# Inaugural-Dissertation

zur Erlangung der Doktorwürde der

Naturwissenschaftlich-Mathematischen
Gesamtfakultät

der

Ruprecht-Karls-Universität
Heidelberg

vorgelegt von

Diego Elias Damasceno Costa (M.Sc.)

aus Paracatu (Brazil)

Tag der mündlichen Prüfung:

# Benchmark-driven Software Performance Optimization

*by*

Diego Elias Damasceno Costa

Supervisor: Prof. Dr. Artur Andrzejak

## Abstract

Software systems are an integral part of modern society. As we continue to harness software automation in all aspects of our daily lives, the runtime performance of these systems become increasingly important. When everything seems just a click away, performance issues that compromise the responsiveness of a system can lead to severe financial and reputation losses. Designing efficient code is critical for ensuring good and consistent performance of software systems. It requires performance expertize, and encompasses a set of difficult design decisions that need to be continuously revisited throughout the evolution of the software. Developers must test the performance of their core implementations, select efficient data structures and algorithms, explore parallel processing when it provides performance benefits, among many other aspects. Furthermore, the constant pressure for high-productivity laid on developers, aligned with the increasing complexity of modern software, makes designing efficient code an even more challenging endeavor.

This thesis presents a series of novel approaches based on empirical insights that attempt to support developers at the task of designing efficient code. We present contributions in three aspects. First, we investigate the prevalence and impact of bad practices on performance benchmarks of Java-based open-source software. We show that not only these bad practices occur frequently, they often distort the benchmark results substantially. Moreover, we devise a tool that can be used by developers to identify bad practices during benchmark creation automatically.

Second, we design an application-level framework that identifies suboptimal implementations and selects optimized variants at runtime, effectively optimizing the execution time and memory usage of the target application. Furthermore, we investigate the performance of data structures from several popular collection libraries. Our findings show that alternative variants can be selected for substantial performance improvement under specific usage scenarios.

Third, we investigate the parallelization of object processing via Java streams. We propose a decision-support framework that leverages machine-learning models trained through a series of benchmarks, to identify and report stream pipelines that should be processed in parallel for better performance.

## Zusamenfassung

Softwaresysteme sind integraler Bestandteil der modernen Gesellschaft. Die immer weitere Durchdringung aller Bereiche des täglichen Lebens durch die Automation durch Software macht das Laufzeitverhalten dieser Systeme immer wichtiger. Wenn alles nur einen Klick entfernt scheint, können Leistungsprobleme, die die Reaktionsfähigkeit eines Systems beeinträchtigen, schwere finanzielle Schäden verursachen sowie rufschädigend wirken. Effizienten Code zu entwerfen ist kritisch wichtig um gute und gleichmäßige Leistung von Softwaresystemen sicherzustellen. Das erfordert Expertise in Softwareleistung, und beinhaltet schwierige Designentscheidungen, die immer wieder überdacht werden müssen, während die Software ihrer Evolution unterliegt. Entwickler müssen die Leistungsfähigkeit ihrer zentralen Implementationen testen, effiziente Datenstrukturen und Algorithmen wählen, parallele Verarbeitung in Erwägung ziehen, falls dies Leistungsgewinne verspricht; neben vielen anderen Gesichtspunkten. Weiterhin macht der konstante Erwartungsdruck hoher Produktivität, der auf Entwicklern lastet, sowie die anwachsende Komplexität moderner Software den Entwurf von effizientem Code zu einer umso größeren Herausforderung.

Diese Arbeit stelle eine Reihe neuartiger, auf empirischen Einsichten beruhender Ansätze vor, die Entwickler beim Entwurf effizienten Codes unterstützen sollen. Wir präsentieren Beiträge in drei Bereichen. Erstens untersuchen wir die Verbreitung von schlechten Programmierpraktiken und ihren Auswirkungen auf Leistungsbenchmarks von Java-basierter Open-Source-Software. Wir zeigen nicht nur dass diese schlechte Praktiken häufig vorkommen, sondern sogar oft die Benchmarkergebnisse erheblich beeinträchtigen. Außerdem entwickeln wir ein Tool, das von Entwicklern genutzt werden kann, um schlechte Praktiken während der Benchmarkerstellung automatisch zu erkennen.

Zweitens entwerfen wir ein Framework auf Anwendungsebene, das suboptimale Implementationen erkennt und optimierte Varianten zur Laufzeit wählt, und so effektiv die Ausführungszeit und den Speicherverbrauch der Zielanwendung optimiert. Weiterhin untersuchen wir die Leistung von Datenstrukturen aus verschiedenen populären Programmierbibliotheken für Collections. Unsere Ergebnisse zei-

gen, dass alternative Varianten so ausgewählt werden können, dass bedeutende Leistungsgewinne unter bestimmten Gegebenheiten erzielt werden können.

Drittens untersuchen wir die Parallelisierung von Objekt-Verarbeitung mittels Java-Streams. Wir stellen ein Framework zur Unterstützung von Entscheidungen vor, das Modelle des maschinellen Lernens nutzt, die mit einer Reihe von Benchmarks trainiert werden, und so Stream-Pipelines erkennt und meldet, die durch Parallelisierung mit höherer Leistung verarbeitet werden können.

# Acknowledgments

First and foremost, I want to express my deepest gratitude to my advisor Prof. Dr. Artur Andrzejak, for giving me the opportunity of a lifetime: pursuing a Ph.D. in a topic very dear to me, in an excellent research environment at Heidelberg University. He gave me the freedom to express my research ideas, and the support to overcome the obstacles a Ph.D. entails. His advice on conducting scientific research and communication will be forever ingrained in my professional career, as I plan to pass on the "Pyramid principle" legacy. I also want to thank Prof. Dr. Holger Fröning, Prof. Dr. Michael Gertz, and Prof. Dr. Björn Ommer, for serving in my doctoral committee.

I want to thank our external collaborators Prof. Dr. David Lo, Prof. Dr. Cor-Paul Bezemer, and Prof. Dr. Philipp Leitner, for the great support and helpful feedback I received in this past few years. I have learned a great deal about Software Engineering with our research collaborations.

I am grateful to my dear friends of the Parallel and Distributed Systems group for their great support and friendship: Dr. Zhen Dong, Mohammadreza Ghanavati, Lutz Büch, Kai Chen, and Thomas Bach. Thank you for creating a productive and engaging group environment. In particular, I would like to thank Mohammadreza, Lutz, and Thomas for providing invaluable feedback to this thesis writing.

My gratitude also goes to Catherine Proux-Wieland and Anke Sopka for their support in the (seemingly) endless administrative work. Thank you for taming the bureaucracy beast and for keeping everything working as smooth as possible.

I gratefully acknowledge the financial support for traveling and the stipend I received for the first three years of my doctoral research by the *Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences* (HGS MathComp), represented by Dr. Michael J. Winckler.

I want to thank the people that help made Heidelberg my home. My dear friends Andreas Spitz, Asha Roberts, Fereydoon Taheri, and Victoria Ponce, thanks for the

best (and sometimes heated!) discussions a friend could ever have. To our Happy-Hour group - Artsiom Sanakoyeu, Danilo Lustosa, Julia Jäger, Mangayarkarasi Rajakrishnan, Michael Gabel, Priyata Kalra, thanks for bringing the fun back to Germany!

To the large crew of past and present Botanik Philosophers, thanks for all the pseudo-scientific/pop-culture/political discussions that energized my afternoons. The coffee has always been just an excuse for the conversation. Please, keep our coffee tradition alive.

Furthermore, I want to thank my family: my parents, Elias and Silvana, for their unconditional support; my siblings, Luan, Ane, and Bruno for being the most joyful part of my life. Finishing a Ph.D. is nothing compared to the struggle of not being there with you.

And finally, to the one that has made this all possible, my wife, and lifetime player-two partner: Priscila. Thank you for encouraging me to pursue an academic career, and for joining me in this adventure with the utmost care, companionship, and love. This thesis and my life are dedicated to you.

# Contents

# 1 Introduction

We live in a technological era. Software systems are the main driver of our technological outburst, embedded in all areas of our modern society: automating our commerce and financial systems, coordinating our transportation, connecting millions of people through social networks, and many more. As we continue to harness the benefits of software automation, the quality attributes of these systems, such as the runtime performance, are evermore crucial to our lives.

Performance issues that ripple through production systems affect the reliability and the reputation of the service and its provider. Consequences of performance problems vary from mild frustration of its user-base to massive monetary and reputation costs. A one-second slowdown in the checkout processes would cost Amazon an estimated $1.6 billion per year [75]. A report from AppDynamic [54] has estimated that among the Fortune 500 companies, more than $46 million is spent annually in labor costs alone, due to performance problems.

Ensuring good and consistent performance in a software system is a difficult multi-layered problem: it starts with the planning of system architecture and underlying technologies; it encompasses efficient code design, with efficient data structures and algorithms; ending up with the appropriate provision of system resources.

The process of writing efficient code alone encompasses a broad range of practices in software development. From measuring and analyzing the performance of implementations, to selecting efficient data structure and algorithms that fits to the problem at hand, these practices focus on preventing performance issues from appearing in the production environment. While a great deal of performance optimization is automatically performed by compilers [3, 170], a number inefficiencies in the code can only addressed by developers and require extensive manual analysis.

This thesis presents a series of empirical insights and approaches that aim at helping developers in developing efficient applications. In particular, we study three important performance-related problems: 1) the creation of sound performance tests through benchmarking; 2) the selection of efficient data structures; and 3) the efficient parallelization of element processing via the Java Stream API.

## 1.1 Benchmark-driven Software Performance Optimization

A recurring theme in this thesis is the use of performance benchmarks to measure, model, and predict software performance. We investigate practices for sound Java performance benchmarking, and design benchmark suites to model the performance of data structures and stream processing via the Java Stream API. These models are used in our proposed tools for supporting developers by providing performance feedbacks and automated optimizations.

In the following, we give a short introduction on the three problems, addressed in the context of Java programming, that shall be discussed throughout this thesis.

### 1.1.1 Sound Java Performance Benchmarking

To avoid performance issues in production, developers must thoroughly test the performance of their systems. Similarly to functional tests, performance tests should be conducted at different granularities, from system performance (e.g., load and stress tests) to the accurate measurement of a method call via (micro/nano/mili)-benchmarking. Developers use benchmarks for precise performance evaluation of an isolated segment of code at the method, loop, or even statement level. Benchmarking are typically used to ensure the performance of critical low-level code components or to compare different implementation alternatives .

Despite the existence of robust benchmark frameworks in Java [136], developers often struggle with writing expressive benchmarks, which accurately represent the performance of such methods or statements. The Java Virtual Machine introduces a series of intricacies and pitfalls [83] that, if not addressed by developers, may

obliterate the setup of benchmarks, distorting the benchmark results and leading developers to the wrong conclusions.

We address this problem in Chapter 3, by presenting a large-scale empirical study on the prevalence of bad practices in open-source projects, and experimentally evaluate their impact in the benchmark results. Moreover, we develop a static analysis tool called SpotJMHBugs, that identifies the bad practices in the source-code of a benchmark, which can be used by developers to avoid said bad practices during benchmark creation.

### 1.1.2 Efficient Collection Selection

Selecting data structures is crucial for developing efficient applications. Java has a rich ecosystem of distinct collection libraries and implementations, providing developers a large pool of data structures with different performance trade-offs. While developers can make good use from the variety of collection variants, developers only rarely tune their data structures [98], and often select variants that are inefficient for their particular workload, leading to significant performance issues in their applications [67, 158, 178, 179].

We address this problem in two chapters of this thesis. In Chapter 4, we conduct an experimental study to explore the performance of alternative collection libraries. Then, in Chapter 5, we propose an application-level framework called CollectionSwitch, that selects collection implementations at runtime to optimize the time and memory performance of applications.

### 1.1.3 Efficient Parallelization of Element Processing via Streams

The Java Stream library [52] provides a concise API for processing streams of elements in a similar fashion to the popularized map-reduce frameworks. The library allows developers to process objects in parallel with a simple change of operators in the pipeline. However, developers need to account for numerous factors to benefit from this parallel processing [110], which are difficult and time-consuming to eval-

uate through manual analysis. Furthermore, if done incorrectly, the parallelization of streams may lead to severe performance issues and incorrect behavior.

We address this problem in Chapter 6, with the introduction of the decision support framework called StreamAssist. We leverage machine-learning models learned from generated stream benchmarks, to report to developers the stream pipelines that are likely to benefit from parallelism, and the ones that are better processed in sequence.

## 1.2 Contributions

In this work we present the following contribution:

1. **Understanding the prevalence and impact of bad practices in Java benchmarks** - We perform the first large-scale empirical study of bad practices in the creation of Java benchmarks under the Java Microbenchmark Harness (JMH) framework. We devise an automated tool that statically identifies five bad practices in a benchmark source code. Using this tool, we empirically investigate the occurrence of bad practices in Java-based open-source projects and experimentally evaluate its impact on benchmark results. Our findings show that bad practices are prevalent in Java-based open-source systems, and they often severely impact the quality of the benchmark, leading to results that substantially deviates from the correct measurements. This work is shown in Chapter 3 and was accepted to appear in [46].

2. **Characterization of collections performance profile from various collection libraries** - We conduct an empirical study on the usage and performance of Java collections. We focus on comparing the performance of non-standard collection implementations against the most commonly used standard variants. Our results show that alternative variants outperform standard collections on several usage scenarios. We devise a guideline that can be used by practitioners to select variants for better execution time and memory consumption of their applications. This is detailed in Chapter 4 and was published in [43].

3. **An application-level framework that selects collection implementation at runtime** - We describe the development of the CollectionSwitch. Given an optimization criteria from developers, our framework identifies allocation sites that instantiate suboptimal implementations and selects optimized variants for future instantiations. We evaluate CollectionSwitch on a series of synthetic benchmarks and real-world applications and observe better runtime and memory usage on several cases, without incurring in a noticeable monitoring overhead. This work is described in Chapter 5 and was published in [42].

4. **A decision-support framework for efficient parallelization of streams** - We propose a support-decision framework for effective stream parallelization. Our framework leverages machine-learning models trained through a series of stream-tailored benchmarks, to identify and report pipelines that can be executed in parallel for better performance. In our evaluation, we show that our trained models can be used to identify the optimal execution mode of stream pipelines of real-applications. We present this work in Chapter 6.

This thesis is partially based on several publications by the author. A list of publications is presented in the reverse chronological order as follows:

- D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. "What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks". *Transactions on Software Engineering*, 2019 (to appear)

- D. Costa and A. Andrzejak. "CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. ACM, Vienna, Austria, 2018, pp. 16–26. DOI: 10.1145/3168825.

- D. Costa, A. Andrzejak, J. Seboek, and D. Lo. "Empirical Study of Usage and Performance of Java Collections". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. ACM, L'Aquila, Italy, 2017, pp. 389–400. DOI: 10.1145/3030207.3030221.

During his Ph.D studies, the author also contributed to the following publications that do not directly relate to the core topic of this thesis:

- M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. "Memory and Resource Leak Defects and their Repairs in Java Projects". *Empirical Software Engineering*, 2019 (to appear).
- M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. "Memory and Resource Leak Defects in Java Projects: An Empirical Study". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. ACM, Gothenburg, Sweden, 2018, pp. 410–411. DOI: 10.1145/3183440.3195032.
- A. Spitz, D. Costa, K. Chen, J. Greulich, J. Geiß, S. Wiesberg, and M. Gertz. "Heterogeneous subgraph features for information networks". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*. 2018, 7:1–7:9. DOI: 10.1145/3210259.3210266.
- Z. Dong, A. Andrzejak, D. Lo, and D. Costa. "ORPLocator: Identifying Read Points of Configuration Options via Static Analysis". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016, pp. 185–195. DOI: 10.1109/ISSRE.2016.37

## 1.3 Thesis Outline

The outline of this thesis is as follows. Chapter 2 presents the context of this work and the basic concepts that shall be used throughout the thesis. In Chapter 3, we present our empirical investigation of bad practices on Java benchmarks. Chapter 4 focuses on analyzing the usage and performance of collection data structures, comparing alternative implementations against the commonly used standard Java collections. In Chapter 5 we present our framework called CollectionSwitch, that selects collection implementations at runtime for better performance. Chapter 6 presents our decision support framework for effective stream parallelization called StreamAssist.

# 2 Context of the Work and Background

In this chapter, we establish the context of this work and introduce the elementary concepts and the terminology that will be used throughout the thesis. As our work addresses three different problems within the broad area of writing efficient code, our related work is partitioned into three different areas. We refrain from presenting the related work in this chapter. Instead, we describe the related literature after introducing the addressed problems and motivating our approaches, within the following chapters.

The structure of this chapter is as follows: We start by positioning this work in the context of academic research fields in Section 2.1, by briefly describing the research areas this touches: empirical software engineering and software performance engineering. Afterward, in Section 2.3, we describe the most relevant performance-related components of the Java Virtual Machine that will be revisited in later chapters. In Section 2.4 we acquaint the reader with the challenges and methods for Java performance measurement. Then, in Section 2.4 we describe the basics of Java collections and how selecting variants are important to optimize Java applications. Lastly, we present in Section 2.5 an introduction to the Java Stream API, together with a glimpse on the factors practitioners need to consider when parallelizing stream pipelines.

## 2.1 Context of the Work

In general terms, this thesis focuses on aiding developers at writing efficient code by providing empirical assessments and methods for performance measurement and optimization. Therefore, the work is situated on two research fields: the em-

pirical software engineering and performance software engineering. In the following, we introduce these two research fields.

### 2.1.1 Empirical Software Engineering

Empirical Software Engineering (ESE) is an area of research that "emphasizes the use of empirical methods in the field of software engineering" [119]. The area focuses on aiding software engineering processes through collecting, analyzing, assessing, and interpreting the data collected from software repositories. The increasing availability of public and open-source repositories, championed by the popularity of GitHub [68], has made Empirical Software Engineering not only effective but essential for producing high quality, low cost, and maintainable software.

The method of mining software repositories is integral to the methodology used in all studies in this thesis. In Chapter 3, the repository mining takes the centerpiece of the chapter, as we investigate the occurrence and impact of bad practices on benchmark creation in Java. In the remaining chapters, the data extracted from open-source repositories are used to identify patterns in software development that will help us devise benchmarks, methods, and models that can be effectively used to optimize real applications.

### 2.1.2 Software Performance Engineering

The Software (and Systems) Performance Engineering (SPE) is a research area that encompasses methods, practices, and disciplines that can be used to ensure the meet of non-functional software requirements, such as latency, throughput, and memory usage [21]. It is a subfield of software engineering and includes the study of topics such as performance measurement, performance modeling, benchmark design, and runtime memory management.

In its core, this thesis is focused on the SPE research area. While we use the empirical data to get insights on how developers perform certain performance-related tasks, we use these insights to provide tools that help practitioners create better benchmarks (Chapter 3), we devise benchmarks to model the performance of commonly used libraries in Chapter 4, and we create tools that optimize the

performance of running applications (Chapter 5) and report possible performance optimizations (Chapter 6).

## 2.2 The Java Language Environment

This thesis is focused on analyzing and optimizing the performance of Java programs. The conceptual contributions presented in later chapters may generalize to other programming languages in performance-related or similar tasks. On the technical side, however, both tasks of performance analysis and optimization require specialization to be effective. Our work exploits the particularities of the Java language environment that need first to be established to the reader. In this section, we discuss the basic concepts of the components that have direct influence on Java's performance, namely the JIT Compiler, the Garbage Collector and the Heap, and the Java Threads component. Such components will be referenced in later chapters.

The Java language environment (or Java for short) is an object-oriented programming language and support runtime environment created by James Gosling [74] and released in 1996 by Sun Microsystems [96]. The language is class-based, as each object state and behavior is defined in classes, and concurrent, giving native support for concurrent programming and allowing developers to write applications that exploit multi-core parallelism.

Oracle acquired Sun Microsystems in 2010, becoming the primary developer and maintainer of the Java language environment [139] to this date. Throughout the years, Java has become one of the most popular programming languages in the world, figuring at the top of most searched programming languages [34] and consistently being the second most popular general-purpose language in StackOverflow [145]. An estimated 3 billion devices run Java today, including Android devices, embedded systems, and Java servers.

Since its genesis, Java has had high performance as one of its design goals [140]. As stated in the introductory paper from Gosling and McGilton [74]: "To live in the world of electronic commerce and distribution, Java must enable the development of secure, **high performance**, and highly robust applications on multiple platforms in heterogeneous, distributed networks". This high-performance goal

has shaped the conception and the evolution of the Java language and the runtime system throughout the years, and made Java a solid choice for developing robust and high-performance applications [94].

### 2.2.1 The Java Virtual Machine

The Java runtime environment was introduced with the slogan "Write Once, Run Anywhere" (WORA) [124], illustrating the benefits of a cross-platform development environment. That means that a compiled Java program can be executed on different platforms (that support Java) without needing any re-compilation. This portability was realized by the introduction of an intermediate program representation (instruction set) called *bytecode*. Java programs are compiled to bytecode as opposed to native machine-code, and the bytecode is then interpreted, compiled and executed by a virtual machine.

The Java virtual machine (JVM) specification describes an abstract computer that reads and executes programs compiled into bytecode [115]. JVM specifies what a virtual machine implementation needs to provide to run bytecode programs, including the execution engine and the dynamic memory management. Currently, the HotSpot [138] is the de-facto implementation of the JVM specification maintained by Oracle. We conduct all our analysis and experiments in this thesis with the HotSpot JVM, and for the sake of simplicity, we here onwards refer to HotSpot simply as JVM.

Figure 2.1 illustrates the JVM and its most important components. We focus on covering the basics of the components highlighted in blue, as they have a direct impact on the performance of Java applications.

JIT Compiler The *Just-In-Time Compiler* (JIT) is part of the JVM execution engine and is responsible for compiling bytecode into native code for better performance [3]. It differs from typical compilers as the compilation takes place during the program's execution. Before the introduction of the JIT compiler, the same method bytecode had to be interpreted into native machine code every time this method was called, incurring in a repeated penalty of a relatively lengthy translation process [49]. The inclusion of JIT Compiler changed this picture, by allowing

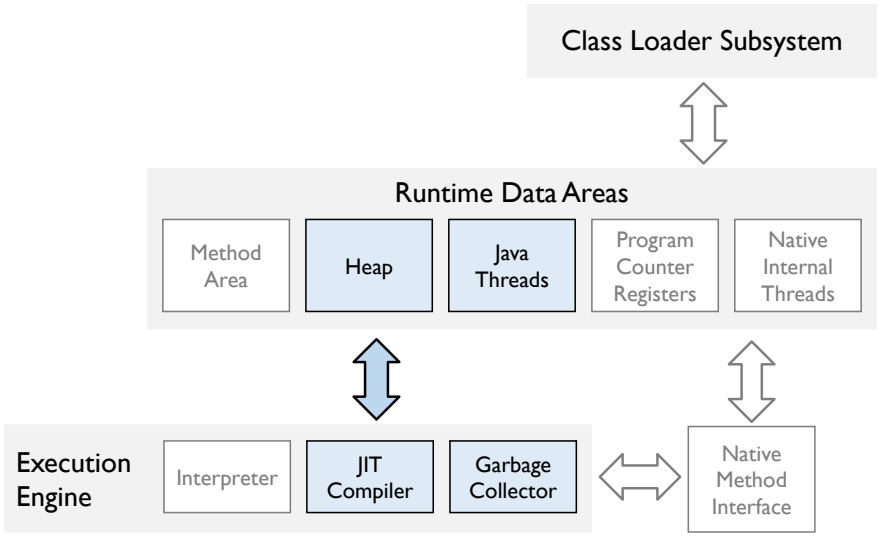



Figure 2.1: Illustration of the JVM architecture. JVM is composed of three subsystems, the Class Loader Subsystem, Runtime Data Areas and the Execution Engine. We highlight in **blue** the components referenced throughout this thesis, as they have a direct influence in the performance of Java applications.

the JVM to compile methods into native machine code and reusing this compiled version on future method calls [12].

While the compiled code is substantially faster than the interpreted counterpart, compiling code at runtime is a costly process [3]. The JVM needs to focus this compilation on methods in which the benefit of the performance gain surpasses the cost of analyzing and compiling it to native code. JVM addresses this problem through a process called *profile-guided optimization* [11], essentially keeping track of the most called methods (typically through sampling) and restricting the analysis and compilation to such cases. Furthermore, JIT Compiler also applies a series of optimizations while compiling the code, such as inlining methods, eliminating dead-code and unrolling loops [24].

Garbage Collector and the Heap The *Garbage Collector* (GC) is a component of the JVM responsible for the automatic management of dynamic memory [87]. In Java, objects created during the program's execution are typically allocated in the *heap*[1], a globally shared and dynamically sized memory space. The GC is re-

sponsible for the memory deallocation, identifying objects no longer in-use and removing them from the heap.

On the one hand, the GC takes away the burden of managing memory from developers, allowing them to focus on other aspects of software development. On the other hand, the GC needs to actively search for unused objects at runtime to free up memory for future allocations. This process consumes resources and might lead to severe performance issues if GC executes for extended periods [26]. In Java, poorly managed memory also leads to performance problems on execution time, as the GC needs to intervene and frequently remove unused memory.

Java Threads Java provides built-in support for *threads*, the concurrent and lightweight paths of program execution, and thread synchronization [91]. Developers can make use of this concurrent support to exploit parallelism and asynchronism in their applications. Threads can be created by either extending the `Thread` class or by defining a `Runnable class` in a very simple manner. On top of the language and environment support, Java ships with a series of libraries tailored for the concurrent environment. In particular, the Fork/Join framework [59] is a framework designed to facilitate parallel processing of a task by dividing it into sub-tasks recursively and processing it with a pool of threads. This framework paved the environment for libraries with friendly-support for parallelization, such as the Java Stream [52], discussed in details in Section 2.5.

## 2.3 Measuring Java Performance

Software performance has never been less transparent [83]. Developers can no longer rely solely on theoretical models to predict the performance of their data structures and algorithms. Java programs are compiled through multi-staged compilers [3], continuously optimized at runtime through analytical methods and heuristics [157, 170], and orthogonal factors such as memory layout may have a more substantial impact on performance than a worse asymptotic algorithm [98].

[1] If an object is guaranteed to have a very restricted local scope, JIT might be able to allocate parts or the entire object in the stack memory as a form of optimizing the allocation and object's access time. This optimization is called Scalar Replacement.

Performance experiments offer an empirical take on performance analysis. It is reasonable to expect that by measuring the performance through experiments we assess the ground truth of a system's performance. Unfortunately, the same factors that make performance less transparent for developers also affect performance tests and may guide developers to the wrong conclusions.

A virtually-managed environment such as the JVM introduces new sources of non-determinism into the performance experiment, which are of less concern on compiled languages such as C and C++. Such new factors include heap size, the memory management strategy, the class loader, the JIT Compiler, thread scheduler, and dynamic optimizations. Consequently, much evidence shows that it is hard to write performance experiments that produce reliable results [16, 66, 78, 128].

A large bulk of optimizations are expected to happen at the start time of a Java application [62, 66]. First, the JVM loads classes on demand. The core classes of a program are expected to be load in the very beginning and are seldom unloaded. Second, several JIT optimizations will also happen soon after the program starts. There is an inherent trade-off with the JIT Compiler between the initial performance loss caused by the compilation of bytecode and the long-term gain of the compiled version. Moreover, most JIT Compilers have a tiered optimization strategy, compiling methods to different levels of optimization, gradually moving them from low to high levels of optimizations at each tier.

Consequently, Java applications have an unstable initial phase, as JVM has to warm-up its optimization heuristics and mechanisms, followed by a more stable (and typically faster) overall performance. The performance at this initial phase is called *start-up performance* and the performance at the second and stable phase is named *steady-state performance* [62]. In this work, we are interested in measuring and optimizing the steady-state performance of applications and will focus on this aspect of Java performance in the following paragraphs. For more information on how to measure start-up performance, we refer to the work of Georges et al. [62].

### 2.3.1 Measuring Steady-state Performance

By measuring the steady-state performance of an application, we focus on the long-running performance, past the initial set of optimizations. This is the in-

dicated method for measuring the performance of applications that have a total running time far longer than the start-up time [62].

There are two challenges in measuring the steady-state performance of an application. First, how to determine when the program reaches the steady-state performance? Each application and input is expected to stabilize at a different point in time. Second, how to ensure that the same steady-state is reached across multiple VM invocations? Non-deterministic factors may play a role in optimizing applications, for instance, optimization heuristics that rely on sampling might select different methods even when running the same program with identical inputs.

In the influential "Statistically Rigorous Java Performance Evaluation", Georges et al. [62] propose the following methodology for evaluating the steady-state performance of Java benchmarks.

S1 Consider $p$ VM invocations, each VM invocation running at most $q$ benchmark iterations. Suppose that we want to retain $k$ measurements per invocation.

S2 For each VM invocation $i$, determine the iteration $s_i$ where the target application reaches the steady-state performance. To determine this, we verify the iteration which the coefficient of variation (CoV [2]) of the $k$ iterations falls below a threshold.

S3 For each VM invocation, compute the mean $\overline{x}_i$ of the $k$ benchmark iterations under steady-state.

$$\overline{x}_i = \sum_{j=s_i-k}^{s_i} x_{ij}.$$

S4 Compute the confidence interval for a given confidence level across the computed means $x_i$ from the different VM invocations. The overall mean equals $\overline{x} = \sum_{i=1}^{p} \overline{x}_i$, and the confidence interval is computed over the $\overline{x}_i$ measurements.

[2] The coefficient of variation (CoV) is defined as the standard deviation $s$ divided by the mean $\overline{x}$

At the end of the experiment, we have the mean $\overline{x}_i$ computed across a single VM iteration, and can compute the confidence interval across multiple VM invocations, as they are, in fact, independent of each other.

Unless otherwise specified, we use this methodology throughout the thesis with the aid of Java performance frameworks. We also use third-party benchmarks that focus on steady-state performance evaluation. The DaCapo benchmark [15], for instance, implements such methodology and reports results after the benchmark has converged to the steady-state performance.

### 2.3.2 Java Microbenchmark Harness (JMH)

Several frameworks have been proposed to facilitate specifying and executing steady-state performance measurements in Java. The Java Microbenchmark Harness (JMH) is a commonly used framework [135], that allows users to specify a benchmark through Java annotations. Practitioners can create benchmarks similarly to unit tests in the JUnit framework [167]. Listing 2.1 shows an example of a simple benchmark that measures the performance in terms of execution time and throughout of the `Math.log` method call. Every public method annotated with `@Benchmark` is executed as part of the benchmark suite.

Listing 2.1: Example of a JMH benchmark measuring the performance of the Math.log method call. In this case, JMH will effectively execute two benchmarks, one for x = 10 and the second for x = 1000.

```
@Param({"10.0", "1000.0"})
private double x;

@Benchmark
public void mathLogPerformance() {
  return Math.log(x);
}
```

The JMH framework is designed with an intricate knowledge of JVM optimizations and can help benchmark designers in avoiding related pitfalls. For example, JMH provides the `Blackhole` class, which consumes return values and circumvents dead code elimination [105]. Besides, JMH provides the infrastructure for measur-

ing steady-state performance and aggregate results with rigorous statistical estimates. In Figure 2.2 we illustrate the execution flow that JMH uses to evaluate benchmarks in four major steps.

1. Initially, an optional benchmark fixture is invoked. The benchmark fixture is the code which initializes the benchmark environment (e.g., by filling a data structure with test data).

2. Afterward, a defined number of warmup iterations are executed. These are identical to benchmark iterations, but their results are discarded. Warmup iterations are intended to bring the JVM into the steady state.

3. In the actual benchmark phase, a defined number of benchmark iterations is executed. Each iteration takes a defined amount of time (typically 1s), during which the framework repeatedly calls the method annotated with @Benchmark (a single invocation in JMH parlor) and records all configured performance counters (e.g., throughput, execution time, latency).

4. One entire run of steps 1-3 (executing the benchmark fixture, zero to many warmup iterations, one to many benchmark iterations) is called a trial. By default, JMH executes each trial in a separate VM, and developers may specify how many forks each benchmark will sequentially execute (e.g., we present an example with two forks in Figure 2.2).

After the last iteration, JMH reports a summary of the results to the user and/or saves the results into an CSV or JSON file. Parameters (e.g., the int `x` in Listing 2.1) can be used to easily define different benchmark instances of the same method. In the example in Listing 2.1, JMH will effectively execute the `mathLogPerformance` benchmark twice; once with `x = 10.0` and once with `x = 1000.0`.

## 2.4 Selecting Java Collections

Selecting efficient data structures is a crucial part of writing efficient code. In Java, developers are commonly exposed to data structures in the form of *collections* [141]. Collections, also referred to as containers, are data structures that group multiple

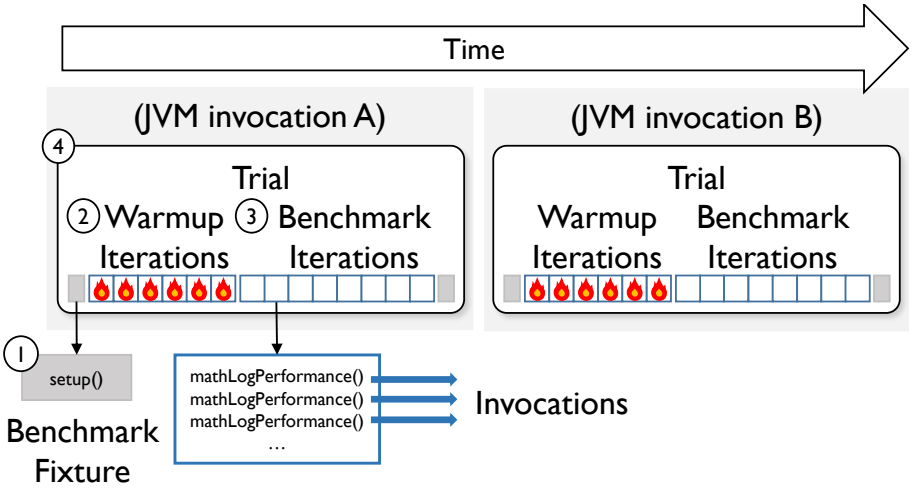



Figure 2.2: JMH execution flow for a benchmark configured with two forks, 6 benchmark warmups and 8 benchmark iterations. In this example, the fixture methods are configured to be executed before and after each trial.

objects or primitives into a single unit, allowing them to be managed as a group. A collection uses one or more *data representations* to efficiently store, manipulate and provide access to the held elements [51]. For instance, an ArrayList is a list implemented with an array representation, while the LinkedHashMap uses both a hash-table and linked-list to provide an ordered map with amortized constant element access time.

Each collection implements a set of operations defined by an abstract data type [116], hereby denoted as the *collection abstraction*, which binds a semantic contract for a collection. For instance, the list abstraction defines collections that maintain the insertion order and provide access to elements by their position. The map abstraction, on the other hand, defines collections that map keys to values, where no duplicated keys are allowed and a key can only map to one value.

The semantic contract imposed by each collection abstraction establishes some ground rules for each collection but leaves the implementation open for different approaches. The map abstraction, for instance, can be implemented using data representations such as hash-tables (HashMap), red-black or AVL trees (TreeMap), or even arrays (ArrayMap), as long as the contract established by the map abstraction is not broken.

Listing 2.2: Example of collection abstractions and variants in Java code. Variant here presented are standard implementations from java.util package.

```
// Array-backed list variant
List<E> list = new ArrayList<>();

// Hash-backed map with double-linked nodes variant
Map<K, V> map = new LinkedHashMap<>();

// Red-black Tree-backed set
Set<E> set = new TreeSet<>();
```

In practice, this abstraction takes the form of commonly used Java interfaces, such as the `java.util.Map`. By using these interfaces, developers may switch to different *variants* of the same abstraction, without affecting the functionality of their code, and may do so to explore functional and non-functional properties of different implementations. Listing 2.2 shows examples of variants of list, map and set abstractions, and how they are normally defined in Java code.

### 2.4.1 Collections Libraries

Collection libraries are so important for software development, that programming languages such as Java, C#, Python, or Ruby include them as a part of the core language environment.The standard implementation of collections in Java is the *Java Collections Framework*, or *JCF* [137]. Albeit the name, JCF is more closely related to a library and was introduced in the early stages of Java development (Java 1.2). The library provides implementations of all major collection abstractions in several variants, e.g., ArrayList, LinkedList and Vector are all JCF variants of the list abstraction.

While JCF offers developers stable and reliable collection implementations, numerous alternative libraries have been developed to provide further options to practitioners [60, 69, 112, 171]. Alternative libraries supply variants with features not supported by JCF (e.g., support for primitive-collections) or richer and domain-specific APIs, such as multisets that are designed to count frequencies of elements. Furthermore, alternative libraries also supply variants to replace commonly-used

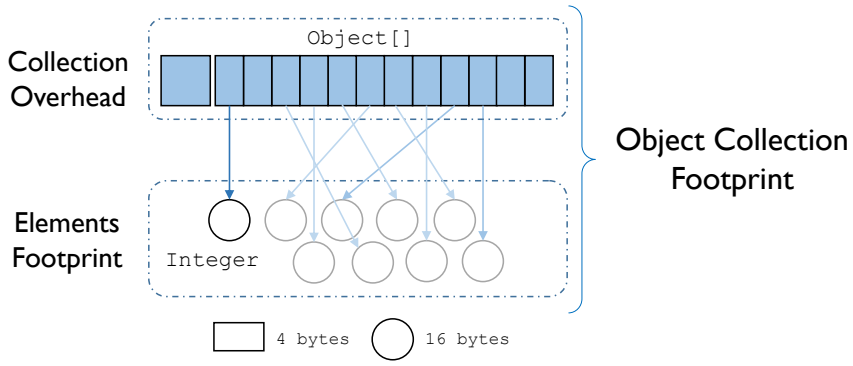



Figure 2.3: Conceptual view of object collections with the example of an ArrayList, and terms related to memory usage.

JCF implementations, providing lower memory overhead and/or faster operations. We perform an in-depth analysis of alternative Java libraries in Chapter 4.

### 2.4.2 Selecting Variants

During software development, programmers select different collection variants for numerous reasons: more expressive APIs, additional features, library reliability and performance. In this thesis, we focus on selecting collection variants for better performance, i.e., lower memory consumption and/or faster operations.

Collections use metadata to track, access and manipulate its elements via specified APIs. Naturally, this metadata incurs extra memory consumption (*collection overhead*) in addition to the memory used by its elements (*element footprint*). In Figure 2.3 we illustrate the collection overhead, element and collection footprint of an ArrayList.

Typically in data structures, execution time and memory consumption have a strong inverse relation: faster collections tend to use more memory and variants with low-footprint suffer from less efficient operations. In some cases, however, variants with lower footprint may have better memory locality, thus boosting time and memory performance. To illustrate both scenarios, we present in the follow-

ing, an example of collection variants that have clear time-space trade-off relations, and a second case where alternative variants might positively impact both time and space performance of an application.

#### Memory-Time Trade-off: Arrays vs. Hashed-Tables

As the name suggests, ArrayMap and HashMaps use respectively arrays and hash-tables to represent their key-map element relations. At first glance, it may appear ill-advised to use array-backed implementations of maps. Array-backed maps consume a small fraction of the memory required for the hash-backed counterparts but rely on binary searches, posing a considerable time penalty in element lookups. The hash-table essentially gives amortized constant lookup time, at the cost of higher memory consumption, caused by the hash-table overhead.

However, when holding a small number of elements (typically at most hundreds), performing binary search on an array have comparable performance to hash-table lookups, due to effects of caching and memory locality. This motivated the Android JDK to provide variants of ArrayMap [8] and ArraySet [9], as mobile applications are sometimes optimized for memory consumption.

#### When Memory gains Performance: Objects vs. Primitives

If a collection contains wrappers of primitive objects like Integer, Long, or Double, *primitive collections* can be used to reduce the collection memory footprint. The key difference is that a primitive collection stores data directly in an array of primitives (int, long, double), instead of using an array of objects. Figure 2.4 illustrates an ArrayList of integers as a primitive collection on the right-side, compared to the standard object collection (left). The primitive variant reduces collection footprint in two ways.

1. The primitive collection needs only a single reference to an array of primitives instead of an array of references. By itself, this reduces the memory-footprint of the collections in 4-bytes/8 bytes per element depending on the JVM configuration.

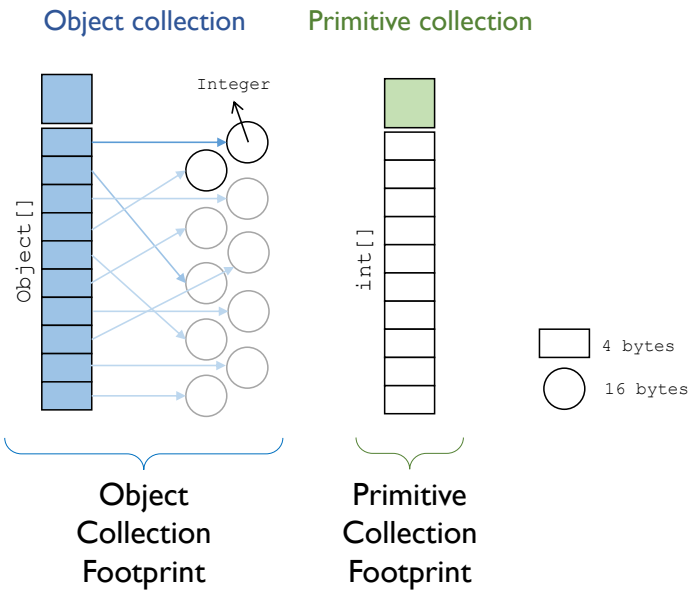



Figure 2.4: Conceptual view of object collections vs. primitive collections with the example of ArrayList, and terms related to memory usage.

2. Each primitive data element (type int) requires only 4 bytes instead of 16 bytes of an object Integer.

In the example of Figure 2.4, using an int-primitive collection can slash memory footprint of collections by a factor of 4. Furthermore, primitive-collections often also improve the execution time of operations, by considerably improving memory locality and making better use of the processor's cache. We investigate this performance gain in Chapter 4.

## 2.5 Parallelizing Java Stream Pipelines

The introduction of lambda expressions in Java 8 marked a shift in Java development paradigm [122], enabling functional programming in the Object Oriented environment and paving the path for expressive APIs such as the Java Stream [52]. The Java Stream API provides the support for writing functional-style operations to process a stream of elements. Element processing is described as a pipeline of aggregate operations (e.g., `distinct`, `filter`), aiming at a declarative programming style: the code focus on "what" it does as opposed to "how". To illustrate the

Listing 2.3: Example of two code snippets that compute the sum of the weights of distinct red widgets. The first uses the traditional procedural approach, while the second exploits the expressiveness of functional programming

```
// #1 Procedural form
int sum = 0;
Set<Widget> distinctWidgets = new HashSet<>(widgets);// removing duplicates
for(Widget widget: distinctWidgets) {
  if(widget.getColor() == RED) {
    sum += widget.getWeight();
  }
}

// #2 Stream pipeline (functional) form
sum = widgets.stream()
  .distinct()
  .filter(b -> b.getColor() == RED)
  .mapToInt(b -> b.getWeight()).sum();
```

expressiveness of stream pipelines, we present in Listing 2.3 an example of two semantically equivalent code snippets. In the first expression, we observe a procedural form that uses a for-loop and a `Set` to remove duplicates. The second code snippet, using Java Stream, declares each process as an operator on a pipeline.

### 2.5.1 The Java Stream Library

The Java Stream Library implemented in the `java.util.stream` package [52] is used to process sequences of objects and primitives. A *stream* is essentially a view on a sequence of elements organized by an underlying data structure, a *(stream) source*. The stream source can be any object that is associated with a `Spliterator` [161], object for traversing and partitioning elements of a data structure (similar to Iterators). Typical stream sources are collections, arrays, and I/O channels. Developers may process this sequence by defining a *stream pipeline* of functional *operations*, such as `filter`, `map`, and others.

Operations are either *intermediate* or *terminal*. An intermediate operation, such as `mapToInt`, produces another stream as a result, while a terminal operation (e.g., `sum`, `count`) results in another object, effectively marking the end of the stream pipeline. Each operation produces a result without modifying its source and is

typically implemented in a lazily fashion, i.e., operations are only computed when the entire pipeline is defined, enabling cross-operation optimizations.

### 2.5.2 Parallelizing Stream Pipelines

On the level a single element in a stream, the operations are performed in the same order as defined in the pipeline. The entire stream, however, may be processed in parallel by several threads. A developer can achieve this by simply replacing the `stream` method to `parallelStream` or `parallel` methods. For the pipeline in Listing 2.3 this would require changing the first code line to `widgets.parallelStream()`. The task of handling the parallel processing of a stream of elements is delegated to the Fork/Join framework [109], that divides the processing into smaller tasks and submits it to a pool of threads.

On the surface, using parallel stream processing is rather programmer-friendly. However, there are no guarantees that parallel stream pipelines will yield better performance than sequential ones nor even lead to a correct result. The correctness of the parallel stream depends primarily on *behavioral parameters* [52], i.e., the functions or predicates used to process each element [103]. In Listing 2.3, e.g., the lambda expression `b -> b.getWeight()` is such a behavioral parameter.

Potential performance improvement due to parallel stream processing depends on a larger number of factors than the behavioral parameters: the number of elements to be processed, the cost of processing each element, the pipeline operations, the data type of the stream source and output. All such factors make the manual assessment of parallelizing stream pipelines a time-consuming and error-prone endeavor. We propose an automated approach for analyzing and reporting the possible benefits of parallelizing stream pipelines in Chapter 6.

# 3 A Study of Bad Practices on Java Benchmarks

As we have established in Chapter 3, measuring the performance of Java applications is a challenging undertaking. Academia has devised methods for sound performance measurement, and the industry has provided tools that allow the creation of fine-grained performance test suites. However, due to the complexities of the Java Virtual Machine, developers still struggle with writing sound benchmarks that accurately represent the performance of the code under test. In this chapter, we present a large-scale empirical study of bad practices in the creation of Java benchmarks under the Java Microbenchmark Harness (JMH) framework.

**Contributions.** In this chapter, we present the following contributions:

I A tool that leverages static analysis to identify five bad practices in JMH benchmark creation automatically.

II An empirical investigation of the prevalence of bad practices in 123 open-source Java projects.

III An experimental evaluation of the impact of each bad practice in multiple case studies.

IV An assessment of the receptivity of fix-patches to projects with benchmarks largely impact by bad practices.

**Reference.** This chapter is based on the following manuscript accepted to appear in [46].

## 3.1 Introduction & Motivation

In large-scale applications, performance issues found during production may cost millions of dollars in maintenance expenses and lost clients [75]. Therefore, practitioners are in a constant battle to identify performance issues during the development phase, where issues can be cheaply fixed and without public scrutiny.

To that aim, practitioners need to test the performance of their system during development. Similar to functional testing (e.g., through integration and unit tests), performance testing should be done at different granularities. The ultimate goal of performance testing is to achieve a good end-to-end performance. Even though the impact of the performance of each component on the end-to-end performance of a system is not necessarily linear, obvious performance issues at the component-level are likely to ripple through to the system-level. Hence, practitioners must first ensure that the performance of each component of the system is adequate.

Benchmarking and performance unit testing are two related approaches to assess the performance of program code. *Benchmarks* evaluate the performance of a (typically small) code segment [5, 107, 135, 154]. In contrast to stress or load tests, which test the end-to-end performance of a system, benchmarks are relatively short-running and aim at measuring the fine-grained performance of specific units of program code. For instance, a benchmark may measure method-level execution times of a class, the performance of a specific data structure, or the implementation of an algorithm.

The outcome of a benchmark run is a set of one or multiple performance counters, such as the execution time or the throughput of an operation. A developer or quality engineer uses these performance counters to compare different implementation alternatives or detect slowdowns (i.e., by statistically comparing the performance counters produced in a new version of the product with counters produced by previous releases). Performance unit tests similarly operate on small code segments, but they entail real target values, akin to asserts in unit testing for functional defects [50, 163].

We have briefly described in Section 2.3, the challenges of performance measurement in Java and how academy and industry have both provide tools and methodologies for accurate steady-state performance evaluation. Recall that ex-

perts developed the Java Microbenchmark Harness (JMH) to provide the scaffold to measure steady-state performance and an API to deal with some of the dynamic optimizations the JVM performs in the code. While the JMH offers a reliable infrastructure for benchmarking, the responsibility of creating a reliable and correct benchmark remains with the developer.

Unfortunately, developers still struggle with the creation of sound JMH benchmarks that represent the performance of the target code [154]. Previous work established several pitfalls developers need to avoid [62] and the execution results of real-life benchmarks of large open source projects can vary significantly across repeated executions in identical environments [107].

In this chapter, we conduct the first empirically study on bad practices of writing JMH benchmarks that can lead to misleading benchmark results. In particular, we attempt to answer two key open-questions:

**RQ1** **How frequently do bad JMH practices occur in real-life open source software?** To assist developers with avoiding bad practices, we implement a plugin (SpotJMHBugs [44]) for the SpotBugs static analysis tool. SpotJMHBugs can automatically identify the bad JMH practices and is used in this study to identify the bad practices in a set of 123 open-source Java projects.

**RQ2** **What is the impact of the identified bad JMH practices on the benchmark results?** We manually fix benchmarks found across six projects and measure the impact of the bad practices by comparing the results before and after the fix.

## 3.2 Related Work

The accurate performance measurement of a software is a problem that has continuously attracted research attention in the past decades. We categorize the related work into three categories: 1) pitfalls of benchmarking, 2) errors in performance evaluation, and 3) methodologies for robust performance analysis.

### 3.2.1 Pitfalls of Benchmarking

A major challenge of correct benchmarking is to prevent misleading results due to the compiler and JVM optimizations, including constant folding, loop unrolling, and method inlining [5]. Frameworks such as JMH [135] are designed with an intricate knowledge of JVM optimizations and can help benchmark designers in avoiding related pitfalls.

Especially close to our study on bad practices in benchmark creation are approaches and methodologies which ensure the correctness of benchmarks or performance unit tests [5, 83, 97, 154]. Rodriguez-Cancio et al. [154] proposed a combination of static and dynamic analysis and code generation to synthesize benchmarks evaluating code segments extracted from large applications. Their tool, named AutoJMH, generates payloads which prevent dead code elimination and constant folding, optimizations which can lead to common mistakes in benchmarks.

In the investigation that follows, we consider a broader set of problems related to JMH, including more complex ones, e.g., using accumulation to consume loop computation in a benchmark. We also focus on detecting such bad practices via a static analysis tool and conduct empirical studies on the prevalence of bad practices and the impact of their fixes.

Further work focusing on the correctness of benchmarking discusses the issues that can hinder the experiment or mislead the evaluation in Java [83], or describe how JMH can be used to avoid typical pitfalls [97]. Additional contributions in this area propose methods for generating benchmarks for scenarios in which JVM optimizations are not harmful [106, 149]. Kuperberg et al. [106] proposed an automated solution for benchmarking any set of APIs, e.g., the Java Platform API. Contrary to the approaches mentioned above, this solution specifically induces the JIT optimizations to "obtain realistic benchmarking results". Pradel et al. [149] introduced SpeedGun, a technique which can automatically discover performance regressions in thread-safe classes. Also, in this scenario, JVM optimizations do not affect the results (assuming that both code versions are optimized).

### 3.2.2 Performance evaluation errors

A large number of work studies the reasons for incorrect or biased performance evaluation results. A commonly identified problem is non-determinism of individual executions caused by the complexity of the runtime environments, in particular, the JVM [16]. Another source of performance variations is the OS Jitter phenomenon [131], e.g., the impact of the thread affinity values and settings on the execution time.

While the impact of non-determinism on the results can be addressed by repetition of experiments and rigid statistical procedures [25, 62], a more serious problem is a measurement bias caused by presumably innocuous factors [129]. Such so-called *hidden factors* can take various forms, such as link order of code segments/libraries, or UNIX environment size [29].

Mytkowicz et al. [129] conducted one of the first comprehensive studies on hidden factors and their causes and proposed a method for their detection (via causal analysis) and for avoiding them (setup randomization). Curtsinger et al. [50] addressed the impact of the layouts of code, stack, and heap objects at runtime on the performance. They developed a tool for randomizing the memory layout, which is combined with statistical techniques such as ANOVA for sound performance optimization (in particular related to the impact of compiler optimization levels).

Other studies reported different types of hidden factors. Kalibera et al. [99] demonstrated the impact of the initial state of the system on the performance results. Harji et al. [79] showed that the Linux kernel had multiple performance-related regressions, resulting in performance variation as much as 45% between two subsequent versions. Consequently, the choice of the kernel version might have a significant impact on the benchmark results. In an earlier version of the JVM, changing the names of symbols significantly affected the cache miss count and thus the performance of applications [77]. In another study of Java performance, the authors reported that simply restarting the virtual machine could cause performance variations as high as 3% [66].

Our work focuses on the incorrect usages of the JMH framework which can result in measurement biases. The reasons for such errors are in most cases unwanted JVM optimizations. Contrary to the approaches to avoid hidden factors

such as setup randomization, unwanted optimizations can be eliminated via a thorough understanding of the optimization process and correct benchmarking code. Nevertheless, detection of measurement biases due to novel optimizations or JVM internals remains an open challenge, similarly to generic detection approaches for hidden factors.

### 3.2.3 Methodologies for robust performance analysis

Prior work proposed methodologies, frameworks, and specific techniques to ensure correctness and robustness of performance evaluation results in the face of non-determinism inherent in complex computer systems. In most cases, this work is complemented by empirical studies or literature analyses which demonstrate the problems.

Georges et al. [62] focus on the data analysis aspects of the performance evaluation of Java programs. Their study of reported Java performance results available at that time uncovered a need for a statistically rigorous evaluation methodology in the face of the non-determinism of the Java runtime. They proposed and evaluated several statistical measures to address this problem while considering practical aspects, such as best practices for quantifying startup and steady-state performance. The conclusions of this work were partially extended by Bulej et al. [25], where the authors showed the pitfalls of applying basic statistical methods to data from a real performance benchmark (SPECjbb2015).

Blackburn et al. [16] showed how the complexity and a large number of degrees of freedom of the Java runtime system could lead to misleading performance results. They argue that benchmark designers must use relevant workloads, principled experimental design, and rigorous analysis to produce meaningful results, and illustrate their reasoning on the design choices for the DaCapo benchmark [15].

Other work focused on specific aspects of performance evaluation. Alexander et al. [6] proposed using nonlinear time series analysis techniques to capture the complex dynamics of computer performance data. Kalibera and Jones [100] addressed the trade-off between the cost of experiments (in terms of the number of repetitions) and the statistical validity of the results. They introduced a mathematical model for adjusting the number of repetitions to the level of uncertainty and

evaluate it using the DaCapo and SPEC CPU benchmarks. De Oliveira et al. [132] proposed DataMill, a distributed infrastructure for rigorous performance evaluation with a particular focus on eliminating the impact of hidden factors via their automated variation. Moreno and Fischmeister [127] discussed a simple technique to eliminate the systematic error introduced by the `get_current_time()` system call, a relevant problem in benchmarking.

The work we present in the following focuses on the particular domain of Java benchmarks, which is a relatively new class of evaluation methods. Our study quantifies the prevalence and impact of misuse of the JMH benchmarks, attempting to raise the awareness of correct benchmarking. We also provide a set of recommendations for practitioners and researchers as a "soft" methodology for robust performance analysis.

## 3.3 Bad JMH Practices in Benchmark Creation

The art of designing sound benchmarks is a hard craft to master. Reading and becoming acquainted with the JMH documentation and API is the first of many steps, in the long path for mastering JMH benchmark creation. JMH offers in its documentation a series of 38 samples benchmarks [142] that illustrate coding pitfalls and bad JMH practices that can affect the reliability and correctness of a benchmark. In the remainder of this section, we discuss five of the most important bad JMH practices described in the JMH documentation (see Table 3.1 for an overview). All code examples in this section were taken from the JMH samples.

Table 3.1: Bad JMH practices collected from the JMH documentation.

| ID | Bad JMH Practice Description | Undesired Effect |
|---|---|---|
| RETU | Not using a returned computation | Dead code optimization |
| LOOP | Using accumulation to consume values inside a loop | Loop optimization |
| FINAL | Using a `final` primitive for benchmark input | Constant folding |
| INVO | Running fixture methods for each benchmark method invocation | JMH overhead |
| FORK | Configuring benchmarks with zero forks | Profile-guided optimization |

### 3.3.1 RETU: Not consuming a result from a method call

*Description:* A benchmark typically calls one or more methods from the main application code. If such a method returns a result that is not used in the benchmark, the JVM may consider the called method "dead code" and eliminate the call either partially or entirely.

*Symptoms:* Because the call to the benchmarked method was eliminated, the code will appear faster than in actual usage. Listing 3.1 shows an example of the RETU bad JMH practice and two possible solutions. In the benchmark `measureWrong()`, `Math.log(x1)` is redundant and may be eliminated by the JVM.

*Solution:* Every object that is returned by a method called directly from the benchmark should be used in the benchmark method. In the benchmark `measureRight1()`, `Math.log(x1)` is used as a return of the benchmark method, and therefore, not eliminated. Alternatively, the JMH infrastructure offers a `BlackHole` object which can be used to prevent dead-code elimination by consuming the result. As shown in `measureRight2()`, `Math.log(x1)` is consumed by a `BlackHole` object.

Listing 3.1: Example of the RETU bad JMH practice and two possible solutions.

```
private double x1;

@Benchmark
public double measureWrong() {
   Math.log(x1); // Call is redundant and will be removed by the compiler
}

@Benchmark
public double measureRight1() {
   return Math.log(x1); // JIT will not remove the call as it has no
                        // guarantee the return is not used
}

@Benchmark
public double measureRight2(Blackhole bh) {
   bh.consume(Math.log(x1)); // Blackhole object tricks the JIT compiler
                             // into "thinking" the result is being used
}
```

### 3.3.2 LOOP: Using accumulation to consume loop computation

*Description:* Developers often have to design benchmarks that measure a method call within a loop. If the method call returns a numeric variable, it is intuitive to accumulate the returned objects as a way of avoiding dead-code elimination [105]. However, this leads to another set of JVM optimizations that optimizes the code beyond what would be expected in real usage.

*Symptoms:* The code appears faster than in actual usage, as the loop can be extensively optimized by the JVM. Listing 3.2 shows an example of the LOOP bad JMH practice. The `measureWrong()` method executes every `work()` method as intended, but the JVM is able to unroll the loop and merge operations between two distinct `work()` calls. Such optimizations are only performed because accumulation is used instead of a proper consume method, and will not hold in a scenario where the application actually uses or stores the return of each method call.

*Solution:* The user should avoid using accumulation as a method of consuming the numeric return and use the `Blackhole` facilities instead. The `measureRight()` method shows how a loop can be benchmarked safely.

Listing 3.2: Example of the LOOP bad JMH practice and a possible solution.

```
@Benchmark
public int measureWrong() {
int acc = 0;
  for (int x : xs) {
    // Loop can be unrolled and multiple work() calls can be merged
    // optimizations that would not be possible without the += accumulator
    acc += work(x);
  }
  return acc;
}

@Benchmark
public void measureRight(int x, Blackhole bh) {
  for (int x : xs) {
    bh.consume(work(x));
  }
}
```

### 3.3.3 FINAL: Using final primitives for benchmark input

*Description:* If the JVM realizes that the result of a computation is predictable, it will optimize the computation (i.e., using *constant folding* [173]), thereby affecting the runtime of the benchmark.

*Symptoms:* Because the constant computation was folded, the code will appear faster than in actual usage. Listing 3.3 shows an example of the FINAL bad JMH practice. The `measureWrong()` method does a computation using the `Math.PI` constant, making the result of the computation predictable and hence foldable.

*Solution:* Benchmark inputs should always be read from non-final primitive instance fields. The `measureRight()` method in Listing 3.3 conducts the computation in a proper way, since its result is not predictable at compile time.

Listing 3.3: Example of the FINAL bad JMH practice and a possible solution.

```
private double x = Math.PI;
private final double wrongX = Math.PI;

@Benchmark
public double measureWrong() {
   // Computation is predictable
   return Math.log(wrongX);
}

@Benchmark
public double measureRight() {
   // Computation is not predictable
   return Math.log(x);
}
```

### 3.3.4 INVO: Using invocation-level fixture methods

*Description:* Fixture methods are methods that are used to setup or tear down a benchmark. In Figure 3.1 we illustrate the three levels at which JMH allows fixture methods to run: (1) before/after a benchmark trial, (2) before/after a benchmark iteration and (3) before/after a benchmark method invocation. In most cases, it is a bad JMH practice to use the third option, as the overhead of the JMH infrastructure might be large compared to the actual benchmark runtime.

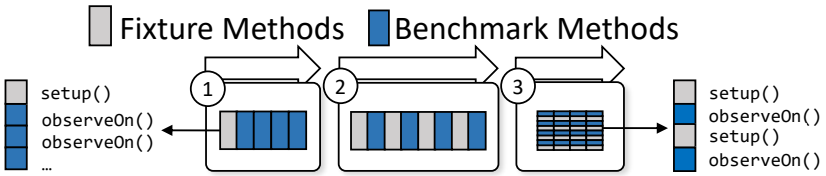



Figure 3.1: JMH setup execution flow when configured to run (1) before a benchmark trial, (2) before each benchmark iteration and (3) before each benchmark invocation. The third option has several drawbacks and is classified as a bad practice (INVO) for short-running benchmarks.

*Symptoms:* The JMH infrastructure must add a timestamp to each method invocation to calculate its execution time, as the time spent in the fixture methods is excluded from the performance measurement. On short-running benchmarks, which typically run for less than a millisecond, JMH saturates the system with timestamp requests, offsetting the measurements. According to the JMH documentation, this level might also omit hiccups from time measurement, introducing unexpected and surprising results. Listing 3.4 shows an example of the INVO bad practice. As fixture methods are shared among benchmarks of the same class, the overhead caused by JMH will offset the measurements of all benchmarks, including the ones that do not access objects created in the setup/teardown methods.

*Solution:* Every invocation-level fixture method should be checked to make sure that it is necessary to be called at the invocation level. This necessity is rare and can be avoided in most situations by including the contents of the fixture method in the benchmark method (causing less overhead).

Listing 3.4: Example of the INVO bad JMH practice.

```
// This method will be executed before and after every benchmark invocation
@TearDown(Level.Invocation)
public void check() {
  assert x > Math.PI : "Nothing changed?";
}
```

### 3.3.5 FORK: Configuring benchmarks with zero forks

*Description:* The JVM is good at profile-guided optimization, i.e., optimization that is based on the usage profile of a method [11]. However, such optimizations should be avoided in benchmarking, since a profile that was optimized for one benchmark may be reused across other benchmarks. In addition to the profile-guided optimization, running a non-forked benchmark may cause the JMH infrastructure to omit JVM options and compiler hints. These options and hints could be paramount to ensuring the correctness of the benchmark results. To avoid profile-guided optimization and risk affecting the execution correctness, each benchmark should be executed in its own VM. Running a benchmark per VM is the JMH default behavior, however, it is possible to override this behavior using the `@Forks` annotation.

*Symptoms:* The code can appear faster or slower than in actual usage, depending on the optimized profile that was used by the JVM. The Listing 3.5 contains an example of a case where profile-guided optimization leads to unreliable benchmark results.

*Solution:* Do not override the default JMH behavior for running a benchmark trial per VM unless there is a very good reason to do so.

Listing 3.5: Example of the FORK bad JMH practice.

```
// Forces JMH to execute the benchmark in the same JVM invocation
@Fork(0)
@Benchmark
public int measureWithNoForks() {
  return work();
}
```

## 3.4 Methodology

We devise the methodology of this study focusing on three main goals:

1. Identify how frequently bad JMH practices occur in open source software projects (Section 3.5).

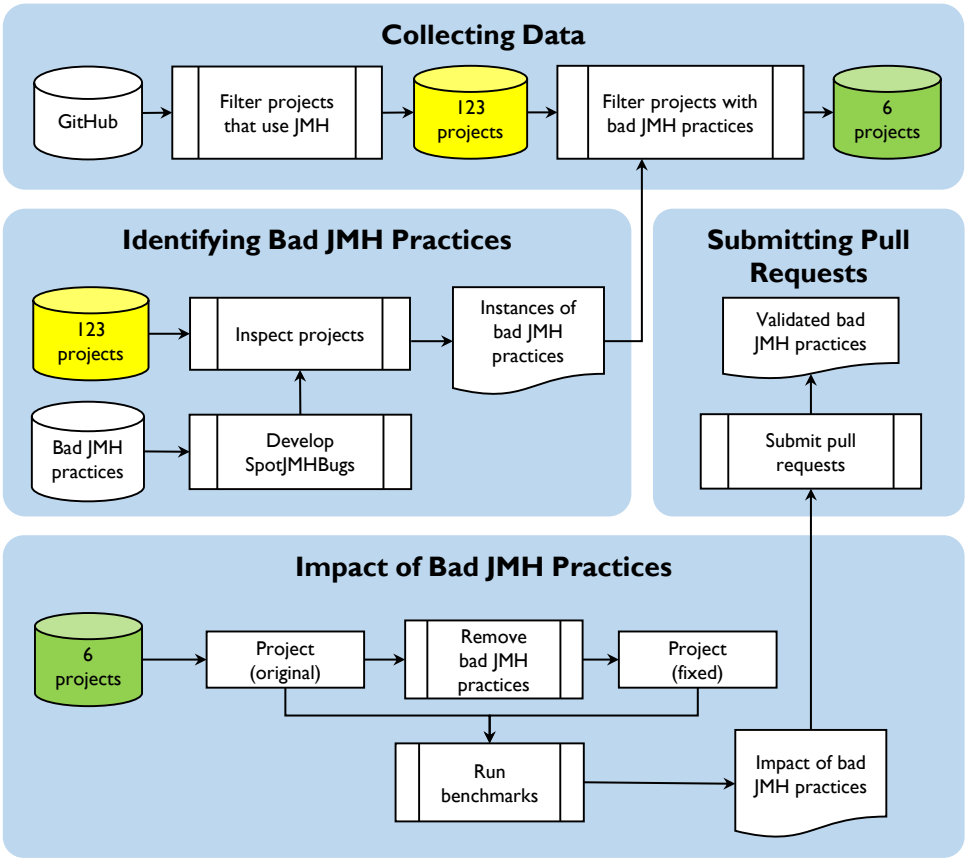



Figure 3.2: Overview of our study approach. Each step of our methodology is detailed in the respective sections.

2. Study the performance impact of the used bad practices on the benchmark results of those projects (Section 3.6).

3. Validate our findings with the developers of those projects, by proposing patches to address the identified bad practices (Section 3.7).

We present in Figure 3.2 the overview of our methodology, further detailed in the following sections.

### 3.4.1 Identifying Instances of Bad JMH Practices

To identify the usage of bad JMH practices in the studied projects, we built a static code analyzer, SpotJMHBugs, for the JMH benchmarks. Our analysis tool was

implemented as a plugin for the SpotBugs[1] tool, the successor of FindBugs [58]. SpotJMHBugs is a rule-based tool that analyzes Java byte-code, identifies bad JMH practices, and reports them to the developer. We discuss the SpotJMHBugs tool in more detail in Section 3.5.1.

### 3.4.2 Collecting Data

Below we present our methodology for collecting the data for our study on (1) the frequency of bad JMH practices and (2) the performance impact of these bad JMH practices on the benchmark results of the projects. The full list of projects and data sets that we base our results on can be found in our online appendix [45].

#### Data Collection for Studying the Frequency of Bad JMH Practices

We queried the 2017 GitHub snapshot (the latest snapshot available at the time of our study) using Google Bigquery[2] to identify open source Java projects that contain at least one JMH benchmark. Concretely, we query for source files that import `org.openjdk.jmh.annotations.Benchmark` and have at least one method annotated with `@Benchmark`. This led to a full data set of 839 projects.

In a second step, we remove forked projects to avoid biasing our analysis towards popular programs' characteristics. Projects such as `RxJava` have 16 forked versions in our dataset (aside from the original from ReactiveX) and would distort our results. The set without forked projects contains 506 projects. Because our SpotJMHBugs tool analyzes JVM bytecode, it requires a compiled project to analyze. As manually compiling and executing tests for a large number of projects is extremely time-consuming, if at all possible [107], we select a subset of 123 projects that could be built automatically or with minimal intervention using Gradle or Maven for our study. Figure 3.3 shows the descriptive statistics of the selected projects in terms of their number of stars and subscribers. Our selected projects cover a range of popular and less-popular projects.

[1] https://spotbugs.github.io/
[2] https://cloud.google.com/bigquery/

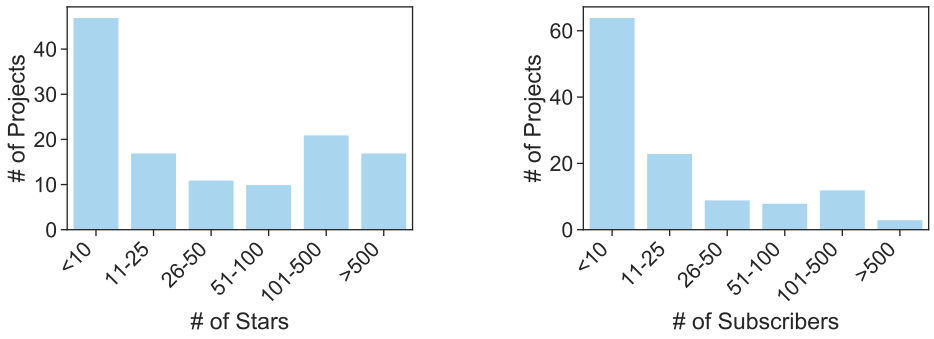



Figure 3.3: Distribution of stars and subscribers of the 123 projects used to identify bad JMH practices.

### Data Collection for Studying the Performance Impact of Bad JMH Practices

In the second part of our study, we study the performance impact of the bad JMH practices (Section 3.6). We limit our project selection further for this part of our study, as it requires manually addressing the instances of the identified bad JMH practices (and therefore requires knowledge about the project). To study the performance impact of bad JMH practices, we select projects that match the following criteria:

- The project is in the top-3 projects ranked by the number of stars on GitHub for a specific bad JMH practice.
- The project contains at least 2 instances of the identified bad JMH practice.

These selection criteria help us focus our efforts on popular projects and projects with multiple instances of a bad JMH practice, thereby reducing the effort that is necessary to address bad JMH practices.

Table 3.2 gives an overview of the projects for which we study the performance impact of the followed bad JMH practices. Initially, our set of projects was composed of three projects per bad JMH practice. However, after careful inspection, we decide to remove two projects initially considered for the impact assessment of the FORK bad practice: `oopsla15-artifact` and `benchmark-arraycopy`. The first project was created as a paper artifact for the OOPSLA conference and the second

Table 3.2: Selected projects per bad JMH practice. We select the top-3 most starred projects with at least two instances of each bad JMH practice. Column # BP shows the total number of bad practice instances that were identified in the three projects.

| | Selected Projects | | | # BP |
|---|---|---|---|---|
| RETU | netty | gs-collections | logging-log4j2 | 54 |
| LOOP | netty | druid | logging-log4j2 | 29 |
| FINAL | netty | druid | logging-log4j2 | 9 |
| INVO | netty | druid | h2o-3 | 18 |
| FORK | pgjdbc | | | 2 |
| **Total number of evaluated instances of bad JMH practices** | | | | 112 |

is a series of benchmarks created to evaluate a specific library function. Hence, both are not good examples of production-quality open-source software, which is the primary subject of our study. Table 3.2 shows the list of projects ultimately selected for the impact assessment. Note that some projects (e.g., the `netty` project) were selected for multiple categories. For the performance impact study, we selected 6 projects which contain a total of 93 instances of the 5 studied bad JMH practices (see Section 3.3).

### 3.4.3 Assessing the Performance Impact of Bad JMH Practices

To assess the performance impact of bad JMH practices, we manually analyze the instances of the bad JMH practices that we identified in the 6 projects in Table 3.2. We then generate an alternative, "fixed" version of the benchmarks, by removing the bad practices according to the solutions proposed by the JMH documentation as stated in Section 3.3. It is of paramount importance that in our fix we do not introduce artificial latency or modify what has been measured in a benchmark. Most fixes are *non-intrusive*, and require simple code refactoring, such as consuming variables, removing the final modifier from a primitive field, or some level of benchmark reconfiguration.

However, one solution for removing the INVO bad practice required us to introduce the code from fixture methods inside the benchmark. We classify this particular solution as *intrusive* and benchmarks fixed with such solution are evaluated using a separate methodology.

Assessing the Impact of Non-Intrusive Fixes. At the end of our experiment we collect the resulting performance counters (in particular, we focus on the execution time of the benchmarks). If a bad JMH practice is irrelevant to the benchmark result, both the original and fixed version should lead to similar performance counter distributions. If the distributions differ significantly, we conclude that the bad practice impacts the benchmark result.

Assessing the Impact of Intrusive INVO Fixes. In the case of the INVO fix, we also collect and compare both performance counter distributions. If inserting the setup/teardown code inside a benchmark method makes the benchmark faster to execute, the JMH overhead takes longer than the time spent on fixture methods. In this case, we conclude that the benchmark was impacted by the INVO bad practice.

To test whether the distributions of performance counters for the original version and fixed version are statistically significantly different, we used the Wilcoxon non-parametric test [176] with a significance level of $\alpha = 0.01$. The Wilcoxon test only indicates whether the distributions have a statistically significant difference. However, the test does not indicate whether the difference is large enough to be noticeable in practice. To quantify the difference, we use Cliff's Delta effect size [33]. We use the following common thresholds [155] for interpreting the effect size:

$$\text{Effect size } d = \begin{cases} negligible(N), & \text{if } |d| \leq 0.147 \\ small(S), & \text{if } 0.147 < |d| \leq 0.33 \\ medium(M), & \text{if } 0.33 < |d| \leq 0.474 \\ large(L), & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

We consider the benchmark as *impacted* if it has at least one benchmark instance where the performance counters of the fixed version differ significantly from the original, with a non-negligible effect size. In the particular case of the INVO intrusive fix, we consider the benchmark as *impacted* if all benchmark instances yield performance counters comparable or faster than the original version.

We further define the *benchmark effect size*, as the highest absolute effect size observed in its instances. The reasoning behind this definition is that a benchmark should yield consistent results on all defined input parameters. A single set of

parameters in a benchmark that is impacted by a bad JMH practice, is sufficient to mislead an analysis and affect the benchmark quality.

### 3.4.4 Evaluating Fixed Versions and Results with Developers

To evaluate the identified instances of bad JMH practices, and their assessed impact, we manually submitted pull requests to six open source projects (selected based on where we found the largest impact on the benchmark results). These pull requests contained the fixed versions that we constructed as part of our study. The goal of this step was two-fold. Firstly, we wanted to see whether the developers agree with our assessment that the original benchmarks produced misleading results. Secondly, we wanted to validate whether the developers agree with the fixes we implemented for the benchmarks.

## 3.5 Identifying Bad JMH Practices

In this section, we present the results of our first RQ: *How frequently do bad JMH practices occur in real-life open source software?*

### 3.5.1 The SpotJMHBugs Tool

To investigate the occurrences of bad JMH practices in Java projects, we first derive a set of rules that can be used to identify such practices via static code analysis. We present a brief description of each derived rule in Table 3.3. Bad JMH practices that relate to the benchmark configuration, such as INVO and FORK are easily verifiable and have unique static rules. For instance, FORK requires a simple check on the occurrence of a `@Fork` annotation with a value of 0.

The RETU, FINAL and LOOP bad JMH practices are related to the source-code and manifest themselves in different ways. In such cases, we use heuristics to identify scenarios in which the undesired JVM optimizations could happen. For instance, the RETU bad practice may occur when a variable is not properly consumed in the benchmark or when developers ignore the return of a static method call. Our rule for identifying unconsumed variables in a benchmark is based on

the following principle. A local variable $V$ is considered consumed if at least one of the following criteria are met:

1. $V$ is stored into a class field.
2. $V$ is returned at the end of the benchmark method.
3. $V$ is consumed by a JMH Blackhole object.
4. There exists another variable $V'$ that has a data dependency on $V$ and $V'$ is a consumed variable. To check if such a dependency exists, we build a data-dependency graph [174], and verify the existence of a path between $V$ and $V'$ in the graph.

Our rule reports every local variable that does not fulfill the above mentioned criteria and is therefore prone to dead-code elimination. For an explanation of the other rules, we refer the reader to the source-code documentation of SpotJMHBugs [44].

Our tool can be executed through a batch command using Maven, Gradle or Ant, or integrated with the Eclipse IDE. If used in conjunction with Eclipse, the warnings about bad JMH practices are shown directly in the editor view. SpotJMHBugs restricts its analysis to classes that contain at least one method annotated with `@Benchmark`, as bad JMH practices are only potentially harmful in the context of JMH benchmarks. Calls to methods outside of benchmark classes are skipped, keeping the analysis time short and primarily dependent on the JMH benchmarks, which tend to be a very small fraction of the overall application code base [111, 163].

Table 3.3: Static rules used to identify bad JMH practices.

| Bad JMH Practice | Static Rule |
|---|---|
| RETU | Variable not consumed in the benchmark<br>Ignored return of static method calls |
| LOOP | Numerical variable is accumulated in a loop |
| FINAL | Final and non-static primitives in @State classes |
| INVO | @Setup/@Teardown with invocation level |
| FORK | @Fork with a value of zero |

### 3.5.2 Results

35 out of 123 projects (28%) contained at least one instance of a bad JMH practice. Table 3.4 shows the number of identified instances of bad JMH practices per project together with the number of benchmarks potentially affected by such instances. If we limit our consideration to the 49 projects with more than 10 benchmarks, the share of projects with at least one bad JMH practice increases to 51%. The number of benchmarks potentially affected by bad JMH practices varies considerably per project. In 23 projects we identified at most 10 instances, while we identified more than 10 instances in 12 other projects. In total, SpotJMHBugs identified 331 instances of bad practices in our dataset.

Table 3.4: Distribution of identified instances of bad JMH practices in *all* 123 projects and on a subset of 49 projects containing at least 10 benchmarks.

| | # of Identified Bad JMH Practices | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Dataset | 0 | 1 | 2 | 3 | 4 | 5 | 6-10 | +10 |
| All projects | 88 | 9 | 6 | 2 | 1 | 1 | 4 | 12 |
| Projects with 10+ benchs | 24 | 3 | 4 | 1 | 1 | 0 | 4 | 12 |

LOOP was the most commonly identified bad JMH practice, with a presence in 13% of the studied projects. Table 3.5 summarizes the number of identified instances for all bad practices. The second most commonly identified bad JMH practice was the RETU bad practice, occurring in 12% of the studied projects. FINAL and INVO occurred in respectively 9 (7%) and 10 (8%) of the studied projects. The FORK bad practice was identified in only 3 projects.

Table 3.6 gives a detailed overview of the distribution of the bad practices across the top 25 projects with the largest number of benchmarks in their benchmark suite in our dataset. Overall, the distribution of identified bad JMH practices appears to be very particular to each project. For example, although `RxJava` has 215 benchmarks, SpotJMHBugs did not identify any bad JMH practice in the project. On the other hand, we identified instances of four different bad JMH practices in the benchmarks of `netty`. In total, we found 22 bad practice instances in this project alone. Aside from FORK (which was found in only three projects) every bad JMH practice was found in at least six projects.

Table 3.5: The number of identified instances of bad JMH practices for all 123 studied projects. The 'Total' column shows the number of instances found in all projects per bad JMH practice.

| Bad JMH Practice | Total | # of Projects | % of Projects |
|---|---|---|---|
| RETU | 89 | 15 | 12.2 |
| LOOP | 128 | 16 | 13.0 |
| FINAL | 25 | 9 | 7.3 |
| INVO | 82 | 10 | 8.1 |
| FORK | 7 | 3 | 2.4 |

**RQ1. How frequently do bad JMH practices occur in real-life open source software?**

28% of the studied projects contained at least one instance of a bad JMH practice. Our results show that the studied bad JMH practices occur frequently in open source projects: LOOP was the most frequently ocurring bad JMH practice, but aside from FORK, all bad practices appeared in at least 6 projects.

## 3.6 Impact of Bad JMH Practices

In this section, we present the results of our second RQ: *How frequently do bad JMH practices occur in real-life open source software?*. To answer the aforementioned question, we fix the identified bad JMH practices in selected projects (Section 3.6.1) and compare the benchmark times of the fixed and original benchmarks (Section 3.6.2).

### 3.6.1 Generating Fixed Versions

We aim to evaluate the impact on performance of each bad JMH practice separately. Thus, in projects that had multiple identified bad practices, we generate multiple fixed versions, each containing fix-patches for a single bad practice. For instance, we identified three bad JMH practices in `druid`, thus we generate three fixed versions. In total, we fix 93 instances of bad JMH practices, generating 13

Table 3.6: The number of instances of bad JMH practices that were found in the 25 studied projects with the largest benchmark suites. The projects in bold were selected for an experimental evaluation of the impact of bad JMH practices instances in the next step of the study. The "Benchs" column denotes the number of `@Benchmark`-annotated methods. We also included the `h2o-3` project in the impact assessment which was not in the listed top-25 projects.

| Project | Stars | Benchs | RETU | LOOP | FINAL | INVO | FORK |
|---|---|---|---|---|---|---|---|
| **gs-collections** | 1652 | 451 | 47 | | | | |
| **logging-log4j2** | 256 | 346 | 5 | 7 | 5 | | |
| RxJava | 23558 | 215 | | | | | |
| oopsla15-artifact | 16 | 213 | 4 | 1 | 2 | 12 | 3 |
| **netty** | 9746 | 159 | 2 | 14 | 2 | 4 | |
| reactive-streams | 106 | 157 | 2 | | | | |
| **druid** | 4743 | 148 | | 8 | 2 | 2 | |
| JCTools | 1053 | 92 | | 1 | | 2 | |
| golo-jmh-benchmarks | 4 | 92 | | | | | |
| zipkin | 5627 | 74 | | | | | |
| microbenchmarks | 7 | 67 | | 17 | 5 | | |
| xodus | 248 | 66 | | 7 | | 18 | |
| lab-java8stream | 4 | 64 | | 6 | | | |
| mini2Dx | 137 | 55 | | | | | |
| fast-select | 3 | 48 | | | | | |
| jenetics | 183 | 47 | | 8 | 1 | | |
| rtree | 482 | 42 | | | | | |
| byte-buddy | 1495 | 39 | | | | | |
| caffeine | 2414 | 38 | 1 | | | | |
| **pgjdbc** | 322 | 35 | | | | | 2 |
| java-final-benchmark | 10 | 34 | | | | | |
| streamalg | 15 | 34 | | 4 | | | |
| pinot | 1475 | 31 | | 16 | | | |
| cache2k-benchmark | 12 | 28 | | | 1 | | |
| template-compiler | 12 | 27 | | | | | |
| **h2o-3** | 1943 | 18 | | 6 | | 12 | |

fixed versions for 6 selected projects. Our concrete process for generating fixes differs per bad JMH practice. We detail how fixed versions were generated when discussing the results for each bad practice.

### 3.6.2 Running Benchmarks

As mentioned in Section 2.3, JMH helps to mitigate the uncontrolled factors by allowing developers to configure the number of warmup iterations, benchmark iterations, and forks. This configuration is highly important to achieve reliable and repeatable results in a benchmark, e.g., too few warmup or benchmark iterations may yield a large variance in the performance counters of some benchmarks [107].

To avoid building our analysis on unreliable benchmark configurations, we repeat each experiment 5 times, while keeping the original benchmark configuration (i.e., the number of warmup and benchmark iterations, and forks). Our experiments generate a median of 780 performance counters (min=100, max=14,400) for each benchmark instance. Finally, we also alternate runs between the original and fixed versions to reduce the chances of circumstantial external influence impacting only one version of the program.

We conducted our experiments on a computational server with an E5-1660-3.3GHz CPU, with 6 physical cores and 64 GB RAM using Linux 3.16.0-53. The benchmarks use the JVM HotSpot 64 bits and jdk1.8.0_65 as the Java version. Aside from the basic operating system functionality, no other process was running during the execution of our experiments.

### 3.6.3 False Positives

We describe in this section, for each bad JMH practice, the criteria for filtering false-positives. False positives are bad JMH practices that were wrongly reported by our tool, due to (1) limitations in our rules or (2) needing to understand the intent behind the benchmark creation.

Table 3.7 shows the number of false positives found (the 'FP' column) per bad JMH practice. Upon manual analysis of the 112 bad practice instances that were initially identified by SpotJMHBugs, we found that 19 (17%) were false positives reported by our tool. Note that the remaining 93 bad JMH practice instances could affect 105 benchmarks, as a single instance of INVO and FORK may affect multiple benchmarks. We further detail the criteria used to filter the encountered false positives.

Table 3.7: Identified bad JMH practice instances instances characterized as false-positives (FP) by further manual inspection. The 'TP' column shows the correctly identified cases by SpotJMHBugs and '# Benchs' shows the number of benchmarks that are potentially affected by the bad JMH practices.

| Bad JMH Practice | Identified | FP | TP | # Benchs |
|---|---|---|---|---|
| RETU | 54 | 11 | 43 | 43 |
| LOOP | 29 | 4 | 25 | 25 |
| FINAL | 9 | 2 | 7 | 7 |
| INVO | 18 | 2 | 16 | 25 |
| FORK | 2 | 0 | 2 | 5 |
| **Total** | **112** | **19** | **93** | **105** |

*RETU*: 11 of 54 instances of RETU were considered false positives after manual inspection. In those cases, developers inserted the variable inside a conditional check, throwing an exception in case of an unexpected value. This can be done by a call to an assert method, or through an explicit `if` clause followed by throwing an exception. This is a valid strategy for checking the value of a variable and preventing dead code elimination, and is currently not considered by our SpotJMHBugs tool.

*LOOP*: In 5 of 29 cases, the accumulation of a numeric variable in a benchmark loop was considered a false positive after manual inspection. In these cases, the accumulation was an integral part of the benchmark, and not only used to consume the return of a method call.

*FINAL*: From the initial set of 9 FINAL cases, 2 were considered a false positive. The final primitives were used in a method unrelated to benchmarks.

*INVO*: From the initially reported 18 INVO bad practices, 2 could not be removed or reduced with our methodology. Both scenarios are comprised of fixture methods that are required to execute on every invocation, and run for too long (longer than 1 ms) to be included in the benchmark without offsetting the measurements. These constitute correct configuration of invocation level fixture methods, and are hence false positives.

*FORK*: We found no false positives in the two instances of FORK reported by SpotJMHBugs.

### 3.6.4 Results

In this section, we describe the methodology used to generate the fixed version and the results of the impact analysis for each of the studied bad JMH practices. For each bad practice, we also detail a specific case where removing the bad JMH practice had a significant impact on the performance counters of the benchmarks.

Table 3.8: The number of benchmarks that were impacted by the bad JMH practices which were removed through non-intrusive fixes. In the 'Effect Size' column we categorize the observed impact after removing the bad JMH practice in small (S), medium (M) and large (L) effect sizes, according to Cliff's delta.

| Bad JMH Practice | Project | Benchmarks | | Effect Size | | |
|---|---|---|---|---|---|---|
| | | Impacted | % | S | M | L |
| RETU | netty | 1/1 | 100.0 | | | 1 |
| | gs-collections | 9/37 | 24.3 | | 1 | 8 |
| | logging-log4j2 | 5/5 | 100.0 | 1 | | 4 |
| | **Total** | **15/43** | **34.8** | **1** | **1** | **13** |
| LOOP | netty | 10/10 | 100.0 | | | 10 |
| | druid | 6/8 | 75.0 | | 1 | 5 |
| | logging-log4j2 | 7/7 | 100.0 | 1 | 2 | 4 |
| | **Total** | **23/25** | **92.0** | **1** | **3** | **19** |
| FINAL | netty | 1/2 | 50.0 | | | 1 |
| | druid | – | – | | | |
| | logging-log4j2 | 4/5 | 80.0 | 4 | | |
| | **Total** | **5/7** | **57.1** | **4** | **0** | **1** |
| INVO | netty | 12/12 | 100.0 | | 1 | 11 |
| | druid | 2/3 | 66.7 | | | 2 |
| | h2o-3 | 6/6 | 100.0 | 2 | 2 | 2 |
| | **Total** | **20/21** | **95.2** | **2** | **3** | **15** |
| FORK | pgdbc | 5/5 | 100.0 | | | 5 |

Table 3.8 shows the analysis of the impact of bad JMH practices for all non-intrusive fixes, and Table 3.9 shows the result of removing the INVO bad practices through the intrusive fix. In both tables we show how often a benchmark was impacted (the 'Impacted' column) by removing the bad JMH practice and the 'Effect Size' of the observed differences compared to the original version.

Table 3.9: Benchmarks impacted by the INVO bad practice, fixed by adding the fixture code inside the benchmark (intrusive fix).

| Bad JMH Practice | Project | Benchmarks Impacted | % | Effect Size S | M | L |
|---|---|---|---|---|---|---|
| INVO | netty | 3/4 | 75.0 | | | 3 |

### Impact of RETU

*Fix Patches.* We can ensure that unconsumed variables will not be eliminated by the JIT in two ways: (1) we can return a variable at the end of the benchmark method, or (2) we can call `Blackhole.consume()` to consume the variables. We apply the first patch to every void benchmark with a single unconsumed variable, as we consider this solution more elegant (which is relevant as we contribute a subset of fixes to the projects, as discussed in Section 3.7). In all other cases, we consume the variable using a Blackhole object.

*Results.* As shown in Table 3.8, we evaluate 43 benchmarks containing the bad JMH practice. In our results, 15 of 43 benchmarks (34%) were impacted by fixing the RETU bad practices. For 13 benchmarks the difference has a large effect size. The differences for the other two impacted benchmarks have smaller effect sizes.

Listing 3.6: Source-code of a benchmark affected by the RETU bad practice in the `logging-log4j2` project.

```
@Benchmark
public void baseline() {
   consume(bytes); // return is not used
}

private static long consume(final byte[] bytes) {
   long checksum = 0;
   for (final byte b : bytes) {
      checksum += b;
   }
   return checksum;
}
```

The highest impact after fixing a RETU bad practice came from a baseline benchmark of `logging-log4j2` presented in Listing 3.6. Developers called a `static`

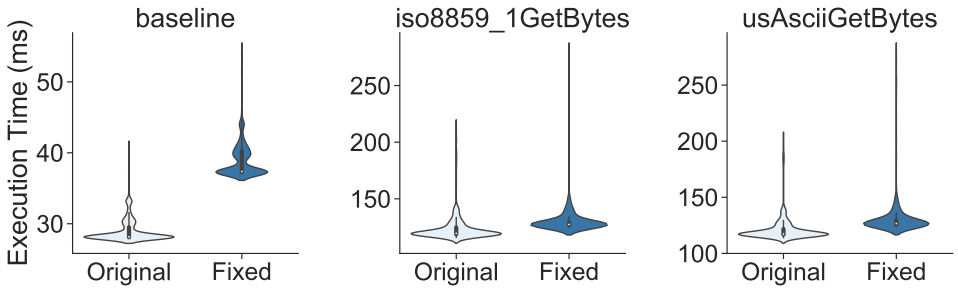



Figure 3.4: Performance counters before and after fixing a RETU bad JMH practice in 3 benchmarks in the `AbstractStringLayoutEncoding` class from `logging-log4j2`. The effect size is large for the baseline benchmark, and medium and small for the other 2 benchmarks.

self-implemented consume method, to consume the computation, but ignored the method return. After changing the code to consume the computation, the resulting performance counters showed that the execution time increased by 32% (see Figure 3.4), which indicates that the developers originally failed to appropriately take JIT optimization into account. In addition, other benchmarks from the same class (e.g., `usAsciiGetBytes`) were also impacted by our fix, although with smaller effect sizes.

**Impact of RETU bad JMH practice**

35% of the benchmarks containing the RETU bad practice were significantly impacted by the bad practice. In 30% of the cases, the impact had a large effect size.

### Impact of LOOP

*Fix Patches.* We refactor the accumulation inside a loop into a `Blackhole.consume` method call.

*Results.* Table 3.8 shows how often an instance of the LOOP bad JMH practice impacted a benchmark. 23 of 25 benchmarks (92%) had their performance counters impacted by fixing this bad JMH practicea. For 19 of these 23 impacted benchmarks, the difference in execution time before and after removing the LOOP bad practice has a large effect size.

Listing 3.7: Source-code of a benchmark in `druid` project affected by the LOOP bad practice.

```
@Benchmark
public void readContinuous(Blackhole bh) {
   ColumnarLongs columnarLongs = supplier.get();
   int count = columnarLongs.size();
   long sum = 0;
   for (int i = 0; i < count; i++) {
      sum += columnarLongs.get(i);
   }
   bh.consume(sum);
   columnarLongs.close();
}
```

To illustrate, we present a benchmark class from the `druid` project, where we observed the largest effect sizes in our experiment. `LongCompressionBenchmark` has two benchmarks defined: the first benchmark reads sequentially from an array (see Listing 3.7) while the second randomly skips array positions. The first example can have its loop unrolled and the operations merged by JIT, artificially speeding-up the benchmark with a median of 22%, and up to 9 times in one extreme case. We present the impact of our fix of the sequential read benchmark in Figure 3.5 for the execution with all 20 benchmark parameters. The second benchmark cannot easily be optimized by JIT, and was not impacted by our fix (see our online appendix [45]).

**Impact of the LOOP bad JMH practice**

23 out of 25 benchmarks containing the LOOP bad JMH practice were impacted by the bad practice. For 19 benchmarks the impact had a large effect size.

### Impact of FINAL

*Fix Patches.* To fix FINAL bad JMH practices, we simply removed the `final` modifier of primitive variables.

*Results.* As shown in Table 3.8, we found that 5 of 7 (71%) benchmarks had their performance counters impacted by fixing the FINAL bad JMH practice. The effect size of the impact was large for one benchmark, and small in the other four cases.

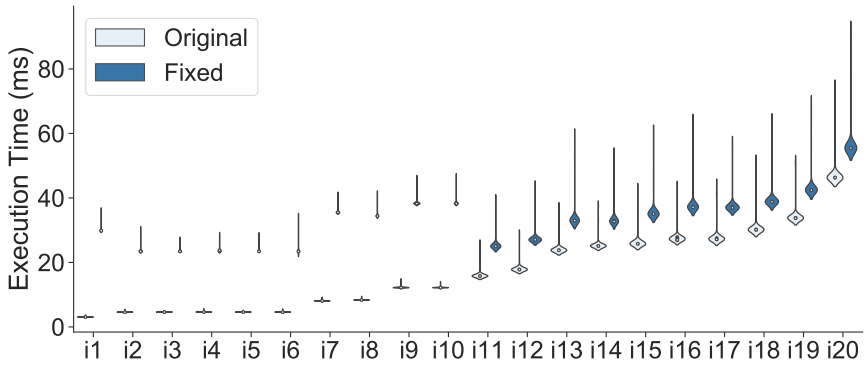



Figure 3.5: Performance counters before and after the fix of an instance of the LOOP bad practice in a `druid` benchmark. The effect size of the differences is large for all 20 benchmark parameters (i1 to i20).

The four benchmarks with small effect size measure the execution time of logging an event in the `logging-log4j2` project. Developers used a final boolean variable to check whether to perform further logging operations as depicted in Listing 3.8. JVM can optimize the first check of the `if`-clause away. The impact of such an optimization is statistically noticeable, and may be relevant in some practical use cases, but the absolute impact of the bad FINAL bad JMH practice is considerably lower than for the previously discussed bad JMH practices (see Figure 3.6).

**Impact of FINAL bad JMH practice**

5 out of 7 benchmarks were impacted by the FINAL bad practice, but the impact typically has only a small effect size.

Listing 3.8: Source-code of a benchmark affected by FINAL bad practice in the `logging-log4j2` project.

```
private final additive = true;
//Method called by a benchmark
private void logParent(final LogEvent event) {
   if (additive && parent != null) {
      parent.log(event);
   }
}
```

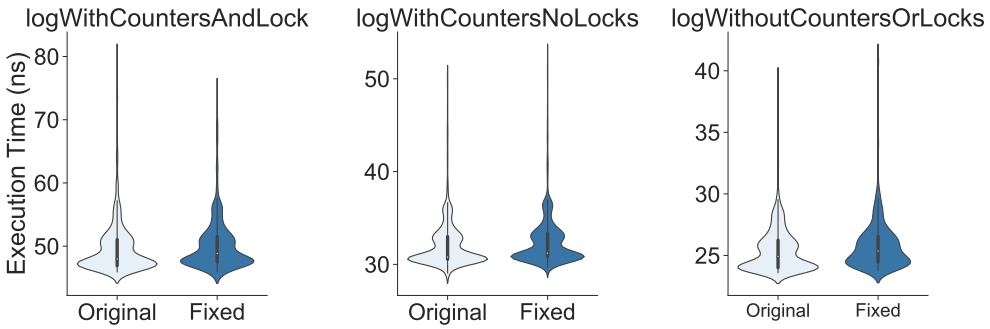



Figure 3.6: Performance counters before and after the fix a FINAL bad practice on 3 benchmarks in `logging-log4j2`. The effect size is small for all 3 benchmarks.

### Impact of INVO

*Fix Patches.* The process of fixing the INVO bad JMH practice requires both the analysis of the benchmark code and a preliminary set of experiments. The JMH documentation [142] explicitly mentions the requirement of an ad-hoc evaluation of the invocation level usage.

We first determined which objects need to be created or cleaned up on every invocation. Every object that does not require this was moved to a fixture method with `Level.Iteration`. Then, we ran a set of preliminary experiments to identify fixture methods that run in less than one millisecond, which we define as short-running. In such cases, we moved the fixture method code into the relevant benchmark method, as suggested by the JMH documentation. We made sure to never refactor class fields into local variables to prevent the JVM from performing further optimizations. The JVM can identify if a local variable is accessed in a restricted scope through a static analysis technique called Escape Analysis [18] and avoid allocating the object in the heap through Scalar Replacement [].

SpotJMHBugs reported 16 INVO instances, which affected 25 benchmarks in total. In 21 benchmarks, the invocation level fixture could be removed or reduced without adding code to the benchmark method (non-intrusive fix). In the remaining 4 benchmarks, we added the setup/teardown code inside the benchmark (intrusive fix).

*Results of the non-intrusive fix.* Table 3.8 shows the impact analysis for instances of the INVO bad practice that could be removed without changing the benchmark

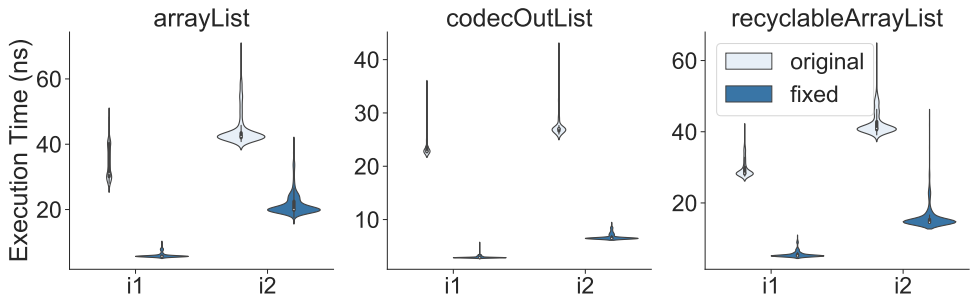



Figure 3.7: Performance counters before and after the non-intrusive fix for the INVO bad practice on 3 benchmarks from `CodecOutputListBenchmark` class. The effect size is large for all benchmarks, each evaluated with 2 sets of parameters (i1 and i2).

code. In our evaluation, 20 of 21 (95%) benchmarks had their performance counters impacted by the INVO bad practice, and in 15 benchmarks the impact had a large effect size.

For instance, the `CodecOutputListBenchmark` class from the `netty` project had its fixture methods unnecessarily configured to invocation level. The created objects were not modified during the benchmark execution and could instead be instantiated on every iteration. After our fix, every benchmark executed on average three times faster (see Figure 3.7).

*Results of the intrusive fix.* As shown in Table 3.9, 3 of 4 benchmarks had significantly faster performance counters after moving the setup code inside of the benchmark. This speedup indicates that the JMH overhead was considerably higher than the time spent in the setup phase.

Figure 3.8 shows the comparison of the performance counters in three of the benchmarks. The benchmarks were up to 20% faster after our fix. More importantly, such benchmarks were defined in a single class `HeadersBenchmark`, which contained nine other benchmarks that were indirectly affected by the invocation level fixture. Therefore, our fix eliminated the JMH invocation overhead from the remaining nine benchmarks as well.

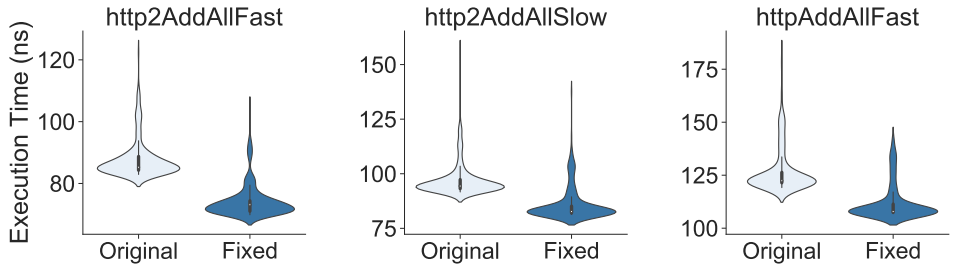



Figure 3.8: Performance counters before and after the fix for 3 benchmarks that had their INVO bad practice removed through an intrusive fix.

**Impact of INVO bad JMH practice**

23 out of the 25 benchmarks that used the INVO bad practice were misconfigured which impacted their performance counters. Including the setup code in the benchmark itself, actually accelerated benchmark execution in 3 of 4 cases.

### Impact of FORK

*Fix Patches.* Benchmarks configured with zero forks can be reconfigured by modifying the annotation `@Fork` or by overriding the fork parameter of JMH directly when starting the benchmark through the command line. We opt for the first approach in our study (as it allows us to fix the bad practice directly in the source-code).

*Results.* We evaluated two instances of the FORK bad JMH practice, which configured the fork parameter for five benchmarks (see Table 3.8). All five benchmarks were impacted by the FORK bad JMH practice. In all cases, the difference in execution time after removing the FORK bad practice had a large effect size.

In our evaluation, the `AddingPaddingZeroes` class from `pgjdbc` showed the highest impact after our fix, with differences of up to three times (see Figure 3.9).

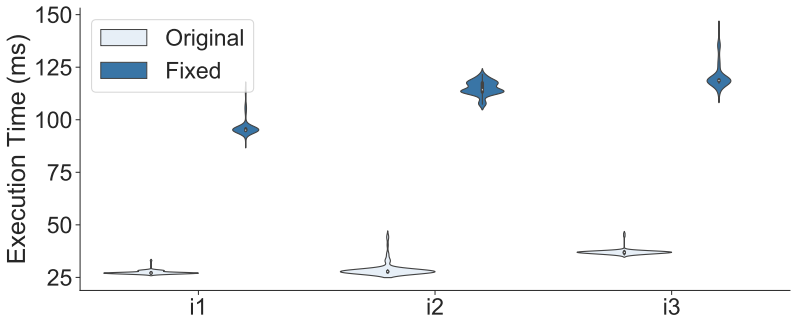



Figure 3.9: Performance counters before and after fixing the FORK bad practice in the `AddingPaddingZeroes` benchmark of the `pgjdbc` project. The impact of removing the FORK bad practice had a large effect size in all three instances.

**Impact of FORK bad JMH practice**

All five benchmarks configured with zero forks had their performance counters largely impacted by removing the FORK bad practice.

## 3.7 Submitting Pull Requests

After evaluating the impact of bad JMH practices, we submitted a number of pull requests to the maintainers of a subset of the studied projects. In this study, the purpose of submitting the pull requests are twofold: (1) To validate our bad JMH practice detection and impact assessment with the benchmark creators, and (2) to contribute to open-source projects by fixing potentially misleading benchmark implementations.

We restricted our pull requests to the cases where our analysis has shown that the effect size of removing the bad JMH practice was large. This limited our efforts to benchmarks where misleading results could have consequences on the project's performance. We also refrained from submitting a pull request for one benchmark from the `pgjdbc` project, because this benchmark did not evaluate project code, but only compared the performance of bitwise operations instead. Table 3.10 shows a description of the pull requests that we submitted to each project, per bad JMH

practice. In total, our pull requests applied changes to 57 benchmarks across 6 open-source projects.

Table 3.10: Pull requests submitted to developers of studied open-source projects. Column Ref links the Github pull-request and Issue description, while column Status shows how developers received our patch.

| Ref | Project | Bad JMH Practice | # of Benchmarks | Status |
|---|---|---|---|---|
| [36] | gs-collections | RETU | 23 | Accepted |
| [38] | logging-log4j2 | RETU | 5 | Accepted |
| [41] | druid | LOOP | 4 | Accepted |
| [40] | druid | INVO | 2 | Accepted |
| [39] | h2o-3 | INVO | 2 | Accepted |
| [35] | netty | INVO | 16 | Accepted |
| [37] | pgjdbc | FORK | 5 | Rejected |
| | | **Total** | **57** | |

*Accepted pull requests.* 6 of 7 issued pull requests were well received and accepted by developers and maintainers of the studied projects. In these pull requests, developers agreed on merging the recommended patch into the main branch. In one case, a developer mentioned having previously identified such unsafe loops, but never had the chance to fix it [41]. In another case [40], the developers agreed with the fix but asked to remove the benchmark altogether instead since it was not in use anymore. The `gs-collections` project has the largest benchmark suite, with almost every class making a good use of Blackhole and variable sinking options. Still, we found 23 cases of the harmful RETU bad JMH practice, fixed by our patch [36].

*Rejected pull requests.* The only rejected pull request was the patch that reconfigured the `@Fork` parameter from zero to one. The developers acknowledged that configuring a benchmark with zero forks could lead to misleading results, but justify that such configuration was only used to debug the benchmark in the IDE. Furthermore, the pull request was not accepted and merged into the `pgjdbc` project for two reasons: (1) The benchmarks were not part of the continuous integration process and are executed on a case-by-case basis, thus would not impact the qual-

ity of the main product and (2) the developers intend to remove JMH from the project due to licensing issues.

## 3.8 Discussion

We now discuss the implications of our results for software developers and researchers, as well as threats to the validity of our study.

### 3.8.1 Implications

Our results show that existing JMH benchmarks even in prominent open source software systems contain bad practices that impact the benchmark results.

Although well-documented in research [62] as well as in the JMH documentation, many open source developers still appear to be struggling to correctly account for the many intricacies of benchmarking Java applications. The fact that our pull requests containing fixed benchmarks have been merged back in 6 of 7 cases indicates that developers generally care about the bad JMH practices we have presented (i.e., they appear to not be cases of conscious trade-offs), but often fail to avoid them in practice. This should be addressed in the following orthogonal ways:

Improve developer training and documentation Based on our results, we speculate that the information that developers currently have is not effective in guiding them towards rigorous benchmarking solutions. One reason may be that without detailed knowledge of the inner workings of the JVM and just-in-time compilation, many bad JMH practices may appear obscure and unimportant to developers. It is possible that developers are aware that their code contains bad JMH practices according to the documentation, but fail to see how these bad practices impact their own projects. Improving this understanding is difficult, but may require more explicitly and extensively discussing the effects of bad practices (for example in the JMH documentation) rather than only listing what they are. However, ultimately, better developer training with regards to performance engineering for Java applications will be required.

Improve developer tooling In addition to training, better tooling should be provided to help developers avoiding bad JMH practices. Our own tool, SpotJMHBugs, is a good starting point. As a SpotBugs plugin, SpotJMHBugs is easy to integrate into standard IDEs and can be used already during development to point out bad JMH practices. Another angle may be to extend JMH itself to, for instance, analyze the configuration input and benchmark code before the execution of the benchmarks. Through this analysis, JMH could produce warning messages when it discovers configurations or benchmark code that appears to contain bad JMH practices (similar to what it already does for the FORK bad practice). A significant advantage of this approach is that at runtime, JMH has access to a much richer set of metrics than our static analysis approach to determine whether any given execution is likely to lead to trustworthy benchmark results. For instance, JMH knows how many warmup iterations, benchmark iterations, and forks are actually being executed, and can use statistical power analysis to evaluate if this configuration is trustworthy given the benchmark value dispersion it observes.

Studying approaches for automatic benchmark repair In our work, we have manually fixed a number of instances of bad JMH practices. However, in doing so, it has become evident that for a subset of bad practices, (e.g., LOOP, FINAL and RETU) fixes actually only require a fairly static and simple transformation of the code. Future studies should investigate automatic benchmark repair, which could help to fix bad JMH practice instances without direct developer involvement. Such a tool would have a large positive impact on performance testing practices, as it (1) could be employed to provide widespread fixes of the many instances of bad JMH practices we have identified in our study, and (2) act as another tool for developer training. That is, such an automatic benchmark repair tool could educate developers in how a rigorous implementation of their benchmark would look.

### 3.8.2 Threats to Validity

The *external validity* concerns the generalizability of our work. One threat is that we selected open source projects from GitHub only. Future studies are necessary to

identify the frequency and impact of bad JMH practices in other types of projects, such as industrial projects.

In addition, our study focused on JMH benchmarks. While JMH has become one of the most popular benchmarking frameworks for JVM-based languages (such as Java, Scala and Clojure), future studies are necessary to investigate whether our findings hold for other benchmarking frameworks in Java and other programming languages. In addition, future studies should investigate the impact of these bad practices when using non-default JIT compilers, such as the Graal compiler.

The *internal validity* concerns the confidence that we have in our findings. One threat is that we consider the fixes that are suggested by the JMH documentation as the current 'industry-standard' for addressing bad JMH practices. It is possible that these JMH-proposed fixes themselves impact the performance of the benchmark. Regardless, our tool can detect the bad JMH practices; if in the future a different 'industry-standard' fix arises, future studies should re-evaluate the impact of the bad JMH practices that our tool can detect.

Another threat is that we use the thresholds that were proposed by Romano et al. [155] for Cliff's Delta effect size to quantify the impact of a bad JMH practice on the benchmark results. Some projects may require different thresholds to meet their performance requirements. Future studies should investigate these differences in performance requirements.

Because our detection rules are heuristic-based, manually verified instances of bad practices investigated in this study could represent only a subset of all existing cases in the studied projects. If a bad practice manifests itself in a different way our tool will not detect it. Future studies should investigate how our detection rules can be extended and improved.

The *construct validity* concerns the construction of our experiments. One threat is that our tool for identifying bad JMH practices is based on heuristics, and is therefore inherently susceptible to false positives. The major challenge is that developer intent is an important factor in deciding whether an identified instance of a bad JMH practice is indeed a bad practice. For example, a developer might actually want to benchmark the impact of JVM optimization on variable accumulation inside a loop. Hence, our tool should be used as a guideline by developers to identify potential instances of bad JMH practices.

In addition, our rules do not cover all possible cases of undesired JVM optimizations. JVM also uses runtime analysis to perform optimizations, e.g., it might eliminate dead code after inlining a method call during the benchmark execution. Because our tool uses static analysis only, it cannot detect instances of bad JMH practices that depend on runtime information.

## 3.9 Summary of the Chapter

In this chapter, we studied bad practices of writing benchmarks using the JMH framework. We presented five bad JMH practices related to not consuming products of computation, using a loop to accumulate computations, using final primitives that are prone to constant folding, incorrect usage of test fixtures, and incorrect usage of JMH forks.

We showed that:

- **The studied bad JMH practices are indeed prevalent in Java-based open source systems.** Half of the projects with more than 10 benchmarks tests contain bad JMH practices in their code base.
- **Bad JMH practices are indeed often severely impacting the outcome of benchmarks** as they lead to benchmark results that substantially deviate from the correct measurements.

We submitted pull requests containing fixed benchmarks to developers of impacted open source projects to validate whether developers agreed with our assessment and analysis. Six of seven submitted pull requests were accepted and merged quickly (one was rejected as the developers plan to remove JMH from the project due to licensing issues), indicating that **developers confirmed the identified issues** after being presented with the results of our study.

Our study results indicate that many open source developers still struggle to account for the many intricacies of benchmarking Java applications. Consequently, we suggest that we need (besides improving developer training as well as the JMH documentation) better tooling to guide developers towards rigorous benchmark implementations. As part of our study, we have developed the SpotJMHBugs,

a static analysis tool that can be used to **assist developers on automatically identifying bad JMH practices during benchmark creation**. As a plugin to SpotBugs, SpotJMHBugs is easy to integrate into standard Java IDEs. However, even more important may be to improve JMH itself. Such improvement could for example consist in generating warning messages when JMH discovers configurations or benchmark code that appears to contain bad JMH practices. Further, we suggest that our work can be used as a starting point for studying approaches for automatic benchmark repair.

# 4 Investigating Collections Usage and Performance

The selection of data structures is the bread and butter of efficient software development. Java has a particularly rich ecosystem of data structure variants, with variants provided by the standard library (JCF) with general-purposed implementations to alternative and more focused collection libraries, giving practitioners a large pool of options to draft their preferred data structures. In this chapter, we investigate how practitioners select their collections in the context of open-source Java programs and how alternative and less-commonly used implementations can be selected to improve both the execution time and the memory usage of applications.

**Contributions.** Hence, in this chapter, we make the following contributions:

I An empirical analysis of the popularity and usage patterns of collection libraries based on mining a large dataset of open-source Java projects.

II A framework for systematic evaluation of collection's performance.

III An experimental evaluation of the performance of JCF and six major alternative libraries in terms of execution and memory under a variety of scenarios.

IV A guideline to practitioners on replacing standard JCF collections based on performance characteristics.

**References** This chapter is partially based on a peer-reviewed publication [43] and the following manuscript in preparation.

D. Costa, E. Schubert, A. Andrzejak, and D. Lo. "Beyond the Java Collections Framework: An Empirical Study of Usage and Performance of Java Collection Libraries and APIs"

## 4.1 Introduction & Motivation

Collections are prominent on current software development and modern programs instantiate such data structures in thousands of program locations [158]. Hence, selecting an appropriate abstraction and variant is a crucial aspect of developing efficient applications. Choosing an inappropriate collection variant may result in *performance bloat*, the excessive use of memory and computational time for accomplishing simple tasks.

Numerous studies have identified the inappropriate use of collections as the main cause of performance bloat [67, 125, 179, 180]. In production systems, the memory overhead of individual collections can be as high as 90% [125], and Chris et al. [30] have concluded that collection usage is related to 7 out of 11 patterns of memory inefficiencies.

Experienced developers also struggle in selecting their data structures on large softwares. A study from researchers at Google [117] found many instances where expert developers have selected inefficient collection variants, or misconfigured collections initial capacity on Google's applications. To exemplify the impact of collections on the performance of real applications, we present in Table 4.1 examples of real applications that suffered significant performance bloat due to collection misconfiguration and/or selection. All cases were fixed by changing a handful of code lines, indicating that the collection selection is a problem hard to identify, with significant performance impact, but can be ultimately easy to fix by developers.

In the case of Java development, the problem of selecting an appropriate collection becomes more complex when accounting for third party libraries. These allow for many more choices compared to the standard Java Collection Framework (JCF), from simple alternatives to existing JCF implementations to collections with extra features such as immutability and primitive-type support.

Despite its importance, we found a gap in experimental studies comparing execution and memory performance of non-JCF collections. Partial benchmarks, especially on the websites of the libraries, are common enough; they, however, do not give a performance comparison for different libraries, nor do they provide an evaluation under a large set of scenarios. The work presented in this chapter at-

Table 4.1: Examples of the inefficient selection as the main cause of performance bloat. We present here one example per study, where changing a few lines of code could impact the application's execution time, memory consumption and energy consumption.

| Study | Application | Metric | Impact |
|---|---|---|---|
| [117] | Anonymous | Time | 17% of application higher execution time due to bad configuration of one `HashMap` allocation-site. |
| [158] | TVLA | Memory | 54% of application higher memory consumption for using `HashMap` instead of `ArrayMap` on instances holding few elements. |
| [80] | Google JSON | Energy | 300% of higher energy consumption when using `LinkedList` instead of `ArrayList`. |

tempts to fill this gap in experimental studies and derive a set of guidelines for developers on how can they improve performance with little code refactoring. To this aim we:

- Study the usage of collections in real Java code via repository mining
- Evaluate the memory consumption and execution performance of collection classes offered by six most popular collection libraries.

The key question to be answered by our study is: **"Can we improve performance of applications in typical scenarios by simply replacing collection implementations, and if yes, to which degree?"**. In particular, we explore alternatives to JCF implementations under the same collection abstraction

## 4.2 Related Work

The experimental evaluation of data structure's performance has been constantly explored by academic and non-academic studies. Due to the emerging concern on limiting energy consumption of applications, the energy efficiency of data structures - and its relation to time and memory consumption - has attracted a lot of attention in the past decade. Hunt et al. [84] evaluated the time performance and its relation to energy efficiency of lock-free data structures. The study results point

to a strong positive correlation of performance and energy consumption, showing that data structures that run for longer also consume more energy.

Hasam et al. [80] and Pereira et al. [147] both presented an experimental work that extracted an energy profile for the most commonly used Java collections via a series of benchmarks. These energy profiles were further used to guide the replacement of collections to evaluate its impact on the overall energy consumption of their subject applications. The result of this analysis showed that collections have a significant role on the energy efficiency of applications, particularly in Hasan experiments, showing that choosing the wrong collection can cost up to 300% more energy consumption on a specific application.

Furthermore, some studies focus on a specific category of collections, Pinto et al. [148] presented a study evaluating the energy efficiency of thread-safe collection variants, and Saborido et al. [156] evaluated the time, memory and energy performance focused solely on map variants in Android.

Prior to our work, we found a surprising lack of academic studies focused on non-standard libraries, and how such alternative collection variants can be used to provide new choices for developers or replace standard implementations. Benchmarks provided by such alternative libraries, or that focus on a specific scenario and/or category of collection are common enough in the web. Lewis [114] compares JCF implementations and proposes a new customized collections for a specific graphical user interface (GUI) usage scenario. Another article [113] exemplifies the time and memory trade-offs of `HashMap` variants through a series of benchmarks, confirming that hash-tables with higher memory footprint are considerably faster than memory-efficient counterparts.

The article entitled “Large HashMap Overview” [172] is the closest study to provide an overview of the performance of variants from alternative libraries. The article focus on `HashMap` variants only, provided by five alternative libraries and the JCF implementation, evaluated under three usage scenarios. Results presented in this article give a glimpse of the advantages alternative libraries might bring to Java applications, but the focus on a single collection type limits the applicability and leaves a gap for further studies.

Similar to the above mentioned studies, we also evaluate different collection variants under a variety of usage scenarios. The focus and scope of our study,

however, differs in two significant ways from past studies. First, we perform a preliminary study to understand how collections are used in a large Java corpus, and use this information to based our experimental plan. Second, we explore the performance of alternative libraries and how non-standard variants can be used to reduce memory usage and improve time performance.

## 4.3 Analysis of Collections Usage

We begin our study by investigating the usage patterns of collections found in a large Java code corpus. Understanding such patterns will help us identify the most popular collection types and implementations, how developers instantiate their collection instances, and what are the most commonly help element type. These patterns of collection usage will serve as a basis of our experimental study on collections performance, described in Section 4.4.3.

### 4.3.1 Data and Static Analysis

We use the GitHub Java Corpus [7], a dataset containing open-source Java projects from GitHub, to perform our analysis on patterns of collections usage. The GitHub Java Corpus has 10,986 Java projects, totalizing more than 268 millions lines of code, and consists only of projects with more than one fork to eliminate small or inactive projects, which prevail at GitHub [101]. In addition, the creators of GitHub Java Corpus analyzed and pruned this dataset manually in order to eliminate project clones. The resulting curated corpus contain projects like Eclipse IDE [55], Clojure [72], RxJava [153] and Elastic Search [56], all popular and mature open-source Java projects

We need to extract from the code every declaration of a collection and every site of a collection instantiation. This is needed to count the usages of collection classes, record the types of held elements, and count how often the initial capacity is specified. To this aim, we develop a project called CollectionsExplorer [1], a static analyzer that uses JavaParser [95] to extract each variable declaration and

[1] Available at https://github.com/DiegoEliasCosta/collections-explorer

instantiation site in the code. We then filter the collection instances by applying a heuristic as defined by the following regular expression:

.*List|.*Map|.*Set|.*Queue|.*Vector

This pattern finds all collection implementations of interest but also retrieves some false positives, e.g., `java.util.BitSet` is not a general purpose collection but is retrieved by our heuristic regardless. We rank the retrieved types and manually inspect and filter out false positive for the top 99% of the retrieved data, ranked by occurrence.

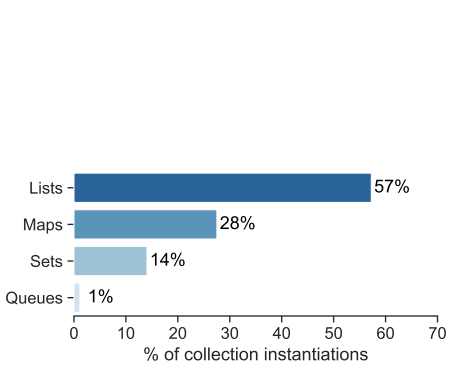
### 4.3.2 Collections Usage in Real Code

In the following, we present the methodology of classification and our results on the usage patterns of collection in Java projects. We focus on the aspects of the most commonly selected collection and held element types, and how often does developers tune their collection creation.

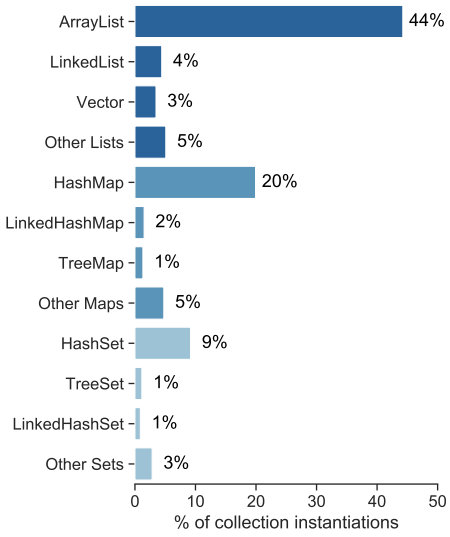
#### What are the Most Commonly Selected Collection Types?

We analyze the instantiations sites and extract the most used collections in the code (see Figure 4.1). Unsurprisingly, list is the most commonly used collection abstraction 57%, followed by map 28%, and finally by set 14%. Contrary to this, queues are rather rare (1% of usages). Our ranking is dominated by JCF as developers only rarely opted for others, not nearly often enough to make it into our charts.

Among lists, ArrayList makes up the bulk of all collections usages, namely 44%. The LinkedList is also rather common, with 4% and is ranked as the fourth most common implementation. In maps, HashMap is most commonly used, representing 20% of all collections instantiation, followed by LinkedHashMap (2%), and the TreeMap (1%). Set instances follow a similar pattern to maps: HashSet is most used (9%), followed by TreeSet and LinkedHashSet with both 1% of occurrences. We found a similar distribution in *Top50*, which have a slightly higher usage of concurrent collections and a lower variety of types.




(a) Collection abstractions.




(b) Collection types.

Figure 4.1: Most selected collection abstraction and collection types in 10,796 Java projects. All top collection types are from JCF as alternative variants are only rarely used.

In summary, the top four most frequently used collections are the JCF implementations of ArrayList, HashMap, HashSet and LinkedList and together they account for approximately 77% of all collections declared in our dataset.

### What are the Most Commonly Held Element Types?

The element type is the type of objects held by collections instances. Understanding the element type distribution can help us design benchmarks that closely resembles the most frequent real scenarios. To simplify, we group the element types into four categories:

- **Strings**: Objects such as `String`, `String[]`, and `String[][]`.
- **Numeric**: The primitive-wrappers such as `Double`, `Float`, `Long`, `Short`, `Integer`, as well as `Boolean`, `Character`, and their respective arrays.
- **Collections**: Collection types that can be identified by our heuristic from Section 4.3.1.

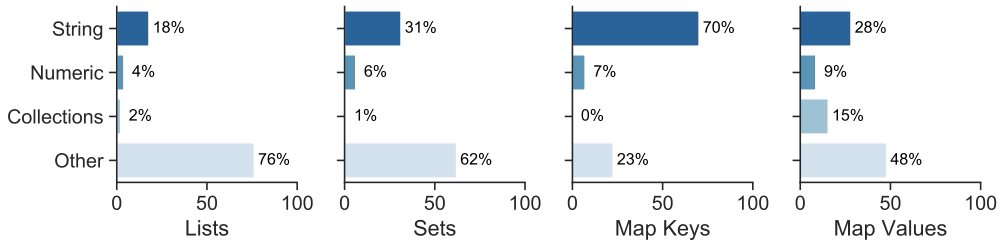



Figure 4.2: Distribution of element types held by lists and set declarations in our dataset.

- **Other**: All the classes and data types not fitting any of the mentioned categories.

We present the result of our analysis in Figure 4.2. A pattern that emerges is the similar usage of primitive wrappers throughout the categories, always ranging from ≈5% to ≈8% of the declarations. This category is particularly interesting because it contains the collections that can be replaced by primitive collections with simple code refactoring.

The remaining categories show a distinct pattern for each of lists, maps and sets. Lists have the highest variability of element types, but hold strings 20% of the time. Sets hold strings more often than lists, 31% of the time. Maps also hold strings very often: 70% of all declared maps use the String as a key and 28% as a value. Map values are often a collection (15%) pointing to a common usage of maps as a collection holder. This is an interesting finding as alternative collection variants called multi-maps are specifically tailored for use-case, offering a richer API than the general purposed maps.

In summary, Strings are the most frequently used category of element type in collections. Maps use String as key in 70% of declared instances, and have their values commonly mapped to collection types (15%). List and sets typically hold a higher variety of element types.

#### How Often do Developers Specify the Initial Capacity of Collections?

Defining an appropriate initial capacity of a collection is a simple but effective method for optimizing runtime and memory. We sample 400 instantiation sites of ArrayList, HashSet, and HashMap, which give us 5% confidence interval on

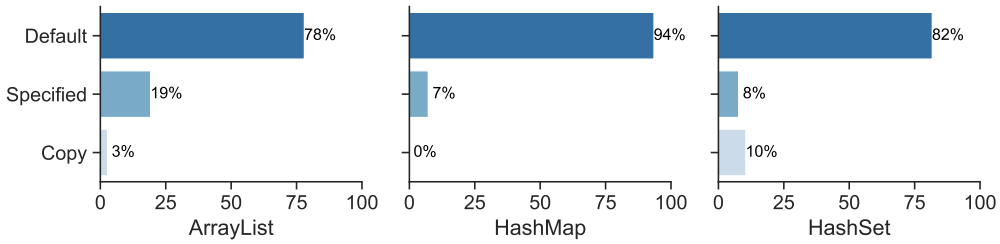



Figure 4.3: Statistics on the style of specifying initial capacity of collections. "Specified" denotes that the developer explicitly set the initial capacity, and "Copy" that a copy constructor was used.

our results. Then we manually categorize whether a developer specifies an initial capacity, copies this instance from another collection, or uses default constructor values.

Figure 4.3 shows that programmers specify initial capacities for ArrayList only in 19% of the declarations. HashMap and HashSet have their capacity specified in 7% and 8% of collection declarations, respectively. Interestingly, HashSet is the category more commonly created through copy instruction (in 10% of cases).

## 4.4 Experimental Design

In this section we describe the design of experiments for evaluating performance of collection implementations in terms of execution time and memory usage. We first identify a suitable set of collection libraries for this evaluation through ranking of libraries in terms of their popularity (Section 4.4.1). Then in Section 4.4.2 we describe a benchmark framework capable of measuring the steady-state performance of collections with high precision. Finally, we design an experimental plan which covers a large set of usage scenarios based on the findings of usage patterns reported in Section 4.4.3. The results of our rigorous statistical analysis of each evaluated scenario are presented then in Section 4.5.

### 4.4.1 Selection of Collection Libraries

As one of the most popular programming languages, Java has a rich ecosystem of third-party libraries. We initially searched the Web for alternative collection libraries implemented in Java and find a total of 14 libraries. All found libraries had stable releases, with more than a year of development time and could be used in most open-source Java projects. Our goal with this study is not to evaluate all available open-source collection libraries, but to provide a glimpse of performance opportunities some alternatives might yield to anaware developers. Therefore, we first select a subset of collection libraries using two different approaches. First, we rank them by popularity of their project in GitHub, see Table 4.2. Second, we analyze how many times they have been included in existing benchmarks (Table 4.3).

The GitHub metrics such as number of stars and number of watches provide a simple but effective method for ranking software projects. Such metrics have also been used as a criterion in other studies [130], and are a good indication of the quality of the project [152].

To account for libraries that are not on GitHub, we retrieve a set of collection benchmarks from the web, and count the occurrence of libraries in collections benchmarks. With this criterion, we expect to capture libraries that have drawn the attention of the performance engineering community, for instance, by providing variants with better performance than standard collections.

Furthermore, we filter libraries that do not provide implementations that can serve as a replacement to JCF collections. This excludes Javaslang as it provides only immutable collections for lambda function usage. In the final step we select the top five libraries from each ranking and merge them into a single list. This yields seven unique libraries (three of them occur in both rankings) to be included in our experimental evaluation.

### 4.4.2 Benchmark Design

In Section 2.3 we describe the plethora of factors that pose a challenge for performance measurement in a managed runtime environment such as the Java Virtual Machine. To reduce the impact of these factors we follow the suggestions by

Table 4.2: Ranking of collection libraries by GitHub popularity. We select the top 5 collections from this ranking. Data was obtained on 28 September 2015. Javaslang library does not provide variants that replace JCF implementation and thus was not considered for our study.

| Rank | Library | # Stars | # Watches |
|---|---|---|---|
| 1 | Guava | 5067 | 641 |
| 2 | GS-Collection | 1293 | 196 |
| 3 | Koloboke | 369 | 69 |
| * | Javaslang | 309 | 31 |
| 4 | HPPC | 189 | 31 |
| 5 | Fastutil | 69 | 6 |
| 6 | HPPC-RT | 8 | 3 |

Georges et al. [63] and implement a four-step methodology for evaluating a steady state performance of collection implementations:

S1 For each scenario we execute ten warm-up iterations to achieve a steady performance [2].

S2 We execute 30 iterations while measuring our response variables. Each iteration executes the same operation (sample) in an uninterrupted fashion for five seconds. After reaching the timeout, we calculate the average of all samples as a result. Each result also contains the experimental error of the iterations at a 99% confidence interval.

S3 We execute steps S1 and S2 twice to avoid circumstantial external influence.

S4 We analyze the results of 2 x 30 iterations using rigorous statistical methods. In detail, we analyze the variance with ANOVA [126] and compare multiple means accounting for their experimental errors.

To evaluate the performance of Java collections through this methodolody, we create a benchmark suite called *CollectionsBench*. CollectionsBench is a benchmark framework tailored to collections performance, built upon JMH. As described in Section 2.3, JMH provides the necessary infrastructure for reliable Java bench-

[2] Our preliminary analysis showed that the execution time of the experiment converges after seven warm-up iterations

Table 4.3: Ranking of collection libraries by number of occurrences in benchmarks. We select the top 5 collections from this ranking.

| Rank | Library | # Occur |
|---|---|---|
| 1 | JCF | 13 |
| 2 | Trove | 10 |
| 3 | HPPC | 6 |
| 4 | Koloboke | 5 |
| 5 | Fastutil | 4 |
| 6 | GS-Collection | 4 |
| 7 | Javolution | 4 |
| 8 | Guava | 3 |
| 9 | Mahout | 3 |
| 10 | Commons | 2 |
| 11 | Brownies | 1 |
| 12 | Colt | 1 |
| 13 | Javaslang | 0 |
| 13 | HPPC-RT | 0 |

marks, with forks, warm-up iterations, and can give the benchmark results with nanosecond precision.

To reliably measure performance using CollectionsBench, we perform the following steps: First, we guarantee a homogeneous benchmark behavior throughout different collection variants by implementing our framework with the Template design pattern [61]. The Template design pattern helps us reuse the same test code in all JCF compliant libraries (see Table 4.4) and to delegate the collection creation to the library specific code. Second, we adopt the best practices on JMH benchmarking (as studied in Chapter 3) with, for instance, consuming every non-void return to avoid dead-code elimination (RETU bad practice).

Finally, to only measure time and memory on the collections operations, we create all elements in the setup phase, i.e., outside the measurement phase. The values of each element are generated randomly through a uniform distribution, and we then reuse the random seed to ensure the same conditions in all tests. In the *benchmark* phase we execute the collection operations and measure the following response variables:

- *Execution time*: Time spent executing the benchmark in nanoseconds, using JMH native support.
- *Memory allocation*: The sum of memory requested during the benchmark execution. Since we allocate the elements in the *setup* phase, this variable shows only the memory requested for the collection overhead (including the collection object header and eventual memory padding), but does not consider the element footprint. We extract this information through the JMH GC profiler.

We also collect some performance indicators through the Perf[3] profiler, such as the number of instructions executed, cache miss rate and branch misprediction rate. In Section 4.6 we analyze those indicators along with the source-code to understand the reasons for performance differences of implementations.

The memory allocated during the benchmark is sensitive to buffers and temporarily allocated objects. For instance, the ArrayList allocates a new and larger buffer to accommodate more elements during its expansion. The previously allocated buffer and the new one will be measured by the *memory allocation* variable. Therefore, the memory allocation alone cannot provide an accurate view of the memory usage of a collection.

To account for this problem, we evaluate the memory usage of a collection by computing their collection overhead. We use the tool Java Object Layout (JOL)[4] to retrieve the collection overhead of each implementation. We then analyze both memory allocation and overhead together, to give a detailed perspective of a collection's memory consumption. CollectionsBench is open-source and is fully available online[5].

### 4.4.3 Experimental Planning

Our experimental plan takes into consideration the findings on collections usage presented in Section 4.3.2. First, we focus on the most frequently used collection types while investigating possible alternatives (from third-party libraries) to

[3] https://perf.wiki.kernel.org/index.php/Main_Page
[4] http://openjdk.java.net/projects/code-tools/jol/
[5] https://gitlab.com/DiegoCosta/collections-bench

the JCF collections. Therefore, we select the four most used collection types, ArrayList (AL), LinkedList (LL), HashMap (HM), and HashSet (HS), which accounts for approximately 77% of all declared collections.

Most of the libraries do not implement an alternative to LinkedList, so we opted to compare JCF LinkedList implementations against ArrayList alternatives, as they are interchangeable under the List abstraction [6]. We also profile primitive alternatives to the top three collection types, as they provide a small-footprint option to collections that hold primitive wrappers (see Table 4.4).

Table 4.4: Selected libraries and evaluated collection implementations. Object collections are marked as "Obj" while primitive implementations are represented with "Prim". The column JCF indicates whether the libraries provide implementations compatible with the Java Collection Framework (only for object collections). In total we evaluate 33 implementations.

| Library | Version | JCF | List(AL/LL) | HashMap (HM) | HashSet (HS) |
|---|---|---|---|---|---|
| JCF [137] | 8.0_65 | yes | Obj/Obj | Obj | Obj |
| Guava (Gu) [73] | 18.0 | no | – | MultiMap | MultiSet |
| Fastutil (Fu) [171] | 7.0.10 | yes | Obj + Prim | Obj + Prim | Obj + Prim |
| Koloboke (Ko) [31] | 0.6.8 | yes | – | Obj + Prim | Obj + Prim |
| HPPC (HP) [143] | 0.7.1 | no | Obj + Prim | Obj + Prim | Obj + Prim |
| GSCollections (GS) [70] | 6.2.0 | yes | Obj + Prim | Obj + Prim | Obj + Prim |
| Trove (Tr) [69] | 3.0.3 | yes | Prim | Obj + Prim | Obj + Prim |

Second, we pay a special attention to collections holding the element type String. Strings are particularly interesting in Java due to their extensive use, and have been studied by various authors [102]. To expand on the results, we evaluate the collection implementations for Integer and Long objects as well. We opt for these two numeric objects as they are representative: primitive-wrappers are the second most common category identified. Moreover, they mainly represent objects with 32 bytes and 64 bytes respectively, a common object size.

Third, we do not specify the initial capacity in our benchmark because from our static analysis results, specifying the initial capacity is done rarely in real code. Since we aim to provide results that are useful to the widest range of programmers, we evaluate collections with their default initial capacity.

[6] LinkedList also implements the Queue abstraction and cannot be replaced by ArrayList under this usage scenario.

Table 4.5: Benchmark scenarios used in the experimental evaluation of collections performance.

| Benchmark | Description |
|---|---|
| Populate | Populate the collection with N random elements. |
| Iterate | Traverse through all elements of a collection. |
| Contains | Check if a collection contains an existing random element. |
| Add | Add a random element to the collection. |
| Get | Retrieves a random existing element from a collection. |
| Remove | Find and remove a random element. |
| Copy | Copy all elements into another collection instance. |

Fourth, we evaluate a collection under seven benchmark scenarios, as shown in Table 4.5. Each scenario stresses a single collection operation, executed uninterruptedly during a benchmark iteration. Albeit simple, this style of benchmark provides insightful results to developers, as each scenario's result can be used as building blocks for more complex use-cases. Furthermore, this style of benchmark is commonly adopted by studies on collections performance [80, 147, 172].

As we could not study the typical size of collections through static analysis, we account for multiple levels of workload, by running our benchmarks with multiple collection sizes ranging from 100 to 1M, with the interval set as a power of ten (five categories of size).

In summary, we perform a *factorial experiment* [126] with three element types and five collection sizes, in a total of 15 different configurations. This experiment is run through the seven scenarios for each of the 33 collection types (see Table 4.4). We execute a total of 3,465 experiments, where each experiment lasts five minutes including replications and forks. Running the full benchmark takes 12 days to finish.

We conduct our experiments on a machine with a E5-1660 3.3GHz CPU, 64GB RAM using Linux 3.16.0-53. We use 64-bits JVM HotSpot, and the jdk1.8.0_65 as our Java version. All tests were executed in a single-threaded environment, as our target collections were built for such settings. Regarding memory measurement, we use the default configuration of JVM, which enables the option of compressed object pointer (`+Compressedoops`).

## 4.5 Experimental Evaluation

Our goal is to find a *superior alternative* that can outperform JCF in terms of execution time and/or memory consumption in several use-case scenarios, while introducing no or minimal penalties in others. To achieve this, we ask the following research questions:

**RQ1. Are there superior alternatives to the most used JCF collections with regard to execution time?** We experimentally evaluate the performance of alternatives to JCF ArrayList, LinkedList, HashMap and HashSet and present variants that could yield performance benefits on execution time on section 4.5.1.

**RQ2. Do primitive collections perform better than JCF collections with regard to execution time?** In this question we focus on evaluating the int-primitive variants of the most used JCF collections. We show in Section 4.5.2 that aside for the expected lower memory footprint, primitive collections indeed provide substantially faster operations than their JCF counterpart.

**RQ3. Are there superior alternatives to the most used JCF collections with regard to memory consumption?** In this question we evaluate the memory footprint and allocation of collection variants in comparison to standard JCF implementations. Our results in Section 4.5.3 show that substantial memory footprint reduction can be reached without complex changes in the source-code.

In this thesis, we focus on the most important results that will lead us to our approach in Chapter 5. The analysis of element type impact and a complete discussion on the results of our experimental study can be found in the original paper [43]. We omit the analysis of primitive-collections on memory consumption, as this question is rather well-explored and exemplified in Section 2.4.2.

### 4.5.1 Alternatives for Faster Collections

We report our results as a comparison to the JCF implementation instead of the absolute performance. First, we extract only comparisons where the errors (99%

confidence interval) do not overlap. Then, we calculate the impact of an alternative collection over JCF using the following speedup/slowdown $S$ definitions:

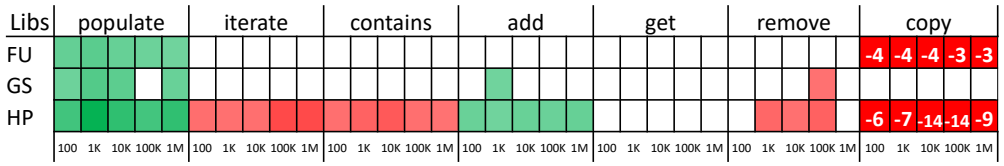
$$S = \begin{cases} \frac{T_{jcf}}{T_{alt}}, & \text{if } T_{jcf} > T_{alt} \\ \\ -\frac{T_{alt}}{T_{jcf}}, & \text{otherwise} \end{cases} \tag{4.1}$$

where $T_{jcf}$ and $T_{alt}$ are the time obtained with using the JCF implementation and the time using an alternative implementation respectively.

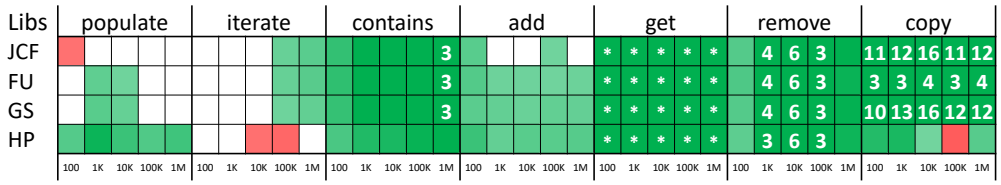
We present our results in a broad heatmap analysis in Figure 4.4, labeled by color. We make an in-depth analysis for each category in the remainder of the section.

ArrayList In the *populate* scenario occurrences of JCF ArrayList can be replaced by the implementation from GSCollections to achieve faster population without compromising the speed of any other operation. In fact, all evaluated alternatives are faster than JCF when populating the list (see Figure 4.4a). The HPPC implementation is, however, slower when iterating the elements in both the *iterate* and the *contains* scenario. In addition, both Fastutil and HPPC copy their lists from 3 to 14 times more slowly.

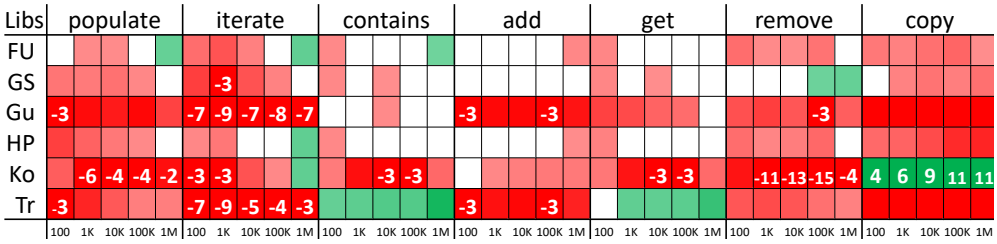
LinkedList JCF LinkedList is outperformed by all ArrayList implementations by a large margin, even in scenarios where LinkedList has a theoretical advantage (Figure 4.4b). This is the case for the *remove* scenario, where we search and remove the element through the `remove(Object)` method. Despite the asymptotic advantage, LinkedList was outperformed by a large margin for collections with more than 1k elements. Furthermore, LinkedList has a comparable performance in the *populate* scenario, even without having to reallocate its buffer (as ArrayList does). The LinkedList is up to three times slower when searching for a random element on collections with one million elements, and it was also outperformed in the *iteration* scenario by some ArrayList variants. Note that in our benchmark



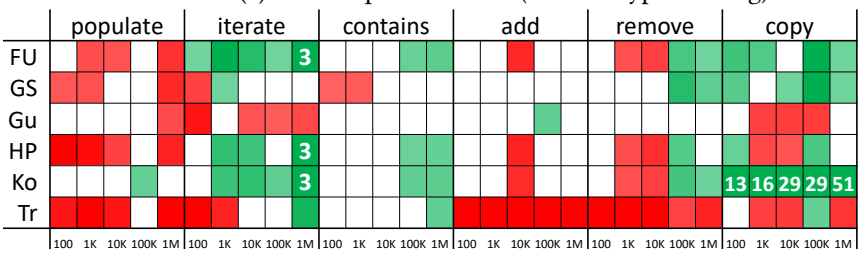

(a) ArrayList alternatives




(b) ArrayList alternatives to LinkedList




(c) HashMap alternatives (element type = String)




(d) HashSet alternatives (element type = String)

Figure 4.4: Heatmap showing speedup/slowdown (as defined in Equation 4.1) of alternative collections compared to JCF, per scenario and size. Green cells indicate a speedup, red cells represent a slowdown. Performance ratios larger than factor 2x are rounded to the nearest integer and printed inside the cell. LinkedList *get* ratios are substantially higher and not shown.

design the LinkedList is unfragmented as the elements are inserted without any removal. Hence, we consider these results a best-case scenario for LinkedList.

HashMap JCF HashMap provides solid performance and cannot be easily replaced by any alternative in terms of execution time improvement. As shown in

Figure 4.4c, some implementations have a comparable performance, like Fastutil, GSCollections and HPPC. Standard HashMap is outperformed only in the *copy* scenario, where Koloboke is able to copy up to 11x faster. However, Koloboke is substantially slower in most of the other remaining scenarios.

HashSet Standard HashSet is outperformed by many alternatives, mostly in the *iterate* and *copy* scenarios (Figure 4.4d). Koloboke is a superior alternative to JCF, outperforming JCF HashSet for large set iteration, and copying at least 10x faster. Fastutil and GSCollections can also be selected as good alternatives: both are slower when populating but faster in the *copy* and and *iterate* scenarios. The JCF implementation is faster than the majority of alternatives when adding elements to the set, but is often outperformed in the remaining scenarios for large workloads (more than 100k elements).

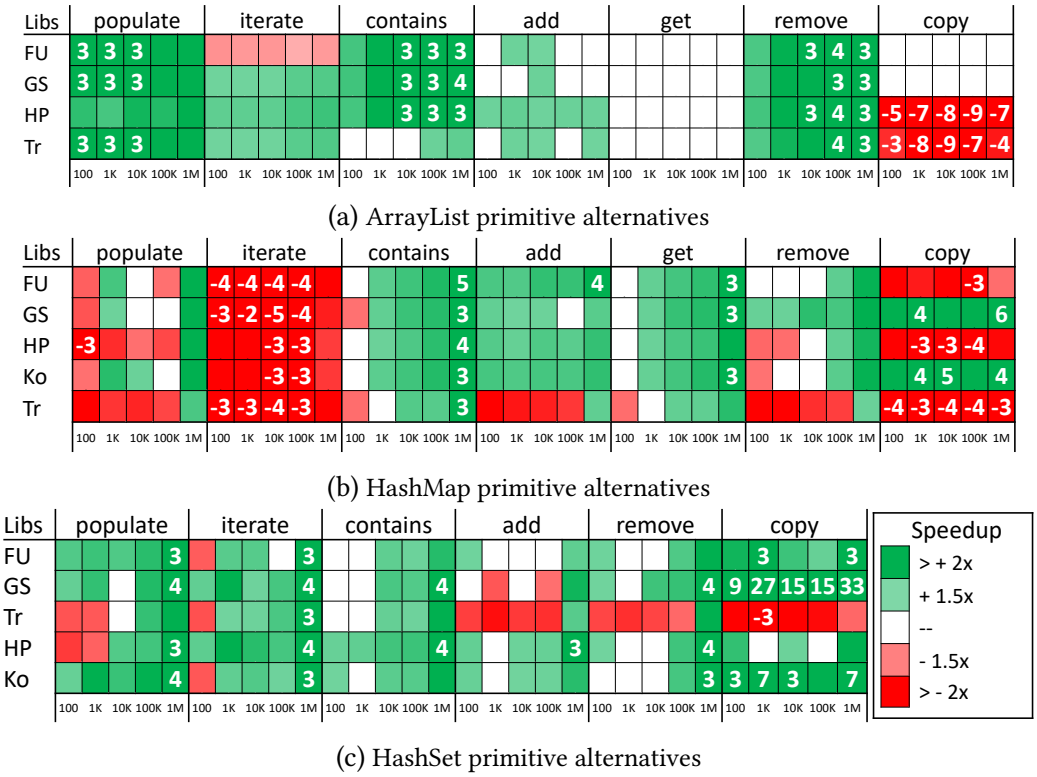
**RQ1. Are there superior alternatives to the most used JCF collections with regards to execution time?**

GSCollections provides a superior alternative to JCF ArrayList. LinkedList is outperformed by any other ArrayList implementation, and both the HashSet of Koloboke and Fastutil are a solid improvement over the standard one. We found no superior alternative to the JCF HashMap, which provides a stable and fast implementation throughout all the scenarios.

### 4.5.2 Alternatives for Faster Primitive Collections

Our results for analyzing the experiments are summarized in a colored heatmap in Figure 4.5.

Primitive ArrayList. Primitive lists provide a superior alternative to JCF ArrayList (Figure 4.5a). The speedup in some cases reaches four times better than the baseline when checking or removing an element from the list. GSCollections consistently outperforms the JCF, followed by Fastutil, which is slightly slower




(a) ArrayList primitive alternatives

(b) HashMap primitive alternatives

(c) HashSet primitive alternatives

Figure 4.5: Heatmap showing the speedup/slowdown (Equation 4.1) of primitive collections (int) compared to the JCF collections holding Integers, per scenario and workload.

when iterating through the list. HPPC and Trove are much slower when copying their lists, and should be avoided if the workload demands this operation often.

Primitive HashMap. As a consequence of the good performance of the standard JCF HashMap, in some scenarios there is no underlying benefit gained from changing to a primitive-type map implementation (Figure 4.5b). Iterations are especially good with the standard HashMap, up to five times faster than the alternatives. For the *contains*, *add*, *get*, and *remove* scenarios, most primitive implementations can provide an improvement. when also considering the *copy* scenario, GSCollections and Koloboke provide the highest performance gain among the alternatives.

Primitive HashSet. Primitive-based sets can be beneficial for the application's performance when it comes to a large number of elements (Figure 4.5c). In fact, the

speedup commonly reaches four times as fast across many scenarios with one million elements. GSCollections, Fastutil, and Koloboke provide superior alternatives and can copy their instances up to 27 times faster. HPPC is also a good alternative but it does not provide the same copy operation speed gain as the other mentioned libraries. Trove's implementation was the only primitive-based HashSet outperformed by the standard HashSet in the *add*, *remove* and *copy* scenarios.

Table 4.6: Comparison of collection overhead and memory allocation of various implementations. The overhead is given in bytes in the form $\alpha \times N + \beta$, where $N$ is the number of elements in the collection, and $\alpha$ and $\beta$ are implementation-specific factors. The average allocated is given in bytes per element.

| **Category** | **Libs** | **Collection Overhead** | **Avg Allocated** | |
|---|---|---|---|---|
| | | | populate | copy |
| Array Lists | JCF | $4 \times N + 24$ | 14.83 | 4.01 |
| | Fu | $4 \times N + 24$ | 10.07 | 4.02 |
| | GS | $4 \times N + 24$ | 13.56 | 4.01 |
| | HP | $4 \times N + 48$ | 15.10 | 4.01 |
| Linked Lists | JCF | $24 \times N + 32$ | 24.07 | 28.11 |
| Hash Maps | JCF | $36 \times N + 48$ | 49.93 | 41.05 |
| | Fu | $8 \times N + 64$ | 35.52 | 42.18 |
| | GS | $8 \times N + 64$ | 59.87 | 24.62 |
| | Gu | $96 \times N + 64$ | 96.90 | 132.36 |
| | HP | $8 \times N + 96$ | 36.41 | 18.16 |
| | Ko | $8 \times N + 232$ | 35.47 | 17.96 |
| | Tr | $8 \times N + 72$ | 47.23 | 18.37 |
| Hash Sets | JCF | $36 \times N + 80$ | 48.85 | 40.43 |
| | Fu | $4 \times N + 40$ | 24.43 | 8.43 |
| | GS | $4 \times N + 32$ | 34.44 | 14.21 |
| | Gu | $52 \times N + 112$ | 64.77 | 58.25 |
| | HP | $4 \times N + 88$ | 16.88 | 8.43 |
| | Ko | $4 \times N + 208$ | 16.84 | 8.43 |
| | Tr | $4 \times N + 64$ | 22.25 | 8.35 |

**RQ2. Do primitive collections perform better than JCF collections with regard to execution time?**

For all three abstraction types (list, map, set), we found a set of primitive implementations that are superior to JCF implementations. GSCollections and Koloboke provide the fastest primitive-based alternatives, followed by Fastutil and HPPC. Trove is often outperformed by JCF implementations in this context.

### 4.5.3 Alternatives for Memory-Saving Collections

In this section, we analyze the difference in the overhead of collections as the main goal for a superior collection replacement. As a supplementary analysis, we also looked at an aspect often neglected by micro-benchmarks: memory allocation. A collection that allocates more memory than indicated by its overhead requests memory for two reasons: (i) allocation as a buffer for future operations, (ii) allocation for temporary objects. The latter reason has a negative impact on performance and can be used as an indicator of how often a collection implementation can trigger the action of the Garbage Collector (GC). Collections that require frequent intervention of the GC can reduce the performance of an application in a long run, a behavior difficult to observe with micro-benchmarks.

Since the elements are pre-allocated in the setup phase, memory allocation only comes into play when elements are added to the collection, namely by *populate* and *copy* operations.

ArrayList. All implementations have a similar overhead (see Table 4.6). Due to its buffer reallocation, each implementation allocates on average three times its own overhead in the *populate* scenario. Regarding only memory allocation, Fastutil can be a good alternative, saving 30% of allocations in the *populate* scenario.

LinkedList. We show in Table 4.6 that the standard LinkedList implementation has an overhead of 24 bytes per element, as opposed to the 4 bytes required in each ArrayList implementation. This is a consequence of the pointers to neigh-

bor elements for each LinkedList entry. Essentially, an ArrayList saves 83% of the memory overhead. Despite the buffer expansion of ArrayList, LinkedList still allocates twice as much memory in the *populate* scenario, and it allocates five times more memory when copying from a previous instance.

HashMap. We can observe in Table 4.6 that the JCF implementation has a considerably higher overhead. Standard HashMap has an overhead of 36 bytes per each entry in the map while almost every other alternative consumes only 8 bytes. This difference occurs because JCF uses a Node object for each entry, a structure that contains three references (12 bytes) and a primitive (4 bytes), but, being an object, also has an overhead of 12 bytes of header and 4 bytes lost due to alignment. The alternatives do not define each entry as an object and instead use two arrays where each key and value pair are stored. As a consequence, they save 77% of memory overhead. Regarding allocations however, the difference of JCF and the alternatives are not quite as drastic as in the collection overhead.

HashSet. Similarly to Maps, JCF HashSet is implemented with a larger overhead than almost any of the potential alternatives. In fact, it uses a HashMap internally to store the elements, causing the same additional overhead of 36 bytes per element, due to the HashMap's Node object. The alternatives, on the other hand, implement the HashSet as a simple single array. This allows them to save 88% of memory overhead compared to the JCF HashSet. Unlike HashMap, this difference holds in the memory allocation as well, where JCF allocates twice as much memory in the *populate* and five times as much memory in the *copy* scenario.

**RQ3. Are there superior alternatives to the most used JCF collections with regards to memory consumption?**

We found no superior alternative to the JCF ArrayList in terms of memory overhead, but Fastutil allocates less memory when populating its own list implementation. Numerous alternatives offer a superior alternative to the JCF HashMap and HashSet. LinkedList has a higher overhead than any considered ArrayList alternative.

## 4.6 Discussion

### 4.6.1 Reasons for Performance Differences

Our investigation of the source code of collections revealed some implementation patterns responsible for performance differences in collections. We discuss these in the following.

Distinct API calls. The implementations often differ in usage of the API call for copying an array. Here developers use two distinct methods: `System.arraycopy()` and `Arrays.copyOf()`. The latter is just a wrapper for `System.arraycopy()` which requires only the original array, as opposed to the System version, where the target must be passed as a parameter. Our experiments showed that `System.arraycopy()` is 25% faster than `Arrays.copyOf()` for arrays up to 1 million elements. This is one of the reasons why GSCollections ArrayList (which calls `System.arraycopy()`) is faster than JCF counterpart, since *populate* scenario is dominated by the buffer expansion cost.

Add copy versus memory copy. Collections copy is the scenario with the largest discrepancy in performance of our experiments. We found that libraries implement two main approaches: they either add all elements one by one to a new created collection instance, or they perform a memory copy. Needless to say, the memory copy is faster as adding elements one by one has the burden of manipulating objects individually. This explains why Koloboke executes up to 50x faster than JCF when copying a set with 1 million elements.

Nevertheless, libraries often opt for adding the elements into the new instance for the simplicity of the workflow. The copy constructor receives a collection reference as a parameter and must be able to polymorphically handle all kinds of collection types. A memory copy is restricted only to copies from the same type. GSCollections and JCF use memory copy for ArrayLists but rely on addAll for HashSet and HashMap. Koloboke is the only library that use memory copy for HashSet/HashMap. It is important to address that copy is called 11% of the HashSet instantiations. Therefore applications can strongly benefit from memory copy implementation.

Sub-optimal primitive API. Iterating elements of FastList's ArrayList primitive collection is slower than JCF ArrayList, and far more time-consuming than any other primitive implementation. To understand this, note that each primitive library provides a `forEach(IntProcedure)` method to iterate and process the list. However, FastList provides a `forEach()` method defined by JCF `AbstractCollection`, which accepts an `Object` and implicitly converts each primitive to its wrapper, degrading the performance by a factor of 5x. This case illustrates the complexity of a collection API, which often provides multiple ways of performing the same task, but with distinct performance costs.

Sub-optimal loop implementation. The `contains()` method of Trove primitive ArrayList is 3 times slower than any other primitive implementation (see Fig 4.5a). The code inspection reveals that contrary to other primitive alternatives, Trove defined the array loop using an unconventional for loop, namely `for(int i = offset; i- > 0; )`. This loop, albeit correct, produces three times more branches and 30% more branch misses than a backwards loop defined with the decrement in third position of the `for`-statement. In fact, after refactoring the `contains()`-method in an obvious way in our own extension of Trove's primitive ArrayList the performance bottleneck was fixed.

### 4.6.2 Implications for Practitioners

Each additional alternative collection implementation used in a software project increases its complexity and maintenance effort. Consequently, developers should consider replacing collection implementations only if such a change will result in a significant benefit. It is unlikely that a developer is willing to include multiple alternative collection implementations to optimize time and memory profiles for each collection usage as the complexity of dealing with multiple implementations might overshadow the performance benefits. In order to support programmers in such decisions, we present a guideline (Tables 4.7 and 4.8) showing the potential benefits and drawbacks of using alternative implementations for several relevant scenarios and/or optimization objectives.

Table 4.7: A guideline showing the impact of replacing JCF collections with alternative collection implementations for some relevant scenarios/optimization objectives.

| Rule | To reduce JCF collection overhead and improve time performance | Overhead Savings | Speedup/Slowdown 1M elements |
|---|---|---|---|
| R1 | JCF LinkedList → JCF, FU or GS ArrayList | 84% | 3x faster contains<br>2x faster remove<br>4x faster copy |
| R2 | JCF HashSet → Ko HashSet | 88% | 3x faster iterate<br>1.5x faster contains<br>51x faster copy |
| | **To reduce overhead with smallest time penalty** | | |
| R3 | JCF HashMap → GS HashMap | 78% | 1.5x slower copy |
| | **To reduce overhead with faster copy** | | |
| R4 | JCF HashMap → Ko HashMap | 78% | 11x faster copy<br>2x slower populate<br>4x slower remove |

Table 4.8: A guideline showing the impact of replacing JCF collections with alternative primitive-collection ("prim-collection") implementations for some relevant scenarios/optimization objectives. The memory savings include overhead reduction as well as smaller element footprint (results assume replacing Integer objects by the primitive int).

| Rule | To reduce JCF collections footprint and improve time performance | Footprint Savings | Speedup/Slowdown 1M elements |
|---|---|---|---|
| R5 | JCF ArrayList → GS prim-collection | 60% | 2x faster populate<br>4x faster contains<br>2x faster remove |
| R6 | JCF HashMap → GS/Ko prim-collection | 76% | 2x faster populate<br>2x slower iterate<br>6x/4x faster copy |
| R7 | JCF HashSet → GS/Ko prim-collection | 84% | 4x faster populate<br>4x/3x faster iterate<br>33x/7x faster copy |

In this thesis we illustrate the effect of JCF collection replacement on execution time for collections one million elements (large). Note that in case of object collec-

tions the memory savings come solely from a reduced overhead as the elements themselves are not modified. In case of the primitive collections the programmer must replace the object element type by its respective primitive. However, the impact on memory savings is higher as the memory footprint of elements is reduced as well.

Table 4.7 presents four recommendations for replacements leading to a substantial improvements. Recommendation R1 describes the replacement of a LinkedList by an ArrayList from one of three libraries, each offering a consistent improvement in time and memory. In R2 we recommend Koloboke's alternative to JCF HashSet, as it provides a substantial improvement on execution time and memory overhead. For HashMaps it is possible to aim for a memory overhead reduction with a small execution time penalty (R3), or to reduce memory and improve the execution time of copy operations (R4).

An important result is the universal superiority of primitive collections (shown in Table 4.8). It is possible to considerably improve both memory footprint (overhead + elements) and execution time for all three major collections: ArrayList (R5), HashSet (R6) and HashMap (R7). Note that in all cases the replacement is particularly beneficial for large collections as the savings of memory and execution time are, in general, bigger.

## 4.7 Summary of the Chapter

In this chapter, we present an empirical study on the usage and performance of Java collection libraries. We analyzed the usage patterns of collections by mining a large code corpus of Java projects and conducted a rigorous evaluation of the performance of standard Java Collection Framework and six most popular alternative collection libraries.

We showed that:

- Developers **rarely select non-standard collections**, relying heavily on general-purposed collection implementations.
- Developers **seldom tune the performance parameters** of collection creation, such as the initial capacity and load factor.

- We found that alternative implementations can offer programmers a **significant improvement over standard libraries on both execution time and memory consumption** on several usage scenarios.

We have devised a guideline to advise developers on the performance benefits of swapping standard implementations by alternative variants. Moreover, the results of this study served as the main motivation for our work in Chapter 5, where we develop and evaluate a low-overhead method for automatic and dynamic selection of collections, including selecting variants provided by alternative libraries.

# 5 A Framework for Efficient and Dynamic Collection Selection

Our study presented in Chapter 4 have indicated that developers rely heavily on standard and general-purposed variants to develop their applications. While alternative implementations can be selected to improve time and memory of applications, the burden of selecting data structures still lies ultimately in the hands of practitioners. In this chapter, we present an application-level framework for efficient collection adaptation. It selects at runtime collection implementations in order to optimize the execution and memory performance of an application, essentially removing the need for manual analysis from practitioners.

**Contributions.** In this chapter, we present the following contribution:

I An approach for dynamic (runtime) selection of collection implementations. The choice of collection variants optimizes (for a specific allocation site) performance along multiple dimensions according to the workload profiles of monitored instances and according to configurable rules.

II CollectionSwitch: a low-overhead, concurrent implementation of this approach as an application-level library.

III An analysis and implementation of adaptive collections which dynamically switch their underlying data structure according to the size of the collection.

IV An empirical evaluation of both concepts and their implementation on synthetic benchmarks and real applications.

**Reference.** This chapter is partially based on a peer-reviewed publication [42].

## 5.1 Introduction & Motivation

Let us suppose you are a developer of an application that contains the code snippet depicted in listing 5.1. This example shows a Java code with a list use-case comprised of multiple insertions followed by a traversal. A number of list variants could be selected for this use-case, but let us take only LinkedList and ArrayList variants: which variant would provide the fastest implementation in this exemplary scenario?

Listing 5.1: Example of Java code using list collection abstraction.

```
List<Integer> list = new _____________<>();
...
for(Integer toAdd : anotherList) {
   list.add(calculateIndex(toAdd), toAdd); // Insertion
}
...
Integer sum = 0
for(Integer toSum : list) { // Traversal
   sum += toSum;
}
```

The standard LinkedList implementation tends to perform better when insertions are done in the middle of the list, while it performs comparably worse at traversals due to bad memory locality of linked nodes. On the counterpart, the ArrayList variant provides faster iterations, and if the `calculateIndex` method returns an index near the end of the list, the cost of shifting every subsequent element in the array can be relatively small compared to the traversal of linked nodes. In fact, if the collection fits in the processor cache, adding an element in the middle is often faster than finding the node to concatenate in a LinkedList in modern processors. Therefore, in this code, the decision to use either an array-backed list or a linked-list depends on 1) the amount of insertions, 2) the position of inserted elements given by `calculateIndex`, 3) the amount of traversals performed and last, but not least 4) the characteristics of the underlying system hardware.

Following the first commandment of performance-engineering *"Don't guess: Measure!"*, the advisable approach in this case is to measure the performance of both

LinkedList and ArrayList using workloads that represent the system in production. Naturally, this approach raises the common challenges of creating performance tests that accurately represent the workload of their program. Does the underlying system has a well-defined known workload or is it composed by a complex interweaving of multiple algorithms?

Furthermore, as software evolves through continuous development, the usage of its collections might also be modified beyond what has been designed in the first place. A second developer might want to add a patch that checks if the element is already in the list, through the method `contains`, before inserting it again. Hence, this new usage would be better suited for a set implementation, or at the very least would benefit more by using an ArrayList variant, as the new `contains` method call effectively adds another traversal to the list.

### Shortcomings of Manual Selection

The sub-optimal selection of collection may cause significant performance degradation in both execution time [117] and memory consumption of applications [158]. Everyday, developers face the problem of selecting collections for their application and have to do such tasks manually. In this chapter, we make the case that manually selecting data structure for applications have the following shortcomings:

1. Developers have to rely solely on theoretical models, such as asymptotic analysis, which may yield sub-optimal results in practice.

2. Developers need to be aware of the performance benefits provided by specialized collection variants tailored for specific use-cases (e.g., primitive collections).

3. Developers need to have a realistic understanding of the application's workload during production, to select the variant that optimally matches the application's demand.

Limitation of Asymptotic Models. Developers have at their disposal the asymptotic analysis to guide their decisions. Albeit being a suitable approach for understanding time and memory complexity of different data structure operations,

the asymptotic model might lead to wrong conclusions in a real usage context. In the realm of collections performance, constant matters. For example, take the TreeSet (red-black tree) and the HashSet (hash-table) Java implementations. The TreeSet has worse asymptotic behavior but almost always has faster lookup times on modern architectures when holding fewer than 200 data elements. In a similar fashion, ArraySet outperforms HashSet on lookups with less than 100 elements (as described in Section 2.4.2), which is extremely common in real Java applications [30]. Therefore, developers need empirical models to guide the selection of their data structures.

Specialized variants. In Chapter 4 we investigate the most frequently selected collection types by developers of Java open-source projects, and concluded that Java programmers rely heavily on the four JCF implementations of ArrayList, HashMap, HashSet, and LinkedList. While such variants are designed to perform reasonably well on general use-cases, our experimental evaluation showed that some alternative variants provide superior performance in particular scenarios. We conjecture that developers are typically not aware of the benefits of alternative variants and have a narrow set of general-purpose collection variants to choose from. Thus, it stands to reason that developers need tools to increase the space-search of variants at their disposal, and identify specialized variants that fit better to their application's workload.

Dynamic Workload. The third shortcoming related to manual collection selection is the fact that the workload of a program is often dynamic and might drastically change during execution. Many studies have shown that a real-world execution often consists of multiple phases, and workloads (as well as program behavior) can change many times during a single run [92, 159]. In many cases, it has proven to be impossible to find a single optimal solution for the whole program run [178].

Our Approach. While crucial for performance, selecting a suitable collection type and implementation essentially creates additional burden to developers during software development. This problem demands automated solutions that can

analyze the application workload, and decide which collection variant to use based on the current usage scenario.

We propose an approach and its implementation for dynamic (runtime) selection and optimization of collection variants. Particular attention was given to efficiency, resulting in a negligible overhead despite of runtime adaptation capabilities. Our approach works at two levels of granularity: at the level of a collection allocation site, and at the level of individual collection instances.

## 5.2 Related Work

There is a substantial body of research on proposing new data structures and recommendation tools that aid developers on identifying collection inefficiencies. In this section, we present the related work containing approaches for different programming languages, as in many cases the same technique can be applied or compared to proposed ideas for Java collections. We categorize these works into the following areas:

- **Design of new collection variants** groups studies that propose new versions of the standard collections that produce better performance, for a certain usage scenario and performance criteria.
- **Detection of collection inefficiencies** is comprised by studies that propose approaches to identify the inefficient usage of collections, but do not focus on the collection selection problem. Such studies focus on pinpointing collections API misconfiguration, that yield software non-functional issues such as memory leak or memory bloat.

Furthermore, we dive into more details on studies that aim specifically at automatically selecting collection variants for better performance. This field of study can be further divided into two categories, according to their methodology and scope of variants selection:

- **Offline collection selection** groups studies that monitor the application workload and report to developers a series of beneficial code transformations to improve application's performance. Such studies are denoted offline

because the process of changing the collection is performed by developers, after the program execution. It also may involve a larger scope of collection variants, including non-functionally equivalent candidates - as developers have the chance to adapt the code accordingly.

- **Adaptive collection selection**, consist of approaches that monitor the application and select the appropriate collection variant at runtime. As all operations have to be performed during application's execution, the time and space overhead is the greatest challenge of adaptive approaches. Furthermore, the changes need to preserve the functional aspect of previously selected collections as the swap is performed without the consensus of developers.

### 5.2.1 Design of new collections variants

This related field of research aims at providing new collection implementations that either replace or complement commonly used variants. As collections have been pointed as one of the main cause of memory overhead in applications [30, 125], memory efficiency is the most commonly tackled aspect of new collection design studies. Gil and Shimrom [67] studied strategies to reduce the memory overhead of maps and sets in Java. Using a series of memory compaction techniques, including new hashing mechanisms, the authors re-implemented hash-backed and tree-backed variants, and were able to reduce memory overhead from 20% to 70% depending on the VM (32 or 64bits) and collection implementation.

Bergel et al. [13] focus on the Pharo language[1] , and propose the use of lazy initialization as a measure to reduce the memory wasted on empty collections, and a mechanism of recycling internal arrays to mitigate the memory cost of collections expansions. Furthermore, authors investigated the usage of a hybrid data structure from Lua [85], which has both array and hash data representations, and can be used to reduce the memory overhead of Pharo collection's by 19%.

Moreover, Bolz et al. [20] presents a series of storage strategies that can be applied to dynamically typed languages (such as Python) to reduce memory waste.

[1] Pharo is an open-source object-oriented language. More information can be found on the project website: http://pharo.org/

Such techniques exploits the homogeneity of collection elements to reduce the memory overhead, essentially trading type flexibility for better memory usage and faster execution time. The evaluation showed an improvement of time performance on average by 18% and lowering peak memory usage by 6% in some Python benchmarks.

The work of Steindorfer and Vinju [164] propose a new version of immutable hash-array mapped trie [2] of Scala and Clojure programming languages. Authors optimize the object layout - formally bloated by suboptimal port from C++ - and introduce a new encoding for the tree data structure, that reduces memory overhead and improves the memory locality of studied collections. This new variant outperformed Clojure and Scala collections in both memory footprint and runtime efficiency.

Overall, studies that propose new collection variants further enhance the possibilities for developers to optimize their applications. While finding inefficiencies on current implementations and proposing new variants certainly contribute towards better performance of applications in general, developers still need to acknowledge the existence of these alternatives, and select them when the application workload matches their intended use.

This chapter presents an approach that focus on automatically selecting variants for better performance, and newly customized variants can be included in our approach to increase the search-space that our framework analyzes.

### 5.2.2 Collection inefficiencies

The sub-optimal selection of collections is only one among many collection-related pitfalls practitioners may fall when developing applications. This body of research focus on identifying patterns of collection inefficiencies such as collection misconfiguration, calls to redundant operations and the unnecessary creation of intermediate data structures.

Xu and Rountev [179] proposed a tool that combines both static and dynamic analysis to find underutilized and overpopulated collections. The former occur

[2] Also referred as a radix tree, a trie is essentially an ordered tree data structure for finite stsrings and acts like a Deterministic Finite Automaton without loops.

when a collection holds a very small number of elements, but waste memory with a largely defined capacity. The latter problem occurs when a collection holds too many elements, but most elements are never accessed again during program's execution. Authors re-framing all collections operations into a unified ADD-GET model, and use the reachability algorithms from graph theory to detect underutilized and overpopulated collections. Yang et al. [181] build upon Xu's work and implement a collection-element flow graph, obtained through dynamic analysis, which is able to also identify unnecessary intermediate collections.

Olivo et al. [133] focus on statically identifying redundant traversals on collections, i.e., cases where the program iterates repeatedly over the same collection not modified between repeated traversals. The proposed approach uses the asymptotic model to establish an upper-bound between collection relations, and any identified code fragment that breaches this threshold is flagged as a performance bug.

The inappropriate usage of collections is also linked to memory-leak issues in Java [64]. Typically, elements never removed from collections are never released back to the JVM, until the collection itself becomes eligible for garbage collection. Xu and Rountev [180] propose a profiling technique that records each collection operation. After a program run, the approach combines a set of heuristics (e.g., an element never removed is likely to be a leak) and builds a ranking of collection instantiations that are prone to memory-leak.

Studies on collection inefficiencies target performance problems related to collections without exploring the effect of different collection variants. Similarly to works on design of new variants (Section 5.2.1), studies on collection inefficiencies tackle an orthogonal problem to the collection selection. The above mentioned studies could be combined to automated collection selection approaches to provide developers a more complete feedback of performance bloats caused by collections.

### 5.2.3 Automated Collection Selection

Studies that tackle the collection selection problem aim at aiding developers on properly choosing their data structures, either by providing a report of beneficial code transformations (offline approaches) by changing their collections automat-

ically (adaptive approaches). In all cases, selecting an appropriate collection requires three fundamental steps:

1. **Understanding the application workload.** To perform a sensible selection, approaches need first to understand the usage context of a particular collection. This effectively requires the monitoring and profiling of collection objects.

2. **Searching for better variants.** Given a particular workload, an automated approach needs to find in the search-space of collection variants a better candidate. Essentially, the search mechanism needs to match the usage context with a model that guides which variant should be selected.

3. **Selecting a new variant.** After the identification of a better variant, the approach then needs to perform the selection. As aforementioned, selecting a new variant means either reporting to developers potential optimization opportunities, or changing the program during runtime to use better variants.

All approaches described in the following subsections differ on how they implement the above mentioned three-steps methodology. Particularly, the model used for searching new variants has been extensively explored in the past decade, with studies proposing the use of rule-based system, machine-learning techniques such as genetic algorithm and neural network, or the use of empirical models guided by benchmark results. We present the related work in chronological order of publication.

#### Offline Collection Selection.

Offline collection selection is comprised by approaches that monitor applications, find a better variant and report to developers a list of optimization opportunities. These approaches are not fit to adapt the code in production, and are sometimes designed to be executed on a specific test environment.

One of the first proposed tools for guiding developers on selecting better collections was Chameleon [158]. Chameleon profiles the access patterns and space

utilization of each collection instance, aggregating the information by call-site. The profile information is then combined with user-written rules for collection selection, reporting back to developers any matched criteria. The strong suit of Chameleon is its ability to monitor a variety of collection features with low-overhead, allowing the tool to be used during production, and work essentially as a runtime verifier of optimization opportunities on collection usage. However, Chameleon still relies on developers rules to guide the framework during search, still requiring an expertize in the performance of data structure from developers.

Liu et al. [117] propose Perflint, a performance adviser tool for C++ programs. Perflint monitors the usage of collections to infer the collection usage context, but searches on the space of collection variants through an empirical cost model. This model calculates each operation cost through a series of benchmarks, and extrapolates the obtained metrics to an observed context usage. This creates a hardware-sensitive search method, which tends to be more accurate, without requiring any input from developers. Perflint also tackles other categories of optimizations, such as sorting algorithms and string usage and due to monitoring of several objects, it may incur in a prohibitive overhead for a production environment.

In 2011, Jung et al. proposed Brainy [98], an approach that relies on a machine-learning model to advise the developer. In this approach, the process of searching for better variants for a determined usage context is learned by an Artificial Neural Network (ANN). Before the learning phase, authors generate thousands of random and synthetic programs that make extensive usage of collections, using different variants, and evaluate their performance. This data is then feed into an ANN model, that attempts to learns what variants would fit best in a specific use-case scenario. Brainy combines hardware awareness with a flexible model that can be applied to different applications, at a cost of a long initial training section of its ANN model.

As energy consumption has emerged as an important performance concern for the research community, Manotas et al. [120] propose SEEDS, a decision-support framework for energy optimization. As its core, SEEDS uses exhaustive search to find the best variant, guided by developer transformation rules and functional tests. Compared to previous approaches, SEEDS not only evaluates the energy consumption of a program under specific test-cases, but also modifies the program's

byte-code automatically to cover all combinations of variants, reporting the set of collections that yield the lowest energy consumption. On one hand, by measuring all combinations SEEDS approach is likely to find a set of variants with empirically accurate energy improvement. On the other hand, however, the absence of a model to limit the search-space may incur in a prohibitive cost on large projects, as the number of combinations in projects where thousands of collections are declared is too large to be evaluated.

Compared to adaptive approaches in general, approaches that apply offline collection selection techniques are able to perform more extensive searches, use more complex models and report different aspects of collections inefficiencies. However, such approaches ultimately rely on developers to apply the suggested transformations, often require a testing environment that is representative of the application's workload, or incur in a prohibitive overhead to be used in production.

Adaptive approaches - such as the one to be described in Chapter 5 - aim at supplying developers with tools that overcome the limitations of offline tools, through the use of simpler but faster search models. Hence, our approach is fundamentally different from the above mentioned in a core aspect: our framework identifies and select better collection variants at runtime, effectively moving from the role of a performance adviser to an automated performance optimizer, adapting the collections to the current application demand.

#### Adaptive Collection Selection.

Online solutions propose to shift the responsibility of selecting an implementation from the developer to the runtime, realized by adaptive collections. As the name suggests, an adaptive collection adapts its own implementation to another variant, better suited for its current usage scenario. Apart of taking away this burden from the developer, the adaptive collection also tackles the problem of optimizing programs with multiple workload patterns. Several works [10, 51, 178] have shown that a single implementation and algorithm is often not optimal for the entire program execution, specially on large-scale softwares.

Moving the entire process of selecting a suitable collection variant to runtime environment is a complex task, which is often tackled by adding an intermediate

layer between the collection and the application source code. Adaptive approaches essentially encapsulate standard collections to implement an adaptive behavior, but differ on what the level of adaptivity, how to reduce the inevitable overhead and on the search scope of new variants.

CoCo [178] was the first fully automated approach to tackle the collection selection in Java and served as an inspiration for our framework: CollectionSwitch. In CoCo, each collection instance monitors its usage and carries multiple data structure representations, gradually transitioning to the most appropriate when seems fit. As its core, CoCo trades memory for better execution time, amortizing the cost of switching to a new variant by carrying all possible data representation in a single instance. Hence, this approach is not suited for memory-constraint applications - memory overhead of CoCo was reported as high as 85% in one application - nor can be used to optimize their memory usage. Another limitation of CoCo is its user defined model to replace the collections at runtime, relying solely on asymptotic analysis to select the most suitable collection type. As shown in [98], the asymptotic analysis sometimes fail to capture important nuances of collections performance. Furthermore, CoCo does not consider alternative collection libraries, or variations of the same data representation, and focus only on standard implementations from JCF, as opposed to our approach.

Orsterlund and Lowe [144] propose the use of context composition to identify scenarios in which transforming the collection would lead to performance benefits. Essentially, context-composition focus on learning a dynamic dispatcher, i.e., a state machine that reads a sequence of collection operations and transitions to a data representation that best fits the current scenario. As opposed to CoCo, the approach here performs an instant transition to the best variant, copying all elements in the process. Copying elements into a new data structure is an expensive operation, and could negatively impact the application's execution time if performed multiple times during the lifetime of a single collection instance. Authors address this problem by analyzing a sequence of operations and only transforming the data representation if this sequence indicates a more stable usage scenario.

De Wael [51] presents "Just-In Time Data Structures", a proposal to shift the focus from selecting the best data representation, which may not be unique throughout the program's life-cycle, to selecting the best sequence of data structures. Au-

thors focus on the general approach of designing adaptive collections and propose the taxonomy used throughout this thesis. Furthermore, authors have implemented an extension of Java language, called JitDS-Java, where experts could provide the set of rules for swapping data representations, and their compiler could create the adaptive collections.

Our work extends the adaptivity proposed of the collection instance to the call-site of the collection instantiation (allocation context). This extension allows us to reduce the overhead impact of the proposed adaptive instances while improving the performance of applications in both execution time and memory consumption. Furthermore, our framework can be combined with the adaptive instances proposed by Xu [178] and Osterlund [144], by including their implementations in our set of variant candidates.

## 5.3 A Framework for Dynamic Collection Selection

In this section, we present the fundamental aspects of CollectionSwitch, our framework for efficient collection adaptation. The CollectionSwitch selects at runtime collection implementations in order to optimize the execution and memory performance of an application. We propose a novel approach that uses workload data on the level of collection allocation sites to guide the optimization process. Our framework identifies allocation sites which instantiate suboptimal collection variants, and selects optimized variants for future instantiations.

The optimization of collection variants of our approach takes place at two levels of granularity: at the level of a collection allocation site, and at the level of the individual collection instances.

- **Allocation site-level adaptation.** We modify the allocation sites of collections in order to enable the workload-aware selection of variants created during future instantiations (Section 5.3.1). To this aim we monitor and analyze the behavior of previous instances created by a specific allocation site, and decide on switching to other variants according to user-defined performance rules. The overall approach is illustrated in Figure 5.1.

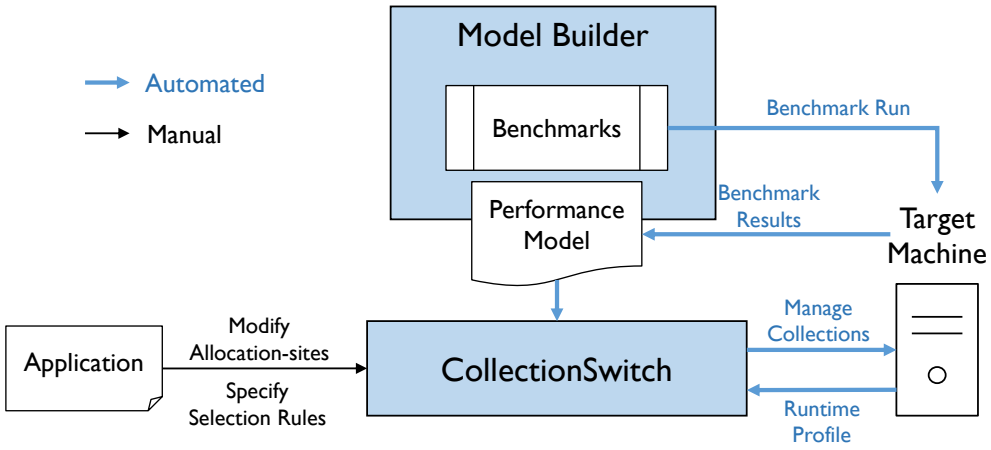



Figure 5.1: Overview of the processes and dependencies used in our approach. Blue arrows denote processed that are automatically handled by our framework, while white arrows are manual processes required by developers.

- **Instance-level adaptation.** We introduce adaptive collection variants capable of changing the internal data structures depending on the collection size (Section 5.3.2). Such variants are suitable if collection instances created by the same allocation sites can have both only few or a large number of elements.

In the allocation-site level adaptation, our framework exploits previously computed performance models, and the runtime data characterizing the workloads of the deployed collection instances (Figure 5.1).

### 5.3.1 Adaptation on Allocation Site-Level

Technically, selected allocation sites are instrumented with a code layer called *allocation context* which creates, monitors and adapts the collections (see Figure 5.4). Each allocation context initially instantiates default collections variants, as specified by the developer.

A sample of all created collection instances is instrumented and monitored in order to obtain the *workload profiles* of these instances. A workload profile comprises the number of executed *critical collection operations* (listed in Section 5.4.1, such as *populate*, *contains*, or *iterate*) and essentially the maximum size of a collection.

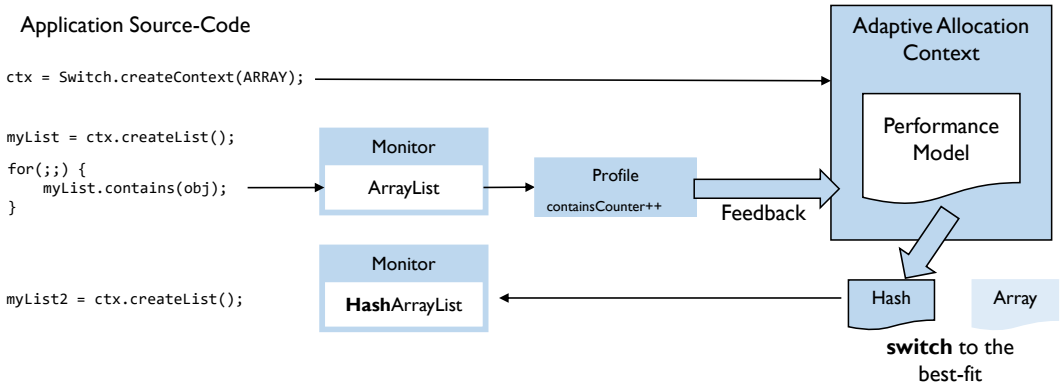



Figure 5.2: Selection of collection variants by allocation contexts based on the workload profiles on collection instances.

The allocation context periodically evaluates these profiles to decide whether future instantiations should use another collection variant than the current one (i.e., whether to "switch"), and if yes, which variant (Figure 5.2). After switching to a new variant a fraction of the instances is monitored to allow a continuous adaptation process.

VARIANT SELECTION ALGORITHM The allocation context in our approach selects a collection variant by considering multiple *cost dimensions* such as execution time and memory overhead. While the interplay of these dimensions is described in Section 5.3.1, we assume in the following a fixed cost dimension $D$.

We compare the collection variants $V$ (per cost dimension $D$) according to the *total cost* $tc(V)$ metric. This metric depends on the observed workload profiles $W$ of the monitored collection instances. In particular, $W$ comprises the numbers of executed critical operations $N_{op}$ during the lifetime of a monitored collection instance, and the maximum size $s$ of this instance. Moreover, $tc$ depends on the performance models obtained for each variant that will be further described in this chapter (Section 5.4.1). These models yield the averaged costs $cost_{op,V}(s)$ of a critical operation $op$ of the collection $V$ depending on $s$, the maximum size of a collection.

With these preliminaries, we define $tc(V) = tc_W(V)$ as:

$$tc_W(V) = \sum_{op} N_{op,W} * cost_{op,V}(s).$$

Note that $tc(V)$ only estimates the total cost of all operations under a workload, since we use the *maximum* collection size $s$ as an argument to a performance model $cost_{op,V}(s)$, and not the actual collection size when a specific operation is executed. As the cost of an operation are typically larger with growing $s$, the value of $tc(V)$ is an overestimate of the real performance cost of a variant instance.

The above description assumed workload data from only a single monitored collection instance. In reality we monitor multiple instances per allocation context. We exploit all this data by summing up the total cost over all monitored instances. These sums $TC_D(V)$ (per collection variant $V$ and a cost dimension $D$) are used when applying the selection rules described below.

Configurable Selection Rules As described in Chapter 2, time and memory have a strong trade-off relation in data structures and an improvement on one cost dimension might incur penalties on other cost dimensions. To account for such trade-offs, we introduce configurable *selection rules*.

A selection rule $R$ consists of one or more criteria (predicates) $C_1, C_2, \ldots$, each corresponding to a unique cost dimension (e.g., execution time, memory overhead, or energy usage). A collection variant is selected by $R$ if all of the criteria are satisfied. A criterion $C_i$ is satisfied if the ratio of the total cost $TC_D(V_{new})$ of a candidate collection variant $V_{new}$ (i.e., a potential replacement) by the total cost $TC_D(V_{cur})$ of the current variant $V_{cur}$ is not larger than a user-specified threshold $T_D$. In other words, $C_i$ is satisfied if

$$\frac{TC_D(V_{new})}{TC_D(V_{cur})} \leq T_D.$$

Note that $T_D < 1$ enforces a cost reduction, while $T_D \geq 1$ expresses a maximum penalty incurred by the candidate variant. Table 5.5 shows examples of selection rules, each focusing on optimizing applications for a particular performance di-

Table 5.1: Example of Selection rules and their interpretation in the performance optimization of applications.

| Rule | Improvement | Max Penalty | Interpretation |
|---|---|---|---|
| $R_{time}$ | $TC_{time}$ < 0.8 | – | Select variants with predicted time performance at least 20% lower |
| $R_{alloc}$ | $TC_{alloc}$ < 0.8 | $TC_{time}$ < 1.2 | Select variants with predicted memory allocation at least 20% lower with maximum of 20% of time penalty. |

mension. We use these same rules later when evaluating our approach in further sections.

During periodical evaluation of workload data for a given allocation context, we switch the collection variant if a selection rule finds a variant different from the currently used one. If multiple candidates satisfy all the criteria, we select a variant with a largest improvement on the first criterion $C_1$.

### 5.3.2 Adaptation on Instance-Level via Adaptive Collections

Adaptive collections are able to change their internal data structure depending on the current size of the collection (i.e., number of contained elements). The motivation for such data structures is that for small collection sizes, operations such as element search (or set `contains`) require comparable or even shorter time if it is implemented as linear search on an array as compared to a lookup in a proper hash table. However, using an array as the underlying data structure reduce significantly the memory footprint.

We studied adaptive collections that change the underlying data structure from an array (lower memory overhead but linear search) to a hash table (higher memory overhead, constant lookup time). For the hash tables, *hash* denotes an implementation which creates a bag of keys in case of hash function conflicts (similarly to JDK's HashMap implementation). The *openhash* version solves hash function conflicts by shifting the key placement to the next free position in the underlying table. Table 5.3 lists the three adaptive data structures and their transition types.

Table 5.2: Adaptive collection types studied in this work, their transition types, and the optimal transition thresholds (in terms of the collection size).

| Col. Variant | Transition | Threshold |
|---|---|---|
| AdaptiveList | $array \rightarrow hash$ | 80 |
| AdaptiveSet | $array \rightarrow openhash$ | 40 |
| AdaptiveMap | $array \rightarrow openhash$ | 50 |

In CollectionSwitch, such adaptive variants are considered as candidates for future instantiations only if the corresponding allocation context has identified that the previously created collection instances had widely ranging sizes.

Transition threshold of adaptive collections. Adaptive collections use local criteria to change its internal implementation, based on the collection size (number of elements). This criteria has a significant impact on the performance of such data structures. Transitioning from array to hash too soon may jeopardize the memory-benefits while even introducing unnecessary transitions to small collections, while having a large collection with array representation is ill-advised.

To create performance models of these data structures, we had to optimize and fix the transition thresholds for all studied variants. We used the lookup search as the scenario for finding this threshold, since our adaptive collections attempt to optimize element search.

We calculate the transition threshold by finding the collection size for which the cost of transition to a hash table would be surpassed by the cost of calling the lookup operation for every collection element. Figure 5.3 illustrates this method for the AdaptiveSet. The optimal thresholds for each adaptive collection are shown in Table 5.2.

## 5.4 Implementation

In this section, we describe the design decisions and implementation details of the CollectionSwitch. Our framework is composed by two components: the *performance model builder* and a *library*. The performance model builder creates performance models of collection variants via benchmarking, essentially calibrating our

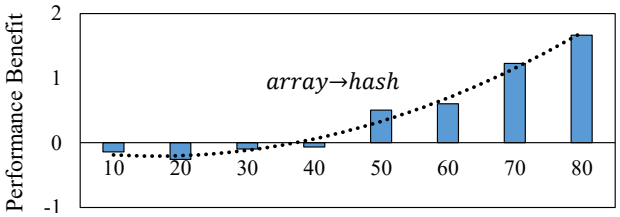



Figure 5.3: Transition threshold analysis of AdaptiveSet. The performance benefit is calculated subtracting the cost of transition from array→hash by the aggregated lookup cost for every element in the Set. The optimal threshold for transitioning from array to hash table is at collection size of 40.

adaptive framework to a specified hardware. The CollectionSwitch library exploits these performance models at runtime for adaptive selection of collection variants (Section 5.3.1).

### 5.4.1 Performance Models via Benchmarking

In any performance-related issue, the hardware configuration is a fundamental component of influence. Experiments in related work have shown examples of different hardware architectures yielding distinct best data structure for the same application [98, 178]. Hence, the architectural changes of the underlying hardware can make the data structure suboptimal as the workload of the application changes.

This substantiates the need for hardware-specific benchmarking and performance modeling as a prerequisite to optimization of collection selection. Another benefit of such benchmarking is uncovering the performance differences hidden by theoretical models such as the asymptotic analysis.

Considered Collection Variants Motivated by the findings presented in Chapter 4 we consider in this work multiple implementations of the most used collection abstraction types: Lists, Sets, and Maps. In order to have a compelling search space to explore, we select implementations from both JCF and alternative collection libraries. We consider implementations from Koloboke [112], EclipseCollections [60] and FastUtil [57] due to their good overall performance.

Additionally, we include implementations not provided by a collection library, such as the ArraySet/ArrayMap provided by Google HTTP Client [71] and Stanford

Table 5.3: Collection implementations identified as candidates for variants.

| Variant | Implemented by | Description |
|---|---|---|
| ArrayList | JDK | Array-backed list |
| LinkedList | JDK | Double-linked list |
| HashArrayList | Switch | ArrayList + HashBag for faster lookups |
| AdaptiveList | JDK, Switch | ArrayList on small sizes and HashArrayList for large sizes |
| HashSetMap | JDK | Chained hash-backed set/map |
| OpenHashSet | Koloboke, Eclipse, Fastutil | Open-address Hash-backed set |
| LinkedHashSet | JDK | Chained hash-backed with double-linked entries |
| ArraySet | Fastutil, Google, NLP | Array backed set/map |
| CompactHashSet | VLSI | Byte-serialized map for high memory efficiency |
| AdaptiveSet | NLP/Google, Koloboke | Array-backed on small sizes and Hash-backed on large sizes |
| HashMap | JDK | Chained hash-backed map |
| OpenHashMap | Koloboke, Eclipse, Fastutil | Open-address Hash-backed map |
| LinkedHashMap | JDK | Chained hash-backed with double-linked entries |
| ArrayMap | Fastutil, Google, NLP | Array backed set/map |
| CompactHashMap | VLSI | Byte-serialized map for high memory efficiency |
| AdaptiveMap | NLP/Google, Koloboke | Array-backed on small sizes and Hash-backed on large sizes |

NLP [76]. Those variants have a narrow best-case scenario, but offer a substantial improvement when used in the right circumstances. We show the variants used in this study in Table 5.3, in total CollectionSwitch is able to adapt a list to 4 different variants, and a map/set to 10 distinct variants.

Computing the Performance Models The CollectionSwitch uses performance models to guide its search for a better variant during program's execution. To compute the performance models, we run a set of benchmarks using a factorial experimental plan [126], designed to evaluate each collection variant in a wide scope of usage scenarios (see Table 5.4). Each usage scenario is composed of a single op-

Table 5.4: Factors and levels adopted in the empirical evaluation of collections.

| Factor | Levels/Categories |
|---|---|
| Abstraction | List,Set,Map |
| Library | JDK, Koloboke, EclipseCollections |
| Collection Size | [10,50,100,150,..,1000] |
| Scenarios | populate, contains, iterate, middle |
| Data Type | Integer |
| Data Distribution | Uniform |

eration executed on a range of collections size from 1 to 10K. To reduce the time of the benchmark, we only evaluate *critical operations*, i.e., operations that have in at least one variant a linear or above asymptotic cost ($O(n)$). Consequently, we evaluate collections when adding elements to the collection (*populate*), searching for an element (*contains*), traversing (*iterate*) and adding/removing an element in the middle (*middle*), which is linear on array and linked implementations.

We build the empirical model considering the `Integer` element data type. Albeit having an impact on the performance of some operations, we believe this impact will be dwarfed by the differences of performances caused by different collection implementations. The data distribution, on the other hand, can impact hash and sorted structures as it has a direct influence on element collisions. We only consider uniform distribution in this model.

We build the benchmark using the *CollectionsBench*, our collections benchmarking tool introduced in Section 4.4.2. We use the JMH native profiling method for collecting the execution time, and we use the GC profiler to retrieve the memory allocated and footprint required in each scenario. Each iteration runs for five seconds, executing the defined scenario continuously and returning the average of the measured performance indicators. We run 20 unmeasured iterations to achieve the steady-performance, and use the average results of 40 measured iterations in our performance model, a similar methodology used to evaluate collections performance in Chapter 4

Modeling collection performance. We model the cost of each critical operation as a polynomial function of the collection size $s$:

$$cost_{op}(s) = \sum_{k=0}^{d} a_k s^k$$

The coefficients are calculated using the least squares polynomial fit on the results of the benchmark. For this work we use polynomials of third degree ($d = 3$), as this choice provided the small residuals, while polynomials of higher degree did not increase the least-square fit significantly.

### 5.4.2 The CollectionSwitch Library

We designed the CollectionSwitch as an application library as opposed to a customized Virtual Machine (VM). This design choice was based solely on facilitating the adoption of our framework by developer teams. A modified VM would incur on a harsh constraint, as to use CollectionSwitch applications would be required to deploy using our personalized VM. The CollectionSwitch is open-source library and is available online[3].

The core component of our library is the instrumented version of a collection-allocation site, the *allocation context*. The allocation context creates collections and monitors a subset of the created instances to obtain workload data, characterizing the current usage scenario. When collections finish their life-cycle, typically when collected by the garbage collector, the workload data is passed to the allocation context. As described in Section 5.3.1, this initiates the performance analysis of collection variants. If a better variant is found, the allocation context switches the current implementation for future collection instantiations and starts another monitoring round.

Specifying an Adaptive Context. The CollectionSwitch is essentially a middle layer between the application and the collection libraries, collecting information and switching the adaptive allocation sites to the appropriate variant. The allocation context is implemented as a Java object, instantiated before the creation of

[3] https://github.com/DiegoEliasCosta/collectionSwitch

```
// Original allocation site
List<?> list = new ArrayList<>();
// Modified code with allocation context
static ListContext ctx = Switch.createContext(CollectionType.ARRAY);
List<?> list = ctx.createList();
```

Figure 5.4: Instrumenting collection allocation sites with allocation context.

the collections. The context creation is specified by the programmer via API and is exemplified in Figure 5.4.

We leave the scope of the allocation context in charge of developers. Developers may choose to use static or non-static allocation context, which drastically change the scope of collection objects monitored by the context. A static context is created as soon as the class is loaded in the class-loader and is kept alive until the end of the application [4]. The usage of static context greatly reduces the potential overhead incurred by the framework, and it is closely related to the concept of tuning the allocation site. However, a developer could use a non-static context if the collections behavior is dependent on the instance that creates it, moving the adaptive behavior from allocation-site to object instance granularity.

Monitoring the Collections Usage. Each allocation context collects metrics on a subset of created instances to characterize their overall collections usage. Example of metrics include the maximum collection size and the amount of critical operation calls. We analyze only a subset of created collections to avoid a potential overhead in case of a huge amount of collection instantiation in a short period of time. The size of this monitored subset is defined by the monitored *window size*, and can be parametrized by the developer. The monitored collections are created with an extra layer called *monitor*, a wrapper that logs the metrics in the context and forwards the collection logic to the proper implementation.

Analyzing the Collections Usage. A vital aspect of the CollectionSwitch implementation is when the allocation context should use the feedback to perform its

[4] In the rare cases where the class-loader is displaced during program's execution the context would also be removed from the application and thus stop the monitoring and adaptation.

transitions. In principle, a feedback should be used only when it provides a complete context, i.e., when collections have already ended their life-cycle. In practice, this delays the decision of the allocation context when collections are retained for too long in memory, hurting the context adaptivity. To address this we created the *finished ratio*, which defines the ratio of monitored collections that needs to be finished, before the context can take any decision. For instance, a *finished ratio* of 0.6 implies that the allocation context will only take action when at least 60% of the instances have finished their execution. It is important to note that we always use the whole set of metrics to analyze the collections usage, the ration only determines when the feedback should be analyzed.

To assess whether the collection object has finished its execution, the allocation context saves a `WeakReference` to the instance. As soon as the collection is eligible for garbage collection, this weak reference returns null, when asked for the referenced object. This method is more reliable and does not incur the substantial overhead caused by the `finalize` method [19].

We implement the analysis of the collections usage using a thread pool to analyze every collections context. A periodic task is scheduled to analyze collections metrics every interval of a parametrized (*monitoring interval*). To further reduce the overhead of CollectionSwitch, this thread pool can be assigned to a specific processor, hence shielding the impact of the analysis on the monitored application time.

### 5.4.3 Limitations

While developing our framework we have identified two important conceptual limitations of CollectionSwitch: 1) estimation error of our performance cost model, and 2) the potential increase of susceptibility to faults due to increased complexity. We elaborate more on each limitation in the following paragraphs.

Estimation errors. Adaptive approaches have the constraint of using fast and simplified models during variant search, and our simplified performance cost model has the following shortcomings. First, we use accumulated execution cost, which might hide the true behavior of collections in a real application. For example,

short-lived instances and collections executed in parallel can have distinct impact on the application's performance. Also factors such as memory locality and branch misprediction are not considered in our performance model and are two major factors for predicting collections performance. On the other hand, CollectionSwitch only needs accuracy sufficient to expose the performance *differences* between collection implementations. Using a more or fully accurate model might increase the runtime overhead and thus limit the benefits of the approach.

Increased susceptibility to faults. By introducing alternative libraries we enhance not only the search-space of collection variants, but also the complexity of programs using the CollectionSwitch. Our framework comes with new library dependencies, and increases the deployed application binary by approximately 20MB. Different scenarios might yield distinct collection implementations, increasing the chance of functional bugs and raising the cost of diagnosis. We mitigate this by selecting well-tested implementations, and by providing a detailed log system for tracing our framework events. A production-grade version of CollectionSwitch should be released with JCF and one alternative library implementation, in order to reduce the possibilities of bugs and the dependency list of our framework.

## 5.5 Evaluation

In this section we evaluate the effectiveness of CollectionSwitch on selecting data structures to optimize applications for better execution time or memory usage. Our framework is evaluated through two benchmark suites:

1. The *CollectionsBench* [43], a set of micro-benchmarks specifically designed to evaluate collections performance. We use this benchmark to observe CollectionSwitch monitoring overhead and to assess the gains in performance our framework could give in a scenario dominated by collections operations.

2. The *DaCapo* benchmark, a set of real-world application benchmarks [15]. In this experiment we effectively evaluate how CollectionSwitch can be used

Table 5.5: Selection rules $R_{time}$ and $R_{alloc}$ used in our evaluation.

| Rule | Improvement | Penalty |
|---|---|---|
| $R_{time}$ | $TC_{time}$ < 0.8 | – |
| $R_{alloc}$ | $TC_{alloc}$ < 0.8 | $TC_{time}$ < 1.2 |

to optimize real applications by adapting their collection to their specific demands.

We conduct all our experiments on a machine with the i7-2760QM 2.40GHz CPU and 8GB RAM under Ubuntu Linux 3.16.0-53 (64 bits).

To optimize applications for time and space dimensions we use two rules shown in Table 5.5. The rule $R_{time}$ target at optimizing execution time while $R_{alloc}$ focus on reducing memory allocation of our target applications and benchmarks. Note that $R_{alloc}$ comprises a maximum penalty allowed on the execution time, otherwise CollectionSwitch would only select array-backed implementations due to their low memory footprint.

For the whole experiment we use a *window size* of 100 instances, a level that showed a good compromise between fast analysis and stable transitions in our preliminary analysis. Furthermore, the *finished ratio* is set 0.6, that is, our framework defers the adaptation until we have 60% of monitored collections finished in the current monitoring round. We configure our dedicated thread to wake-up every 50ms (monitoring interval) to inspect and adapt each allocation-context configured in an application.

### 5.5.1 Micro-benchmarks

We extended the CollectionsBench benchmark to evaluate the CollectionSwitch through experiments covering scenarios dominated by a single collection operation (single-phased), and scenarios with the dominant operation varying over time (multi-phased). We follow the methodology [62] for evaluating the steady-state performance of Java programs. In particular, we run each test scenario with 20 unmeasured iterations for warm-up, followed by 40 measured executions.

Albeit having trained our models with the `Integer` element type, we evaluate our framework adapting collections holding `String` objects. Strings are a very commonly used data type (as shown in Chapter 4), and by changing the content type we attempt to appraise the robustness of the performance model.

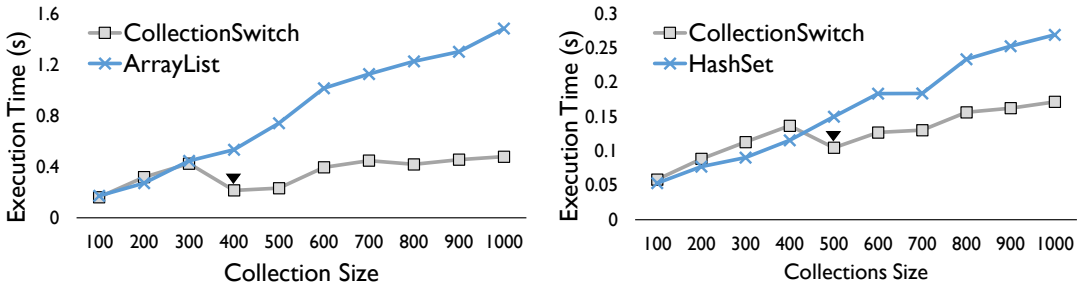
Single-Phased scenario. In this experiment, each scenario consists of creating and populating 100k collection instances, followed by 100 lookup searches (`contains()`). We focus on the lookup operations for this experiment as this operation have different asymptotic complexities on array/hash implementations and showcase an interesting trade-off between time and memory consumption.

Figure 5.5 shows the time performance against JDK implementations (ArrayList, HashSet, HashMap) for varying collections size. CollectionSwitch was able to select variants with better performance on all collection abstractions. In Figure 5.5a, the performance is gained by switching to a `HashArrayList` implementation. On sets and maps (Figures 5.5b and 5.5c) this performance was achieved by switching to the Koloboke OpenHash implementation.

In case of $R_{alloc}$, CollectionSwitch switches multiple times for both sets and maps. We present the performance in terms of allocated memory during our benchmark in Figure 5.6. On small collections ($size < 400$), FastUtil OpenHash implementation is selected (the most memory efficient variant). For medium size collections, the time penalty for using FastUtil lookup crosses the threshold established by $R_{alloc}$, and EclipseCollection is selected. A yet better implementation (Koloboke) is identified and used when the collection size reaches 700.

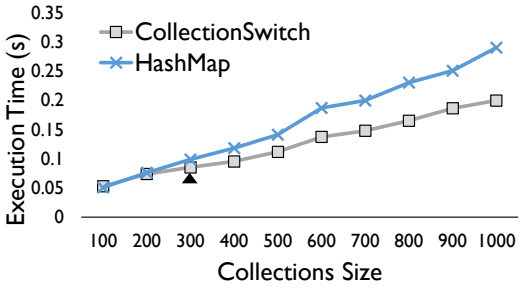
Multi-Phased scenarios. We craft this experiment to evaluate the effectiveness of our dynamic collection adaptation in a controlled multi-phased scenario. Each iteration of this experiment is comprised of the creation and population of 100k instances followed by an execution of 100 operations. Every five iterations we change the operation type resulting in the phases depicted in Figure 5.7.

In our experiments our framework switched to the expected best-fit implementation for all phases except for the phase "search and remove". Here the `HashArrayList` instead of the optimal `ArrayList` was used. We attribute this to a limitation in our performance model, our model assumes that cost of removing an




(a) Time comparison against JCF ArrayList

(b) Time comparison against JCF HashSet



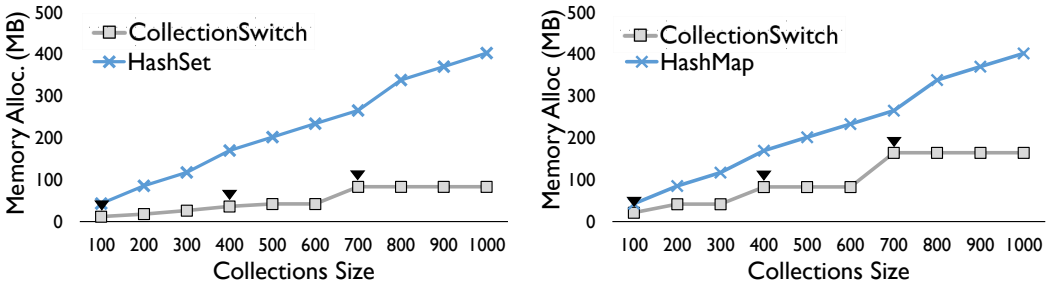

(c) Time comparison against JCF HashMap

Figure 5.5: CollectionSwitch time performance comparison on populating 100k collections and executing 100 random lookups (`contains`), evaluated with the $R_{time}$ selection rule. The marker indicates a size where our framework performed a switch to a different variant.

element by index is identical on both variants. In reality, `HashArrayList` implementation is slower as it searches on both hash and array structures.

### 5.5.2 Evaluation on Real Applications

DaCapo is a benchmark suite [15] comprised by a set of 14 open-source, real-world Java applications, carefully created by researches to be used as a tool by the community. Each benchmark contains realistic and non-trivial memory-intensive workloads, and for this particular reason is widely used as an evaluation tool for a variety of scientific studies. The latest DaCapo version (dacapo-9.12) contains 14 benchmarks from which we use the following five: *avrora*, *fop*, *h2*, *lusearch* and *bloat* (2006 version). We select this subset as inefficiencies related to collections were previously reported [144, 158, 178] on those projects.




(a) Allocation comparison against HashSet (b) Allocation comparison against HashMap

Figure 5.6: CollectionSwitch memory allocation performance comparison on populating 100k collections and executing 100 random lookups (`contains`), evaluated with the $R_{alloc}$ selection rule. The marker indicates a size where CollectionSwitch performed a transition to a different variant.

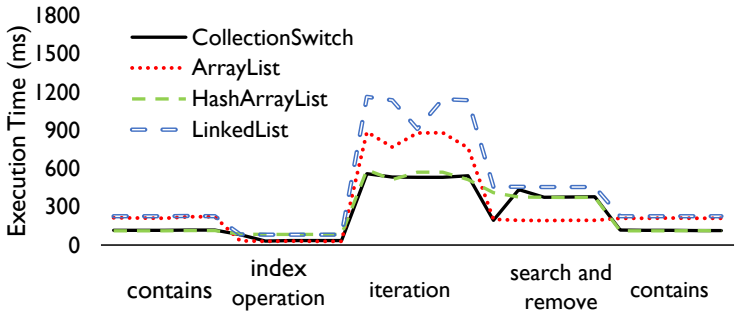



Figure 5.7: CollectionSwitch time performance on a multi-phased scenario, evaluated with $R_{time}$. As each phase is comprised by one dominant operation and hence has a different variant as the most optimal choice. CollectionSwitch switches to the best-fit variant in almost but one scenario.

CollectionSwitch requires modifications of existing code to use the adaptive allocation contexts. To limit this effort, we modify only allocation-sites that yield at least one thousand instances and that comply with JCF interfaces (possibly with little refactoring effort). To this end, we first monitor a regular benchmark execution and rank the allocation sites by the number of generated instances.

To compare the effectiveness of the full framework against the benefits of a simple adaptivity at instance-level we use two modified versions of DaCapo. In the first version (*FullAdap*), each target allocation site is modified to use our allocation context, providing full framework capabilities. In the second version (*InstanceAdap*), the target allocation sites are simply changed to always instantiate

an adaptive variant (e.g., ArrayList → AdaptiveList), i.e., without the capability of selecting the collection types through the analysis of previous instances.

In this experiment we run the original and the two modified DaCapo benchmarks 35 times with a maximum heap size of 1 GB (first 15 runs are discarded as warm-ups).

Table 5.6: Results of our approach on DaCapo benchmarks. Section "Original Run" reports the execution time ($T$) and the peak of memory consumption ($M$) of the original Dacapo run. Section "Full CollectionSwitch" shows the results for the full optimization level (*FullAdap*) under both selection rules $R_{time}$ and $R_{alloc}$. We present the gain/loss percentages for the significant differences (Tukey HSD test [126]) compared to the original run (positive values are better). Non-significant differences are reported as –.

| Bench | Original Run | | Full CollectionSwitch (*FullAdap*) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $R_{time}$ | | | | $R_{alloc}$ | | | |
| | T(s) | M(MB) | $T_1(s)$ | | $M_1(MB)$ | | $T_2(s)$ | | $M_2(MB)$ | |
| avrora | 4.1 | 72.4 | 4.2 | – | 72.1 | – | 4.4 | -7% | 65.4 | **+10%** |
| bloat | 30.3 | 96.7 | 28.9 | – | 96.9 | – | 26.6 | **+12%** | 89.4 | **+8%** |
| fop | 0.5 | 53.4 | 0.5 | – | 57.0 | -7% | 0.5 | – | 53.9 | – |
| h2 | 40.1 | 509.0 | 38.3 | **+6%** | 508.7 | – | 44.6 | -11% | 470.1 | **+8%** |
| lusearch | 3.6 | 282.4 | 3.1 | **+15%** | 269.4 | **+5%** | 3.4 | **+6%** | 268.2 | **+5%** |

TIME IMPROVEMENT ($R_{time}$) FOR *FULLADAP*. The largest improvement of the execution time (15%) was obtained for *lusearch*. The dominant transition in *lusearch* was performed on map variants, as most of its HashMap instances held less than 20 elements, and were replaced by AdaptiveMap and Koloboke OpenHashMap. Thus, as a side effect, CollectionSwitch also reduced the memory peak consumption of *lusearch* by 5%. Benchmarks *fop* and *h2* showed similar characteristics of collection transitions. Both of them have allocations sites that extensively instantiate lists exposed to large amount of lookup calls. CollectionSwitch has correctly transitioned them to AdaptiveList ($array \rightarrow hash$). This transition improved the execution time of *h2*, but provided no significant improvement for *fop*.

MEMORY IMPROVEMENT ($R_{alloc}$) FOR *FULLADAP*. In case of $R_{alloc}$, CollectionSwitch managed to reduce the peak of memory consumption of most of the applications.

Typically, HashSet and HashMap were replaced by adaptive and open-hash variants. Interestingly, the *bloat* benchmark had a lower execution time in this case ($R_{alloc}$) than when aiming at the time reduction ($R_{time}$). We conjecture that a reduction of memory usage implied in a better cache utilization and lower Garbage Collection time.

Comparison of *FullAdap* and *InstanceAdap*. Table 5.7 shows the comparison of the original runs of DaCapo benchmark and the limited version of CollectionSwitch with only adaptive instances (*InstanceAdap*). Compared with the full-version presented in Table 5.6, the simpler version *InstanceAdap* featured comparable (but not better) improvement grades for memory usage as the full framework under the rule $R_{alloc}$. However, the *InstanceAdap* version failed to achieve any improvement on the execution time, especially in comparison with the full CollectionSwitch under the $R_{time}$. These results indicate that allocation-site adaptivity is essential for improvement of execution time Such mechanism help to consider multi-dimensional criteria (memory and *time*), which prevents the uncontrolled performance degradation which might be introduced by adaptive collection.

Table 5.7: Results of our approach on DaCapo benchmarks considering only pure adaptive instances as opposed to the full version of CollectionSwitch. Section "Original Run" reports the execution time ($T$) and the peak of memory consumption ($M$) of the original Dacapo run. We present the gain/loss percentages for the significant differences (Tukey HSD test [126]) compared to the original run (positive values are better). Non-significant differences are reported as –.

| Bench | Original Run | | CollectionSwitch (*InstanceAdap*) | | | |
|---|---|---|---|---|---|---|
| | T(s) | M(MB) | $T_3(s)$ | | $M_3(MB)$ | |
| avrora | 4.1 | 72.4 | 4.4 | -7% | 64.9 | **+10%** |
| bloat | 30.3 | 96.7 | 29.5 | – | 89.6 | **+8%** |
| fop | 0.5 | 53.4 | 0.5 | – | 53.8 | – |
| h2 | 40.1 | 509.0 | 44.9 | -12% | 493.2 | **+3%** |
| lusearch | 3.6 | 282.4 | 3.5 | – | 275.7 | **+2%** |

Common Transitions. Table 5.8 shows the most frequently selected transitions (by application and by selection rule). Only 11 out of the 25 possible variants were

Table 5.8: Most commonly performed transitions (AL = ArrayList, LL = LinkedList, HS = HashSet, HM = HashMap).

| Benchmark | $R_{time}$ | $R_{alloc}$ |
|---|---|---|
| avrora | HS → OpenHashSet | HS → AdaptiveSet |
| bloat | LL → AL | HS → AdaptiveSet |
| fop | AL → AdaptiveList | AL → AdaptiveList |
| h2 | AL → AdaptiveList | HS → ArraySet |
| lusearch | HM → OpenHashMap | HM → AdaptiveMap |

used in our evaluation. This indicates that a small set of collection variants might be sufficient to cover most of the real cases found in the applications. Moreover, in most allocation sites our framework replaced the original implementation. We conjecture that our approach offers unexploited potential for improving performance in applications.

**How effective was CollectionSwitch in selecting collections for real-applications?**

CollectionSwitch managed to positively impact the execution time and the peak of memory consumption in most evaluated applications. The time and memory improvement vary considerably by application and selection rule, with the most substantial speedup observed in *lusearch* (15%) and the highest memory peak reduction observed in *avrora* (10%).

### 5.5.3 Overhead of the CollectionSwitch Framework

To evaluate the time and space overhead incurred by our approach on DaCapo benchmark, we compare the performance statistics of the unmodified benchmarks against such statistics under CollectionSwitch (*FullAdap*) with disabled optimization actions. We achieve the latter by setting an impossible selection rule (required 1000x time/space improvement for a transition). We found no significant difference (Tukey HSD) in the execution time in any DaCapo benchmark when using our standard configuration.

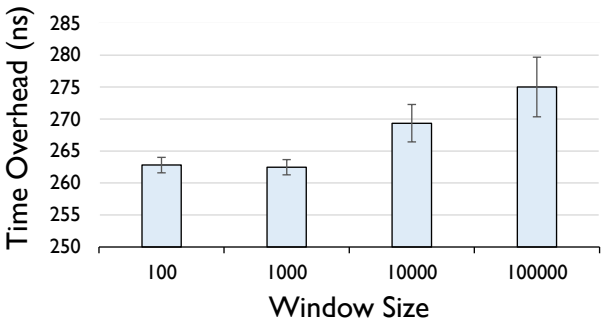



Figure 5.8: Overhead of analyzing the collections metrics by *window size*.

Additionally, we run a micro-benchmark to evaluate the cost of analyzing a set of collection metrics while varying the *window size* from 100 to 100k. Figure 5.8 confirms that the overhead is negligible (<285ns), and can be easily amortized by a multi-threaded environment.

Regarding the memory overhead, each allocation context has a footprint of approximately 1KB. As the amount of allocation context is limited by the amount of allocation sites, we consider the memory overhead of the CollectionSwitch non-significant for real applications.

## 5.6 Summary of the Chapter

In this chapter, we presented CollectionSwitch, an application-level framework for efficient collection adaptation. It **selects at runtime collection implementations in order to optimize the execution and memory performance of an application**. Unlike previous works, we use workload data on the level of collection allocation sites to guide the optimization process. Given a set of performance rules, our framework identifies allocation sites which instantiate suboptimal collection variants, and selects optimized variants for future instantiations. As a further contribution, we propose adaptive collection implementations which switch their underlying data structures according to the size of the collection.

We implement this framework in Java and **demonstrate the improvements in terms of time and memory peak consumption across a range of benchmarks**. While implemented for collections, the concept of exploring allocation-

site regularities can be applied to different programming languages and domains, to optimize the execution time and memory consumption of applications with low overhead.

# 6 A Decision Support Framework for Effective Parallelization of Java Streams

The Java Stream library aims at facilitating element processing while providing friendly support for parallelization. Object processing can be parallelized with a simple switch of method calls. The performance benefits of this parallelization, however, require careful deliberation from developers, that need to account for a myriad of different factors through manual analysis, potentially leading to suboptimal choices and performance issues. In this chapter, we address this problem by introducing a framework that combines machine learning models with static and dynamic analyses of the target application to identify stream pipelines that can be effectively parallelized for better runtime performance of the application.

Hence, in this chapter, we make the following contributions:

I An approach for identifying streams that can benefit from parallelization, by leveraging machine learning models and synthetically generated stream benchmarks.

II An implementation of this approach in the form of a decision support framework that monitors the application workload and reports to developers opportunities for optimizing Java streams.

III An evaluation of the accuracy of our models on three benchmark suites from real applications.

This chapter is partially based on the following manuscript under-revision.

D. Costa, J. P. Sandoval, and A. Andrzejak. "Leveraging Machine Learning Models for Effective Parallelization of Java 8 Streams". In: *Automated Software Engineering*. Under revision

## 6.1 Introduction & Motivation

As introduced in Chapter 2, the Java Stream provides a concise API for processing objects and primitives, with a programmer-friendly method for parallelization: processing streams in parallel requires a simple call to `parallel` in a stream pipeline. The benefits of parallelizing streams, however, require careful deliberation from developers [110]. As evidenced by some of the most viewed stream questions in Stack Overflow [118, 121, 150], developers struggle on understanding when to use parallel streams for better performance.

Unfortunately, the JVM cannot automatically parallelize streams. The problem of fully unguided automated multi-core parallelism remains a grand challenge [151], and the JVM takes a conservative approach. As it cannot guarantee the correctness neither the performance benefits of the parallel code [110], the JVM leaves to developers the decision to use parallelism in their stream processing. In turn, developers need to account for the following factors to decide whether using parallel streams is beneficial or not in terms of performance:

1. The **number of elements** and **the amount of computation** performed on each element. In essence, the task of processing the entire stream needs to be large enough to justify the need for parallelization and compensate the costs of splitting and combining each sub-tasks running in parallel.

2. The **side-effects** of the behavioral parameters. Stateful lambda expressions [89] might introduce external thread contention that may jeopardize the benefits of parallelization. For instance, behavioral parameters that perform I/O operations are unlikely to benefit from stream parallelization.

3. The **stream source** determines how efficient a stream of elements will be partitioned. Every stream source in a pipeline has to provide a `Spliterator` [161] data structure, which defines how a source can be split and tra-

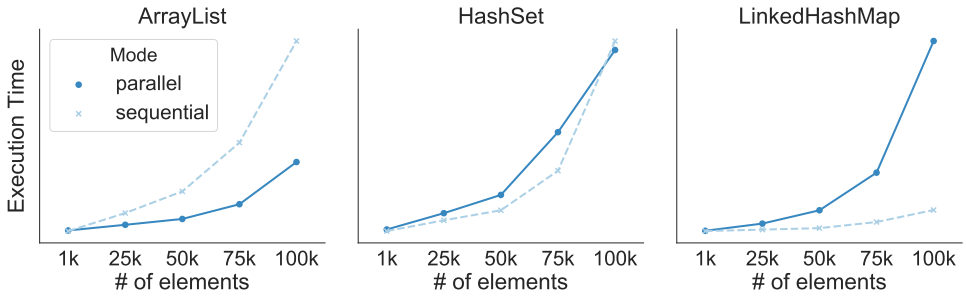
versed. Furthermore, the source data type may enforce a specific order of the elements (e.g., ArrayList guarantees the insertion order), which in combination with short-circuit pipeline operations (such as `findFirst` or `limit`) may lead to early termination of pipeline processing. This can, in turn, profoundly influence the benefit of potential parallelization. As the order of the source elements potentially needs to be preserved in the output, the costs of coordinating the split and merge would rarely pay-off when processing a pipeline.

4. The **stream output** determines the costs of combining results from parallel tasks. For example, combining partial sums in parallel `sum` is fast. On the other hand, merging sets is more expensive as it might involve copying large amounts of data.

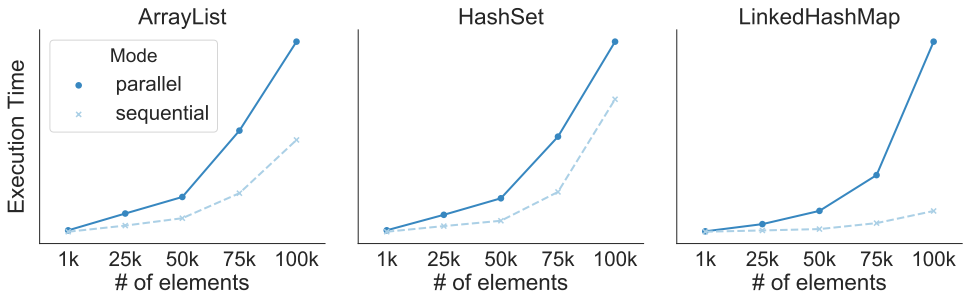
5. Each **pipeline operation** also influences the performance of parallel streams. Some operations such as `distinct` needs to process the entire sequence of elements before producing any result, which may penalize the parallelization. A short-circuit operation mentioned above finishes as soon as a criterion is met but still has to coordinate with executing threads, jeopardizing the performance gains of parallel execution.

6. Aside from factors related directly to the stream pipeline and stream source, factors like the number of CPUs in the **underlying hardware**, the data locality of the source, and the cost of managing the thread pool might also affect the performance gains from parallel streams.

To illustrate the effect of stream source and the pipeline operations, we show in Figure 6.1 the performance of two pipelines under different stream sources, in both execution mode. In Figure 6.1a, using parallel streams with ArrayList is highly beneficial for performance, while streams created from HashSet showed performance benefits or parallelization only with streams containing more than 100 thousand elements. The streams from LinkedHashMap, however, show no trend of being beneficial in parallel. The pipeline containing the `distinct()` operation showed no benefits of parallelization for all stream source types, up to 100 thousand elements (Figure 6.1b).




(a) `filter().collect()`




(b) `distinct().collect()` pipeline

Figure 6.1: Execution time comparison (lower is better) of two stream pipelines with distinct stream sources in both parallel and sequential mode. Results were obtained with streams containing `Integer` elements, and filtering is performed with a trivial behavioral parameter (`x -> x.hashCode() % 2 == 0`). The tests were performed on a machine with E5-1660-3.3GHz CPU, with 6 physical cores.

Table 6.1: Example of Java projects in GitHub that make extensive use of stream pipelines, ranked by the number of pipelines declared in its code-case (test files included).

| **Project** | **# of stream pipelines** |
|---|---|
| OpenJDK | 4,818 |
| Cyclops | 4,705 |
| Intellij-Community | 3,154 |
| Onos | 2,185 |
| Sonarqube | 2,164 |

Furthermore, large software projects that make extensive use of the Stream API have thousands of stream pipelines defined in their source-code (Table 6.1). For instance, the project Intellij-community has more than 3 thousand stream pipelines

defined in its code-base. This makes it practically unfeasible for developers to analyze each pipeline with performance tests that reflect the real workload of the application, to foster possible benefits of parallelization.

All the factors mentioned above and their potential interactions pose a non-trivial challenge to developers to decide whether parallelization is beneficial (and even semantically equivalent), or not. In this chapter, we address only the performance aspect and focus on the following problem.

Problem definition Given a specific use-case and environment, identify an optimal *execution mode* (sequential or parallel) for each stream pipeline to optimize the application's runtime performance. The execution mode is decided by classifying each stream pipeline into one of the following two types:

- Sequential-optimal: a stream pipeline that has faster runtime performance processing streams in the sequential mode (`stream()`).
- Parallel-optimal: a stream pipeline that has faster runtime performance processing streams in the parallel mode (`parallelStream()`).

## 6.2 Related Work

In this section, we categorize the related literature into 1) stream programming model in Java, 2) adapting the Java Stream library, and 3) supporting parallel streams.

### 6.2.1 Stream Programming Model in Java

The notion of stream processing programs has been around for decades [2]. In the stream programming model, a program is a collection of independent filters (functional unit) communicating through uni-directional data channels. This model lends itself naturally to concurrent and efficient implementations on modern multiprocessors [162]. After the popularization of streaming media in the early 2000s, the research community has devoted more attention to modern languages and compilers for stream processing applications [1, 23, 168, 184].

In Java, approaches for distributed stream processing were proposed a decade prior to the introduction of the Java Stream library [162, 168]. StreamIt [168] provide a high-level data flow abstractions for stream programming, represented with autonomous actors that communicate through FIFO data channels, and a compiler infrastructure that generated both Java and native code. More closely related to the Stream Library, was the approach proposed by Spring et al. [162]. Authors propose an extension to Java that enabled the development of programs using the stream programming model, with traditional object-oriented components.

Furthermore, frameworks for Big Data processing are notorious for using the stream programming model, typically by implementing the Map-Reduce algorithm. The most popular Apache Hadoop [175] and Apache Spark [183] provide extensive support for Java developers.

### 6.2.2 Adapting the Java Stream Library

A substantial body of work aims at improving the current specification and implementation of Java Stream library to improve data locality, or provide support for distributed and real-time processing.

Chan et al. [28] studied the data locality of streams for processing very large datasets. While loading such datasets from disk, the authors showed that the lack of thread affinity and data locality seriously degraded the performance of standard Java streams. Authors proposed an approach called JUNIPER, that allows applications to tailor their levels of parallelism, thread affinity, and to better exploit data locality to improve the performance of Java streams when processing Big Data systems.

Su et al. [166] proposed a new abstraction layer built on top of Java Streams, that supports the execution of stream queries over a set of distributed machines. By reusing patterns and concepts of standard Java streams, their approach provided a friendly way of integrating distributed queries into Java programming language.

Chan et al. [27] presented a set of extensions to Java Stream library to support the processing of large datasets. Authors point a set of issues with the current Java Stream programming model, that makes it suboptimal for distributed processing, such as the constraint of using only in-memory sources (e.g., collections).

Similarly, to [166], a prototype is proposed that enables distributed collections, distributed streams, and a new set of eager operations for distributing data across multiple computers.

Biboudis et al. [14] address the lack of extensibility of current streams libraries. In most implementations (Java included), stream libraries are not flexible enough to enable the customization of the semantics of pipeline operators. For instance, developers have to change the library code to add new combinators or to log intermediate operations. Authors propose the usage of parametrized pipelines that can be infused with customized semantics. This new extensible stream is shown to have no negative impact on performance.

Furthermore, Mei et al. [123] conduct a study on the feasibility of using Java streams for the Real-Time Specification (RTSJ) [4]. Authors conclude that the current framework design cannot supply a pool of real-time worker threads, due to the lack of flexibility of the Fork/Join framework [109], used to process parallel streams.

Hayashi et al. [81] leverage machine learning models to extract a set of performance heuristics for computing streams of primitives in a GPU. Customized JIT compilers [93, 182] have also been proposed that optimizes parallel streams APIs for GPU execution. All approaches focus on the `IntStream` use-case alone, as GPU processing is a better fit for array processing, and do not tackle the more general use-case of processing objects with Java streams.

### 6.2.3 Supporting parallel streams

To the best of our knowledge, there is a surprising lack of studies and approaches that aim at helping developers on using parallel streams.

To guide practitioners in deciding when to use parallel streams, Lea [110] created a simple model called the *NQ model*. Given an efficiently splittable source and an independent behavioral parameter, the NQ model consists on multiplying $N$ (number of elements) by $Q$ (cost per element) and if $NQ > 10000$ the parallelization is beneficial. The cost per element can be roughly estimated, in a tiny and fast function such as `x -> x + 1`, $Q$ can be assigned to one. While this model provides an easy thought model to be used in practice, it has a non-trivial assumption of

efficiently splittable source and may yield sub-optimal solutions by neglecting all the remaining factors described in Section 6.1.

Khatchadourian et al. [103] proposed a tool for safe stream refactoring that uses a novel typestate analysis [165] to identify when it is safe to parallelize, by taking into account the possible lambda side-effects and stream source type. While this work is the most closely related to this chapter, our approach targets an orthogonal aspect of stream parallelization, by focusing on the performance benefits as opposed to the safeness of the refactoring. While the work of Khatchadourian et al. [103] aims at identifying whether a pipeline **could be refactored**, we focus on the performance aspect of stream parallelization, effectively identifying whether a stream pipeline **should be refactored**. A practical method can be devised by combining both approaches into a cohesive tool, tackling both the safeness and the performance benefits of using parallel streams.

## 6.3 A Decision Support Framework for Effective Stream Parallelization

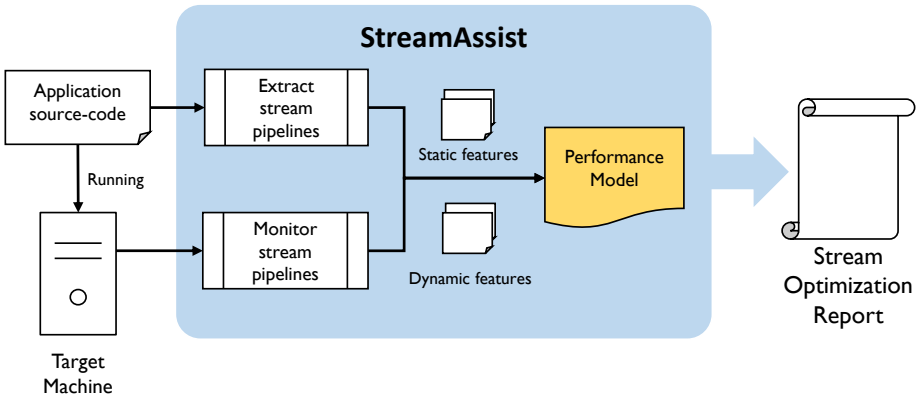



Figure 6.2: Overview of the StreamAssist framework.

To address the problem of finding the optimal execution mode for stream pipelines, we devise a decision-support framework that analyzes the application at runtime and reports back to developers the stream pipelines that are likely to benefit

from sequential or parallel processing. To accomplish this, our approach extracts all stream pipelines in the application source code and monitors the streams created during the runtime of the application, to construct a *pipeline profile*. This profile is then used as an input of a *performance model* that classifies whether the stream pipeline is sequential or parallel-optimal.

In the following, we detail how the stream pipelines are extracted from the source-code (Section 6.3.1), how we monitor streams at runtime (Section 6.3.2), and how pipeline profiles are fed to machine learning models to classify stream pipelines (Section 6.3.3).

### 6.3.1 Extracting stream pipelines

We build a tool using the support of the Java Parser library [95] to parse the Java code and extract each stream pipeline from the application source code. To identify a stream pipeline, our parser analyzes the return type of every method call in the Java code of the target application. If the return type inherits from the base class for all stream objects (`java.stream.BasicStream`), the tool considers the method call an entry-point of a stream pipeline. Every subsequent method call is recorded until a terminal operation is identified, marking the end of the pipeline.

The parser extracts stream pipelines defined within a method scope. Streams that are passed as a parameter to other methods are currently not considered for our prediction. Our tool records the call-site of the pipeline (class name and line number) and the sequence of stream operations that compose the pipeline.

### 6.3.2 Monitoring Stream Pipelines

Aside from the stream pipeline, which can be extracted statically, the factors that may influence the performance of stream pipelines can only be obtained at runtime. We develop a customized Java agent within StreamAssist, with the support of the ByteBuddy library [177] to monitor the streams created on each stream pipeline during application runtime. A Java agent is a software that uses the Java Agent framework [88] to instrument programs running in the JVM by modifying the bytecode of their methods.

Our agent monitors the creation of streams by instrumenting the class responsible for creating streams, the `java.streams.StreamSupport` [90]. This class provides a low-level utility method for creating and manipulating streams and constitutes a single entry-point for our data collection. Naturally, this class is also used by third-party libraries and the JVM itself to create stream objects. We filter the streams created by the target application code by inspecting the package of the caller in the stack trace. As inspecting the stack-trace can incur on substantial monitoring overhead, we search only the five first frames in the stack, as all standard methods for creating streams in Java reach to `StreamSupport` within at most five method calls.

After identifying a stream created by the target application, our tool records: the number of elements $e$ and the type of the data source $s$. This info is grouped by pipeline call-site (class and line number). To avoid the expensive overhead of tracing every stream creation, we only count the unique occurrences of the pairs $(e', s)$, with $e' = e/100$, essentially grouping the recorded stream size in groups of hundreds.

The information collected is serialized either when the agent is uninstalled from the JVM - characterizing the end of the data collection; or at the end of the JVM execution. Our agent subscribes for a JVM shutdown hook [86], a mechanism provided by JVM to execute methods when the Java process is finalizing.

### 6.3.3 Predicting the Pipeline Optimal Execution Mode

A stream pipeline can generate numerous streams during the program's life-cycle. Consequently, the pipeline profile is comprised of multiple stream profiles, each potentially differing in terms of the number of elements and source type. We group the pipeline extracted statically with the information collected at runtime, constructing a pipeline profile. Given a stream pipeline $p$, we denote the pipeline profile as $X_p = \{x_{p1}, x_{p2}, ..., x_{pn}\}$, where $x_{pi}$ is the profile of the i-th stream.

We formulate the problem of finding the optimal execution mode as a classification task of a pipeline profile $X_p$ to an execution mode $y$, where $y \in Y = \{seq, par\}$. We denote the time of running a pipeline $p$ in the execution mode $y$ under the profile $X_p$ as $T_y(X_p)$. Hence, in general terms the goal of our perfor-

mance model is to find a function $f : X_p \rightarrow y$ such that $T_y(X_p) < T_{y'}(X_p)$, with $y'$ denoting the opposite execution mode to $y$. We detail in the following section, the methodology used to create the performance model.

## 6.4 A Benchmark-driven Model Builder

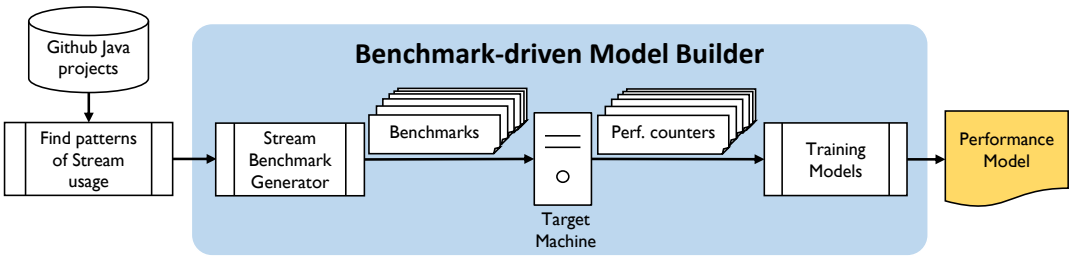



Figure 6.3: Overview of our approach for building the stream performance model.

The model builder has the goal to create a model that accurately predicts the performance of stream pipelines in both parallel and sequential mode. We approach the modeling of stream performance from an empirical viewpoint: we use benchmarks to evaluate the most commonly used pipelines in open-source software under a variety of workloads and usage scenarios (Section 6.4.1), and use such patterns to design a benchmark generator (Section 6.4.2). The benchmarks generated are run in both sequential and parallel modes. The results are fed into machine learning models, trained to predict the optimal execution mode for each stream (Section 6.4.3), and aggregated into a prediction for a stream pipeline. The overview of our model builder is depicted in Figure 6.3.

### 6.4.1 Finding Patterns of Stream Usage

We mine repositories to identify the most commonly used stream operations and pipelines in open-source Java projects. In particular, we query the GitHub repository for the 10,000 most starred Java projects, aiming at analyzing a range of very popular to less popular projects.

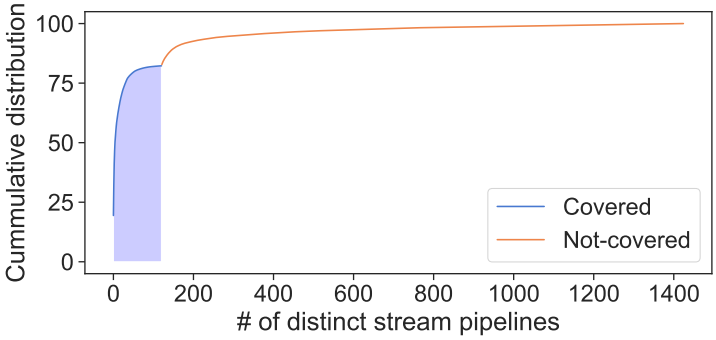



Figure 6.4: Cumulative distribution of the 1,426 unique pipelines found in the dataset. The highlighted area corresponds to the 120 pipelines selected to compose our training set, which covers 82% of all found stream pipelines.

We use the same parser used to extract the static features of stream pipelines (Section 6.3) to extract the pipelines from our dataset of projects. Recall that our tool extracts stream pipelines defined entirely within a method scope. Hence the number and variety of pipelines observed in this section are a lower bound of what projects may use in reality. For our purposes, however, the statistics can still be used to guide our efforts towards optimizing the more commonly defined stream pipelines.

Table 6.2: Descriptive statistics of stream pipelines extracted from the top 10k most-starred Java projects in GitHub.

| **Statistics** | **#** |
|---|---|
| # of Projects that use Java streams | 1,307 |
| total of # stream pipelines found | 156,096 |
| # of unique pipelines | 1,426 |

We present a preliminary analysis of this investigation in Table 6.2. Out of ten thousand projects, 1,307 projects (13%) have at least one instance of a stream pipeline defined in its source code. In total, more than 156 thousand instances of stream pipelines were retrieved by our tool. Disregarding the behavioral parameters, we identify only 1,426 unique pipelines, i.e., 1,426 unique sequence of pipeline operators.

Table 6.3: Top 10 most frequently used stream pipelines in the most starred 10k GitHub Java projects

| **Stream Pipeline** | **#** | **%** | **# Projects** |
|---|---|---|---|
| map-collect | 17571 | 11.2 | 825 |
| filter-collect | 6513 | 4.1 | 612 |
| forEach | 5878 | 3.7 | 535 |
| collect | 5536 | 3.5 | 539 |
| anyMatch | 3548 | 2.2 | 445 |
| filter-forEach | 2834 | 1.8 | 406 |
| filter-map-collect | 2210 | 1.4 | 421 |
| filter-findFirst | 1368 | 0.8 | 315 |
| map-forEach | 1365 | 0.8 | 275 |
| map-toArray | 1290 | 0.8 | 260 |

The distribution of unique pipelines occurrences is presented in Figure 6.4, which shows that the usage of stream pipelines is highly concentrated on a small subset of pipelines. The combinations of the intermediate operators `map`, `filter` with the terminal `collect` make approximately 20% of the all declared pipelines (see Table 6.3). Notably, pipelines with zero and one intermediate operators make the majority of all occurrences. We decided to model all combinations of stream pipelines with zero and one intermediate operation, to include all operations at least once in our performance model. Furthermore, we include the most used pipelines with two intermediate operations, until we reach 120 pipelines that account for 82% of all pipelines retrieved from open-source Java projects.

### 6.4.2 Stream Benchmark Generator

To evaluate the performance of streams under both execution modes, we build a benchmark suite called *StreamBench*. StreamBench can generate benchmarks accounting for all combinations of stream operations, under a variety of workload parameters. Table 6.4 details the factors considered in the current version of StreamBench. Note that these parameters are further combined with stream pipelines with a different number of operations. As described in Section 6.4.1, we configure our StreamBench to generate benchmarks for 120 distinct pipelines.

Table 6.4: Factors evaluated by StreamBench benchmark suite. The column # of levels shows how many different values are considered in StreamBench.

| Factor | Example of levels | # of levels |
|---|---|---|
| # of elements | 100, 1k, 10k, 50k, 100k | 5 |
| Source Type | HashMap, Array | 11 |
| Element Type | Integer or int | 2 |
| Intermediate Op | `filter()`, `map()` | 6 |
| Terminal Op | `collect()`, `toArray()` | 12 |
| Workload Level | trivial, light, medium, heavy | 4 |

Our benchmark simulates the behavioral parameter by artificially consuming CPU cycles when processing each element, through the call of `consumeCPU` [17] from the `Blackhole` object of the JMH infrastructure. As shown in the NQ Model, the larger the processing time per element, the more likely the pipeline can benefit from parallelization. However, at the current stage of our approach, we can not extract with reasonable precision and low-overhead the performance of behavioral parameters on target applications. Hence, we train our models with the lowest workload-level. This translates to essentially taking a conservative approach and favoring sequential streams when predicting the optimal execution mode of a pipeline.

We build the framework upon the Java Micro-benchmark Harness (JMH) [136]. Each benchmark is configured with 20 warm-up iterations to reach the steady-state of the JVM, and 10 measured iterations (in both sequential and parallel mode). This entire process is forked twice to mitigate the effects of external influence.

### 6.4.3 Training Machine Learning Models

Training set. For each generated benchmark (Section 6.4.2) we record the combination of factor levels shown in Table 6.4. This information encoded using one-hot-encoding for multi-category factors (e.g., stream source), and values are normalized to the range of [0, 1]. This preprocessing yields a vector with 31 features per benchmark. Each vector is complemented with the execution time observed in sequence and parallel modes, together with a label (for classification) derived from the execution times of the respective benchmark. In total, we obtain a labeled data

set with 8,135 elements. A machine with E5-1660-3.3GHZ CPU needs eight days to execute all benchmarks.

Machine Learning Models. Given a pipeline $p$ and a stream profile $x_p$, we denote by $t_{seq}(x_p)$ and $t_{par}(x_p)$ the execution time of processing the stream sequentially and in parallel, respectively. We train two types of models:

1. Classification: We train each model to classify each stream pipeline as either sequential-optimal or parallel-optimal. A pipeline is considered parallel-optimal if $t_{par}(x_p) * 1.05 < t_{seq}(x_p)$. As parallel streams consume more system resources, we consider the 5% to be the bare minimum to apply parallel execution mode.

2. Regression: We train each model to predict $t_{seq}(x_p)$ and $t_{par}(x_p)$. The predicted performance will be used to quantify the expected performance improvement of the optimal execution mode against the sub-optimal one.

We use the data set from the synthetic benchmarks described above to compare machine learning models against two baselines and to select the best-performing models. The two baselines are: a stratified model (Baseline I), that uses the distribution of labels for the prediction; and the NQ model described in Section 6.2. Since we train the models with benchmarks at the lowest workload level, we assign the amount of work to $Q = 1$. This is the only method documented and currently used by developers. Hence our model needs to outperform the NQ model to be useful in practice.

Table 6.5 reports the model accuracy over 5-fold cross validation for the best ML estimator types. Our results show that conventional machine-learning models (from Scikit-Learn [146]) such as Decision Tree [22] and Multi-Layer Perceptron (1 hidden layers with 100 units) [82] without further hyper-parameter tuning are able to learn a good classification model for our stream problem, considerably superior to the NQ model.

Feature Importance. Figure 6.5 shows all features ranked by their importance for the classification according to the $\chi^2$ score. A higher value of the $\chi^2$ score indicates a lack of independence between a feature and labels; that is, the feature

Table 6.5: Score obtained during training of machine-learning models.

| Model | Classification Score | Regression Score |
|---|---|---|
| Stratified (Baseline I) | 0.57 | -0.01 |
| NQ Model (Baseline II) | 0.70 | – |
| Linear Regression | 0.32 | 0.25 |
| Logistic Regression | 0.79 | – |
| SVM / SVR | 0.81 | 0.22 |
| Decision Tree | 0.96 | 0.78 |
| Multi-layer Perceptron | **0.97** | **0.88** |

is more significant for the prediction. As expected, the size of the stream source is the factor most significant for the classification, followed by the short-circuit terminal operations (`findAny` and `findFirst`) in the pipeline. Terminal operators dominate the top half part of the ranking, with only three intermediate operations appearing in the top-15: `filter`, `distinct` and `sorted`.

The source-types `LinkedHashSet`, `HashMap` and `TreeMap` also showed significant influence and appeared high in the ranking. Source types that provide efficient Spliterator implementations such as `ArrayList`, `Vector` and `Array`, showed little or no importance to the classification. This indicates that streams created from those sources are far more influenced by the stream size and pipeline operators.

Overall, at least 20 factors showed significant importance score to the stream pipeline classification. The results confirm that, while the number of elements is the primary factor, pipeline operators and the source type still play a crucial role in the performance of stream parallelization.

### 6.4.4 Aggregating Stream Predictions to Stream Pipeline

Our models are trained to predict the performance of a single stream, generated from a stream pipeline. A stream pipeline generates numerous streams during the program's life-cycle. As each stream might differ in terms of its dynamic features (number of elements, source type), our model may predict conflicting optimal execution-modes within the same pipeline. We need to aggregate all predictions to provide a report for a stream pipeline.

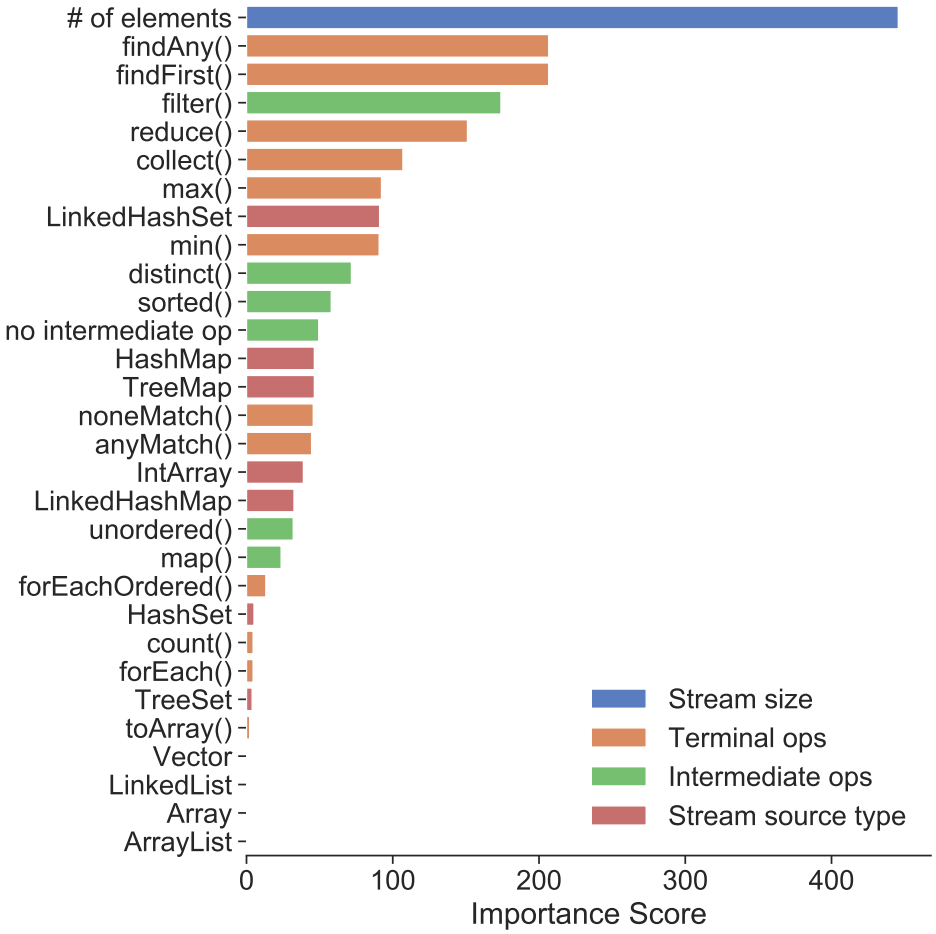



Figure 6.5: Ranking of the most significant features for the classification according to the $\chi^2$ score.

Given a stream pipeline $p$ and a set of streams generated from the pipeline as $X_p = \{x_{p0}, ..., x_{pn}\}$, we denote the $t'_{seq}(x_{pi})$ and the $t'_{par}(x_{pi})$ the predicted execution time of processing the pipeline $p$ under a scenario $x_{pi}$. We aggregate the stream predictions made by our classification and regression models as follows:

1. We aggregate the predictions from the classification models by showing the ratio $R_{cla}$ between the classified sequential and parallel-optimal streams.

2. We aggregate the predictions from regression models by showing the ratio $R_{reg}$ of the summed predicted performances of all streams in both execution modes.

$$R_{reg} = \frac{\sum_{x_p \in X_p} t'_{seq}(x_p)}{\sum_{x_p \in X_p} t'_{par}(x_p)} \tag{6.1}$$

As stated in Section 6.3.3, we consider a pipeline to be parallel-optimal if $R_{reg} > 1$, and sequential-optimal otherwise. In practice, however, developers should use both metrics ($R_{cla}$ and $R_{reg}$) to make an educated decision on whether using parallel streams can provide performance benefits or not. As our $R_{reg}$ is particularly sensitive to outliers - streams with millions of elements might affect $R_{reg}$ substantially - developers can use the $R_{cla}$ to assess the consistency of the pipeline classification. Developers are also encouraged to use higher thresholds values for $R_{reg}$ (e.g., 2 or 4) to only rely on parallel streams if the expected benefit is two or four times faster than the sequential counterpart, to justify the use of all CPUs to stream processing.

### 6.4.5 Limitations

While developing our framework, we have identified two important conceptual limitations of our model: 1) the lack of hardware features, and 2) the lack of behavioral parameter estimation. We elaborate more on each limitation in the following paragraphs.

Lack of Hardware Features. As described in Section 6.1, the underlying hardware is a crucial factor for deciding to parallelize streams. The JVM configures the number of threads used to process parallel streams based on the number of available CPUs to the application [59], hence, a machine with more CPU's is more likely to benefit from parallel streams. On the other hand, a high-speed CPU might be able to sequentially process the entire stream faster than the threads can split and combine the processed task.

Currently, our model does not take into account any hardware feature. To reliably use StreamAssist, a user would have to re-run the benchmarks on its produc-

tion environment. While benchmarks need to be run just once on the production environment, we plan to embed hardware features (e.g., number of CPUs, clock speed) into our training data and to provide benchmark results run in machines with distinct hardware configuration. In this way, our models can learn the effects of the underlying architecture during the classification of stream pipelines, and a user could opt to re-use previously learned models by providing its hardware configuration.

Lack of Behavioral Parameter Estimation. Our benchmark generator is prepared to simulate behavioral parameters with different workload levels and can be used to enrich our machine-learning models by taking into account the cost of processing each element. However, to use this information in the prediction, we have to estimate the cost of behavioral parameters in the target application. An estimation can be performed by statically analyzing the behavioral parameter or by measuring the time spent to process an entire stream during the application runtime.

An approach could be devised to evaluate whether a behavioral parameter is stateless and does not interfere with the stream source, i.e., are non-interfering. Stateless and non-interfering behavioral parameters are more likely to benefit from parallelization, and their performance can be estimated by, for instance, the number and type of generated byte-code instructions. Also, identifying behavioral parameters that call methods hostile for parallelization, such as I/O operation, can be used to filter out stream pipelines from our analysis to prevent reports with misguided information.

Furthermore, we could also provide a Java agent that measures the performance of behavioral parameters and feeds this information back to the models. Measuring the time of stream processing during runtime can lead to high application overhead; hence, this approach is indicated to be used only during application testing.

## 6.5 Evaluation

In this section, we evaluate our trained models and the monitoring of stream operations from StreamAssist. We conducted our experiments on the same machine

Table 6.6: Applications selected for our evaluation. Column "# of Benchs" depicts the number of stream benchmarks selected for the classification.

| Application | Version | Description | # of Benchs |
|---|---|---|---|
| EclipseCollections | 10.0.0 | Collections library | 416 |
| Guava | 27.1 | Google Core Libraries for Java | 32 |
| HTM | 0.6.13 | Hierarchical Temporal Memory | 10 |

where our models were trained, a computational server with an E5-1660-3.3GHz CPU, with 6 physical cores and 64 GB RAM using Linux 3.16.0-53 and Java 12.0.1 with HotSpot JVM. The evaluation focuses on answering the following research questions:

RQ1. **How accurate is our model at classifying streams execution mode of real applications?** As we train our models using synthetically generated pipelines, we evaluate if our models can generalize to classify stream pipelines from real applications.

RQ2. **What is the overhead of StreamAssist when monitoring stream pipelines?** We quantify how intrusive is our Java agent to the application's performance, during the process of collecting the dynamic information of streams from stream pipelines.

### 6.5.1 Selecting Applications

The evaluation of our tool requires selecting applications that make extensive use of Java streams and ideally have sound performance tests that evaluate the performance of stream pipelines. Unfortunately, writing performance tests is not a popular task in open-source projects [111] and projects that make extensive use of streams are an even smaller subset of Java-based projects. We inspect projects that have more than a thousand declared stream pipelines and manually verify if they contain a benchmark suite which evaluates streams performance. The Table 6.6 shows the three selected applications for this preliminary evaluation.

The EclipseCollection and Guava are both collection libraries and have been subject of our experimental studies in Chapter 4. As collections are a common stream

source, both projects include benchmarks dedicated to evaluating their collection implementation through streams pipelines under several scenarios. Moreover, we also include the HTM project, as this project was also selected in the evaluation of a related approach [103].

### 6.5.2 Experiment Preparation

After selecting the applications, we need to establish the *ground-truth* dataset. The ground-truth is a labeled dataset that contains the optimal execution mode for all pipelines per benchmark. Hence, we need to run all benchmarks with streams in both sequential and in parallel mode and compare their performance to establish the optimal-execution mode for each pipeline and benchmark. The EclipseCollection project already contains identical benchmarks that execute each stream in sequential and parallel-mode. We create new benchmarks for Guava and HTM, configured to run with the opposite execution mode of the original benchmark.

Note that a proper evaluation of the impact of multiple pipelines in a benchmark, would require us to perform a factorial experiment [126] with all possible execution mode combinations. To reduce the efforts during this evaluation, we filter out benchmarks that execute multiple stream pipelines.

Table 6.7: The labeled ground-truth dataset per project.

| **Projects** | **Ground-truth** | | |
|---|---|---|---|
| | **# of pipelines** | **Seq-optimal** | **Par-optimal** |
| EclipseCollections | 463 | 115 | 348 |
| HTM | 10 | 10 | 0 |
| Guava | 32 | 31 | 1 |

We then execute both the sequential and parallel versions of each benchmark and compare the performance counters of each benchmark to establish the optimal execution mode for a pipeline. If the faster benchmark version contains a pipeline with the sequential mode, we label the pipeline "seq-optimal", otherwise we label "par-optimal." If no significant difference is found between both benchmark versions, using the Tukey HSD test [126] we by default label the streams as "seq-optimal", as the sequential execution mode consumes fewer resources and should

Table 6.8: Score of the Multi-layer Perceptron model in the classification of the optimal execution-mode of stream pipelines per project.

| Project | Opt. Mode | Accu. | Prec. | Recall | f1-score | Support |
|---|---|---|---|---|---|---|
| EclipseCollections | Seq-optimal | 0.76 | 1.00 | 0.02 | 0.05 | 123 |
| | Par-optimal | | 0.74 | 1.00 | 0.85 | 340 |
| HTM | Seq-optimal | 1.00 | 1.00 | 1.00 | 1.00 | 10 |
| | Par-optimal | | – | – | – | 0 |
| Guava | Seq-optimal | 0.91 | 0.97 | 0.94 | 0.95 | 31 |
| | Par-optimal | | 0.00 | 0.00 | 0.00 | 1 |

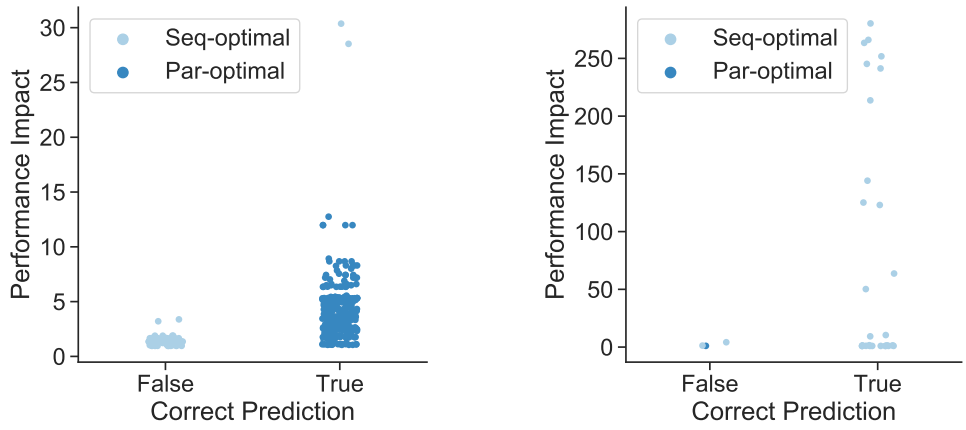
be preferred in this case. The result of this process is the labeled dataset summarized in Table 6.7. EclipseCollection benchmark suite contains 75% of parallel-optimal pipelines, while the other two projects contain almost exclusively sequential-optimal pipelines.

### 6.5.3 Accuracy of the Stream Pipeline Classification

Our models are trained entirely with synthetically generated stream pipelines, with simulated behavioral parameters. Hence, in this question, we attempt to quantify if our models can generalize to predict the performance of real stream pipelines. We present here only the results obtained with the Multi-Layer Perceptron [82], as this was the best model in our training phase.

We show in Table 6.8 the score of our models using standard metrics for classification: accuracy, precision, recall, and the harmonic f1-score metric. Overall, the model showed promising results, classifying all pipelines from HTM correctly and getting an accuracy of 0.91 and 0.76 for Guava and EclipseCollections, respectively. However, our model obtained very low recall value when classifying seq-optimal pipelines in the EclipseCollections application, indicating that our model tends to over-parallelize, classifying as par-optimal pipelines that have better performance is processed in sequence.

We further detail the results of our classification in Figure 6.6, for both EclipseCollections Figure 6.6a and Guava Figure 6.6b. In these two figures, we present each pipeline classified by whether it was predicted correctly (correct prediction axis), its optimal execution-mode, and the performance impact of executing the pipeline




(a) EclipseCollections (b) Guava

Figure 6.6: Evaluation of our model prediction by application, optimal execution mode, and performance impact. The performance impact shows the slowdown factor when the pipeline is executed with the sub-optimal mode. HTM results were omitted as it showed no incorrect predictions.

in its sub-optimal execution mode. For instance, if the performance impact of a parallel-optimal pipeline is 5x, that means the same pipeline would take five times longer if executed in sequential mode. Note that, in some benchmarks, the impact of wrongly defining the pipeline execution mode reaches up to 30x in EclipseCollections and more than 250x in Guava. Our trained model classifies all pipelines correctly with performance impact larger than a factor of four.

Wrong Predictions. We investigate the stream pipelines that were not correctly classified by our models. Upon further inspection, we identify two possible causes for the wrong classification of 120 benchmarks and present them in Table 6.9. First, we encounter pipelines that were not trained by our model (C1). While we model every stream operation, the pipeline as depicted in C1 was not part of our training set. This particular pipeline was used in 64 benchmarks, and our model mistakenly classified it as parallel-optimal in all 64 cases. Second, some benchmarks from EclipseCollections use the `collect(Collectors.groupBy())` to return a result grouped by a particular field (C2). The grouping conveys expensive

Table 6.9: Wrong pipeline predictions grouped by its possible causes.

| ID | **Possible cause** | **Example** | # |
|---|---|---|---|
| C1 | Pipeline not modeled | `filter-map-map-filter-collect` | 64 |
| C2 | Grouping-by not modeled | `collect(Collectors.groupingBy())` | 56 |
| – | Others | `reduce((a, b) -> b)` | 3 |

merge operations that are unfavorable for parallelization. Currently, our benchmark generator only considers the more streamlined `toList`, `toSet` collectors.

### 6.5.4 Overhead of StreamAssist

We design StreamAssist as a tool to be used during the development cycle of an application through the monitoring of performance and integration tests. However, the real benefit of using such framework is the possibility of monitoring streams in a system in production (within a specified time frame) and using the collected data to analyze the stream pipeline's performance. To achieve that, we need to guarantee a reasonable overhead to reduce the performance impact of collecting streams data at every stream creation.

We quantify the overhead of StreamAssist by running all benchmarks from the projects EclipseCollections, HTM, and Guava with and without our monitoring agent, comparing both performance counters to assess the monitoring overhead. The impact of the overhead in the benchmarks is depicted in Figure 6.7. In average, using our agent during benchmarks caused an overhead of 41% in the benchmarks execution time. In the `EclipseCollection` project, we also observe a few cases with more extreme overhead, with benchmarks being up to 9 times slower. These numbers were observed on benchmarks dominated by the stream creation operation. Such benchmarks are very short-running, with execution time below 100 nanoseconds.

While the overhead of StreamAssist is non-negligible, we expect the overhead of each stream creation to be amortized by the processing time of a stream pipeline during the application runtime. Moreover, as our agent can be uninstalled anytime from a running JVM, practitioners can turn off the monitoring, if excessive slowdowns are observed during the dynamic collection.

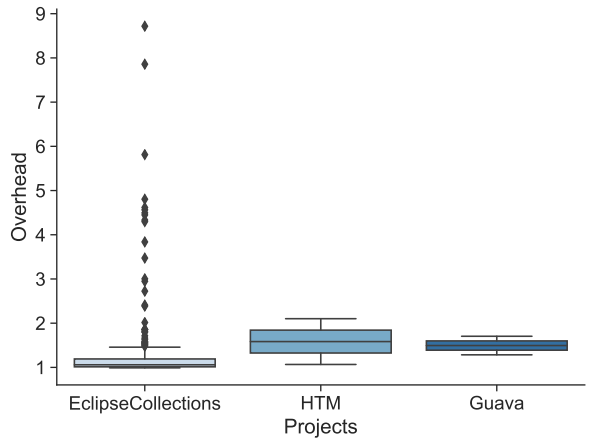



Figure 6.7: Distribution of observed monitoring overhead incurred on the benchmarks of EclipseCollection, HTM and Guava.

## 6.6 Summary of the Chapter

In this chapter, we presented StreamAssist, a decision-support framework for effective stream parallelization. The framework uses static and dynamic analysis to extract the profile of stream pipelines in an application. These profiles are fed into machine-learning models, trained with numerous stream benchmarks, that predicts the optimal execution mode of each stream pipeline. The framework reports back to developers what stream pipelines can benefit from parallel processing and what pipelines should run in sequential mode.

We evaluate the models using benchmarks from three open-source Java applications. The results show that **machine-learning models can capture essential features of stream performance and can identify with high accuracy the best execution mode for stream pipelines** of real applications. While incurring in a non-negligible overhead during the monitoring of a running application, the agent system is flexible enough to be active for a brief period to **extract the workload of a system in production**, without compromising its performance.

While in its early stage, we plan to augment StreamAssist with hardware features to improve the generalization of its models and devise approaches to estimate the performance of behavioral parameters. Both additions can make StreamAssist a more practical tool to support Java developers on stream programming.

# 7 Summary and Outlook

Developing efficient applications is a complex and multi-faceted problem that requires the knowledge and practices of multiple computer science disciplines. Developers have to continuously measure the performance of core implementations, use appropriate data structure and algorithms, have an in-depth knowledge of hardware specificities, and optimize sub-optimal program routines.

The goal of this thesis was to provide practical insights and novel methods to support developers in the laborious task of designing efficient applications. In the following, we summarize the previous chapters and highlight the most important contributions presented in this thesis.

## 7.1 Summary and Contributions

In Chapter 1, we start by motivating the need for performance efficiency in the design of software applications and by introducing the three problems tackled in this thesis: 1) the creation of sound performance tests through benchmarking, 2) the selection of efficient data structures, and 3) the efficient parallelization of element processing via the Java Stream API.

The following Chapter 2 establishes the research context of this work and describe the performance-related components of the Java programming environment. We then proceed to introduce the basic concepts and the terminology used throughout the thesis.

### Creating Sound Performance Tests through Benchmarking

In Chapter 3, we present an empirical study on bad practices in the creation of Java benchmarks under the Java Microbenchmark Harness (JMH) framework, referred

to as bad JMH practices. In this study, we empirically investigate the occurrence of bad JMH practices in open-source projects and experimentally evaluate its impact on benchmark results. The investigation yielded the following findings:

- **The studied bad JMH practices are prevalent in Java-based open-source systems.** Bad practices in Java benchmark creation were found in half of the projects with more than ten benchmarks tests. Using accumulation to consume method calls in a loop (LOOP) was the most frequently occurring bad practice, but all bad practices appeared in multiple projects.
- **Bad JMH practices often severely impacting the outcome of benchmarks** as they lead to benchmark results that substantially deviate from the correct measurements. Our findings were also confirmed by developers that promptly accepted the pull requests containing the fixed benchmarks.

Moreover, we devise a static analysis tool called SpotJMHBugs [44], that **automatically identifies the studied five bad practices**. This tool can be integrated into Eclipse IDE and support developers in the creation of sound performance benchmarks.

### Selection of efficient data structures

In Chapter 4, we conduct an experimental study on the usage and performance of Java collections. We analyzed the usage patterns of collections by mining software repositories and experimentally compare the performance of non-standard collection variants against the most commonly used variants. The findings in this study showed that:

- **Developers only rarely select non-standard collections or tune their collection instantiations.** Developers rely mostly on standard and general-purposed variants, while performance-related parameters such as the initial capacity are seldom specified.
- **Alternative implementations can offer programmers a significant improvement over standard libraries** on both execution time and memory

consumption on several usage scenarios. We devise a guideline that can be used by practitioners to identify scenarios where non-standard variants offer substantial performance benefits.

In Chapter 5, we present an application-level framework for efficient collection adaptation named CollectionSwitch. Our framework **selects at runtime collections variants that better suits the application workload**, in order to optimize the execution and memory performance of the target application. To accomplish this with low-overhead, we propose the use of adaptive allocation-sites, that monitor the workload of past collection instances to decide for a better variant on future collection instantiations. We implement the framework and experimentally evaluate our approach on a range of synthetic and real-application benchmarks. Our evaluation showed that:

- **CollectionSwitch can improve the execution time and memory consumption** of several applications by selecting better collection variants at runtime. The framework uses empirical models to predict the performance of each collection variant and switches to variants according to customized selection rules given by developers.
- **Collections can be adapted at runtime with very low-overhead**. We accomplish this by centering the monitoring, analysis, and selection of variants on the collection allocation-site - as opposed to each collection instance - and only selecting better variants for future instances.

### Efficient Parallelization of Element Processing via Streams

Finally, in Chapter 6, we present a decision-support framework that identifies stream pipelines that can be effectively parallelized to improve the runtime performance of an application. To that aim, the framework monitors stream pipelines of the target application at runtime to extract their profiles and combine it with a performance model that predicts the fastest execution mode (sequential or parallel) of stream pipelines. The performance model is crafted by training machine learning models on thousands of synthetically generated stream benchmarks, executed in sequential and parallel mode. We implement this framework and evaluate the

accuracy of our performance models and the overhead of monitoring the stream pipelines:

- **Machine-learning models can capture essential features of stream performance and can identify with high accuracy the best execution mode for stream pipelines**. While trained with synthetically generated benchmarks, our performance model showed reasonably good accuracy when classifying stream pipelines of real applications.
- **Monitoring stream pipelines can yield noticeable runtime overhead.** We mitigate this limitation by devising a flexible profiling method with agents that can be removed after collecting pipeline profiles for a determined period.

## 7.2 Outlook

In this thesis, we presented a series of empirical studies and automated methods that aim at facilitating the development of efficient applications. Although we have discussed the quality of the insights provided and the usefulness of our approaches, there are some interesting areas in which our work could be extended as detailed in the following paragraphs.

A study of even more bad practices on Java benchmarks. The study in Chapter 3 focused on five bad practices on Java benchmark creation. The JMH documentation [142] lists 38 samples containing several pitfalls not investigated by our study, that have the potential for substantially impacting the benchmark quality. For instance, false-sharing [169] is a difficult problem to handle when writing multi-threaded benchmarks. While JMH attempts to mitigate this problem by automatically padding fields declared within State objects [134], the internals of State objects are not padded and might affect the benchmark results. Our SpotJMHBugs can be further extended to identify new bad practices statically, and a new study should investigate its prevalence and impact, to give further insights into the problems developers face when writing benchmarks.

Automated Benchmark Repair. Our work, presented in Chapter 3, can be used as a starting point for automatic benchmark repair. Except for the INVO bad practice, our study showed that the removal of bad practices from benchmarks could be done with patches touching only a few lines of code. In many cases, the fix encompasses reconfiguring a benchmark parameter, or making proper use of the JMH infrastructure (e.g., using `Blackhole` object to consume method call returns). Current automated program repair techniques can fix highly localized bugs, with fix-patches spanning few lines of code [108]. In particular, template-based techniques [104] could be extended to fix several bad practices of Java benchmarks.

Benchmark-driven Models for Performance-Sensitive Libraries. The approaches presented in Chapter 5 and Chapter 6 used models crafted from benchmarks to predict the performance of Java collections and streams, using this prediction to optimize the application's performance and provide feedback to developers. Both the Java Collections Framework (JCF) and the Java Stream are performance sensitive libraries. Developers have to manually identify the performance trade-offs of using a particular data structure or processing a determined pipeline in parallel, with suboptimal decisions potentially leading to performance issues in their application.

Benchmark-driven models offer a cheap and hardware-sensitive approach for modeling the performance of core libraries. Benchmarks need to be run once in the production environment, and derived models can be integrated into the developer's workstation, providing valuable feedback and increasing the performance awareness during the development cycle. Since Java 12, the JDK releases a basic micro-benchmarking suite [32]. This suite can be used as a good starting point, as it is likely to cover core and performance-sensitive implementations, and will be maintained and enriched with new benchmarks on further JDK releases.

# Bibliography


1. D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. "The design of the borealis stream processing engine". In: *In CIDR*. 2005, pp. 277–289.
2. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. 2nd. MIT Press, Cambridge, MA, USA, 1996.
3. A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. "Fast, Effective Code Generation in a Just-in-time Java Compiler". *SIGPLAN Not.* 33:5, 1998, pp. 280–290. ISSN: 0362-1340. DOI: 10.1145/277652.277740.
4. Aicas. *Realtime Specification for Java 2.0*. https://www.aicas.com/cms/en/rtsj. [Online; accessed 3-April-2019].
5. Aleksey Shipilëv. *Java Microbenchmark Harness - (the lesser of two evils)*. https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf. [Online; accessed 19-December-2018]. 2013.
6. Z. Alexander, T. Mytkowicz, A. Diwan, and E. Bradley. "Measurement and Dynamical Analysis of Computer Performance Data". In: *Proceedings of the 9th International Conference on Advances in Intelligent Data Analysis*. Springer-Verlag, 2010, pp. 18–29. DOI: 10.1007/978-3-642-13062-5_4.
7. M. Allamanis and C. Sutton. "Mining Source Code Repositories at Massive Scale Using Language Modeling". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. IEEE Press, San Francisco, CA, USA, 2013, pp. 207–216.

8. Android. *ArrayMap Development Kit*. https://developer.android.com/reference/android/support/v4/util/ArrayMap. [Online; accessed 3-April-2019]. 2018.

9. Android. *ArraySet Development Kit*. https://developer.android.com/reference/android/util/ArraySet. [Online; accessed 12-January-2019]. 2018.

10. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. ACM, Dublin, Ireland, 2009, pp. 38–49. DOI: 10.1145/1542476.1542481.

11. M. Arnold. *Online Profiling and Feedback-directed Optimization of Java*. 2002.

12. J. Aycock. "A Brief History of Just-in-time". *ACM Comput. Surv.* 35:2, 2003, pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

13. A. Bergel, A. Infante, S. Maass, and J. P. S. Alcocer. "Reducing resource consumption of expandable collections: The Pharo case". *Sci. Comput. Program.* 161, 2018, pp. 34–56. DOI: 10.1016/j.scico.2017.12.009.

14. A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. "Streams a la carte: Extensible Pipelines with Object Algebras". In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by J. T. Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015, pp. 591–613. DOI: 10.4230/LIPIcs.ECOOP.2015.591.

15. S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. ACM, Portland, Oregon, USA, 2006, pp. 169–190. DOI: 10.1145/1167473.1167488.

16. S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century". *Commun. ACM* 51:8, 2008, pp. 83–89. ISSN: 0001-0782. DOI: 10.1145/1378704.1378723.

17. *Blackhole (JMH Core 1.6.3 API)*. http://javadox.com/org.openjdk.jmh/jmh-core/1.6.3/org/openjdk/jmh/infra/Blackhole.html. [Online; accessed 10-April-2019].

18. B. Blanchet. "Escape Analysis for JavaTM: Theory and Practice". *ACM Trans. Program. Lang. Syst.* 25:6, 2003, pp. 713–775. ISSN: 0164-0925. DOI: 10.1145/945885.945886.

19. J. Bloch. *Effective Java*. 3rd ed. Addison-Wesley, Boston, MA, 2018.

20. C. F. Bolz, L. Diekmann, and L. Tratt. "Storage Strategies for Collections in Dynamically Typed Languages". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM, Indianapolis, Indiana, USA, 2013, pp. 167–182. DOI: 10.1145/2509136.2509531.

21. A. B. Bondi. *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. 1st. Addison-Wesley Professional, 2014.

22. L. Breiman. *Classification and regression trees*. Routledge, 2017.

23. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. ACM, Los Angeles, California, 2004, pp. 777–786. DOI: 10.1145/1186562.1015800.

24. Z. Budimlic and K. Kennedy. "Optimizing Java - Theory and Practice". *Concurrency, Practice and Experience* 9, 1997, pp. 445–463.

25. L. Bulej, V. Horký, and P. Tůma. "Do We Teach Useful Statistics for Performance Evaluation?" In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 2017, pp. 185–189. DOI: 10.1145/3053600.3053638.

26. M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas. "A Performance Study of Java Garbage Collectors on Multicore Architectures". In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '15. ACM, San Francisco, California, 2015, pp. 20–29. DOI: 10.1145/2712386.2712404.

27. Y. Chan, A. Wellings, I. Gray, and N. Audsley. "A Distributed Stream Library for Java 8". *IEEE Transactions on Big Data* 3:3, 2017, pp. 262–275. ISSN: 2332-7790. DOI: 10.1109/TBDATA.2017.2666201.

28. Y. Chan, A. Wellings, I. Gray, and N. Audsley. "On the Locality of Java 8 Streams in Real-Time Big Data Applications". In: *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '14. ACM, Niagara Falls, NY, USA, 2014, 20:20–20:28. DOI: 10.1145/2661020.2661028.

29. Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. "Evaluating Iterative Optimization Across 1000 Datasets". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2010, pp. 448–459. DOI: 10.1145/1806596.1806647.

30. A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. "Patterns of Memory Inefficiency". In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. Springer-Verlag, Lancaster, UK, 2011, pp. 383–407.

31. Chronicle. *Koloboke*. http://chronicle.software/products/koloboke-collections/. 2015.

32. R. Claes, F. Staffan, and S. Aleksey. *JEP 230: Microbenchmark Suite*. https://openjdk.java.net/jeps/230. [Online; accessed 20-April-2019].

33. N. Cliff. “Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions”. *Psychological Bulletin* 114, 1993, pp. 494–509. DOI: 10.1037/0033-2909.114.3.494.

34. T. T. S. Q. Company. *TIOBE Index*. https://www.tiobe.com/tiobe-index/. [Online; accessed 3-April-2019]. 2019.

35. D. Costa. *Avoiding the usage of Jmh Invocation level on setup/teardown*. https://github.com/netty/netty/pull/8005. [Online; accessed 19-December-2018]. 2018.

36. D. Costa. *Fix 627 - Consume every object created in a benchmark to avoid DCE*. https://github.com/eclipse/eclipse-collections/pull/628. [Online; accessed 19-December-2018]. 2018.

37. D. Costa. *fix: configure benchmark forks to >= 1*. https://github.com/pgjdbc/pgjdbc/pull/1214. [Online; accessed 19-December-2018]. 2018.

38. D. Costa. *LOG4J2-2478 Return the computed variables on each benchmark to avoid DCE*. https://github.com/apache/logging-log4j2/pull/219. [Online; accessed 19-December-2018]. 2018.

39. D. Costa. *PUBDEV-6081: Reducing the Invocation JMH level setup/teardown to only the training*. https://github.com/h2oai/h2o-3/pull/3070. [Online; accessed 19-December-2018]. 2018.

40. D. Costa. *Reducing the usage of Jmh Invocation level on setup*. https://github.com/apache/incubator-druid/pull/6459. [Online; accessed 19-December-2018]. 2018.

41. D. Costa. *Use Blackhole objects to sink the method return in a benchmak loop (JMH)*. https://github.com/apache/incubator-druid/pull/5908. [Online; accessed 19-December-2018]. 2018.

42. D. Costa and A. Andrzejak. “CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. ACM, Vienna, Austria, 2018, pp. 16–26. DOI: 10.1145/3168825.

43. D. Costa, A. Andrzejak, J. Seboek, and D. Lo. "Empirical Study of Usage and Performance of Java Collections". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE '17. ACM, L'Aquila, Italy, 2017, pp. 389–400. DOI: 10.1145/3030207.3030221.

44. D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. *SpotJMHBugs*. https://github.com/DiegoEliasCosta/spotjmhbugs. [Online; accessed 21-December-2018]. 2018.

45. D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. *Studying Bad Practices in JMH Benchmarks*. https://github.com/DiegoEliasCosta/badJMHpractices-study. [Online; accessed 21-December-2018]. 2018.

46. D. Costa, C.-P. Bezemer, P. Leitner, and A. Andrzejak. "What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks". *Transactions on Software Engineering*, 2019 (to appear).

47. D. Costa, J. P. Sandoval, and A. Andrzejak. "Leveraging Machine Learning Models for Effective Parallelization of Java 8 Streams". In: *Automated Software Engineering*. Under revision.

48. D. Costa, E. Schubert, A. Andrzejak, and D. Lo. "Beyond the Java Collections Framework: An Empirical Study of Usage and Performance of Java Collection Libraries and APIs".

49. T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. "Compiling Java Just in Time". *IEEE Micro* 17:3, 1997, pp. 36–43. ISSN: 0272-1732. DOI: 10.1109/40.591653.

50. C. Curtsinger and E. D. Berger. "STABILIZER: Statistically Sound Performance Evaluation". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013, pp. 219–228. DOI: 10.1145/2451116.2451141.

51. M. De Wael, S. Marr, J. De Koster, J. B. Sartor, and W. De Meuter. "Just-in-time Data Structures". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Onward! 2015. ACM, Pittsburgh, PA, USA, 2015, pp. 61–75. DOI: 10.1145/2814228.2814231.

52. O. Docs. *java.util.stream (Java Platform SE 8 )*. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. (Accessed on 05/11/2019).

53. Z. Dong, A. Andrzejak, D. Lo, and D. Costa. "ORPLocator: Identifying Read Points of Configuration Options via Static Analysis". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016, pp. 185–195. DOI: 10.1109/ISSRE.2016.37.

54. A. Dynamics. *Infographic: Cost of Performance Issues | APMdigest - Application Performance Management*. https://www.apmdigest.com/infographic-cost-of-performance-issues. [Online; accessed 10-April-2019].

55. I. Eclipse Foundation. *The Platform for Open Innovation and Collaboration - The Eclipse Foundation*. [Online; accessed 13-February-2019]. 2019.

56. *Elasticsearch: Open Source, Distributed, RESTful Search Engine*. https://github.com/elastic/elasticsearch. [Online; accessed 13-February-2019]. 2019.

57. *fastutil*. http://fastutil.di.unimi.it/. [Online; accessed 25-February-2019]. 2019.

58. *FindBugs™ - Find Bugs in Java Programs*. http://findbugs.sourceforge.net/. [Online; accessed 20-April-2019].

59. *ForkJoinPool (Java Platform SE 8 )*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html. [Online; accessed 20-April-2019].

60. T. E. Foundation. *Eclipse Collections - Features you want with the collections you need*. https://www.eclipse.org/collections/. [Online; accessed 25-February-2019]. 2019.

61. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994.

62. A. Georges, D. Buytaert, and L. Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. ACM, 2007, pp. 57–76. DOI: 10.1145/1297027.1297033.

63. A. Georges, L. Eeckhout, and D. Buytaert. "Java Performance Evaluation Through Rigorous Replay Compilation". *SIGPLAN Not.* 43:10, 2008, pp. 367–384. ISSN: 0362-1340. DOI: 10.1145/1449955.1449794.

64. M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. "Memory and Resource Leak Defects in Java Projects: An Empirical Study". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. ACM, Gothenburg, Sweden, 2018, pp. 410–411. DOI: 10.1145/3183440.3195032.

65. M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. "Memory and Resource Leak Defects and their Repairs in Java Projects". *Empirical Software Engineering*, 2019 (to appear).

66. J. Y. Gil, K. Lenz, and Y. Shimron. "A Microbenchmark Case Study and Lessons Learned". In: *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. ACM, 2011, pp. 297–308. DOI: 10.1145/2095050.2095100.

67. J. Y. Gil and Y. Shimron. "Smaller Footprint for Java Collections". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '11. ACM, Portland, Oregon, USA, 2011, pp. 191–192. DOI: 10.1145/2048147.2048201.

68. I. GitHub. *GitHub*. https://github.com/. [Online; accessed 5-April-2019]. 2019.

69. GNU Trove. *Trove*. http://trove.starlight-systems.com/. 2015.

70. Goldman Sachs Group, Inc. *GS Collections*. https://github.com/goldmansachs/gs-collections. 2015.

71. Google. *Google HTTP Client Library for Java*. https://github.com/google/google-http-java-client. [Online; accessed 10-September-2017]. 2013.

72. Google. *google/closure-library: Google's common JavaScript library*. https://github.com/google/closure-library. [Online; accessed 13-February-2019]. 2019.

73. Google. *Guava*. https://github.com/google/guava. 2014.

74. J. Gosling and H. McGilton. *The Java Language Environment: A White Paper*. Technical report. Mountain View, CA, USA: Sun Microsystems Computer Company, 1995.

75. V. Green. *Impact of slow page load time on website performance*. http://bit.ly/2SGmbM4. [Online; accessed 19-December-2018]. 2016.

76. S. N. Group. *Stanford CoreNLP*. https://github.com/stanfordnlp/CoreNLP. [Online; accessed 10-September-2017]. 2013.

77. D. Gu, C. Verbrugge, and E. Gagnon. "Code Layout as a Source of Noise in JVM Performance". *Stud. Inform. Univ.* 4:1, 2005, pp. 83–99.

78. D. Gu, C. Verbrugge, and E. M. Gagnon. "Relative Factors in Performance Analysis of Java Virtual Machines". In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE '06. ACM, Ottawa, Ontario, Canada, 2006, pp. 111–121. DOI: 10.1145/1134760.1134776.

79. A. S. Harji, P. A. Buhr, and T. Brecht. "Our Troubles with Linux and Why You Should Care". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2011, 2:1–2:5. DOI: 10.1145/2103799.2103802.

80. S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. "Energy Profiles of Java Collections Classes". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. ACM, Austin, Texas, 2016, pp. 225–236. DOI: 10.1145/2884781.2884869.

81. A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar. "Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. ACM, Melbourne, FL, USA, 2015, pp. 27–36. DOI: 10.1145/2807426.2807429.

82. G. E. Hinton. "Connectionist learning procedures". In: *Machine learning*. Elsevier, 1990, pp. 555–610.

83. V. Horký, P. Libič, A. Steinhauser, and P. Tůma. "DOs and DON'Ts of Conducting Performance Measurements in Java". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. ACM, Austin, Texas, USA, 2015, pp. 337–340. DOI: 10.1145/2668930.2688820.

84. N. Hunt, P. S. Sandhu, and L. Ceze. "Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures". In: *2011 15th Workshop on Interaction between Compilers and Computer Architectures*. 2011, pp. 63–70. DOI: 10.1109/INTERACT.2011.13.

85. R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. "Lua&Mdash;an Extensible Extension Language". *Softw. Pract. Exper.* 26:6, 1996, pp. 635–652. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

86. O. Inc. *Design of Shutdown Hooks API*. https://docs.oracle.com/javase/7/docs/technotes/guides/lang/hook-design.html. [Online; accessed 10-April-2019].

87. O. Inc. *Java Garbage Collection Basics*. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. [Online; accessed 10-April-2019].

88. O. Inc. *java.lang.instrument (Java Platform SE 7 )*. https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html. [Online; accessed 10-April-2019].

89. O. Inc. *Parallelism (The Java Tutorials)*. https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html. [Online; accessed 20-April-2019].

90. O. Inc. *StreamSupport (Java Platform SE 8 )*. https://docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html. [Online; accessed 20-April-2019].

91. O. Inc. *The Java Tutorials: Processes and Threads*. https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html. [Online; accessed 10-April-2019].

92. C. Isci, A. Buyuktosunoglu, and M. Martonosi. "Long-Term Workload Phases: Duration Predictions and Applications to DVFS". *IEEE Micro* 25:5, 2005, pp. 39–51. ISSN: 0272-1732. DOI: 10.1109/MM.2005.93.

93. K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. "Compiling and Optimizing Java 8 Programs for GPU Execution". In: 2015. DOI: 10.1109/PACT.2015.46.

94. *Java leads as high-performance language for mission-critical financial applications*. https://www.cinnober.com/news/java-leads-as-high-performance-language-for-mission-critical-financial-applications. [Online; accessed 10-April-2019].

95. *JavaParser – For processing Java code*. http://javaparser.org/. [Online; accessed 13-February-2019]. 2019.

96. *JavaSoft Ships Java 1.0*. https://tech-insider.org/java/research/1996/0123.html. [Online; accessed 10-April-2019].

97. Julien Ponge. "Avoiding Benchmarking Pitfalls on the JVM". *Java Magazine*, 2014.

98. C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. "Brainy: Effective Selection of Data Structures". *SIGPLAN Not.* 46:6, 2011, pp. 86–97. ISSN: 0362-1340. DOI: 10.1145/1993316.1993509.

99. T. Kalibera, L. Bulej, and P. Tuma. "Benchmark precision and random initial state". In: *in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems*. SCS, 2005, pp. 853–862.

100. T. Kalibera and R. Jones. "Rigorous Benchmarking in Reasonable Time". In: *Proceedings of the 2013 International Symposium on Memory Management*. ACM, 2013, pp. 63–74. DOI: 10.1145/2491894.2464160.

101. E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. ACM, Hyderabad, India, 2014, pp. 92–101. DOI: 10.1145/2597073.2597074.

102. K. Kawachiya, K. Ogata, and T. Onodera. “Analysis and Reduction of Memory Inefficiencies in Java Strings”. *SIGPLAN Not.* 43:10, 2008, pp. 385–402. ISSN: 0362-1340. DOI: 10.1145/1449955.1449795.

103. R. Khatchadourian, Y. Tang, M. Bagherzadeh, and S. Ahmed. “Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams”. In: *International Conference on Software Engineering*. ICSE ’19. Technical Track. To appear. ACM/IEEE. ACM, Montreal, QC, Canada, 2019.

104. D. Kim, J. Nam, J. Song, and S. Kim. “Automatic Patch Generation Learned from Human-written Patches”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. IEEE Press, San Francisco, CA, USA, 2013, pp. 802–811.

105. J. Knoop, O. Rüthing, and B. Steffen. “Partial Dead Code Elimination”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. ACM, Orlando, Florida, USA, 1994, pp. 147–158. DOI: 10.1145/178243.178256.

106. M. Kuperberg, F. Omri, and R. Reussner. “Automated Benchmarking of Java APIs”. In: Paderborn, Germany, 2010.

107. C. Laaber and P. Leitner. “An Evaluation of Open-source Software Microbenchmark Suites for Continuous Performance Assessment”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 119–130. DOI: 10.1145/3196398.3196407.

108. X. B. D. Le, D. Lo, and C. L. Goues. “History Driven Program Repair”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 213–224. DOI: 10.1109/SANER.2016.76.

109. D. Lea. “A Java Fork/Join Framework”. In: *Proceedings of the ACM 2000 Conference on Java Grande*. JAVA ’00. ACM, San Francisco, California, USA, 2000, pp. 36–43. DOI: 10.1145/337449.337465.

110. D. Lea. *When to use parallel streams*. http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html. (Accessed on 05/11/2019).

111. P. Leitner and C.-P. Bezemer. “An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects”. In: *Proceedings of the 8th ACM/SPEC of the International Conference on Performance Engineering*. ACM, 2017, pp. 373–384. DOI: 10.1145/3030207.3030213.

112. R. Leventov. *Koloboke Collections – Java Collections till the last breadcrumb of memory and performance*. https://koloboke.com/. [Online; accessed 25-February-2019]. 2019.

113. R. Leventov. *Time - Memory Tradeoff With the Example of Java Maps*. https://dzone.com/articles/time-memory-tradeoff-example.

114. L. Lewis. *Java Collection Performance*. https://dzone.com/articles/java-collection-performance. 2011.

115. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014.

116. B. Liskov and S. Zilles. “Programming with Abstract Data Types”. In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. ACM, Santa Monica, California, USA, 1974, pp. 50–59. DOI: 10.1145/800233.807045.

117. L. Liu and S. Rus. “Perflint: A Context Sensitive Performance Advisor for C++ Programs”. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’09. IEEE Computer Society, Washington, DC, USA, 2009, pp. 265–274. DOI: 10.1109/CGO.2009.36.

118. E. Loy. *Java 8’s streams: why parallel stream is slower?* https://stackoverflow.com/questions/23170832/java-8s-streams-why-parallel-stream-is-slower. [Online; accessed 15-April-2019]. 2014.

119. R. Malhotra. *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. Chapman & Hall/CRC, 2015.

120. I. Manotas, L. Pollock, and J. Clause. “SEEDS: A Software Engineer’s Energy-optimization Decision Support Framework”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. ACM, Hyderabad, India, 2014, pp. 503–514. DOI: 10.1145/2568225.2568297.

121. Matsemann. *Should I always use a parallel stream when possible?* https://stackoverflow.com/questions/20375176/should-i-always-use-a-parallel-stream-when-possible. [Online; accessed 15-April-2019]. 2016.

122. D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. "Understanding the Use of Lambda Expressions in Java". *Proc. ACM Program. Lang.* 1:OOPSLA, 2017, 85:1–85:31. ISSN: 2475-1421. DOI: 10.1145/3133909.

123. H. T. Mei, I. Gray, and A. Wellings. "Integrating Java 8 Streams with The Real-Time Specification for Java". In: *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '15. ACM, Paris, France, 2015, 10:1–10:10. DOI: 10.1145/2822304.2822314.

124. S. microsystems. *JAVASOFT SHIPS JAVA 1.0*. https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml. [Online; accessed 3-April-2019]. 2019.

125. N. Mitchell and G. Sevitsky. "The Causes of Bloat, the Limits of Health". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. ACM, Montreal, Quebec, Canada, 2007, pp. 245–260. DOI: 10.1145/1297027.1297046.

126. D. C. Montgomery. *Design and Analysis of Experiments*. English. 8 edition. Wiley, Hoboken, NJ, 2012.

127. C. Moreno and S. Fischmeister. "Accurate Measurement of Small Execution Times—Getting Around Measurement Errors". *IEEE Embed. Syst. Lett.* 9:1, 2017, pp. 17–20. ISSN: 1943-0663. DOI: 10.1109/LES.2017.2654160.

128. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. "Evaluating the Accuracy of Java Profilers". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. ACM, Toronto, Ontario, Canada, 2010, pp. 187–197. DOI: 10.1145/1806596.1806618.

129. T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. "Producing Wrong Data Without Doing Anything Obviously Wrong!" In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2009, pp. 265–276. DOI: 10.1145/1508244.1508275.

130. H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. "Mining Preconditions of APIs in Large-scale Code Corpus". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ACM, Hong Kong, China, 2014, pp. 166–177. DOI: 10.1145/2635868.2635924.

131. P. E. Nogueira, R. Matias Jr., and E. Vicente. "An Experimental Study on Execution Time Variation in Computer Experiments". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1529–1534. DOI: 10.1145/2554850.2555022.

132. A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. "DataMill: Rigorous Performance Evaluation Made Easy". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 137–148. DOI: 10.1145/2479871.2479892.

133. O. Olivo, I. Dillig, and C. Lin. "Static Detection of Asymptotic Performance Bugs in Collection Traversals". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, Portland, OR, USA, 2015, pp. 369–378. DOI: 10.1145/2737924.2737966.

134. *Open JMH Sample 22: False-Sharing*. https://hg.openjdk.java.net/code-tools/jmh/file/99d7b73cf1e3/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_22_FalseSharing.java. [Online; accessed 10-April-2019].

135. *OpenJDK: Java Microbenchmark Harness*. https://openjdk.java.net/projects/code-tools/jmh/. [Online; accessed 19-December-2018]. 2018.

136. *OpenJDK: jmh*. https://openjdk.java.net/projects/code-tools/jmh/. [Online; accessed 10-April-2019].

137. Oracle. *Java Development Kit*. https://www.oracle.com/java/index.html. 2015.

138. Oracle. *Java SE HotSpot at a Glance*. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html. [Online; accessed 5-April-2019]. 2019.

139. Oracle. *Strategic Acquisitions*. https://www.oracle.com/sun/. [Online; accessed 5-April-2019]. 2019.

140. Oracle. *The Java Language Environment*. https://www.oracle.com/technetwork/java/intro-141325.html. [Online; accessed 3-April-2019]. 2019.

141. Oracle. *Trail: Collections (The Java Tutorials)*. https://docs.oracle.com/javase/tutorial/collections/. [Online; accessed 10-April-2019].

142. Oracle America Inc. *JMH Samples*. http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/. [Online; accessed 19-December-2018]. 2014.

143. S. Osiński and D. Weiss. *HPPC: High Performance Primitive Collections for Java*. http://labs.carrotsearch.com/hppc.html. 2015.

144. E. Österlund and W. Löwe. "Dynamically Transforming Data Structures". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE'13. IEEE Press, Silicon Valley, CA, USA, 2013, pp. 410–420. DOI: 10.1109/ASE.2013.6693099.

145. S. Overflow. *Stack Overflow Developer Survey 2018*. https://insights.stackoverflow.com/survey/2018/. [Online; accessed 3-April-2019]. 2019.

146. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research* 12, 2011, pp. 2825–2830.

147. R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes. "The Influence of the Java Collection Framework on Overall Energy Consumption". In: *Proceedings of the 5th International Workshop on Green and Sustainable Software*. GREENS '16. ACM, Austin, Texas, 2016, pp. 15–21. DOI: 10.1145/2896967.2896968.

148. G. Pinto, K. Liu, F. Castor, and Y. D. Liu. "A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 20–31. DOI: 10.1109/ICSME.2016.34.

149. M. Pradel, M. Huggler, and T. R. Gross. "Performance Regression Testing of Concurrent Classes". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 13–25. DOI: 10.1145/2610384.2610393.

150. J. Rabcan. *Parallel stream vs serial stream*. https://stackoverflow.com/questions/32799937/parallel-stream-vs-serial-stream. [Online; accessed 15-April-2019]. 2015.

151. T. Rauber and G. Rnger. *Parallel Programming: For Multicore and Cluster Systems*. 2nd. Springer Publishing Company, Incorporated, 2013.

152. B. Ray, D. Posnett, V. Filkov, and P. Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ACM, Hong Kong, China, 2014, pp. 155–165. DOI: 10.1145/2635868.2635922.

153. *ReactiveX/RxJava: Reactive Extensions for the JVM*. https://github.com/ReactiveX/RxJava. [Online; accessed 13-February-2019]. 2019.

154. M. Rodriguez-Cancio, B. Combemale, and B. Baudry. "Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 132–143. DOI: 10.1145/2970276.2970346.

155. J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine. "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices". In: *Annual meeting of the Southern Association for Institutional Research*. 2006.

156. R. Saborido, R. Morales, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. "Getting the most from map data structures in Android". *Empirical Software Engineering* 23:5, 2018, pp. 2829–2864. ISSN: 1573-7616. DOI: 10.1007/s10664-018-9607-8.

157. J. L. Schilling. "The Simplest Heuristics May Be the Best in Java JIT Compilers". *SIGPLAN Not.* 38:2, 2003, pp. 36–46. ISSN: 0362-1340. DOI: 10.1145/772970.772975.

158. O. Shacham, M. Vechev, and E. Yahav. "Chameleon: Adaptive Selection of Collections". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. ACM, Dublin, Ireland, 2009, pp. 408–418. DOI: 10.1145/1542476.1542522.

159. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. "Discovering and Exploiting Program Phases". *IEEE Micro* 23:6, 2003, pp. 84–93. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1261391.

160. A. Spitz, D. Costa, K. Chen, J. Greulich, J. Geiß, S. Wiesberg, and M. Gertz. "Heterogeneous subgraph features for information networks". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Houston, TX, USA, June 10, 2018*. 2018, 7:1–7:9. DOI: 10.1145/3210259.3210266.

161. *Spliterator (Java Platform SE 8 )*. https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html. [Online; accessed 10-April-2019].

162. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. "Streamflex: High-throughput Stream Programming in Java". *SIGPLAN Not.* 42:10, 2007, pp. 211–228. ISSN: 0362-1340. DOI: 10.1145/1297105.1297043.

163. P. Stefan, V. Horky, L. Bulej, and P. Tuma. "Unit Testing Performance in Java Projects: Are We There Yet?" In: *Proceedings of the 8th ACM/SPEC of the International Conference on Performance Engineering*. ACM, 2017, pp. 401–412. DOI: 10.1145/3030207.3030226.

164. M. J. Steindorfer and J. J. Vinju. "Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. ACM, Pittsburgh, PA, USA, 2015, pp. 783–800. DOI: 10.1145/2814270.2814312.

165. R. E. Strom and S Yemini. “Typestate: A Programming Language Concept for Enhancing Software Reliability”. *IEEE Trans. Softw. Eng.* 12:1, 1986, pp. 157–171. ISSN: 0098-5589. DOI: 10.1109/TSE.1986.6312929.

166. X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz. “Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing”. *Proc. VLDB Endow.* 7:13, 2014, pp. 1343–1354. ISSN: 2150-8097. DOI: 10.14778/2733004.2733007.

167. T. J. Team. *JUnit 5*. https://junit.org/junit5/. [Online; accessed 5-April-2019]. 2019.

168. W. Thies, M. Karczmarek, and S. P. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Proceedings of the 11th International Conference on Compiler Construction*. Ed. by R. N. Horspool. CC’02. Springer-Verlag, London, UK, UK, 2002, pp. 179–196. DOI: 10.1007/3-540-45937-5_14.

169. J. Torrellas, H. S. Lam, and J. L. Hennessy. “False sharing and spatial locality in multiprocessor caches”. *IEEE Transactions on Computers* 43:6, 1994, pp. 651–663. ISSN: 0018-9340. DOI: 10.1109/12.286299.

170. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. “Soot - a Java Bytecode Optimization Framework”. In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON ’99. IBM Press, Mississauga, Ontario, Canada, 1999, pp. 13–.

171. S. Vigna. *Fastutil*. http://fastutil.di.unimi.it/l. 2016.

172. M. Vorontsov. *Large HashMap Overview*. http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/. 2015.

173. M. N. Wegman and F. K. Zadeck. “Constant Propagation with Conditional Branches”. *ACM Trans. Program. Lang. Syst.* 13:2, 1991, pp. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136.

174. D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. “Value Dependence Graphs: Representation Without Taxation”. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1994, pp. 297–310. DOI: 10.1145/174675.177907.

175. T. White. *Hadoop: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2009.

176. F. Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Breakthroughs in Statistics: Methodology and Distribution*. Ed. by S. Kotz and N. L. Johnson. Springer New York, 1992, pp. 196–202. DOI: 10.1007/978-1-4612-4380-9_16.

177. R. Winterhalter. *Byte Buddy: runtime code generation for the Java virtual machine*. https://bytebuddy.net/. [Online; accessed 10-April-2019].

178. G. Xu. "CoCo: Sound and Adaptive Replacement of Java Collections". In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Springer-Verlag, Montpellier, France, 2013, pp. 1–26. DOI: 10.1007/978-3-642-39038-8_1.

179. G. Xu and A. Rountev. "Detecting Inefficiently-used Containers to Avoid Bloat". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. ACM, Toronto, Ontario, Canada, 2010, pp. 160–173. DOI: 10.1145/1806596.1806616.

180. G. Xu and A. Rountev. "Precise Memory Leak Detection for Java Software Using Container Profiling". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. ACM, Leipzig, Germany, 2008, pp. 151–160. DOI: 10.1145/1368088.1368110.

181. S. Yang, D. Yan, G. Xu, and A. Rountev. "Dynamic Analysis of Inefficiently-used Containers". In: *Proceedings of the Ninth International Workshop on Dynamic Analysis*. WODA 2012. ACM, Minneapolis, MN, USA, 2012, pp. 30–35. DOI: 10.1145/2338966.2336805.

182. B. Yarahmadi and F. Khunjush. "Bamshad: A JIT compiler for running Java stream APIs on heterogeneous environments". In: *2017 19th International Symposium on Computer Architecture and Digital Systems (CADS)*. 2017, pp. 1–5. DOI: 10.1109/CADS.2017.8310734.

183. M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: A Unified Engine for Big Data Processing". *Commun. ACM* 59:11, 2016, pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

184. D. Zhang, Z.-Z. Li, H. Song, and L. Liu. “A Programming Model for an Embedded Media Processing Architecture”. In: *Proceedings of the 5th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. SAMOS’05. Springer-Verlag, Samos, Greece, 2005, pp. 251–261. DOI: 10.1007/11512622_27.