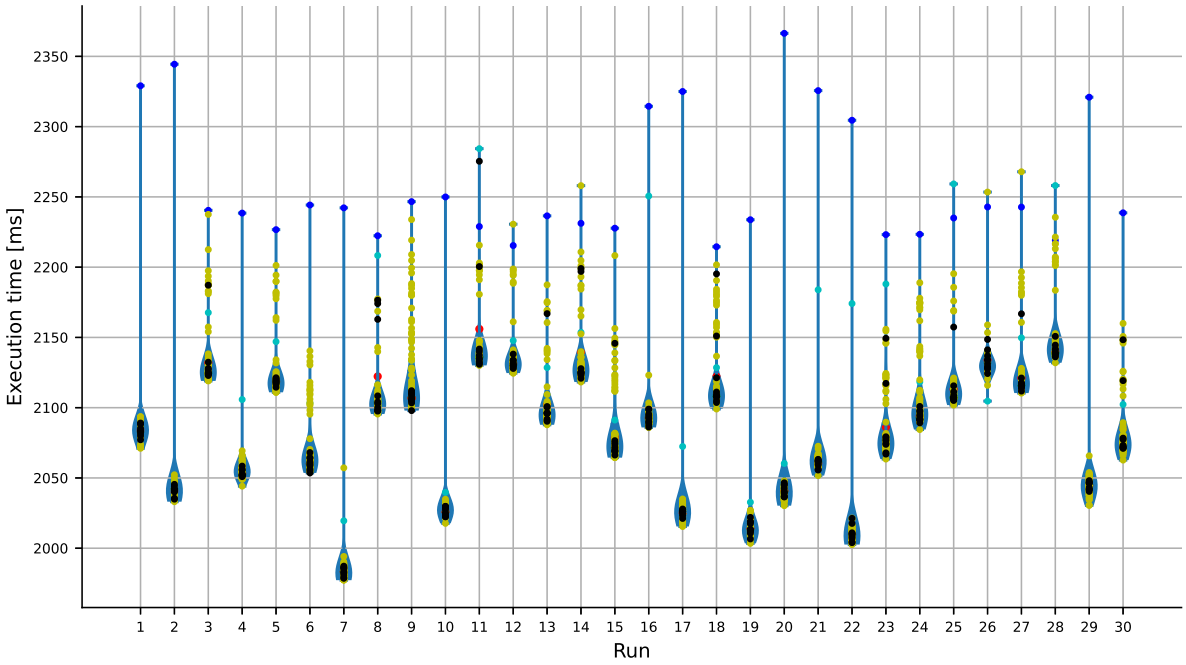


Figure 4.25: Violin plot for runs in par-mnemonics workload on Java 24

Java 23

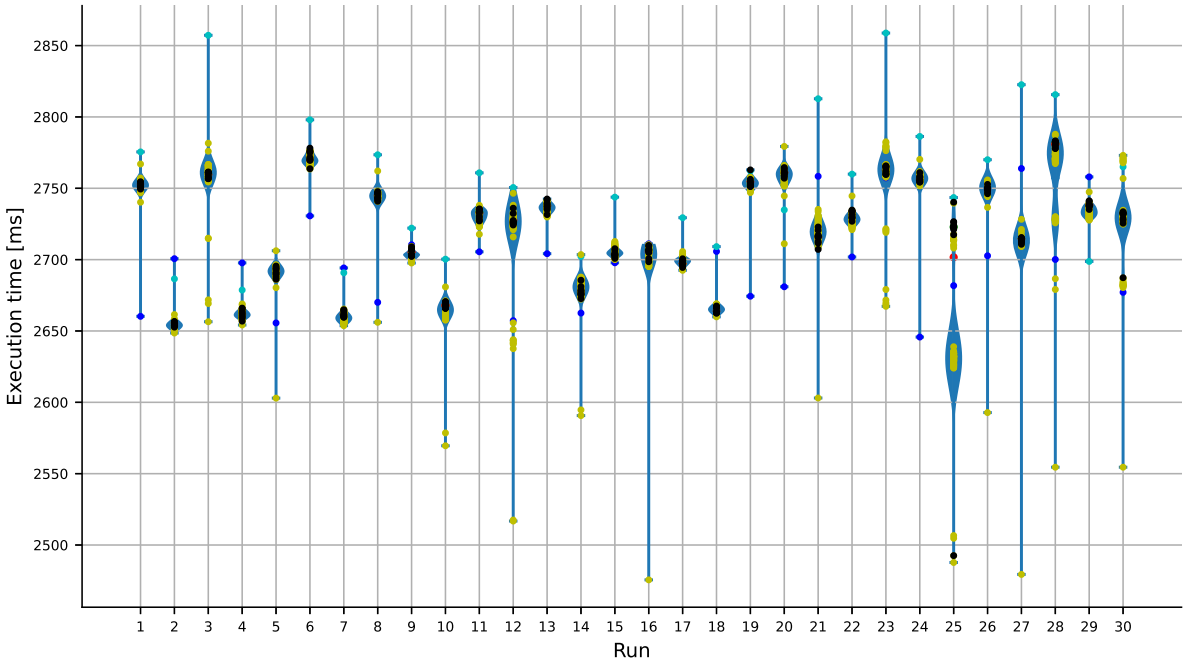


Figure 4.26: Violin plot of mnemonics on platform Java 23

The majority of the runs exhibit performance that falls within the range between the first and second iterations, with the concentration of the curtain concentrated in a single point of the range (Figure 4.27). It has been observed that other runs demonstrate a high degree of variability over the course of their duration (Figure 4.28).

Java 24

In the context of Java 24, a greater number of runs are characterized by a wider range of variability. This phenomenon is attributed to the occurrence of specific iterations within

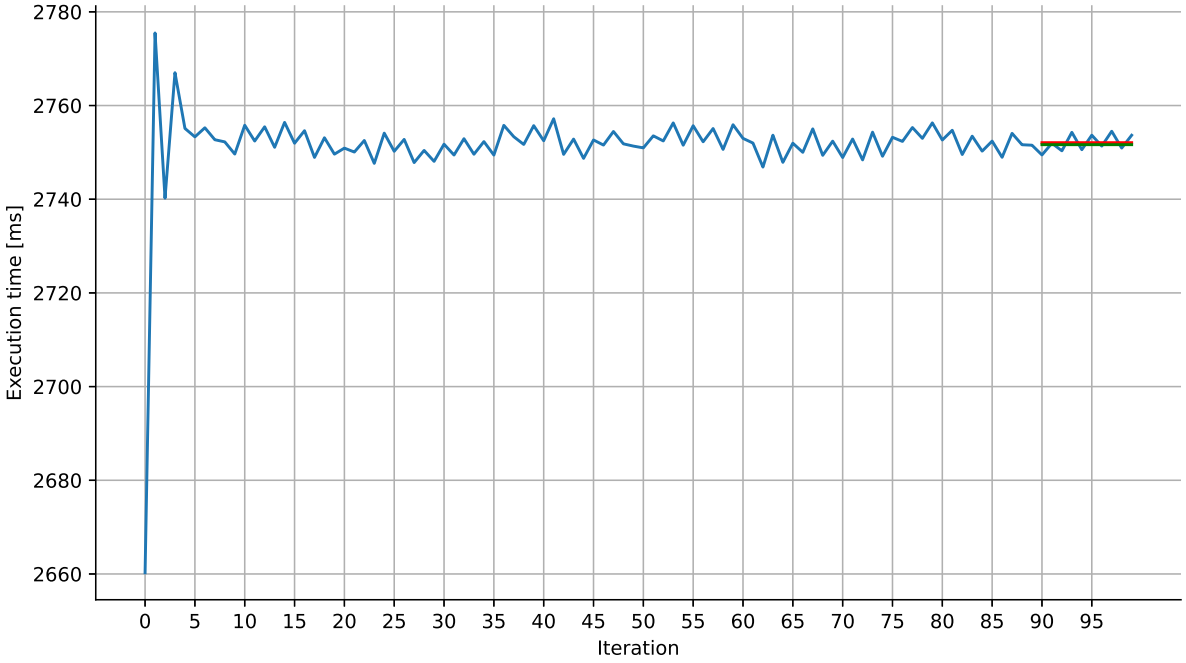


Figure 4.27: Run 1 of mnemonics on Java 23

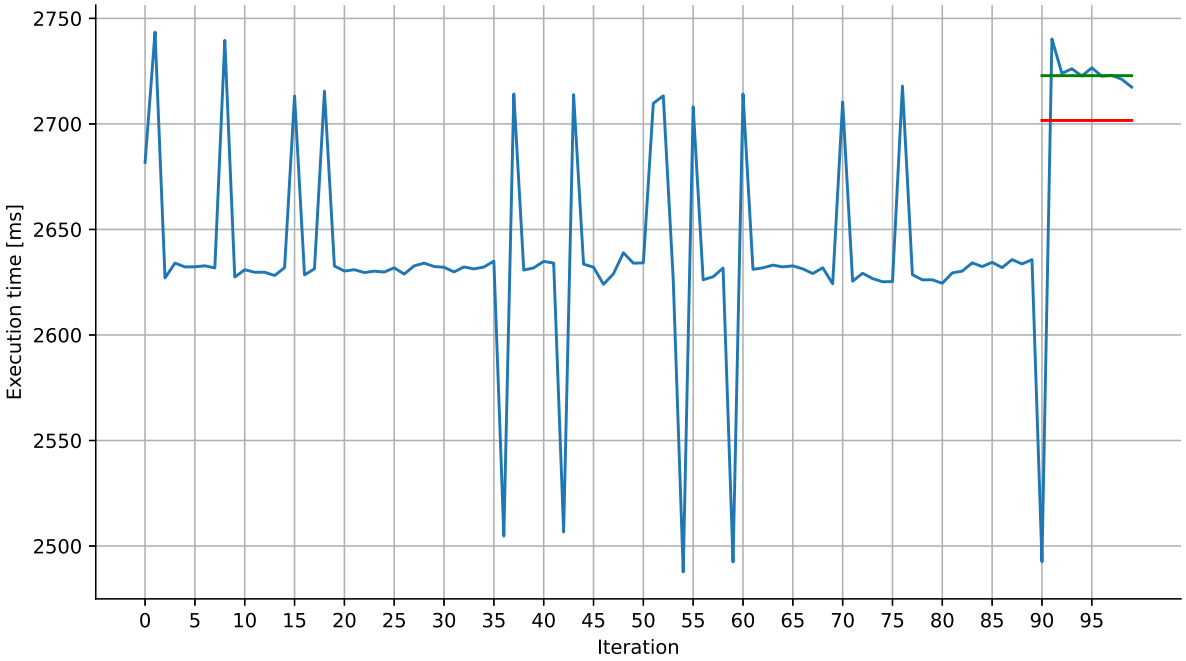


Figure 4.28: Run 25 of mnemonics on Java 23

the sequence, wherein a substantial enhancement in performance is discerned, as it is noticeable in Figure 4.30. However, these iterations are subsequently eliminated in subsequent iterations. In the other cases, the range is considerably more limited due to the absence of spikes, as illustrated in Figure 4.31.

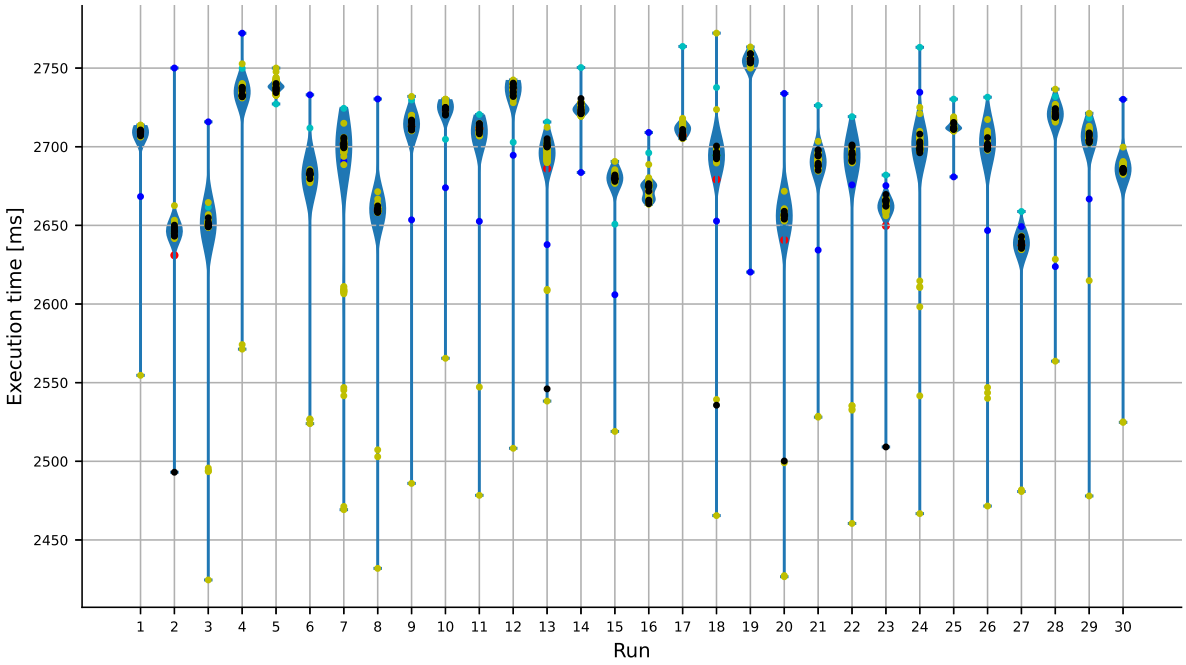


Figure 4.29: Violin plot of mnemonics on platform Java 24

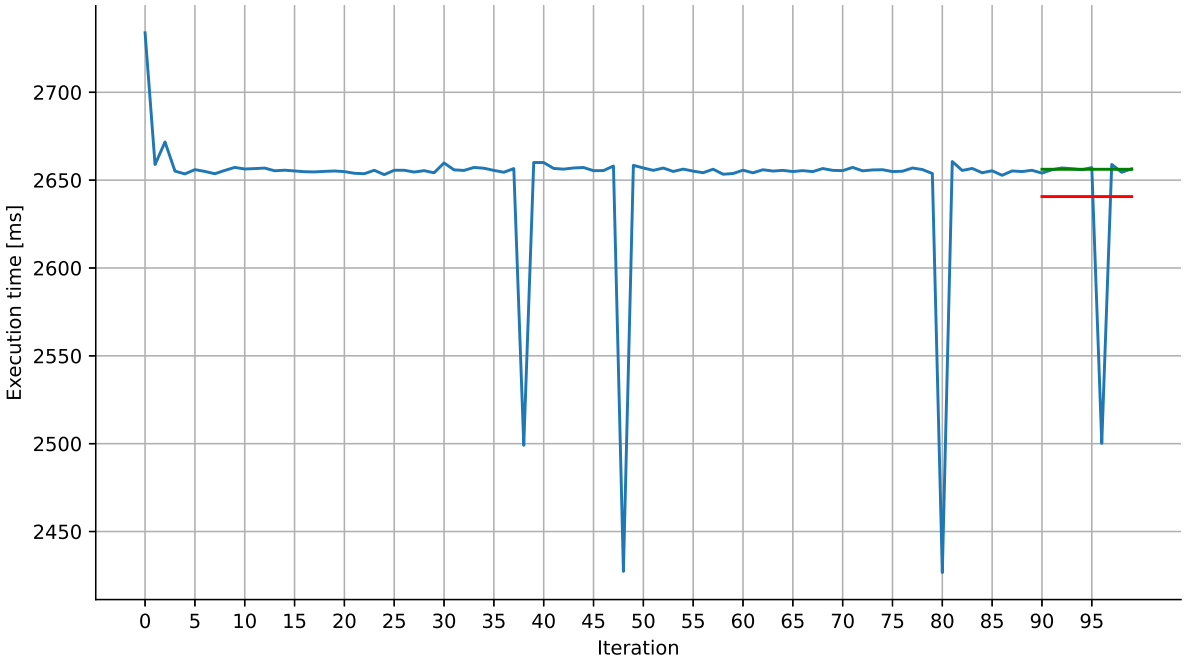


Figure 4.30: Run 20 of mnemonics on Java 24

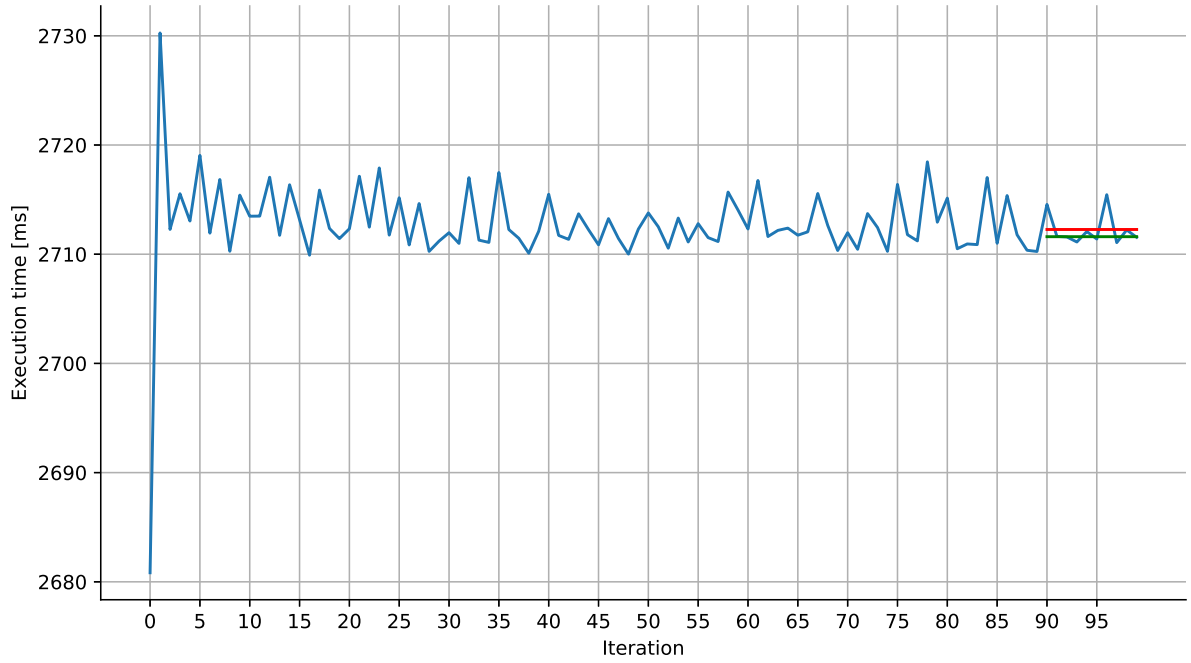


Figure 4.31: Run 25 of mnemonics on Java 24

GraalVM 23

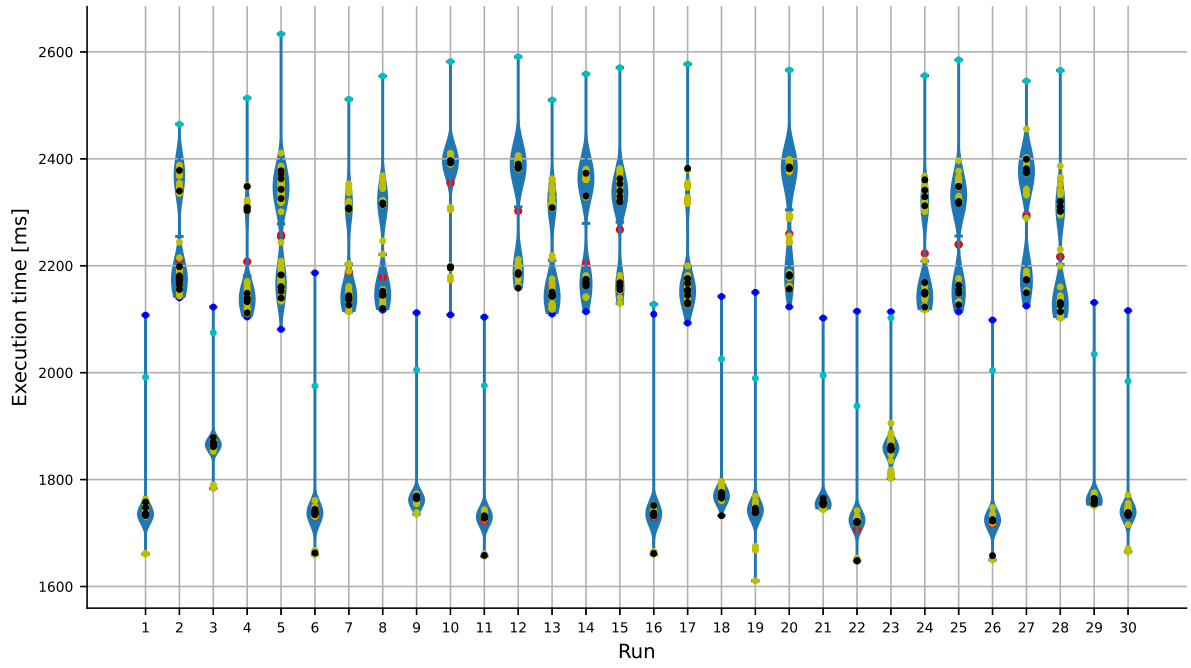


Figure 4.32: Violin plot of mnemonics on platform GraalVM 23

In Graal 23, the run can be categorized into two distinct groups: normal execution and upper fluctuation. In the normal execution scenario depicted in Figure 4.33, subsequent to the initial execution, the performance remains consistent throughout the remainder of the execution. However, in a few instances, minor variations may be observed in the iteration process. In the upper fluctuation, the runs are characterized by a second iteration where the duration is the longest of the entire run. There is a fluctuation between iterations at two accumulation points, as demonstrated in Figure 4.34.

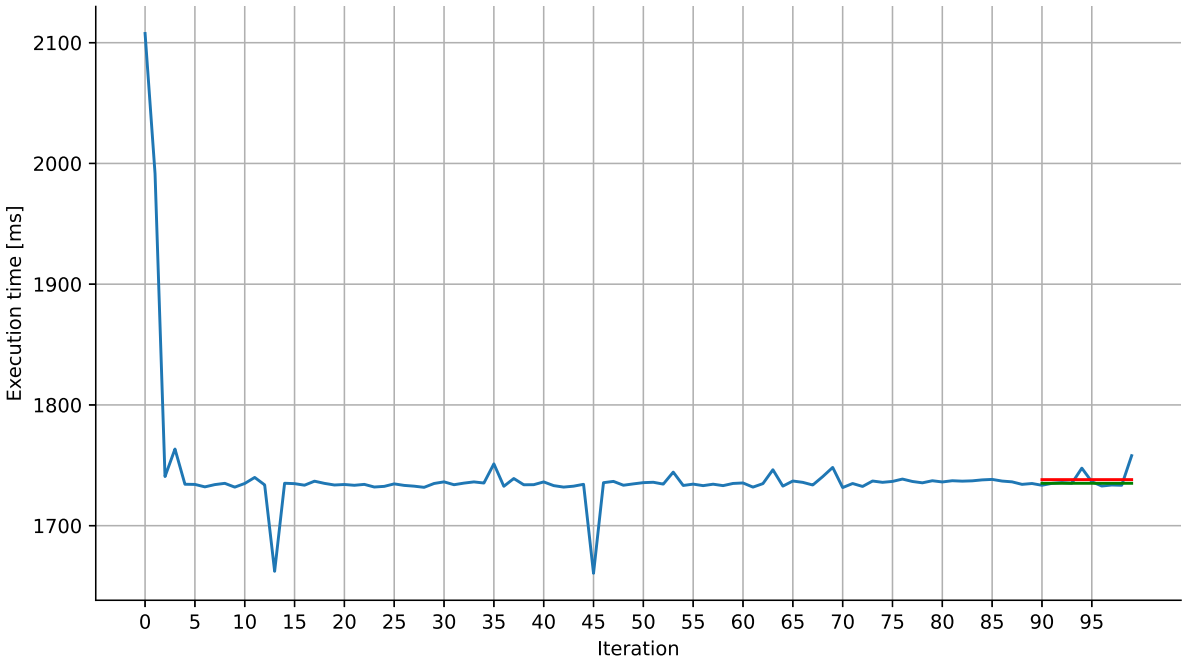


Figure 4.33: Run 1 of mnemonics on GraalVM 23

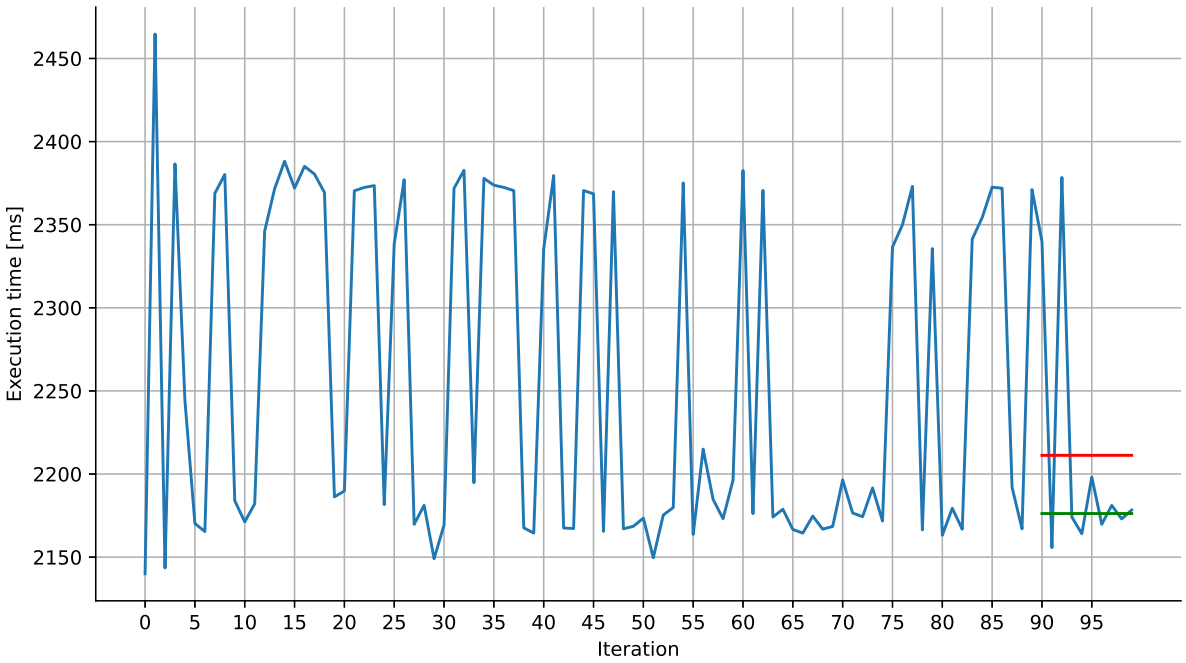


Figure 4.34: Run 2 of mnemonics on GraalVM 23

GraalVM 24

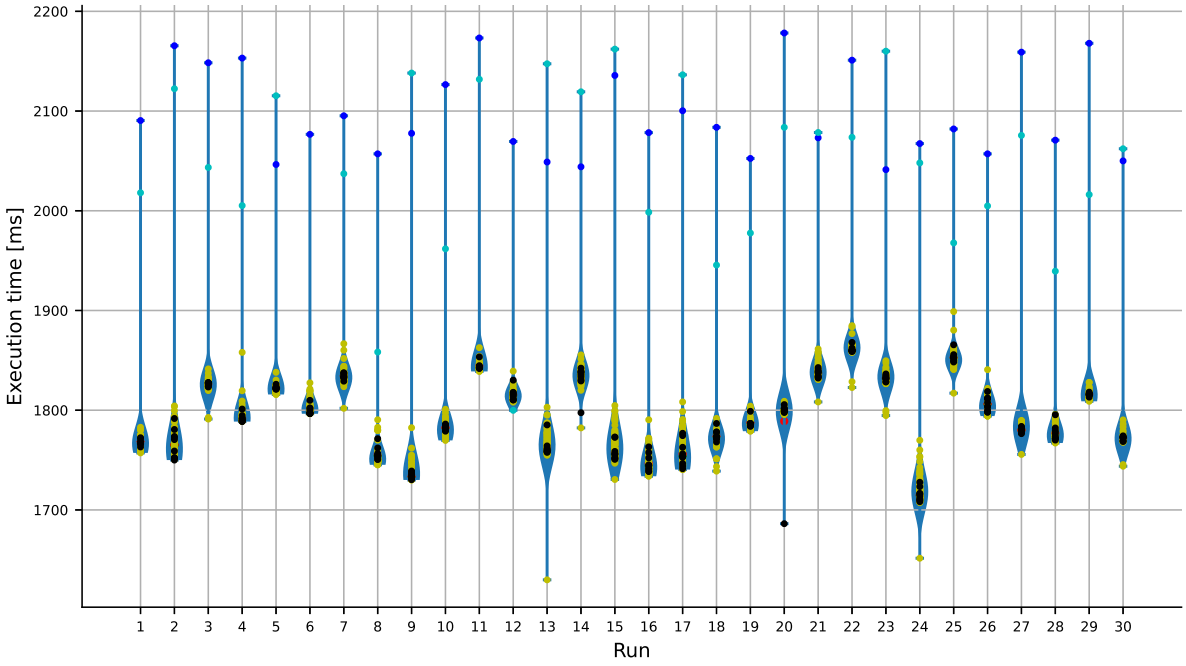


Figure 4.35: Violin plot of mnemonics on platform GraalVM 24

Mnemonics on Graal 24 are distinguished by elevated variability between runs. The iterations can be categorized into two distinct groups, based on the presence or absence of spikes, as illustrated in Figure 4.36 and Figure 4.37. A comparison of the performance of this platform with that of other platforms reveals that this platform exhibits superior overall performance, with an average of 1.7s per iteration. This is followed by Graal 23, which exhibits an average of 2.0s per iteration.

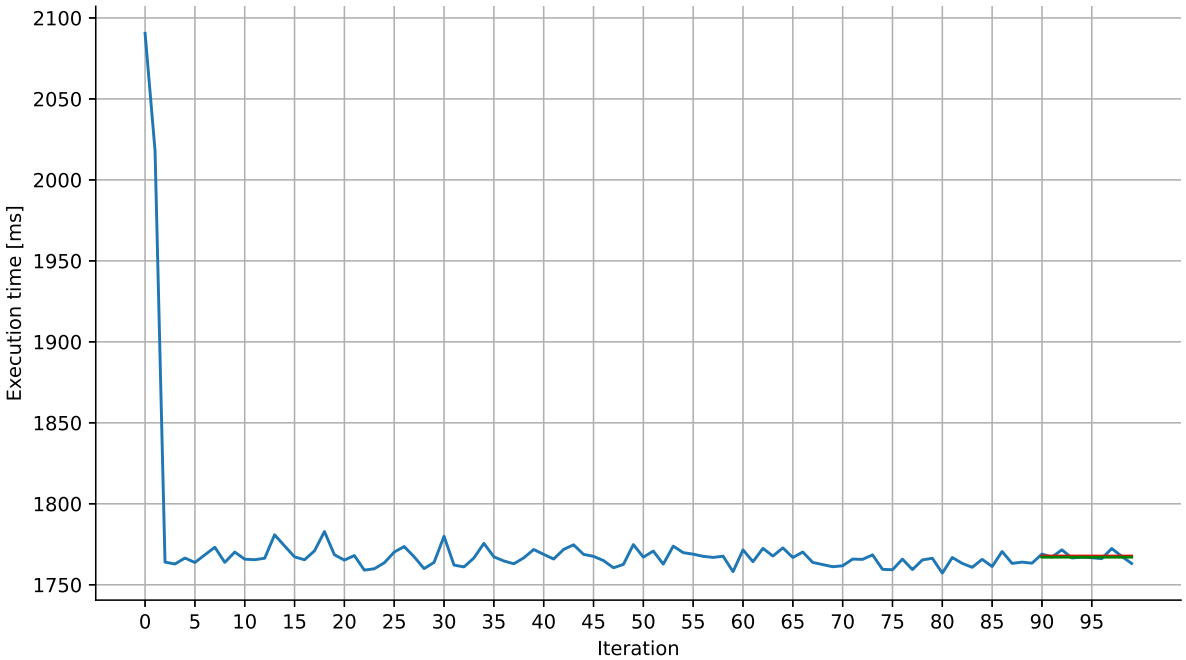


Figure 4.36: Run 1 of mnemonics on GraalVM 23

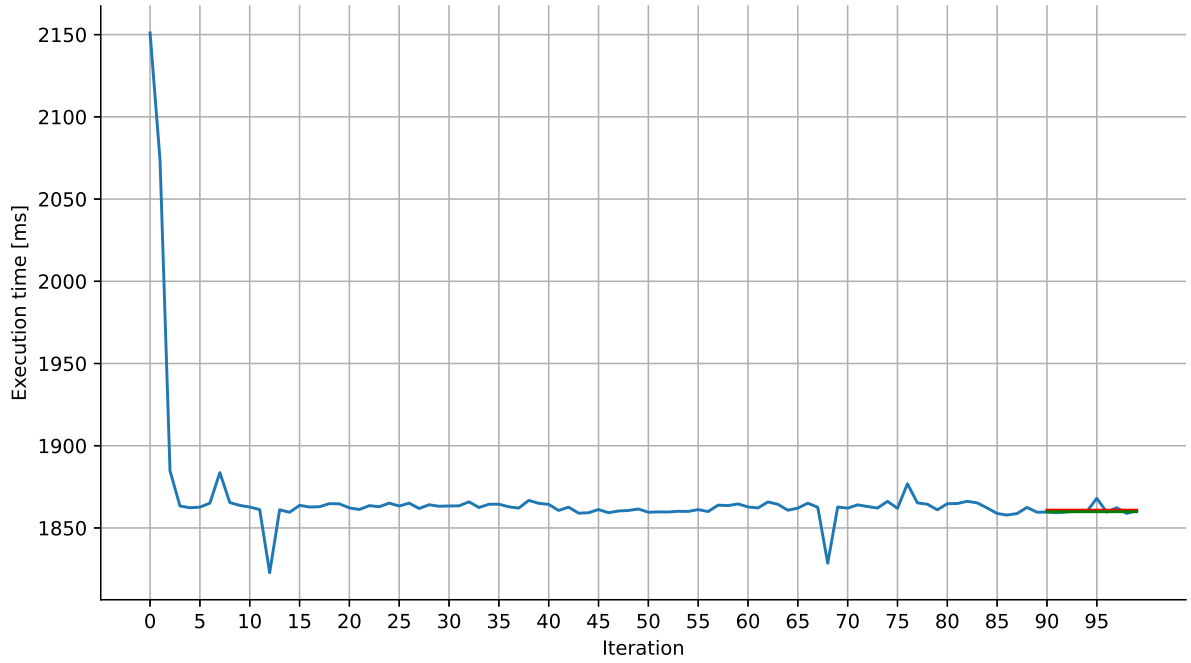


Figure 4.37: Run 22 of mnemonics on GraalVM 23

4.4.7 Scala-doku [R]

Scala-doku is distinct from other workloads for the same reasons as mnemonics. However, it differs from mnemonics in a peculiar way, which only occurs in Java 23, as will be discussed later. Scala-doku is part of the scala workloads from Renaissance together with dotty, philosopher, scala-kmeans and scala-stm-bench7. Graal platforms exhibit greater variability between runs (Figure 4.38a and Figure 4.38b), while demonstrating less significant variability within individual runs. In the case of Java 24 (Figure 4.38d), the variability is found to be significantly more pronounced within a single run than between different runs.

Java 23

The distinctive feature of Java 23 is the manner in which scala-doku functions during its execution. A typical run can be divided into five phases:

1. First iteration
2. Second and third iteration
3. First block of iteration
4. Second block of iteration
5. Third block of iteration

This subdivision of a run enables the identification of three distinct runs: the *low run*, the *high run* and the *long run*. All runs are characterized by a deterioration of performance between blocks. All blocks are characterized by zig-zag between two values.

Low run

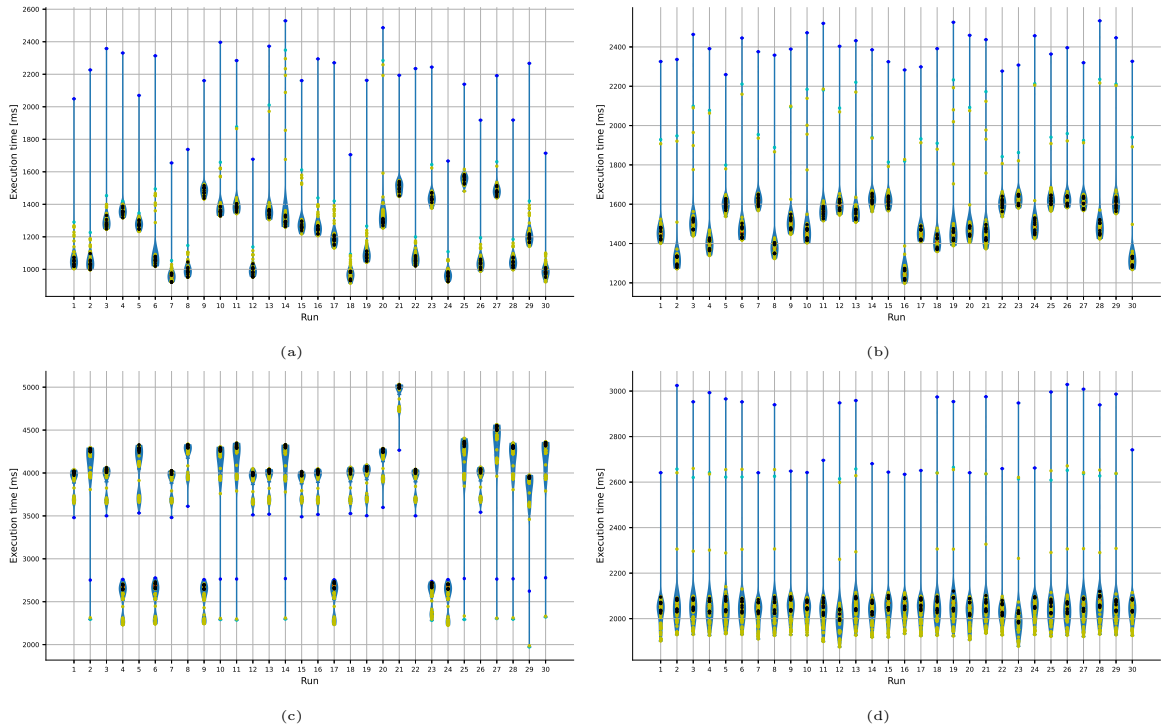


Figure 4.38: Violin plot for runs in scala-doku workload on different VMs. a) scala-doku on GraalVM 23 b) scala-doku on GraalVM 24 c) scala-doku on Java 23 d) scala-doku on Java 24

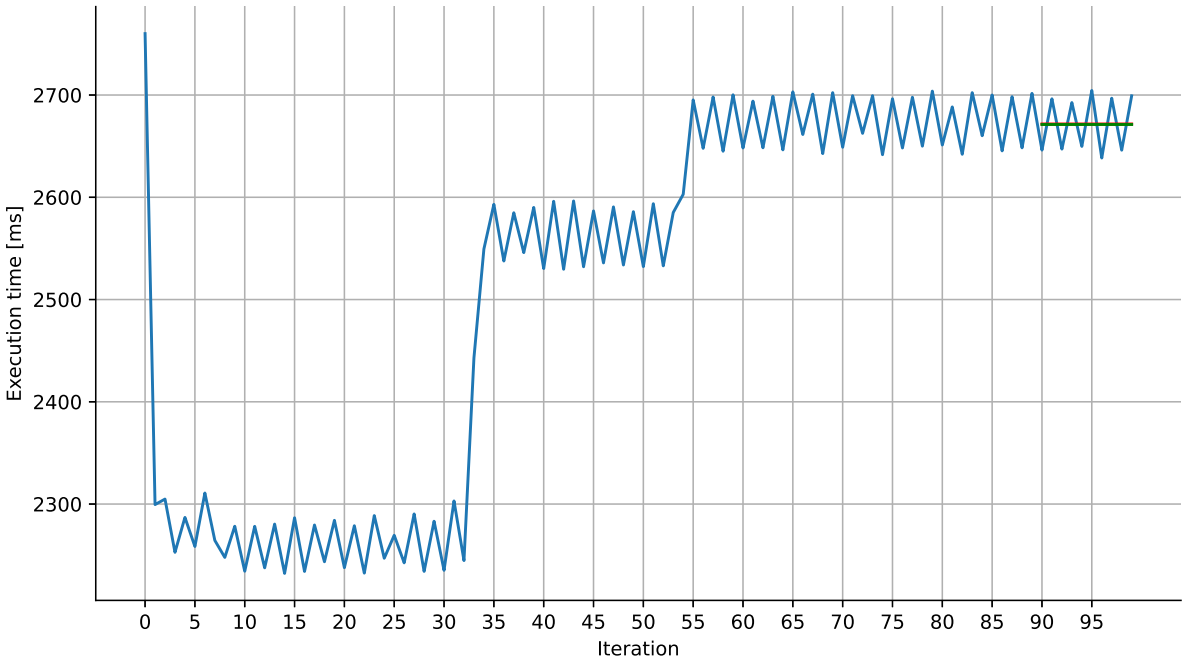


Figure 4.39: Low run in scala-doku (Run 4)

- **First Iteration:** $\sim 2750ms$
- **Second and Third Iteration:** Improvement of performance
- **First Block:** Similar performance to the ones of the previous phase
- **Second Block:** Performance deteriorate significantly from the previous phase without reaching the upper bound from the first iteration.
- **Third Block:** Performance deteriorate more, but not as mush as from the first to

the second block. Execution times are still under the value of the first iteration.

High run

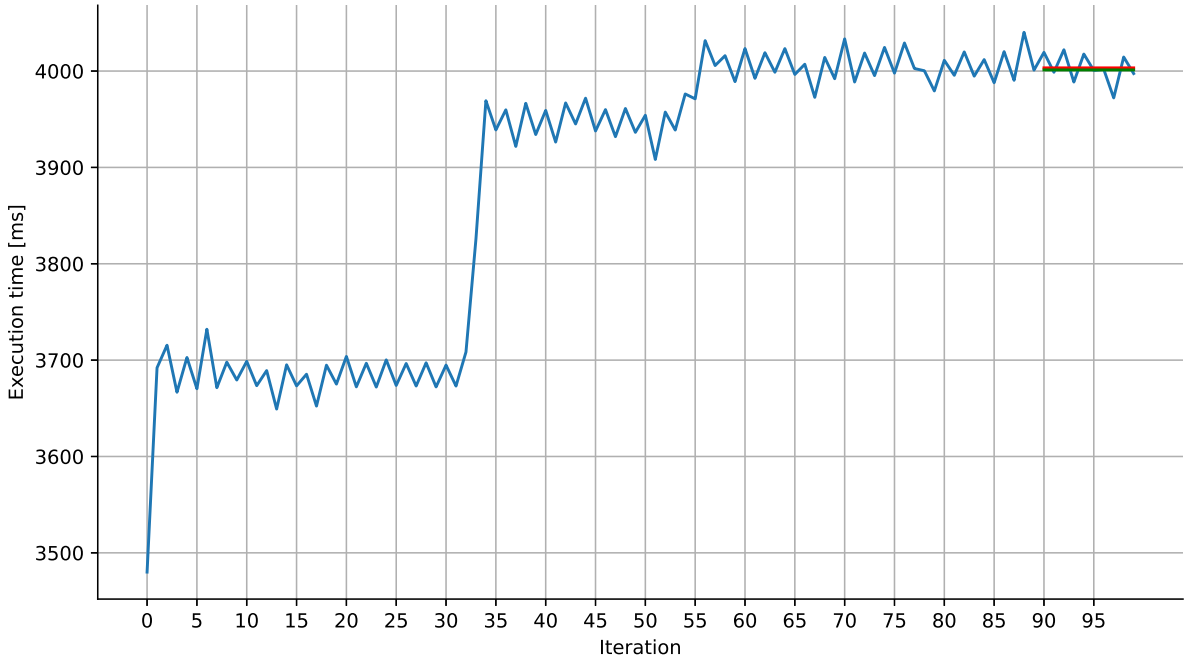


Figure 4.40: High run in scala-doku (Run 1)

- **First Iteration:** $\sim 3500ms$
- **Second and Third Iteration:** Deterioration of performance
- **First Block:** Similar performance to the ones of the previous phase
- **Second Block:** Performance deteriorate significantly from the previous phase.
- **Third Block:** Performance deteriorate more, but not as mush as from the first to the second block.

Long run

- **First Iteration:** $\sim 2750ms$
- **Second and Third Iteration:** Improvement of performance
- **First Block:** Performance deteriorate significantly surpassing the upper bound set by the first iteration.
- **Second Block:** Performance deteriorate from the previous phase.
- **Third Block:** Performance deteriorate more, but not as mush as from the first to the second block.

4.5 Summary

In summary, variability is a phenomenon that is not trivial and is not as rare as commonly thought. The classification of the phenomenon was achieved through the identification of patterns, thus facilitating recognition. Mnemonics and scala-doku have been identified as the most variable and particular workloads of both suites. In the subsequent chapter, an

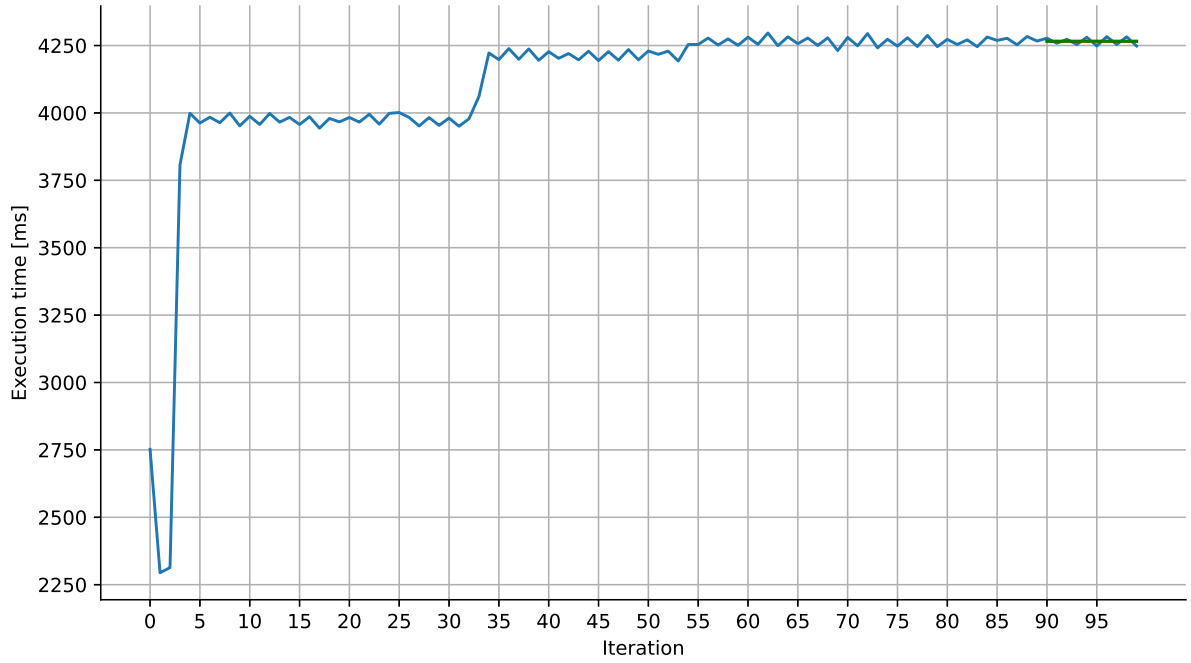


Figure 4.41: Long run in scala-doku (Run 2)

analysis will be conducted of the methodology employed in the identification of causative factors and the attempts made to address variability.
Table 5.3 will demonstrate the pattern of variability present within each workload, indicating whether it is confined to a specific platform.
The first iteration variability manifests in 16 out of 44 workload instances (36.3%), with zxing exhibiting the most variability ($cv = 16.2\%$).
The presence of a periodic pattern is evident in 10 out of the 44 workloads (22.7%), with this pattern manifesting in half of the workloads in each iteration. The chi-squared distribution is the most informative, as its pattern is consistently evident in the majority of iterations.
Intra-run performance variability is characterised by two primary workloads that are distinct from the others. Sunflow exhibits the highest intra-run mean variability across all platforms, while par-mnemonics demonstrate the most significant intra-run variation on Graal. This pattern was identified as a periodic pattern in 10 out of 44 workloads (22.7%).
Inter-run variability is a pattern that involves both stable workloads where intra-variability is not present, as is the case with graphchi, and workloads where some other form of inner variance is present, as is the case with par-mnemonics. It is evident that chi-squared is the most variable among all workloads.
In the subsequent chapter, a more comprehensive analysis of the underlying causes of these patterns will be conducted, with the objective of devising effective mitigation strategies.

	First Iteration				Periodic Pattern				Intra-run				Inter-run			
Workload	J23	J24	G23	G24	J23	J24	G23	G24	J23	J24	G23	G24	J23	J24	G23	G24
akka-uct	X	X	X	X					X	X	X	X				
als																
avrora																
batik		X		X	X	X	X	X								
biojava					X	X	X	X	X	X	X	X				
cassandra																
chi-square			X	X	X	X	X	X	X	X	X	X	X	X	X	X
dec-tree					X	X	X	X								
dotty																
eclipse																
fj-kmeans			X	X												
fop																
future-genetic																
gauss-mix					X	X	X	X								
graphchi	X	X	X	X										X		
h2	X	X														
h2o	X	X	X	X					X	X	X	X				
jme																
jython					X	X	X	X								
kafka																
log-regression																
luindex									X	X	X	X				
lusearch																
mnemonics	X	X	X	X					X	X	X	X	X	X	X	X
movie-lens					X	X	X	X								
naive-bayes					X	X										
page-rank			X	X											X	X
par-mnemonics									X	X	X	X	X	X	X	X
philosophers																
reactors																
rx-scrabble	X	X		X	X	X	X	X								
scala-doku	X	X	X	X	X					X				X	X	X
scala-kmeans	X		X	X	X	X							X	X	X	X
scala-stm-bench7	X		X						X	X	X	X	X	X	X	X
scrabble	X	X	X	X												
spring													X	X	X	X
sunflow		X	X	X					X	X	X	X	X	X	X	X
tomcat																
xalan																
zxing	X	X	X	X												

Table 4.5: Alphabetically sorted workloads evaluated across variation types and platforms.
The X represents the presence of the pattern.

Chapter 5

Tunic Dynamic Compilation and Its Effect on Performance Variability

In this chapter, we present our analysis of different workloads exhibiting variability, starting from mnemonics, using ProfDiff and JitWatch. We also describe our attempts to improve the performance of some of these workloads with a few compiler-level tweaks. Section 5.1 describes the measures taken to ensure that ProfDiff overhead did not impact the variability patterns present in the different workloads. Section 5.2 analyses different metrics to understand the differences in inlining between the various runs. Section 5.3 goes into more detail, analysing the ProfDiff reports and the information provided by JitWatch on the individual runs. Section 5.4 concludes by showing how the choices made impacted the different workloads. Section 5.5 looks at an alternative approach attempted in scala-doku.

5.1 Profdiff & JitWatch Environment

Profdiff [17] requires two things to create its reports:

- *Perf Directory* - A directory containing binary files with informations about the CPU
- *JSON Proftool Report* - Containing information about the inlining and hotness of methods involved in the run.

On the other hand, JitWatch only requires `LogCompilation` to work.
Respect to Figure 4.24, mx instrumentation changes the patterns of the workload drastically adding ∼100% execution time overhead.

We managed to extract the Java command line executed behind the `mx benchmark` command line [30]. We removed unnecessary flags, fixed the heap size and added log compilation, resulting in the following snippet:

```
perf record -k 1 --freq 1000 --event cycles --output "${PROFTOOL_DIR}/perf_binary_file"
    \
    "$HOME_DIR/platforms/graalvm-${version}/files/bin/java" \
    -server \
    -XX:+UnlockExperimentalVMOptions \
    -XX:+UnlockDiagnosticVMOptions \
    -XX:+DebugNonSafepoints \
    -XX:+EnableJVMCI \
    -XX:+UseJVMCICompiler \
```

```
    -XX:+LogCompilation \
    -XX:LogFile="${PROFTOOL_DIR}/log_compilation" \
    --add-exports=java.base/jdk.internal.misc=jdk.graal.compiler \
    -Dgraalvm.locatorDisabled=true \
    -agentpath:"/home/user2/mx/mxbuild/linux-amd64-jdk24/com.oracle.jvmtiasmagent/linux-amd64/glibc/libjvmtiasmagent.so"="${PROFTOOL_DIR}/jvmti_asm_file" \
    -agentpath:"$HOME_DIR/benchmarks/renaissance/files/libubench-agent.so" \
    -Xss2M -Xms12g -Xmx12g \
    -Djdk.graal.CompilationFailureAction=Diagnose \
    -Djdk.graal.DumpOnError=true \
    -Djdk.graal.ShowDumpFiles=true \
    -Djdk.graal.PrintGraph=Network \
    -Djdk.graal.ObjdumpExecutables=objdump,gobjdump \
    -Djdk.graal.CompilerConfiguration=community \
    -Djvmci.Compiler=graal \
    -Djdk.graal.TrackNodeSourcePosition=true \
    -Djdk.graal.OptimizationLog=Directory \
    -Djdk.graal.OptimizationLogPath="$PROFTOOL_DIR/optimization_log" \
    -jar "$HOME_DIR/benchmarks/renaissance/files/renaissance-gpl-0.15.0.jar" \
        --plugin "$HOME_DIR/benchmarks/renaissance/files/plugin-ubenchagent-assembly-0.0.1.jar" \
        --with-arg JVM:compilations,PAPI:ref-cycles,PAPI:instructions,PAPI:cache-references,PAPI:cache-misses,PAPI:branch-instructions,PAPI:branch-misses \
        --plugin "$HOME_DIR/benchmarks/renaissance/files/plugin-jmxmemory-assembly-0.0.1.jar" \
        --plugin "$HOME_DIR/benchmarks/renaissance/files/plugin-jmxtimers-assembly-0.0.2.jar" \
        --csv "/home/user2/profiler/mnemonics-graalvm-${version}/${CSV_FILE}" --repetitions 100 "${benchmark}"
```

Listing 5.1: Profiler Command Line Snippet

This allow us to restore some patterns from the baseline and proceed with analysing more in details the information about inlining and hotness.

5.2 Methodology

Qualitative division of patterns allows us to form an initial impression of what could be a discriminating factor in distinguishing good runs from bad runs. To obtain a quantitative classification of the whole, we expanded the ProfDiff source code by adding a subcommand called `jit-bulk`.
This allows you to collect the ten hottest methods from each run in a CSV file, along with other information such as the percentage of hotness, the method name and the number of cycles. Using this information, you can cluster runs with the same hot methods and choose a representative run from the group. This can then be compared with other clusters (inter-cluster comparison), or with runs within the same cluster (intra-cluster comparison). Inter-cluster analysis enables you to identify differences by determining which method was used most frequently and comparing the optimisation tree between runs. Intra-cluster analysis reveals which part of the inlining process caused differences between runs with the same hot methods but different performance levels.
For this reason, we created three visualisations to help us understand how these groups differ from each other and how they vary internally. Furthermore, using ProfDiff and JitWatch reports made it possible to determine which of these differences were significant. The three visualisations include:

1. A scatter plot showing the distribution of methods relative to hotness percentages
2. A scatter plot showing the relationship between methods and runs.
3. A line chart for comparing selected runs and identifying where and how hot methods differ.

5.3 Mnemonics Analysis

Following a comprehensive review of the two benchmarks that demonstrated the greatest variability, it was determined that mnemonics would serve as the initial workload for in-depth analysis. In view of the fact that there is an intention on the part of the relevant parties to utilise both ProfDiff and JitWatch, the focus is on the Graal platform, and more specifically on GraalVM 23. The second graph is utilised to differentiate between the various clusters, which are categorised based on the colour indicated in Figure 5.1.

As illustrated in Figure 5.2, which provides a more detailed depiction of the distribution of hotness between methods and clusters, it becomes evident that certain methods exhibit consistent hotness across disparate clusters (e.g. `HashMap$EntrySpliterator .forEachRemeaning(Consumer)`). In contrast, others demonstrate variable percentages of hotness across different clusters (e.g. `MnemonicsCoderWithStream .lambda$encode$9 (String, Set, String)`).

Following a comprehensive analysis, the final stage of the research project involved the investigation of the disparities between various clusters comprising a limited number of runs and the largest cluster, which was represented by the orange, blue, and green clusters. It was established that the pink cluster, which exclusively contains run 27, differs from the orange cluster solely in terms of the 10th hottest method (Figure 5.3). This finding enables the amalgamation of these two clusters into a single entity.

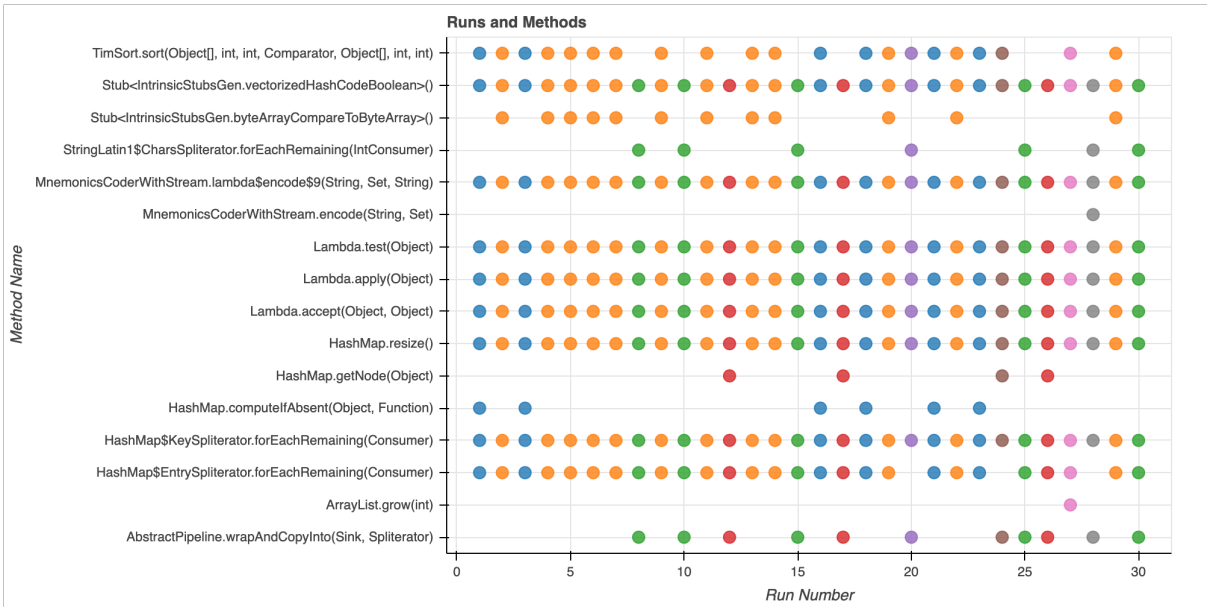


Figure 5.1: Mnemonics Run-Method scatter plot

Utilising the visual data provided by the preceding plot and the comprehensive results from ProfDiff, of which we have an excerpt of a method comparing run 6 and run 12 in GraalVM 23 (Listing 5.2), we hypothesise that inlining performed by C1 and subsequently re-inlined by G1 may yield a distinct outcome when compared to the inline produced by G1 alone. In the next subsection we will analyse how we approach this hypothesis.

```
Experiment 1 with execution ID 126201 (JIT) - Mnemonics GraalVM 23 Run 6
    Collected optimization logs for 311 compilation units
    Collected proftool data for 357 compilation units
    Graal-compiled methods account for 70.27% of execution
    5 hot compilation units account for 60.95% of execution
    534.97 billion cycles total
    Top methods
```

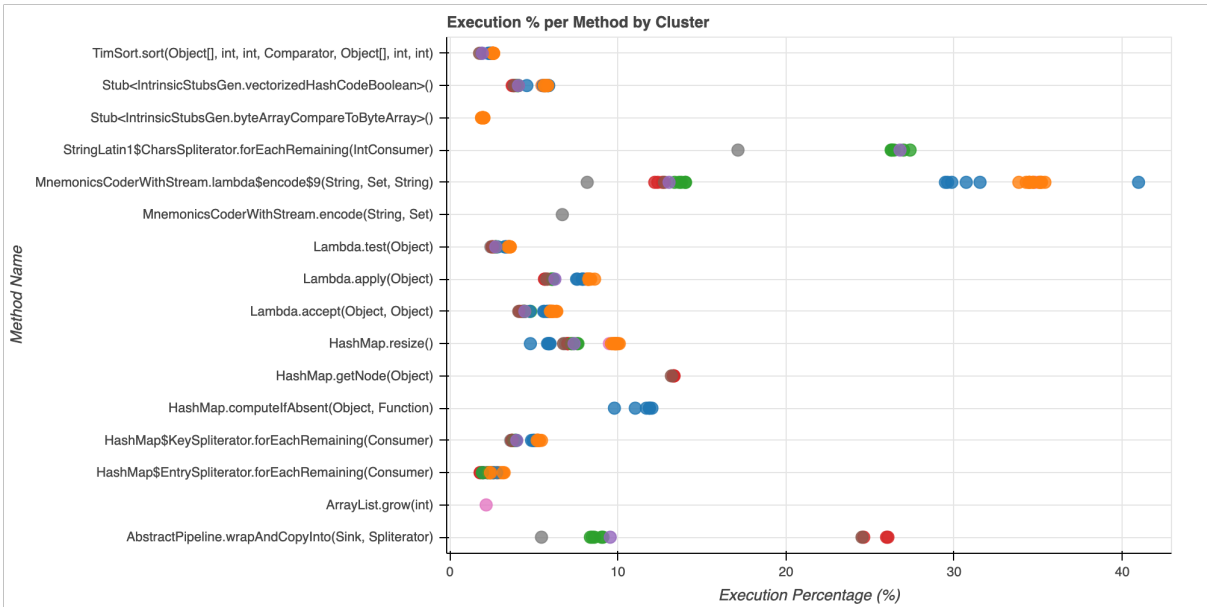


Figure 5.2: Execution % per method by cluster

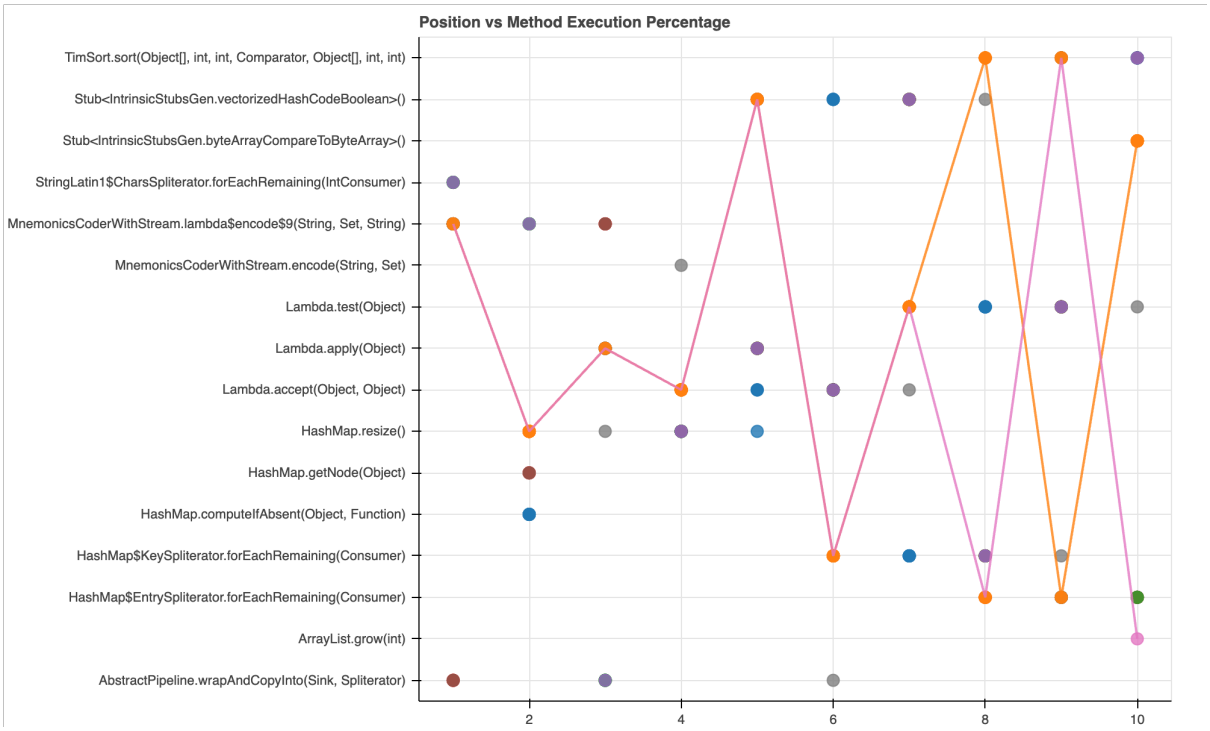


Figure 5.3: Hot methods sorted by position - Pink line run 24 and Orange line run 6

```
        Execution     Cycles  Level      ID  Method
           34.29%     183.45      4    1792  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream.lambda$encode$9(java.lang.String, java.util.Set, java.lang.
    String)
            9.87%      52.82      4    1642  java.util.HashMap.resize()
            8.20%      43.87      4    1582  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream$$Lambda.0x00007f99a313a6d0.apply(java.lang.Object)
            5.99%      32.05      4    1644  java.util.stream.Collectors$$Lambda.0
    x80000004d.accept(java.lang.Object, java.lang.Object)
            5.65%      30.24                 Stub<IntrinsicStubsGen.
    vectorizedHashCodeBoolean>
            5.23%      27.95      4    1558  java.util.HashMap$KeySpliterator.
    forEachRemaining(java.util.function.Consumer)
            3.59%      19.22      4    1538  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream$$Lambda.0x00007f99a3139b18.test(java.lang.Object)
            2.60%      13.90      4    1897  java.util.TimSort.sort(java.lang.Object[],
    int, int, java.util.Comparator, java.lang.Object[], int, int)
            2.47%      13.23      4    1574  java.util.HashMap$EntrySpliterator.
    forEachRemaining(java.util.function.Consumer)
            1.91%      10.24                 Stub<IntrinsicStubsGen.
    byteArrayCompareToByteArray>

Experiment 2 with execution ID 181693 (JIT) - Mnemonics GraalVM 23 Run 12
    Collected optimization logs for 319 compilation units
    Collected proftool data for 320 compilation units
    Graal-compiled methods account for 63.86% of execution
    4 hot compilation units account for 55.82% of execution
    533.00 billion cycles total
    Top methods
        Execution     Cycles  Level      ID  Method
           34.62%     184.54      4    1768  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream.lambda$encode$9(java.lang.String, java.util.Set, java.lang.
    String)
            9.68%      51.57      4    1622  java.util.HashMap.resize()
            8.26%      44.03      4    1559  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream$$Lambda.0x00007f4d3b13a6d0.apply(java.lang.Object)
            6.17%      32.89      4    1624  java.util.stream.Collectors$$Lambda.0
    x80000004d.accept(java.lang.Object, java.lang.Object)
            5.83%      31.08                 Stub<IntrinsicStubsGen.
    vectorizedHashCodeBoolean>
            5.35%      28.53      4    1536  java.util.HashMap$KeySpliterator.
    forEachRemaining(java.util.function.Consumer)
            3.54%      18.87      4    1534  org.renaissance.jdk.streams.
    MnemonicsCoderWithStream$$Lambda.0x00007f4d3b139b18.test(java.lang.Object)
            2.52%      13.46      4    1877  java.util.TimSort.sort(java.lang.Object[],
    int, int, java.util.Comparator, java.lang.Object[], int, int)
            2.33%      12.44      4    1551  java.util.HashMap$EntrySpliterator.
    forEachRemaining(java.util.function.Consumer)
            2.03%      10.81                 Stub<IntrinsicStubsGen.
    byteArrayCompareToByteArray>

Method java.util.Arrays$ArrayItr.next()
    In experiment 1
        2 compilations (1 of which are hot)
        Compilations
            1792#1 consumed 48.80% of Graal execution, 34.29% of total *hot*
              |_ a fragment of java.util.HashSet.<init>(Collection)
                               java.util.AbstractCollection.addAll(Collection) at bci 24
                               java.util.Arrays$ArrayItr.next() at bci 19
             1623 consumed  0.00% of Graal execution,  0.00% of total
    In experiment 2
        1 compilations (1 of which are hot)
        Compilations
             1624 consumed  9.66% of Graal execution,  6.17% of total *hot*
    Compilation fragment 1792#1 consumed 48.80% of Graal execution, 34.29% of total in
    experiment 1 vs
    Compilation unit  1624 consumed  9.66% of Graal execution,  6.17% of total in
    experiment 2
        Inlining tree matching
            There are no differences
        Optimization tree matching
            . RootPhase
                . EnterpriseHighTier
```

```
59                      . PriorityInliningPhase
60                          - RootPhase
61                              - PhaseSuite
62                                  - HotSpotGraphBuilderPhase
63                              - BoxNodeIdentityPhase
64                              - DominatorBasedGlobalValueNumberingPhase
65                          - PartialEscapePhase
66                          [...]
67                          - RootPhase
68                              - PhaseSuite
69                                  - HotSpotGraphBuilderPhase
70                              - BoxNodeIdentityPhase
71                              - DominatorBasedGlobalValueNumberingPhase
72                          - PartialEscapePhase
73                              - SchedulePhase
74                          - PartialEscapePhase
75                              - SchedulePhase
76                      - DisableOverflownCountedLoopsPhase
77                      . IterativeConditionalEliminationPhase
78                          - ConditionalEliminationPhase
79                      - LoopBoundOptimizationPhase
80                      - LoopFullUnrollPhase
81                      - InjectLoopCounterStampsPhase
82                      - LoopPeelingPhase
83                      - InjectLoopCounterStampsPhase
84                      - LoopBoundOptimizationPhase
85                      - LoopUnswitchingPhase
86                      . ConditionalEliminationPhase
87                          - ConditionalElimination FixedGuardElimination at bci 11
88                      - EnterprisePartialUnrollPhase
89                      . DuplicationPhase
90                          - HighTierDuplicationSimulationPhase
91                          - ConvertDeoptimizeToGuardPhase
92                          - HighTierDuplicationSimulationPhase
93                      . FinalPartialEscapePhase
94                          - PhaseSuite
95                              - PullThroughPhiPhase
96                              - DuplicationPhase
97                                  - HighTierDuplicationSimulationPhase
98                              - LoopFullUnrollPhase
99                          - PhaseSuite
100                             - PullThroughPhiPhase
101                             - DuplicationPhase
102                                 - HighTierDuplicationSimulationPhase
103                             - LoopFullUnrollPhase
104                         - PhaseSuite
105                             - PullThroughPhiPhase
106                             - DuplicationPhase
107                                 - HighTierDuplicationSimulationPhase
108                             - LoopFullUnrollPhase
109                         - SchedulePhase
110                         - SchedulePhase
111                 . EnterpriseMidTier
112                     . FloatingReadPhase
113                         + FloatingRead FixedWithFloatingReplacement at bci 1
114                         + FloatingRead FixedWithFloatingReplacement at bci 7
115                         + FloatingRead FixedWithFloatingReplacement at bci 10
116                     - InjectLoopCounterStampsPhase
117                     - LoopBoundOptimizationPhase
118                     . IterativeConditionalEliminationPhase
119                         . ConditionalEliminationPhase
120                             - ConditionalElimination GuardElimination at bci 34
121                             + ConditionalElimination FixedGuardElimination at bci 11
122                     - LoopRotationPhase
123                     - OptimizeAddressesInLoopsPhase
124                     - LoopInversionPhase
125                         - InjectLoopCounterStampsPhase
126                     - OptimizeLoopAccessesPhase
127                     - LoopPeelingPhase
128                     - LoopUnswitchingPhase
129                     - OptimizeAddressesInLoopsPhase
130                     - EnterprisePartialUnrollPhase
131                     - LoopRotationPhase
```

```
                    - LoopFullUnrollPhase
                    - InjectLoopCounterStampsPhase
                    . IterativeConditionalEliminationPhase
                        - ConditionalEliminationPhase
                    - LoopVectorizationPhase
                        - RemoveEmptyLoopsPhase
                    - RemoveEmptyLoopsPhase
                    - LoopPartialUnrollPhase
                    . DeoptimizationGroupingPhase
                        + DeoptimizationGrouping DeoptimizationGrouping at bci 10
                . EnterpriseLowTier
                    . UseTrappingNullChecksPhase
                        + UseTrappingNullChecks ImplicitNullCheck at bci 10
                    - InvertedLoopPhiUsageMinificationPhase
                    - DetectInvertedLoopPhase
```

Listing 5.2: ProfDiff inlining differences between run

5.3.1 Disabling C1 in Mnemonics

The experiment is initiated with the removal of the C1 layer. In order to achieve this objective, it is necessary to add two flags to the command line:

```
-XX:+UnlockDiagnosticVMOptions
-XX:CompilationMode=high-only
```

The findings are remarkable: mnemonics is no longer observed to display unusual patterns on any given platform, and a significant proportion of other workloads now demonstrate a complete absence of peculiar behaviour, particularly those characterised by periodic patterns.
As demonstrated in Figure 5.4, the variability observed on the baseline is mitigated, yet residual inter-run variability remains.

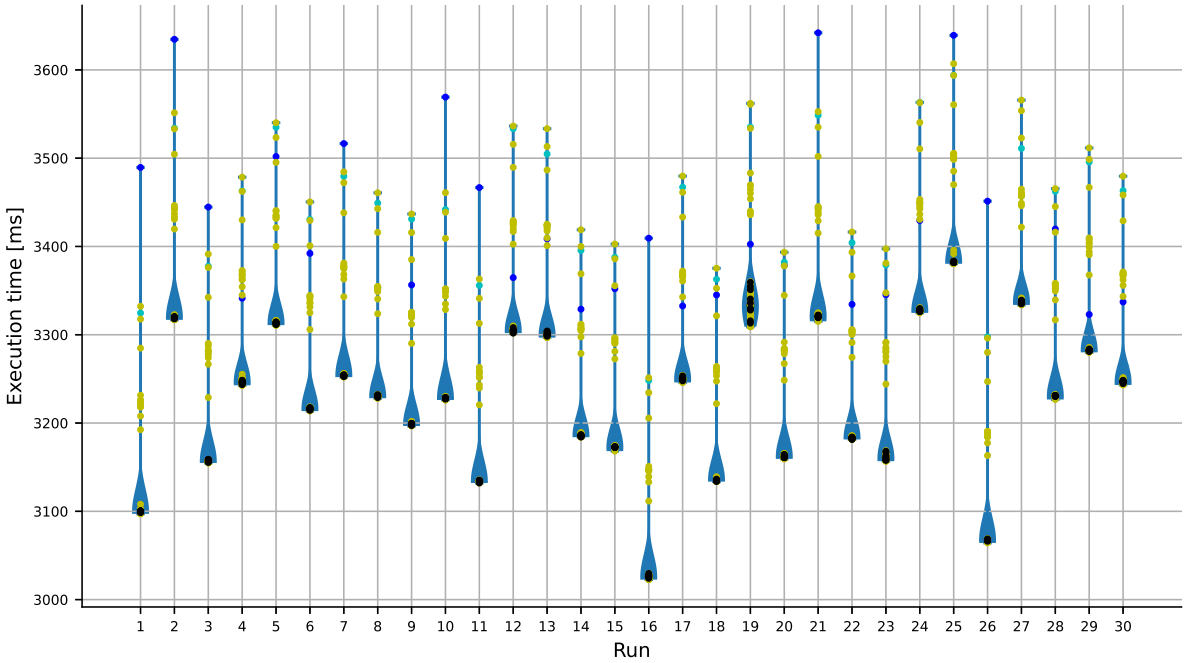


Figure 5.4: Violin plot of mnemonics on Java 23 without C1 tiers

5.3.2 Increasing Inlining Threshold in Mnemonics

Analysis of the results on JitWatch is conducted to ascertain the factors that are yet to be optimised. The findings of the present study demonstrate a consistent correlation

between the number of inlining failures and the overall performance of each run. It is therefore necessary to increase inlining separately for each workload. In order to facilitate a more profound comprehension of the impact of inlining, the default inlining size (5000) is to be scaled, with initial values ranging from 1.25 to 4 times the default size. The flag that is associated with inlining is as follows:

```
-XX:+Tier4InvocationThreshold={value}
```

The findings indicate that while the inter-run pattern remains unimproved, there is an enhancement in the mean performance, as illustrated in Figure 5.5.

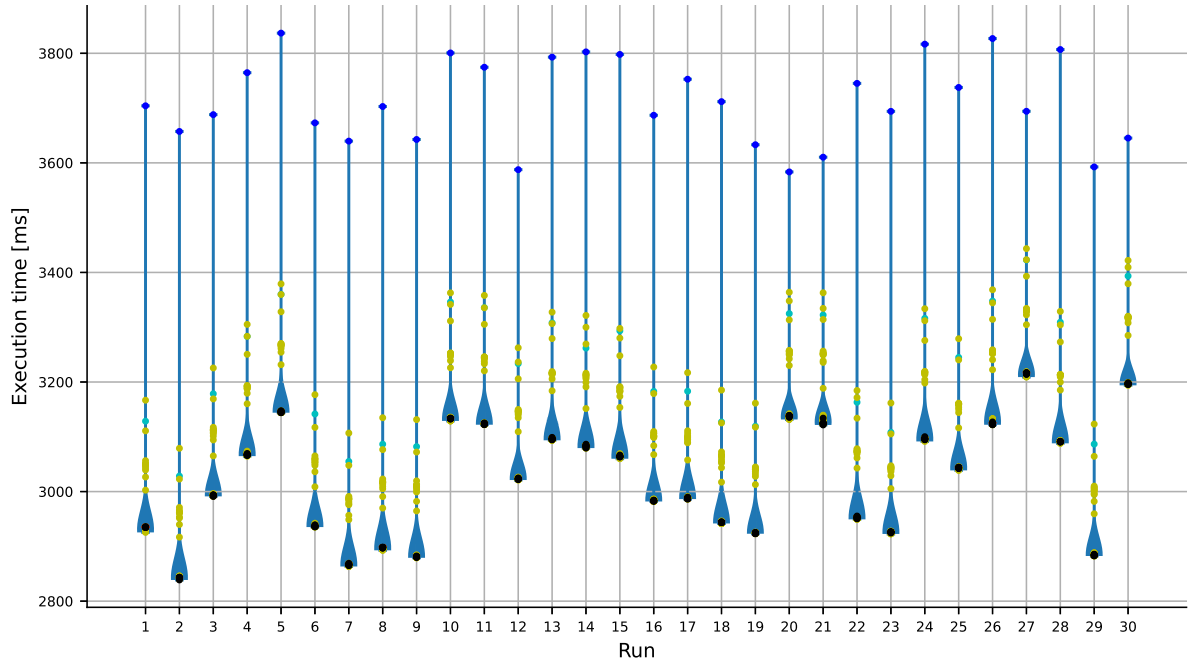


Figure 5.5: Violin plot of mnemonics on Java 23 with inlining size scaled 4 times

In contrast, the performance of runs in graal is more consistent, and inter-run variability is less pronounced.

5.4 C1 Disabling and Inlining Threshold in other Workloads

It is evident that the impact of mnemonics is substantial, and variability is effectively mitigated in Graal and enhanced in Java. Consequently, we conducted experiments to assess the consequences of eliminating the C1 layer and implementing inlining increment on other workloads that exhibit variability, as outlined in Chapter 4.
Initially, the efficacy of C1 removal is tested, followed by the implementation of increased inlining for the workload, with the objective of reducing variability without any mitigation measures.

5.4.1 C1 Disable Results

We summarize, in Table 5.1, the effect of removing the C1 layer on workloads that were previously affected by some form of variability, from which they benefit.
It is observed that of the remaining 23 workloads that exhibited variability, 18 were found to benefit from the removal of C1 (78.3%), with eight of these resolving variability completely (34.8%). The majority of these workloads are found to be affected by periodic variability.

	Java 23	Java 24	Graal 23	Graal 24
batik	Solved	Solved	Solved	Solved
biojava	Inter-run	Inter-run	Inter-run	Inter-run
chi-square	Solved	Solved	Solved	Solved
dec-tree	Solved	Solved	Solved	Solved
fj-kmeans	Solved	Solved	Solved	Solved
gauss-mix	Solved	Solved	Solved	Solved
graphchi	Solved	Solved	Solved	Solved
jython	Solved	Solved	Solved	Solved
luindex	Improved intra-run	Improved intra-run	Improved intra-run	Improved intra-run
movie-lens	Inter-run	Inter-run	Inter-run	Inter-run
naive-bayes	Solved	Solved		
page-rank	Inter-run	Inter-run	Inter-run	Inter-run
rx-scrabble	Solved	Solved	Solved	Solved
scala-doku	Intra-run	Intra-run	Inter-run	Inter-run
scala-kmeans	Solved	Solved	First iteration	First iteration
scala-stm-bench7	Improved performance	Improved performance	Improved performance	Improved performance
scrabble	First iteration	First iteration	First iteration	First iteration
spring	Inter-run	Inter-run	Inter-run	Inter-run

Table 5.1: Summary of effects of removing C1 layer only for workloads that benefit from it
We report only remeaning patterns if present or *Solved* otherwise

5.4.2 Inlining Increased Results

For workloads that benefit from the removal of the C1 layer, we gradually increased the inlining size threshold as we have done with mnemonics. For workloads that did not improve when the threshold was increased, or for which variability was still significant at 4x, we conducted further investigations into other flags related to inlining.

We discovered that improvement is not possible for concurrent workloads due to the causes of variability described in Chapter 2. This is evident from the par-mnemonics workload in Figure 5.6. Along with par-mnemonics, other workloads with similar results are akka-uct, h2 and h2o.

The results pertaining to the workloads that benefit from the increased threshold are collated and displayed in Table 5.2. It was discovered that increasing the threshold by a factor of 3 for the majority of them mitigates variability.

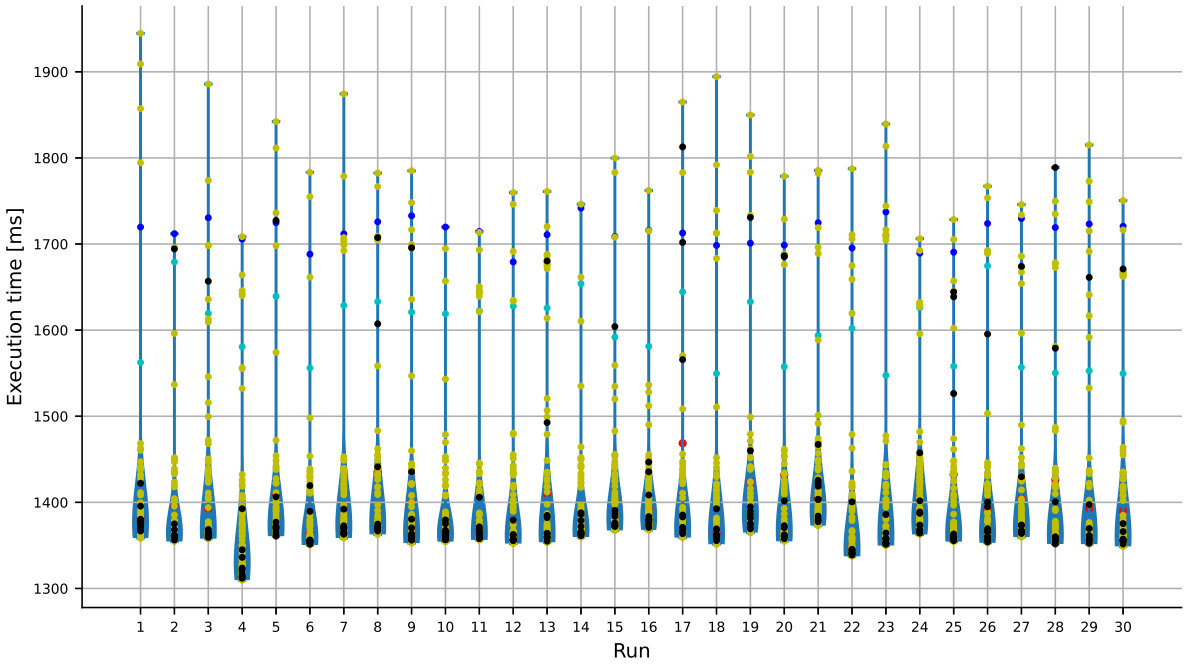


Figure 5.6: Par-mnemonics workload on GraalVM 23 - Example of concurrent workload that present no benefits from the removal of C1 layer

	Java 23	Java 24	Graal 23	Graal 24
biojava	New pattern *	New pattern *	New pattern *	New pattern *
luindex	Solved (3x)	Solved (3x)	Solved (3x)	Solved (3x)
movie-lens	Solved (2x)	Solved (2x)	Solved (2x)	Solved (2x)
page-rank			Inter-run	Inter-run
scala-doku	New pattern *	New pattern *	Solved (3x)	Solved (3x)
scala-stm-bench7	Solved (2x)	Solved (2x)	Solved (2x)	Solved (2x)
scrabble	First iteration	First iteration	First iteration	First iteration
spring	New pattern *	New pattern *	New pattern *	Solved (3x)

Table 5.2: Summary of effects of inlining increasing only for workloads that benefit from it
Factor of scaling on threshold if it mitigates variability or patterns still present with a factor of 4x
(*) Deterioration in performance from the nth iteration

5.5 Degradation from n-th iteration Pattern

As demonstrated in the preceding section, an analysis of the data revealed a novel pattern, which was designated *Degradation from n-th iteration*, in response to varying workloads. It has been demonstrated that, from a certain point onward, there is a substantial decline in performance, relative to the optimal iteration. This phenomenon may occur during iteration 2, as evidenced by the biojava example, or in a shorter range, as demonstrated by the scala-doku illustration.
Due to temporal limitations, our analysis is constrained to a comprehensive investigation of Scala-doku on all the flags associated with tier 4 compilation:

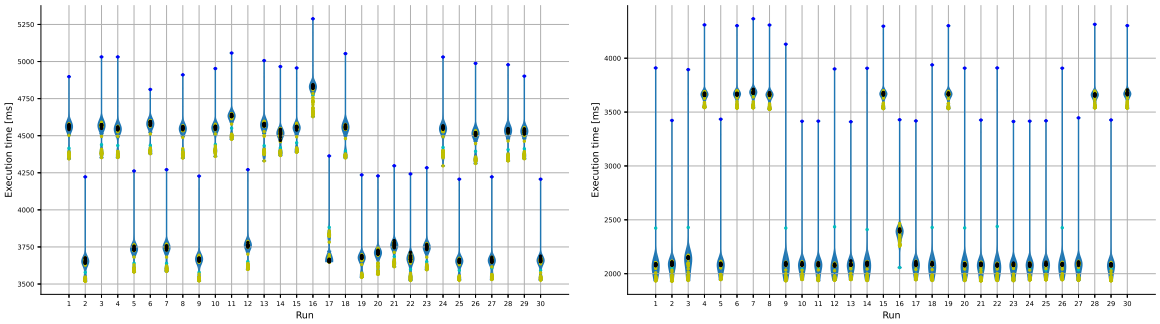
```
-XX:Tier4BackEdgeThreshold={40000}
-XX:Tier4CompileThreshold={15000}
-XX:Tier4InvocationThreshold={5000}
-XX:Tier4LoadFeedback={3}
-XX:Tier4MinInvocationThreshold={600}
```

5.5.1 BackEdge Threshold

- Default value: 40000

The BackEdge threshold is initiated when the number of iterations within a loop exceeds the predetermined threshold value. In the event of a loop being executed on multiple occasions, the JIT compiler is triggered to optimise the method. The elevation of this threshold has the effect of postponing the process of compilation, which may result in a reduction of JIT overhead for loops of a short duration. It has been demonstrated that lowering it results in more aggressive compilation.

Scala-doku exhibits better performance when the threshold size is increased, as demonstrated in Figure 5.7.



(a) 0.5x (b) 2x

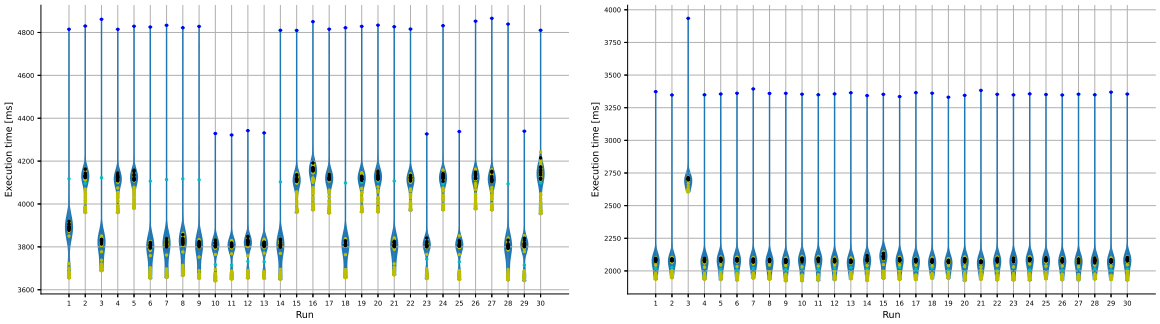
Figure 5.7: BackEdge Threshold effects in scala-doku on Java 24

5.5.2 Compile Threshold

- Default value: 15000

The compilation threshold deals is to be undertaken in conjunction with the compilation of hot methods. The process of lowering it results in the compilation of methods being undertaken at an earlier stage. This can enhance performance, albeit at the expense of increased overhead in terms of compilation. It is important to note that the JVM frequently makes dynamic adjustments to this value based on profiling.

It is evident that increasing the threshold in Scala-Doku results in a more stable workload, with only one run being an outlier. Conversely, in the opposite context, the performance is not stable, as it can be seen in Figure 5.8.



(a) 0.5x (b) 2x

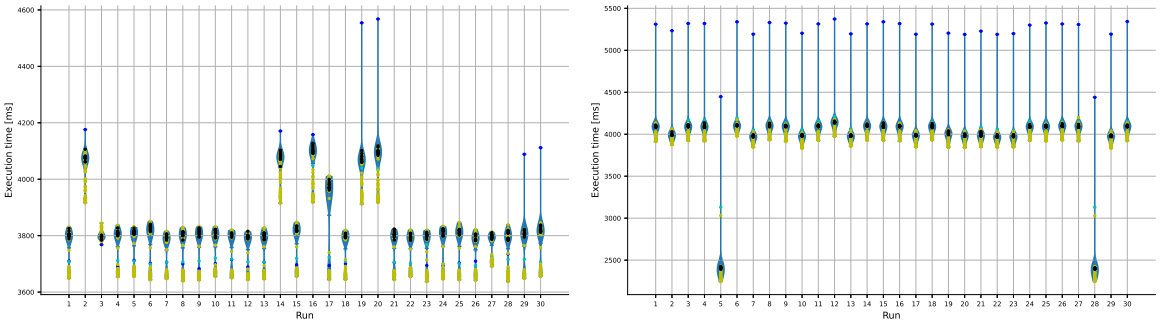
Figure 5.8: Compile Threshold effects in scala-doku on Java 24

5.5.3 Invocation Threshold

- Default value: 5000

The invocation threshold trigger compilation on tier 4 is initiated when a method is invoked a specified number of times in the lower tiers prior to being considered for tier 4 compilation. The function of this mechanism is to regulate the extent to which the JVM escalates a method through tiers.

As is the case with the compile threshold, scala-doku shows more stable behaviour when assigned a higher value (Figure 5.9).



(a) 0.5x (b) 2x

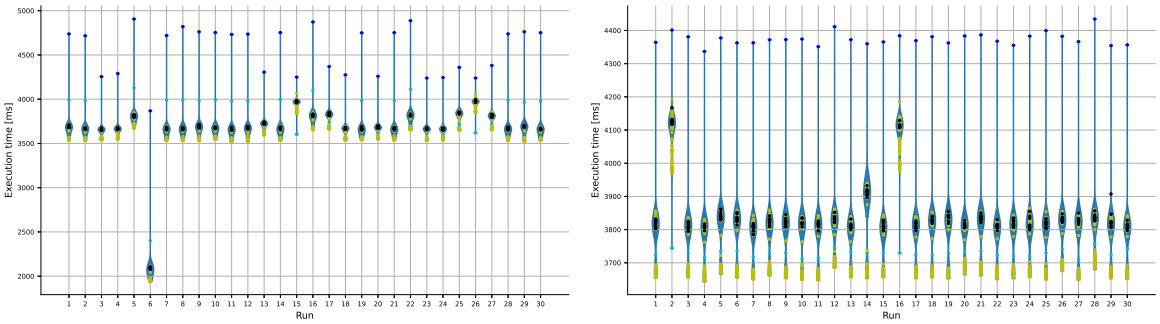
Figure 5.9: Invocation Threshold effects in scala-doku on Java 24

5.5.4 LoadFeedback

- Default value: 3

The LoadFeedback flag serves to determine whether the Tier-4 compiler should utilize runtime profiling information (e.g. type feedback, branch probabilities, etc.) collected by lower tiers. Enabling this feature enables C2 to produce machine code that has been optimized based on actual runtime behaviour. Disabling this flag results in a reduction in the quality of optimisation.

Rreduction in this value results in an enhancement of the mean performance of scala-doku (Figure 5.10). In this particular instance, a value of 1 has been assigned to the lower case and 5 to the higher case.



(a) `value = 1` (b) `value = 5`

Figure 5.10: Invocation Threshold effects in scala-doku on Java 24

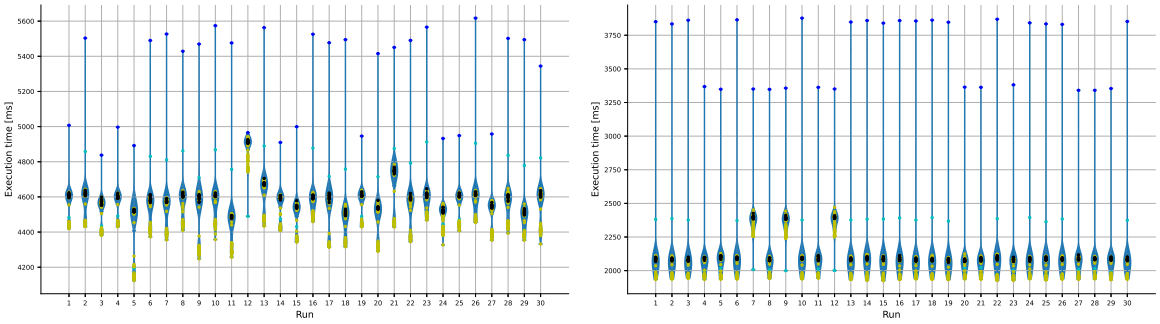
5.5.5 MinInvocation Threshold

- Default value: 600

This is the minimum number of invocations before tier 4 compilation can even be considered, regardless of other thresholds. This approach is designed to prevent the wastage of compilation resources that would otherwise be allocated to methods that are utilised

with infrequency.

This flag exerts a considerable influence on the performance of scala-doku, signifying that a substantial number of low-invoked methods are compiled, thereby resulting in the wastage of resources during the process. As demonstrated in Figure 5.11, the discrepancy is clearly evident.



(a) 0.5x (b) 2x

Figure 5.11: Invocation Threshold effects in scala-doku on Java 24

5.5.6 Result Combining Flag

Following a series of empirical trials on scala-doku, the optimal combination of flag to mitigate variability is ascertained:

```
    -XX:Tier4BackEdgeThreshold={200000}
    -XX:Tier4CompileThreshold={60000}
    -XX:Tier4InvocationThreshold={5000}
    -XX:Tier4LoadFeedback={1}
    -XX:Tier4MinInvocationThreshold={3000}
```

As demonstrated in Figure 5.12, the degradation pattern remains unresolved; however, the presence of other forms of variability has been eliminated.

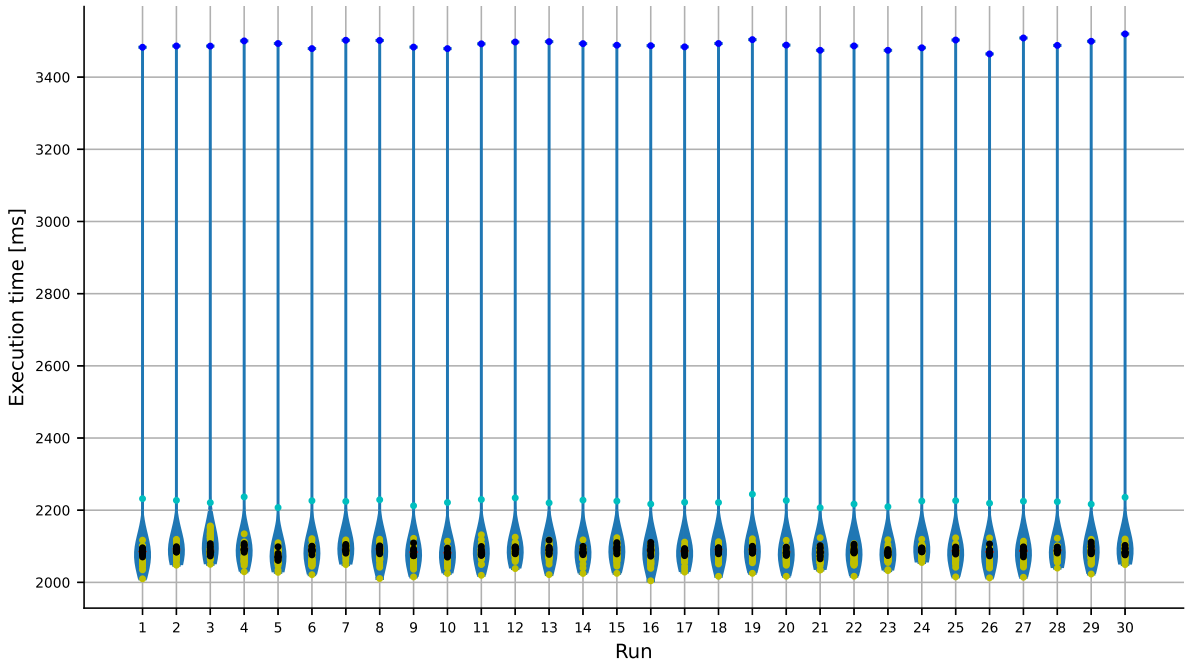


Figure 5.12: Violin plot of scala-doku on Java 23 - Only degradation pattern left

5.6 Summary

As illustrated in Table 2, the results of the experiment are presented in relation to the initial conditions outlined in Table 1. In the table we can see:

- ✓ - Variability mitigated removing C1
- ✓ - Variability mitigated removing C1 and increasing inlining
- X - Variability decreased removing C1 and increasing inlining

Of the 25 workloads affected by variability, we have mitigated the phenomenon in 10 cases only removing C1 layers, and 5 more increasing inlining too. Of the remeaning 10 workloads we decrease variability in 6 more workloads. In conclusion 60.0% of the workload can have variability mitigated using our approach and only 16.0% result uneffected by our approach.

	First Iteration				**Periodic Pattern**				**Intra-run**				**Inter-run**			
Workload	J23	J24	G23	G24	J23	J24	G23	G24	J23	J24	G23	G24	J23	J24	G23	G24
akka-uct	X	X	X	X					X	X	X	X				
batik		✓		✓	✓	✓	✓	✓								
biojava					✓*	✓*	✓*	✓*	✓	✓	✓	✓				
chi-square			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
dec-tree					✓	✓	✓	✓								
fj-kmeans			✓	✓												
gauss-mix					✓	✓	✓	✓								
graphchi	✓	✓	✓	✓										✓		
h2	X	X														
h2o	X	X	X	X					X	X	X	X				
jython					✓	✓	✓	✓								
luindex									✓	✓	✓	✓				
mnemonics	✓	✓	✓	✓									X	X	X	X
movie-lens					✓	✓	✓	✓								
naive-bayes					✓	✓										
page-rank			✓	✓											X	X
par-mnemonics									X	X	X	X	X	X	X	X
rx-scrabble	✓	✓		✓	✓	✓	✓	✓								
scala-doku	✓	✓	✓	✓	✓					✓			X*	X*	✓	✓
scala-kmeans	✓				✓	✓							✓	✓	✓	✓
scala-stm-bench7	X		X						X	X	X	X	X	X	X	X
scrabble	X	X	X	X												
spring													✓*	✓*	✓*	✓
sunflow		X	X	X					X	X	X	X	X	X	X	X
zxing	✓	✓	✓	✓												

Table 5.3: Alphabetically sorted workloads evaluated across variation types and platforms.
(*) introduced degradation pattern, but removed previous patterns

Chapter 6

Discussions

In this chapter, the initial discussion focuses on the limitations (Section 6.1), while the concluding remarks identify avenues for future work (Section 6.2).

6.1 Limitations

6.1.1 Hardware Architecture

The study on variability was carried out on a single server, despite the fact that it uses resources and an operating system that are well known in the scientific community and also widely used in industrial environments. In order to obtain a more general picture of the situation, it would be necessary to compare the results on different machines with different resources and operating systems.

6.1.2 Managed Runtime Environment

The experiments were performed exclusively on versions 23 and 24 of Java and Graal, two state-of-the-art JVMs. Due to temporal limitations, it was not possible to conduct the same experiments on the recently released versions 25 and 26 (released a few months ago and a few days ago respectively).
Furthermore, a significant number of companies continue to utilise obsolete versions of Java, including versions 8 and 17.
In the near future, it will be possible to repeat the experiment using both newer and older versions.

6.1.3 Limited Focus

In this study, the primary focus was on the impact of the compiler on workloads. It was determined that the primary design objective of the compiler was to achieve optimal performance, often at the cost of phenomena such as this.
However, there are still workloads that exhibit forms of variability that have not been resolved, or where we have reduced the phenomenon but without resolving it entirely, even discovering a new pattern.
The potential involvement of the garbage collector in this phenomenon remains a subject for future exploration, as our current research has not delved into this aspect.

6.2 Future Work

The present study has established a fundamental understanding the behaviour of the phoenomenon, but there are several avenues for further exploration that could enhance its practical applicability and robustness. These include the expansion of research in the field of garbage collection (GC), the automation of pattern recognition in code behaviour, and the optimisation of runtime flag configurations to mitigate variability in performance. Each of these directions addresses critical limitations identified in the current work and aligns with the evolving needs of managed runtime environments.

6.2.1 Expanding Research on Garbage Collection

A key limitation of the current study was its narrow focus on compilation, which overlooked the dynamic behaviours of managed runtimes, particularly garbage collection. It is recommended that future research focus on conducting a more in-depth investigation into the GC strategies employed by the G1, ZGC, Shenandoah and other contemporary collectors. This encompasses the analysis of how GC pause times, memory fragmentation, and throughput. Furthermore, the study should compare our research results across different GC implementations to determine its adaptability to diverse runtime environments. To illustrate this point, latency-sensitive applications may benefit from low-pause collectors such as ZGC, while throughput-oriented systems may prioritise G1. The incorporation of these analyses into future research endeavours has the potential to yield actionable insights for developers seeking to optimise their applications within managed environments.

6.2.2 Automating Pattern Recognition in Code Behavior

The present work is founded upon manual analysis of compilation phases, a limitation that hinders its capacity to adapt to runtime behaviours such as garbage collection, memory allocation, and thread scheduling. It is recommended that future research endeavours focus on the automation of pattern recognition in order to identify recurring code structures or runtime behaviours that have the potential to be optimised. To illustrate this point, machine learning models or rule-based systems could be trained to detect patterns such as memory leaks, inefficient object allocations, or suboptimal resource usage. These patterns could then be mapped to specific GC settings or code transformations to enhance performance. The utilisation of tools such as static analysers, dynamic monitoring systems, or hybrid approaches combining both methods has the potential to automate this process. The integration of such capabilities would result in the framework transitioning from a static optimization tool to a dynamic, self-adaptive system, capable of addressing runtime challenges in real time.

6.2.3 Automating Flag Configuration for Performance Stability

Variability in performance has been identified as a persistent challenge in managed runtimes, driven by factors such as workload changes, GC behaviour, and system resource constraints. It is recommended that future research endeavours concentrate on the automation of the configuration of runtime flags with a view to mitigating this variability. The development of a system capable of dynamically adjusting GC flags, memory allocation parameters, and other runtime settings based on observed workload patterns could be a fruitful avenue of research. This could involve the utilisation of reinforcement

learning or heuristic-based models to ascertain the optimal configuration for a given application. In scenarios involving high-throughput processing, the system may prioritise throughput-oriented garbage collection (GC) settings. Conversely, in scenarios with low-latency requirements, the system may transition to low-pause collectors. Real-time monitoring and adaptive adjustment mechanisms would ensure that the framework remains responsive to changing conditions. The automation of this process has the potential to reduce manual intervention and enhance the framework's reliability and scalability in production environments.

By addressing these areas, future work could significantly expand the framework's utility, ensuring it remains relevant in the face of evolving runtime technologies and industry demands. The integration of GC-aware optimisations, automated pattern recognition, and dynamic flag management would enable the framework to adapt to a broader range of applications, ultimately bridging the gap between theoretical design and practical deployment.

Chapter 7

Conlusions

In this work, we analyse and mitigate performance variability in managed runtime environments, thereby addressing a critical challenge in the optimisation of applications for dynamic and unpredictable workloads. To the best of our knowledge, our novel approach is the first to identify and reduce variability in managed runtimes. This approach involves systematically removing the C1 compiler layer and increasing the inlining threshold. These modifications significantly reduce runtime fluctuations, with some benchmarks achieving near-total variability mitigation. Our results demonstrate that compiler design choices, such as the inclusion of C1 and the configuration of inlining thresholds, play a pivotal role in determining performance stability in managed environments.

To validate our approach, we used advanced analysis tools such as ProfDiff and JitWatch to show how compiler decisions directly affect variability. ProfDiff allowed us to compare performance profiles across various configurations and pinpoint regions where variability was most pronounced. By isolating the impact of compiler layers and inlining decisions, we identified key contributors to instability, such as the overhead of C1's speculative optimisations and suboptimal inlining, which led to fragmented code execution. JitWatch, on the other hand, provided a visual and temporal analysis of Just-In-Time (JIT) compilation, enabling us to trace the translation of compiler choices into runtime behaviours. For instance, we found that removing C1 reduced the number of speculative optimisations, resulting in more predictable execution paths and reduced variance in garbage collection and memory management.

Our findings highlight the importance of aligning compiler strategies with application-specific requirements in order to achieve stable performance. The present study demonstrates the efficacy of compiler-level interventions in mitigating variability in managed runtimes. The analysis reveals that variability in managed runtimes can be mitigated in 60% of cases and reduced in 80% of instances. This work paves the way for further research into automating such optimisations by leveraging machine learning or heuristic-based systems to adapt compiler settings dynamically in real time. Future work will expand the scope to other benchmarks, explore the interplay between garbage collection and compiler choices, and integrate these insights into production-grade tools to enhance the reliability of managed runtime environments. Ultimately, our approach bridges the gap between theoretical compiler design and practical performance optimisation, offering a scalable framework for mitigating variability in a wide range of applications.

References

[1] Ali Abedi and Tim Brecht. «Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments». en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. L'Aquila Italy: ACM, Apr. 2017, pp. 287–292. DOI: 10.1145/3030207.3030229. URL: https://dl.acm.org/doi/10.1145/3030207.3030229 (visited on 07/20/2025) (cit. on p. 8).

[2] *AdoptOpenJDK/jitwatch*. original-date: 2013-09-30T14:18:17Z. Aug. 2025. URL: https://github.com/AdoptOpenJDK/jitwatch (visited on 08/09/2025) (cit. on pp. 17, 18).

[3] A.R. Alarneldeen and D.A. Wood. «Variability in architectural simulations of multi-threaded workloads». en. In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* Anaheim, CA, USA: IEEE Comput. Soc, pp. 7–18. DOI: 10.1109/hpca.2003.1183520. URL: http://ieeexplore.ieee.org/document/1183520/ (visited on 08/01/2025) (cit. on p. 9).

[4] Juan Pablo Sandoval Alcocer and Alexandre Bergel. «Tracking down performance variation against source code evolution». en. In: *Proceedings of the 11th Symposium on Dynamic Languages*. Pittsburgh PA USA: ACM, Oct. 2015, pp. 129–139. ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816718. URL: https://dl.acm.org/doi/10.1145/2816707.2816718 (visited on 01/27/2025) (cit. on pp. 1, 5).

[5] Sebastian Angel et al. «End-to-end Performance Isolation through Virtual Datacenters». en. In: () (cit. on p. 10).

[6] Edd Barrett et al. «Virtual machine warmup blows hot and cold». en. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3133876. URL: https://dl.acm.org/doi/10.1145/3133876 (visited on 01/29/2025) (cit. on p. 4).

[7] Abhinav Bhatele et al. «There goes the neighborhood: performance degradation due to nearby jobs». en. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503247. URL: https://dl.acm.org/doi/10.1145/2503210.2503247 (visited on 01/29/2025) (cit. on pp. 2, 6, 19).

[8] Stephen M. Blackburn et al. «Rethinking Java Performance Analysis». en. In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. Rotterdam Netherlands: ACM, Mar. 2025, pp. 940–954. DOI: 10.1145/3669940.3707217. URL: https://dl.acm.org/doi/10.1145/3669940.3707217 (visited on 07/13/2025) (cit. on pp. 2, 19, 20).

[9] Adnan Bouakaz, Isabelle Puaut, and Erven Rohou. «Predictable Binary Code Cache: A First Step towards Reconciling Predictability and Just-in-Time Compilation». en. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Chicago, IL, USA: IEEE, Apr. 2011, pp. 223–232. DOI: 10.1109/rtas.2011.29. URL: http://ieeexplore.ieee.org/document/5767154/ (visited on 08/11/2025) (cit. on p. 15).

[10] *DaCapo Benchmarks*. en. URL: http://dacapobench.org/ (visited on 01/27/2025) (cit. on pp. 2, 19).

[11] *Documentation | Renaissance Suite*. URL: https://renaissance.dev/docs (visited on 08/12/2025) (cit. on p. 20).

[12] Matthieu Dorier et al. «CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination». en. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, May 2014, pp. 155–164. ISBN: 978-1-4799-3800-1

978-1-4799-3799-8. DOI: 10.1109/IPDPS.2014.27. URL: https://ieeexplore.ieee.org/document/6877251 (visited on 01/29/2025) (cit. on pp. 2, 7, 19).

[13] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. «Speculation without regret: reducing deoptimization meta-data in the Graal compiler». en. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*. Cracow Poland: ACM, Sept. 2014, pp. 187–193. DOI: 10.1145/2647508.2647521. URL: https://dl.acm.org/doi/10.1145/2647508.2647521 (visited on 08/10/2025) (cit. on p. 14).

[14] Gilles Duboscq et al. «An intermediate representation for speculative optimizations in a dynamic compiler». en. In: *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*. Indianapolis Indiana USA: ACM, Oct. 2013, pp. 1–10. DOI: 10.1145/2542142.2542143. URL: https://dl.acm.org/doi/10.1145/2542142.2542143 (visited on 08/10/2025) (cit. on p. 13).

[15] Andy Georges, Dries Buytaert, and Lieven Eeckhout. «Statistically Rigorous Java Performance Evaluation». en. In: () (cit. on p. 4).

[16] Andy Georges, Lieven Eeckhout, and Dries Buytaert. «Java performance evaluation through rigorous replay compilation». en. In: () (cit. on p. 6).

[17] *graal/compiler/docs/Profdiff.md at master · oracle/graal*. en. URL: https://github.com/oracle/graal/blob/master/compiler/docs/Profdiff.md (visited on 07/19/2025) (cit. on pp. 2, 14–17, 49).

[18] *GraalVM*. en. URL: https://www.graalvm.org/ (visited on 01/29/2025) (cit. on pp. 2, 13–15).

[19] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. «Relative factors in performance analysis of Java virtual machines». en. In: *Proceedings of the 2nd international conference on Virtual execution environments*. Ottawa Ontario Canada: ACM, June 2006, pp. 111–121. DOI: 10.1145/1134760.1134776. URL: https://dl.acm.org/doi/10.1145/1134760.1134776 (visited on 07/20/2025) (cit. on p. 6).

[20] Wim Heirman et al. «Runtime variability in scientific parallel applications». en. In: () (cit. on p. 9).

[21] Torsten Hoefler and Roberto Belli. «Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results». en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas: ACM, Nov. 2015, pp. 1–12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807644. URL: https://dl.acm.org/doi/10.1145/2807591.2807644 (visited on 01/29/2025) (cit. on p. 9).

[22] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. «On the Performance Variability of Production Cloud Services». en. In: *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Newport Beach, CA, USA: IEEE, May 2011, pp. 104–113. ISBN: 978-1-4577-0129-0. DOI: 10.1109/CCGrid.2011.22. URL: http://ieeexplore.ieee.org/document/5948601/ (visited on 01/29/2025) (cit. on p. 8).

[23] *Java Software*. en. URL: https://www.oracle.com/uk/java/ (visited on 01/29/2025) (cit. on pp. 2, 11).

[24] *JRockit to HotSpot Migration Guide*. en-us. URL: https://docs.oracle.com/en/java/javase/11/jrockit-hotspot/compilation-optimization.html#GUID-92C9119B-C9F4-4BF6-91C8-527FE5C63657 (visited on 08/10/2025) (cit. on p. 12).

[25] Tomas Kalibera and Richard Jones. «Rigorous Benchmarking in Reasonable Time». en. In: () (cit. on pp. 1, 5).

[26] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. «Predicting unstable software benchmarks using static source code features». en. In: *Empirical Software Engineering* 26.6 (Nov. 2021). Publisher: Springer Science and Business Media LLC. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-021-09996-y. URL: https://link.springer.com/10.1007/s10664-021-09996-y (visited on 07/20/2025) (cit. on p. 6).

[27] Philipp Leitner and Jürgen Cito. «Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds». en. In: *ACM Transactions on Internet Technology* 16.3 (Aug. 2016), pp. 1–23. ISSN: 1533-5399, 1557-6051. DOI: 10.1145/2885497. URL: https://dl.acm.org/doi/10.1145/2885497 (visited on 01/29/2025) (cit. on p. 8).

[28] K.M. Lepak, H.W. Cain, and M.H. Lipasti. «Redeeming IPC as a performance metric for multithreaded programs». en. In: *Oceans 2002 Conference and Exhibition. Conference Proceedings (Cat. No.02CH37362)*. New Orleans, LA, USA: IEEE Comput. Soc, pp. 232–243. DOI: 10.1109/pact.2003.1238019. URL: http://ieeexplore.ieee.org/document/1238019/ (visited on 07/20/2025) (cit. on p. 9).

[29] Aleksander Maricq et al. «Taming Performance Variability». In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 409–425. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/maricq (cit. on p. 1).

[30] *mx/README-proftool.md at master · graalvm/mx*. URL: https://github.com/graalvm/mx/blob/master/README-proftool.md (visited on 08/09/2025) (cit. on pp. 15, 49).

[31] Todd Mytkowicz et al. «Producing Wrong Data Without Doing Anything Obviously Wrong!» en. In: () (cit. on p. 4).

[32] Andrej Pečimúth, David Leopoldseder, and Petr Tůma. «Diagnosing Compiler Performance by Comparing Optimization Decisions». en. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. Cascais Portugal: ACM, Oct. 2023, pp. 47–61. DOI: 10.1145/3617651.3622994. URL: https://dl.acm.org/doi/10.1145/3617651.3622994 (visited on 08/09/2025) (cit. on p. 15).

[33] *perf: Linux profiling with performance counters*. URL: https://perfwiki.github.io/main/ (visited on 08/09/2025) (cit. on p. 15).

[34] *Pharo - Welcome to Pharo!* URL: https://pharo.org/ (visited on 08/06/2025) (cit. on p. 5).

[35] Aleksandar Prokopec et al. «Making collection operations optimal with aggressive JIT compilation». en. In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. Vancouver BC Canada: ACM, Oct. 2017, pp. 29–40. DOI: 10.1145/3136000.3136002. URL: https://dl.acm.org/doi/10.1145/3136000.3136002 (visited on 08/10/2025) (cit. on p. 13).

[36] Aleksandar Prokopec et al. «Renaissance: benchmarking suite for parallel applications on the JVM». en. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Phoenix AZ USA: ACM, June 2019, pp. 31–47. DOI: 10.1145/3314221.3314637. URL: https://dl.acm.org/doi/10.1145/3314221.3314637 (visited on 07/13/2025) (cit. on pp. 2, 19).

[37] *Renaissance Suite, a benchmark suite for the JVM*. URL: https://renaissance.dev/ (visited on 01/27/2025) (cit. on pp. 2, 19).

[38] David Shue, Michael J Freedman, and Anees Shaikh. «Performance Isolation and Fairness for Multi-Tenant Cloud Storage». en. In: () (cit. on p. 10).

[39] D. Skinner and W. Kramer. «Understanding the causes of performance variability in HPC workloads». en. In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* Austin, TX, USA: IEEE, 2005, pp. 137–149. ISBN: 978-0-7803-9461-2. DOI: 10.1109/IISWC.2005.1526010. URL: http://ieeexplore.ieee.org/document/1526010/ (visited on 01/29/2025) (cit. on pp. 2, 6, 19).

[40] soulteary@gmail.com. *Basic Skills | Java Just-in-Time Compiler Principle Analysis and Practice*. zh-CN. Oct. 2020. URL: https://tech.meituan.com/2020/10/22/java-jit-practice-in-meituan.html (visited on 08/10/2025) (cit. on p. 13).

[41] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. «Partial Escape Analysis and Scalar Replacement for Java». en. In: *i f* () (cit. on p. 5).

[42] Xiongchao Tang et al. «vS ensor: leveraging fixed-workload snippets of programs for performance variance detection». en. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Vienna Austria: ACM, Feb. 2018, pp. 124–136. DOI: 10.1145/3178487.3178497. URL: https://dl.acm.org/doi/10.1145/3178487.3178497 (visited on 08/01/2025) (cit. on p. 7).

[43] «The Tail at Scale – Communications of the ACM». en. In: *Communications of the ACM* () (cit. on p. 8).

[44] *Tmpfs — The Linux Kernel documentation*. URL: https://docs.kernel.org/filesystems/tmpfs.html (visited on 08/12/2025) (cit. on p. 19).

[45] Ozan Tuncer et al. «Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning». en. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (Apr. 2019), pp. 883–896. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2018.2870403. URL: https://ieeexplore.ieee.org/document/8466019/ (visited on 01/29/2025) (cit. on p. 7).

[46] *Ubuntu 22.04.5 LTS (Jammy Jellyfish)*. URL: https://releases.ubuntu.com/22.04/ (visited on 08/13/2025) (cit. on p. 19).

[47] *VM Options Explorer - OpenJDK23 HotSpot*. URL: https://chriswhocodes.com/hotspot_options_openjdk23.html (visited on 08/09/2025) (cit. on p. 17).

[48] Wei Wang et al. «Testing Cloud Applications under Cloud-Uncertainty Performance Effects». en. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*

(ICST). Vasteras: IEEE, Apr. 2018, pp. 81–92. DOI: 10.1109/icst.2018.00018. URL: https://ieeexplore.ieee.org/document/8367038/ (visited on 08/01/2025) (cit. on p. 8).

[49] Nicholas J. Wright et al. «Measuring and Understanding Variation in Benchmark Performance». In: *2009 DoD High Performance Computing Modernization Program Users Group Conference*. June 2009, pp. 438–443. DOI: 10.1109/HPCMP-UGC.2009.72 (cit. on pp. 1, 7).

[50] Kejiang Ye and Yunjie Ji. «Performance Tuning and Modeling for Big Data Applications in Docker Containers». en. In: *2017 International Conference on Networking, Architecture, and Storage (NAS)*. Shenzhen, China: IEEE, Aug. 2017, pp. 1–6. DOI: 10.1109/nas.2017.8026871. URL: http://ieeexplore.ieee.org/document/8026871/ (visited on 08/01/2025) (cit. on p. 5).

[51] Yudi Zheng, Lubomír Bulej, and Walter Binder. «An Empirical Study on Deoptimization in the Graal Compiler». en. In: *LIPIcs, Volume 74, ECOOP 2017* 74 (2017). Ed. by Peter Müller. Artwork Size: 30 pages, 764369 bytes ISBN: 9783959770354 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 30:1–30:30. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECOOP.2017.30. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2017.30 (visited on 08/01/2025) (cit. on p. 14).