UiO : University of Oslo

Sanaz Tavakolisomeh

User-Centric Approaches to Garbage Collector Selection and Heap Size Optimization for Java Applications

Thesis submitted for the degree of Philosophiae Doctor

Department
Faculty of Mathematics and Natural Sciences

2024

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo

ISSN ISSN

Cover: UiO.
Print production: Graphic Center, University of Oslo.

To my dearest and unwavering best friend, my guiding light through every challenge, my beloved husband, Amirhossein,
To my, the best I could have, parents, whose boundless love has been always with me,
To my caring siblings, whose encouragement has always brightened my path,
With a heart filled with gratitude, I dedicate this thesis to you, my cherished family.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted at the Department of Informatics, UiO, under the supervision of Professor Paulo Ferreira, Professor Rodrigo Bruno, and Professor Erik Bartely Jul.

The thesis is a collection of three papers, presented in peer-reviewed scientific journals and conferences. The papers are preceded by an introductory chapter that relates them to each other and provides background information and motivation for the work. These introductory sections give an overall perspective of the thesis and document the coherence of the thesis. I have been involved as a main contributor in each one of these papers, and they were written between 2020 and 2023 for the purpose of supporting the thesis.

Acknowledgements

I would like to extend my gratitude to my three supervisors, Professor Paulo Ferreira, Professor Rodrigo Bruno, and Professor Erik Bartely Jul, for all their support, guidance, and invaluable mentorship throughout the entire duration of my doctoral journey.

My deep appreciation goes to Professor Paulo Ferreira, whose wisdom, dedication, years of experience, and encouragement have been pivotal in shaping the direction of my research. Also, I am thankful to Professor Rodrigo Bruno for his mentorship. His expertise in the GC domain has greatly enriched my research and provided invaluable perspectives on my work. Their constructive feedback and commitment to my academic growth have had a key role in the successful completion of my PhD and helped me through the most challenging phases of this journey.

My appreciation extends to the University of Oslo, where I have had the privilege of studying and conducting my research. The university's academic environment, facilities, and financial support have been crucial in facilitating my research endeavors.

I would also like to express my gratitude to Professor Tobias Wrigstad from Uppsala University for the opportunity to collaborate on my last research efforts. His insights, expertise, and willingness to share his knowledge have enriched my research and broadened my horizons.

My thanks extend also to Erik Österlund at Oracle Corporation for his valuable insights and support in conducting valuable research on ZGC, one of the most powerful GCs available for JVM. His technical perspective and knowledge have added a practical dimension to my research.

The importance of my parents in my academic journey cannot be overstated. Their unwavering belief in me, endless encouragement, and sacrifices have been my source of strength. Their love and support have been the foundation upon which I have built my academic pursuit without any worries.

Lastly, but most importantly, I want to express my deepest gratitude to my husband, Amirhossein Hassaneini. Throughout the ups and downs of this challenging journey, he has been my constant source of emotional support and motivation. His understanding, patience, unwavering belief in my abilities, and high technical knowledge and intelligence have sustained me during the most demanding times.

Pursuing a PhD has been a pivotal experience for me that sparked substantial personal growth. The journey wasn't without its struggles, I faced obstacles and doubts along the way. However, through determination, hard work, and support from others, I found the resilience to push forward. Through the program, I've expanded my academic knowledge and research skills and I feel more prepared

to make meaningful contributions in my field. The PhD taught me the value of hard work, critical thinking, intellectual humility, and maintaining enthusiasm for learning. These lessons will stay with me well beyond graduation as I move forward in my career.

Sanaz Tavakolisomeh
Oslo, April 2024

Abstract

Garbage collection plays a vital role in Java applications by freeing up unused memory allocated by applications. Selecting an optimal Garbage Collector (GC) and heap size is crucial for application performance yet poses significant challenges for users and developers lacking expertise in this domain. This thesis comprises a progression of research papers, each extending the groundwork of the preceding one to provide solutions where prior works have not adequately covered. Together, these contributions form a coherent sequence of investigations that collectively address the intricate challenges of GC and heap size selection. It begins by conducting a comprehensive evaluation of several widely used GCs in OpenJDK HotSpot, considering performance metrics such as application throughput, GC pause time, and memory usage. Our findings enable the selection of the most suitable GC for CPU and I/O-intensive applications based on user-defined performance goals. Building on these findings, a system is proposed to automatically recommend a proper heap size and the most suitable GC based on user-defined application requirements. Further, an adaptive heap sizing technique is introduced that adjusts heap size dynamically based on a user-defined GC CPU utilization limit rather than estimating application memory needs. Evaluations demonstrate this strategy matches the memory usage to the application memory usage pattern. This thesis offers a user-centric perspective on simplifying the GC and heap size selection process by empowering users to meet application goals without extensive GC expertise.

List of Papers

Paper I

Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira, "Selecting a GC for Java Applications", in: *2021 Norsk IKT-konferanse for forskning og utdanning (NIKT) Conference*, pp. 2–15, November 2021, Trondheim, Norway. https://ojs.bibsys.no/index.php/NIK/article/view/912

Paper II

Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira, "BestGC: An Automatic GC Selector", in: *IEEE Access Journal*, vol. 11, pp. 72357–72373, July 2023, https://ieeexplore.ieee.org/document/10177931

Paper III

Sanaz Tavakolisomeh, Marina Shimchenko, Eric Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad, "Heap Size Adjustment with CPU Control", in: *20th International Conference on Managed Programming Languages and Runtimes (MPLR)*, October 2023, Lisbon, Portugal, https://dl.acm.org/doi/10.1145/3617651.3622988

Other Publications

Doctoral Symposium

Sanaz Tavakolisomeh, "Selecting a JVM Garbage Collector for Big Data and Cloud Services", in: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*, pp. 22–25, December 2020, Virtual Event / Delft, The Netherlands. https://dl.acm.org/doi/abs/10.1145/3429351.3431745

Extended Abstract/Poster

Sanaz Tavakolisomeh, Rodrigo Bruno and Paulo Ferreira, "BestGC: An Automatic GC Selector Software", in: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR)*, pp. 144–146, September 2022, Brussels, Belgium. https://dl.acm.org/doi/10.1145/3546918.3560804

Other Contributions

Collaborative Work

Research and development of a novel Java Virtual Machine (JVM) option for Z Garbage Collector (ZGC). This groundbreaking feature allows users to set an upper bound for the garbage collector CPU utilization while adjusting the heap size automatically. As a result of this work, the feature is set to be included in the Open Java Development Kit (OpenJDK). The work has been done in collaboration with Oracle Corporation in Sweden, Stockholm, and a research group at Uppsala University led by Professor Tobias Wrigstad.

Software

- **Buffer Bench (B2)**: A simple software (written in Java) designed to stress GCs through extensive read and write operations.

  GitHub Repository: https://github.com/SaTaSo/B2

- **BestGC**: This tool suggests the most proper GC solution and an optimal heap size for a user application based on the user's preferences regarding application throughput and GC pause time.

  GitHub Repository: https://github.com/SaTaSo/BestGC-Software

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the Java ecosystem, the Java Virtual Machine (JVM) is a core component that provides a runtime environment for executing Java bytecodes (resulting from compiling the Java source code). Automatic memory management plays a key role in the JVM and includes the garbage collection process. The garbage collection process frees up memory that is no longer in use by an application and ensures the optimal performance and stability of Java applications while minimizing the effort of the developer.

Java applications generate a substantial amount of objects during their execution. These objects become dead, and therefore garbage, when they are no longer being used by the application, which can cause memory usage to grow rapidly if not managed effectively. A Garbage Collector (GC) helps to identify and free up dead objects, thus preventing memory leaks and ensuring the application remains responsive. In the absence of an efficient GC, Java applications can experience performance degradation, leading to increased resource usage, longer response times, and even failures. In addition, a GC increases the developers' productivity by eliminating the need for them to worry about managing unused objects.

There are several collection algorithms available in the JVM with different characteristics and goals. Consequently, some GCs may better meet different application requirements. For instance, a throughput-oriented GC may be a better option for applications that need a high rate of processing, while a low-pause GC is a better choice for an application with strict latency requirements. To select a suitable GC for a Java application, the user would need to understand the characteristics of different GC algorithms, their resource requirements, their impact on application performance, and how a GC aligns with the application's goals. A poorly chosen GC negatively impacts the application's overall performance and results in longer GC pause times, higher CPU usage, or increased memory usage.

In this chapter we begin by explaining the motivation behind this research, providing a clear understanding of the driving forces that prompted this thesis. Subsequently, we articulate the research questions and objectives, specifying the specific goals and the key inquiries that this thesis aims to address. Then, we explain the system requirements, the constraints that direct our research and implementations in this thesis. By identifying unaddressed gaps in previous works, we show the distinctive contributions of this research to the existing body of knowledge. Moreover, we outline the evaluation approach and the research findings. Finally, this chapter concludes with a summary of the key points discussed and an overview of the thesis structure, providing readers with a roadmap for the forthcoming chapters, which will prepare the details of our

research and its outcomes.

1.1 Motivation

Selecting an appropriate GC and optimizing heap size stands as a pivotal challenge in the development and management of applications. The complicated process of choosing the right GC for a user or developer who is not knowledgeable and an expert in GCs makes this process even harder. To make a proper GC selection, users would need to understand the benefits and limitations of available GC solutions and match them with their application's requirements (e.g., GC pause times, application throughput, etc.). Moreover, considering certain application demands, like memory requirements, determining an adequate memory size can be a complex task, often necessitating users to engage in multiple rounds of trial and error to identify the optimal memory size for their applications. Alternatively, users could try to increase the resource's capacity to overcome issues such as Out-of-Memory (OOM) errors and subsequent failures, or slow application execution. However, these approaches are time-consuming, expensive, and inefficient in terms of resource utilization.

In this thesis, we aim to address these challenges (selecting the best GC and optimizing the heap size) by providing solutions that simplify them. Our focus is on assisting users and developers (hereafter referred to as "users") who may lack expertise in this field, enabling them to make informed decisions without doing costly hardware upgrades or spending time in a time-consuming process.

1.2 Research Questions

As modern applications continue to grow in complexity, the evolution of GC algorithms becomes vital to meet the diverse memory requirements. Nevertheless, selecting the ideal GC algorithm remains a tough challenge. This challenge becomes even more critical when considering performance metrics like application throughput, GC pause time, and memory usage. Consequently, addressing these difficulties calls for the development of solutions to aid users in choosing the most suitable GC for their applications. In addition, it highlights the need for innovative solutions to assist users in determining the optimal amount of memory that aligns with their application's requirements, all while preserving other crucial performance metrics.

In the following, we present a set of research questions that aim to address these challenges, including:

- RQ 1: How do different GC solutions perform in terms of performance metrics, such as application throughput, GC pause time, and memory usage, for various applications?

- RQ 2: How can users without GC expertise select an appropriate GC algorithm or an optimal heap size for their applications?

- RQ 3: Can a software tool be developed to suggest the most appropriate GC algorithm for an application based on desired performance metrics, such as application throughput, GC pause time, and memory usage?

- RQ 4: Given the complexity of preventing memory stalls by determining an optimal memory size for an application, are there alternative metrics available that allow users to manage GC overhead and memory usage effectively?

1.3 Objectives

This thesis aims to explore the GC domain with a user-centric approach, offering solutions to address the challenges mentioned above (i.e., GC selection and heap size determination). By taking into account a user's application requirements, the research strives to empower users with the tools and insights needed to make well-informed decisions about GC and heap size configuration.

To begin our exploration, we initiate an evaluation and comparison of various GC solutions across different application classes and offer a GC for each class. Subsequently, based on our previous findings, we focus on automating the GC and heap selection processes for users. To this end, we introduce a system designed to recommend an appropriate GC (within the types of concurrent or non-concurrent GCs) based on the requirements of the user's application. This step includes determining an optimal heap size for running the application and assessing whether the application is CPU-intensive, considering the pivotal role of the CPU as a shared resource between the GC and the application. Then, continuing our investigation into the provision of an optimal GC and heap size for users, we address the critical concern of determining an optimal heap size for users' applications. Our emphasis is on overcoming the challenge of selecting the most suitable heap size for users. Considering the advantages of concurrent collectors, which operate simultaneously with the application and are designed to function independently of the application's critical path, we implement our adaptive heap size strategy within one of the concurrent collectors that showed the best results in the previous steps.

In pursuit of these aims, our objectives are:

- to analyze and compare available GC solutions for different application categories; it allows for gaining a better understanding of the trade-offs in the GC solutions regarding application throughput, GC pause time, and application memory usage.

- to propose a software system, BestGC, that suggests the most suitable GC solution for an application based on the user's application performance goals, in addition to suggesting an appropriate heap size for running the application.

- to propose a strategy to adapt the memory size based on the GC CPU utilization, to follow the previous research and offer users an alternative approach for optimizing their application's heap size.

1.4 System Requirements

To effectively frame the scope and methodological approach of this thesis, it is important to outline the system requirements, more precisely, specific parameters and constraints to direct our research in this thesis. Establishing these requirements helps ensure the research contributions are effective and practical. Formulated in alignment with our research objectives, these system requirements serve as guiding principles, and keep the research focused while highlighting the foundations on which the findings are built.

To meet our first objective (as outlined in the previous section), the primary requirement is to establish an approach and define the categorization of applications into CPU-intensive and I/O-intensive categories. It is important that the suggested GC for each application category results from comprehensive evaluations and maintains a high level of accuracy. Furthermore, the strategy developed should have the flexibility in integrating new GC algorithms. This research forms the foundational framework of our thesis by enabling evaluation and a deeper understanding of various GCs, particularly Stop The World (STW) and concurrent GCs.

For implementing and developing BestGC, the requirements include the following key aspects:

- BestGC should run for a specific duration, by default set to 30 seconds but modifiable by the user. The BestGC runtime should be shorter than the user's application to ensure efficiency in its utilization.

- BestGC should be flexible to user preferences, enabling them to specify their optimization preferences in terms of application throughput versus GC pause time.

- The methodology employed by BestGC should be extensible to new JDK versions, diverse GCs, and varying heap sizes.

- Utilizing the GC suggested by BestGC should provide better overall performance compared to simply using the default GC provided by the JDK.

To implement a strategy to adapt the heap size based on the GC CPU utilization, the following requirements are considered:

- The strategy should automatically adjust the heap size in response to the user's application memory usage pattern, eliminating the need for manual heap tuning.

- The strategy should not negatively impact application performance metrics like application throughput and latency compared to the default heap management strategy of the GC.[1]
- The strategy should converge towards a user-defined limit for GC CPU utilization throughout the application's execution.
- The strategy should avoid excessive changes to the heap size during each round of adjustment.

These system requirements establish a framework and serve as the foundation for our implementations. This framework additionally ensures the effectiveness and reliability of our proposed strategies and contributions.

1.5 Unaddressed Gaps in Previous Works

While previous studies have evaluated and compared GCs and suggested heap adjustment strategies, at the time of conducting our research contributions and to the best of our knowledge:

- Previous works have not adequately covered the process of selecting the best GC solution based on specific application characteristics, such as whether it is CPU or I/O intensive. Additionally, these studies have not encompassed a thorough evaluation of the array of GCs employed in our research with a broad range of benchmark applications, regarding their impact on key performance metrics, including application throughput, GC pause time, and memory footprint.
- Previous works did not offer an automatic GC selection mechanism. BestGC is the first system to automate GC selection. It would be easily extensible to work with a new JDK version, new heap configurations, and new GCs beyond those it was initially trained with. Furthermore, although we developed BestGC for the JVM, its underlying algorithm has the flexibility to be adapted to other runtime environments and programming languages.
- Existing works have not introduced a heap size adjustment strategy based on GC CPU utilization for an in-production and well-known concurrent GC in widely used OpenJDK. We adjust the heap size to meet a specific GC CPU target, instead of estimating the amount of memory needed by the application. Our CPU-driven heap size adjustment is particularly

[1] It is important to note that energy constraints were also taken into account during this research process. However, we need to clarify that the energy consumption metric falls beyond the scope of this thesis. This metric was the primary focus of the research group with whom we collaborated for this research. Consequently, the emphasis on energy constraints within their research objectives influenced the consideration of this metric within the scope of the third paper.

important for concurrent collectors (like ZGC) which compete with the application thread (mutator) for CPU resources to collect memory, unlike the STW collectors used in prior studies. In other words, rather than directly controlling the heap as in previous works for STW collectors, we specify the desired GC CPU target and adjust the heap accordingly.

1.6 Contribution

The significance of this PhD thesis lies in the way it contributes to the understanding of how different GC solutions perform for different applications in terms of performance metrics like application throughput, GC pause time, and memory usage. This contribution is realized by developing a software system, `BestGC`, that suggests a heap size for executing the application and the most appropriate GC solution based on the application's requirements.

Also, due to the importance and complexity of choosing an optimal heap size for a user's application, this thesis aims to propose a strategy that helps users overcome the struggles of finding a proper memory size for their application; instead, they use a more intuitive metric, CPU, to control the frequency of GC invocations, and therefore, GC adjusts the heap size based on the application's memory usage pattern.

In summary, our contributions include: 1) classifying user applications into CPU/I-O intensive and offering a GC for each category, 2) designing and developing BestGC, and 3) adjusting heap size based on the GC CPU limit. Our research papers that formulate our contributions represent a progression from the initial GC analysis to an automated selection system and to a fully adaptive heap tuning mechanism. Each research paper is built on the insights from the previous paper to incrementally advance capabilities for optimizing garbage collection. Our techniques have matured from manual methodologies to automated systems for GC and heap optimization.

Figure 1.1 illustrates how our contributions, as exemplified in our research papers, directly and comprehensively address the research questions outlined in Section 1.2.

1.7 Evaluation

The evaluation criteria in this thesis is based on choosing applications that effectively stress GCs, have various memory usage patterns, report relevant performance metrics essential for GC algorithm examinations, and represent real-world scenarios. Consequently, as indicated next, we chose well-known and broadly utilized benchmark suites within the GC domain.

In the first research paper, "Selecting a GC for Java Applications" (published in NIKT'21), we employed DaCapo [Bla+06] and Renaissance [Pro+19] benchmark suites. Additionally, we introduced a custom benchmark, Buffer Bench (B2), designed to do a large volume of read and write operations to study

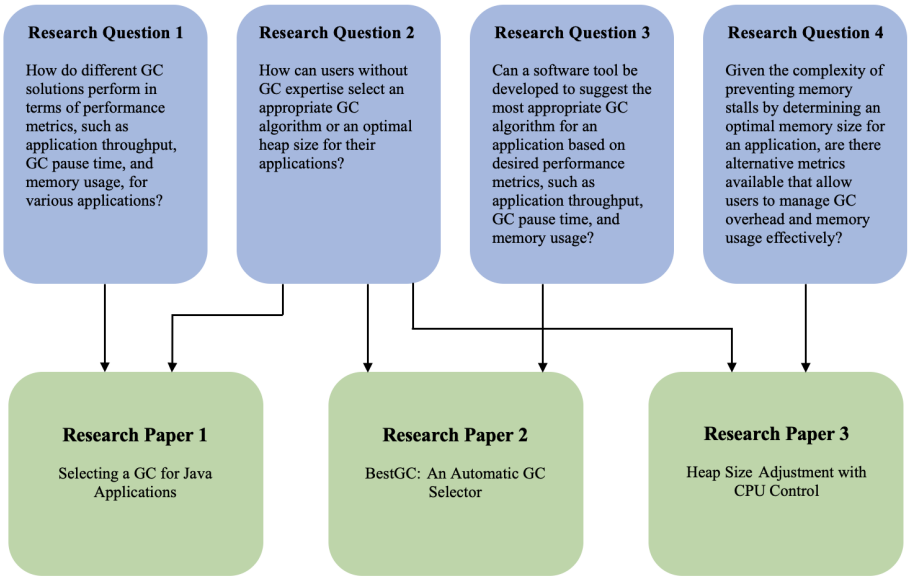


Figure 1.1: How each of the key research questions are addressed in our research papers.

GCs' behavior and the cost associated with the read and write barriers[2] that GCs use. We further utilized the PetClinic [Spr07] application in our evaluations, a web-based model of a veterinary clinic management system. These applications enabled us to assess GCs with varying data profiles. We categorized applications as CPU-intensive and I/O-intensive (basically, we define as I/O-intensive all applications which are not CPU-intensive) and assessed performance metrics such as application throughput, GC pause time, and memory usage before and after GC; then, we identified the most suitable GC solution (among CMS [Ora15], G1 [Det+04], ZGC [CK22], and Shenandoah [Flo+16]) for each category and metric. CMS excelled in application throughput, while ZGC proved to be the top choice for minimizing GC pause times in both CPU and I/O-intensive applications. In terms of memory usage before GC, CMS performed well in both application categories, with Shenandoah being equally effective, for I/O-intensive applications. Regarding memory usage after GC, G1 outperformed the other GCs in CPU-intensive applications in freeing memory after garbage collection, whereas Shenandoah matched G1's success in I/O-intensive scenarios.

For the second research paper, "BestGC: An Automatic GC Selector" (published in IEEE Access Journal), Renaissance and DaCapo's Chopin version[3] provided us with several standardized applications for training our system,

[2]A read barrier (also known as a load barrier) is executed when reading or accessing object references in memory. In contrast, a write barrier's code snippet is invoked to track and manage changes to object references [PGM19].

[3]https://github.com/dacapobench/dacapobench/tree/dev-chopin

BestGC. Subsequently, we tested BestGC's performance using a third benchmark suite, SPECjvm 2008 [Sta08], as a representative of applications that might be used by users of BestGC. BestGC recommends the optimal GC solution in approximately 51.24% of cases and correctly identifies a GC with the appropriate concurrent or non-concurrent category about 85.95% of the time for user applications. Even when BestGC falls short of suggesting the best GC, utilizing the recommended GC still results in a slight improvement compared to using default OpenJDK GC (G1).

In the third research paper, "Heap Size Adjustment with CPU Control" (published in ACM MPLR'23), following the discussions with the Oracle team located in the Stockholm office, we decided to do the experiments using the latest version of DaCapo Chopin. To include more latency-sensitive applications, we utilized the Hazelcast [Gen+21] benchmarks in our evaluation. The findings revealed that setting GC CPU limits higher than what an application truly utilizes leads to reduced memory usage due to an increased frequency of garbage collections. The proposed dynamic heap sizing strategy did not have a significant impact on the execution time of most of the applications. In terms of application latency, the proposed approach did not have a negative effect on latency and even demonstrated the potential for latency reduction in some instances. Most notably, the evaluations suggested that a 15% GC CPU target could be a favorable default choice for optimizing the trade-off between memory usage, execution time, and energy consumption while eliminating users' concerns about finding an optimal heap size for their applications.

1.8 Summary

In this thesis, we delve into the realm of GCs within the JVM ecosystem, with an emphasis on simplifying the challenges associated with selecting the appropriate GC and managing heap sizes for Java applications. These challenges particularly arise for users lacking expertise in GC and heap-related issues.

We propose a GC selection methodology that provides a framework for analyzing application characteristics and guiding the choice of suitable GC algorithms based on the application performance goals. This lays the groundwork for developing more sophisticated optimization approaches.

Building upon these fundamental insights, the BestGC system represents an approach to automate GC selection. By removing the need for tedious manual GC selection, BestGC streamlines GC selection for production environments; while it also suggests a heap size for application execution.

The introduction of a dynamic heap tuning technique controlled by GC CPU overhead further enhances BestGC's capabilities. Adaptively resizing heap memory based on real-time feedback from GC CPU utilization pushes GC optimization to the next level.

These projects have advanced GC optimization from manual profiling methodologies to automated systems. These advances promise to streamline GC efficiency in emerging application domains like cloud computing.

The insights gained and solutions developed in this thesis open avenues for further exploration and advancement in the field of Java memory management, impacting both research and industry practices.

1.9 Thesis Structure

This PhD thesis adopts a *collection of articles* format while maintaining a well-structured and coherent narrative throughout several core chapters.

In Chapter 2, various aspects of garbage collection are explored, including performance metrics related to the GCs, GC methods and strategies, and an overview of existing GCs. Chapter 3 offers a review of prior research on GCs, serving as a vital foundation for the present study.

Chapter 4 elaborates on the overall architecture underpinning the key concepts presented in the research papers. Chapters 5 and 6 provide practical insights into our proposed solutions and present the results we have obtained, respectively.

Finally, Chapter 7 consolidates the findings of our research, offering a comprehensive summary of the thesis. This organizational framework is designed to provide insights into our solutions, which are aimed at simplifying user interactions with GCs and mitigating the complications associated with GC and heap size selection.

For a more detailed exploration of the topics discussed, the published research papers, which are referenced throughout the thesis, are available in Paper I, Paper II, and Paper III.

Chapter 2

Garbage Collection Foundations

Memory management plays a key role in utilizing the available memory resources. There are several programming languages in which the process of memory (de)allocation is performed explicitly by the developer, like C and C++. In contrast, several programming languages, like Java, benefit from automatic memory management achieved through the usage of garbage collection techniques. In the presence of a GC, developers no longer need to struggle with the process of managing memory (de)allocation for the objects in their applications.

The Java Virtual Machine (JVM)[1] is a widely-used runtime system for executing Java bytecodes that employs GCs to manage the memory automatically. The JVM uses a heap as a memory pool to provide dynamic memory allocation in Java programs. This heap is a designated region of memory that can dynamically expand or contract as the application requires during runtime. Whenever objects are created in Java, the JVM assigns memory from the heap to store these objects and accommodate their data.

In Java, each object in memory includes a header and the object's data. The header contains objects' class-related and control information. The data component includes different fields that hold the object's characteristics with different data types (such as Integer, Boolean, etc.); it also holds references to other objects. Interconnection between objects, through the references, forms an object graph in the heap. The object graph's structure is changed dynamically when an object is created, modified, and destroyed during an application's execution time. JVM employs a GC to identify live objects that are currently accessible or referenced by other objects and reclaims dead objects that are no longer needed or accessible by the application.

In the most commonly used collection approach (called tracing, see Section 2.2.2), the GC traverses the object graph starting from root objects, e.g., global variables, static variables (variables associated with a class rather than specific instances of that class) or local variables (i.e., ones that are on the stack). Then, it marks live objects and reclaims the dead ones. Figure 2.1 illustrates four Java objects in the heap and how they connect to each other through references. The corresponding object graph of the existing objects in the heap is also shown in the same figure (on the right-hand side). Object 2 is not referenced by any live object in the object graph. Therefore, it is identified as a dead object, and the GC will collect it.

In the subsequent sections, we explain various performance metrics that play a crucial role in evaluating the performance of GCs. These metrics are important for evaluating GCs in managing memory in Java applications. Following this, we

[1] https://docs.oracle.com/en/java/javase/20/vm/java-virtual-machine-technology-overview.html

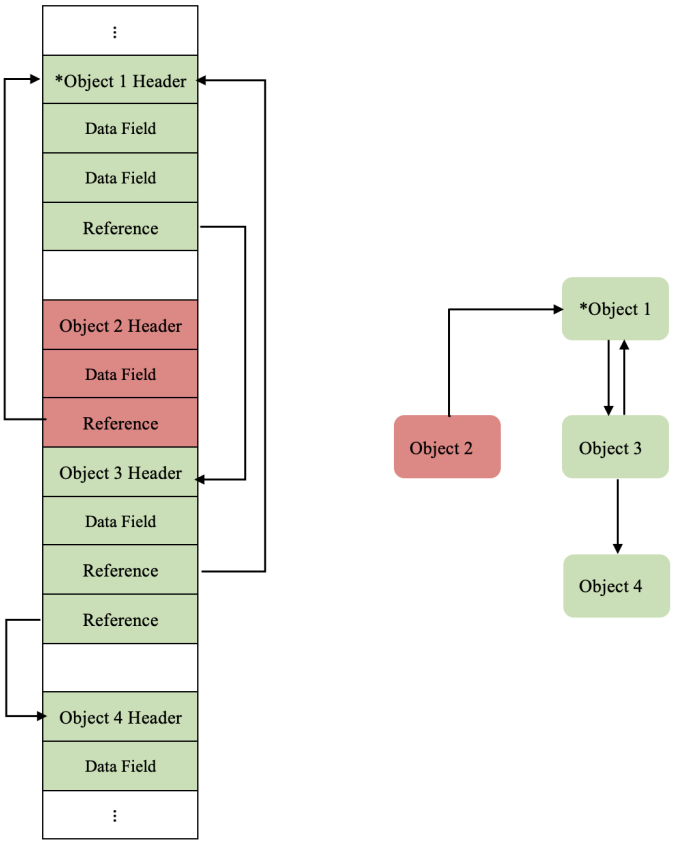


Figure 2.1: Objects in the heap (left) and corresponding application-level's object graph (right). Object 1 (marked with *) is the root object. Live objects are highlighted in *green* and dead objects in *red*.

will explore diverse algorithms and strategies employed by GCs in the process of reclaiming memory from objects no longer in use. To offer a more comprehensive perspective, we will also provide an overview of the specific GCs utilized in our thesis, highlighting their features and how they contribute to the overall memory management scheme.

2.1 Performance Metrics

In the realm of evaluating GC algorithms, a set of key performance metrics, some application-related and others GC-related, comes to the forefront. However, reaching an optimal balance among all these metrics is not possible at the same time and GCs may need to sacrifice some of them to enhance the others. It

depends on the goals each GC follows as well as the resource limitations on the machine and the requirements of a particular application. The most important of them are explained next.

Application Throughput is the number of operations per amount of time. The goal for every GC is not to decrease the application throughput (but to increase it, if possible). Thus, the maximum application throughput goal focuses on optimizing the time spent in the GC to increase the application throughput. Taking OpenJDK HotSpot JVM as a reference implementation for Java, hereafter we call it "Java"[2], users can define a throughput goal using the command line `-XX:GCTimeRatio=nnn`, where the ratio of GC time to application execution time is 1/(1+nnn) [Ora22]. For instance, if a user wants the GC time to take 10% of the application execution time, he/she should set `GCTimeRatio=9`[3]. Accordingly, the GC changes the heap size to meet the throughput goal.

GC Pause Time is the duration of pauses in which the GC halts the mutator. The GC pause time goal focuses on minimizing such pauses. In Java, users can define a maximum GC pause time goal using the command line option `-XX:MaxGCPauseMillis=<nnn>`. If the average pause time exceeds the specified maximum pause-time goal, the GC determines that the goal is not being met, and then it adjusts relevant parameters like heap size to ensure that GC pauses stay within the specified limit. The adjustments may lead to more frequent collections, resulting in a potential decrease in overall application throughput.

Application Memory Usage refers to the amount of memory used by an application. When an application throughput and GC pause time goals are achieved, GC embarks on reducing the size of the heap to optimize memory usage. The GC tries to make a trade-off in such a way that the application's memory usage is optimized, GC pauses are minimized, and the overall throughput of the application is maximized. However, users are allowed to specify a minimum and maximum heap size using the JVM options `-Xms=<nnn>` and `-Xmx=<mmm>`. Therefore, they can control the initial heap size at application startup and the maximum size of the Java heap can reach during application runtime.

CPU Utilization This metric shows the percentage of CPU resources allocated to garbage collection activities. Balancing CPU utilization between application tasks and garbage collection tasks is important for overall system performance. Defining an upper bound for this metric affects the frequency of the GC invocations within the JVM.

[2]When we mention 'Java', we are referring to both the Java language and its implementation, OpenJDK HotSpot JVM.

[3]This means that the application should get 9 times more working time compared to the time given to garbage collection.

Application Latency This refers to the delay experienced in application responsiveness imposed by the GC for the collection. Low-latency GC is essential for maintaining responsive applications, especially in real-time and interactive scenarios.

2.2 GC Algorithms

Continuing from the discussion of the above-mentioned performance metrics in the context of GC, this section explores the distinct strategies used in collection algorithms, with a focus on two pivotal approaches: reference counting and reference tracing.

2.2.1 Reference Counting Algorithm

As the name implies, the Reference Counting (RC) algorithm [BM03] keeps, for each object, the number of incoming references from other objects in a counter. When a new reference to an object is created, it increments the counter. It also decrements the counter whenever a reference pointing to that object is removed. In fact, when an object is destroyed, the counters of all objects referenced by it are decremented, potentially leading to further reclamations. If an object's counter reaches zero, it becomes inaccessible, and its memory can be reclaimed. It should be noted that these operations have to be atomic to prevent wrong counter updates.

Reference counting is considered a direct collection algorithm as it identifies unreachable objects directly. It aims to reduce the collection pauses and improve responsiveness. However, it comes with the overhead of keeping a counter for each object in the heap, along with the overhead of updating the counter for all the referenced objects when an object is destroyed. Another major drawback of the reference counting algorithm, as shown in Figure 2.2, is that it can't reclaim cyclic references, which leads to memory leaks. In Figure 2.2, if Object 1's counter field falls to zero, it becomes dead, and it immediately will be collected by the GC; consequently, the counter of Object 2, referenced by Object 1, will be decremented by one and will become 1. So, the RC algorithm does not recognize Object 2 as garbage in the current collection cycle. Therefore, Objects 2 and 3, which have cyclic references, remain in the heap; while they are unreachable from other objects, and it is not possible for the RC algorithm to reach these objects.

To address this issue, other techniques, like the reference tracing algorithm, are employed to trace object reachability from the root objects.

2.2.2 Reference Tracing Algorithm

Reference Tracing (RT) [McC60] is the primary class of algorithms used in HotSpot JVM (the default JVM implementation in OpenJDK). RT, also known as the mark-and-sweep algorithm, aims to trace and mark the live and reachable

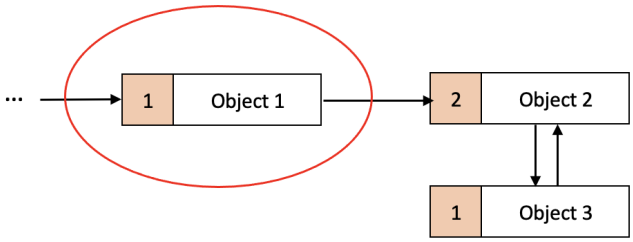


Figure 2.2: Cyclic reference issue in RC algorithm. Highlighted fields in *orange* are counter fields. When Object 1 is collected, Objects 2 and 3 remain in the heap without any path from other objects reaching them.

objects starting from the root objects in the object tree and then reclaim the memory used by dead objects.

The RT algorithm starts traversing the object graph from the root objects (static variables, global variables, etc.); then, it follows references to other objects. If the GC finds an object throughout traversing, it marks the object as *live*. It shows that the marked object is still reachable and, therefore, in use by the application. During the marking phase, collectors can use various data structures to keep track of marked objects. Since the RT algorithm marks live objects instead of dead ones, it is also called an indirect collection algorithm.

After finishing marking the live objects in the entire object graph, the sweeping phase is started. In this phase, GC reclaims all the memory occupied by the objects not being marked *live*. GCs may also use different techniques during the sweeping phase to update the reference fields of live objects and the data structure it uses to keep track of the objects.

By sweeping the objects that are not referenced by other objects, i.e., not marked as *live*, the RT algorithm handles the cyclic referenced (non-reachable) objects. However, RT also comes with some overheads. RT needs to traverse the entire object graph, the entire heap, to mark live objects. It introduces a performance overhead, especially when the heap size is too large since the overhead of tracing is proportional to the number of live objects. This overhead becomes more problematic when the GC needs to stop the application threads (mutators) to perform the marking/sweeping. In addition, scattered free memory segments appear in the heap after memory deallocations (this is called memory fragmentation). So, it might be hard to find a contiguous heap space to allocate new (big) objects. GCs may need techniques to remove the fragmentation to provide bigger chunks of memory as well as increase locality.

In order to deal with memory fragmentation, GCs may utilize copying or compaction methods to defragment live objects:

- Copying: By dividing the heap into two semispaces, in each GC cycle, GC uses one semispace as *from-space* and the other as *to-space*. Then, it uses

the *from-space* to allocate objects and, by moving and grouping all the live objects to the *to-space*, tries to eliminate the gap that occurred between the live objects. This approach works efficiently when the number of live objects scattered in different heap spaces is low; thus, copying them will not cost a lot.

- Compaction: GC rearranges and moves the live objects to a part of the heap space. By compacting all the live objects, GC removes the gaps between the live objects and creates a larger contiguous block of empty memory.

It is worth mentioning that GCs also may use a combination of copying and compaction techniques to use the heap optimally.

2.3 Garbage Collector's Strategies

In this section, we explain different GC modes including serial, parallel, stop-the-world, incremental, and concurrent. Additionally, we explore the main heap layouts employed in the GCs.

2.3.1 GC Modes

GCs may use different modes to do the collection, such as:

- Serial: In this mode, the GC uses only one single thread to perform the collection task. A serial GC best suits machines with a single processor, it cannot take advantage of multiprocessors, and is also a good choice for simple applications. Serial GC eliminates the overhead of syncing different GC threads. However, it causes long GC pauses in mutators' execution, making it less suitable for applications that demand high responsiveness.

- Parallel: In the parallel implementation, multiple threads are used to perform the collection task. In this way, it speeds up the garbage collection process, reduces the GC pause time, and improves application throughput compared to utilizing a single thread for the GC. A parallel approach is good to use on machines having multiprocessors or multithreaded hardware available. Due to utilizing several threads, GCs implementing the parallel approach use more system resources.

- Stop-The-World (STW): To ensure a consistent state during the garbage collection process, this strategy results in GC pauses where all the mutators stop from execution during the collection process. In this way, it allows the GC to perform efficiently without being affected by the mutators. Thus, the GC performs an accurate collection. However, STW induces delays in application responsiveness and negatively affects the application's latency. Both Serial and Parallel GCs stop mutators for garbage collection.

- Incremental: An incremental GC performs collection in steps, e.g., per memory page, per sub-heap, per sub-graph. This approach helps to minimize the impact of garbage collection on responsiveness by reducing garbage collection pause times.

- Concurrent: Employing concurrent implementation allows GC threads to work with mutators concurrently. Therefore, concurrent GCs minimize GC pause times while improving user experience, compared to STW GCs. However, implementing a concurrent strategy is complicated. The GC should manage the heap while the state of the heap is frequently changing, and references pointing to objects may be destroyed or updated at the same time as the GC is examining the heap. So, GC needs to employ different data structures and various approaches to provide concurrency, which may cause memory overheads.

Figure 2.3 illustrates how the serial, parallel, STW, and concurrent GCs work with mutators.

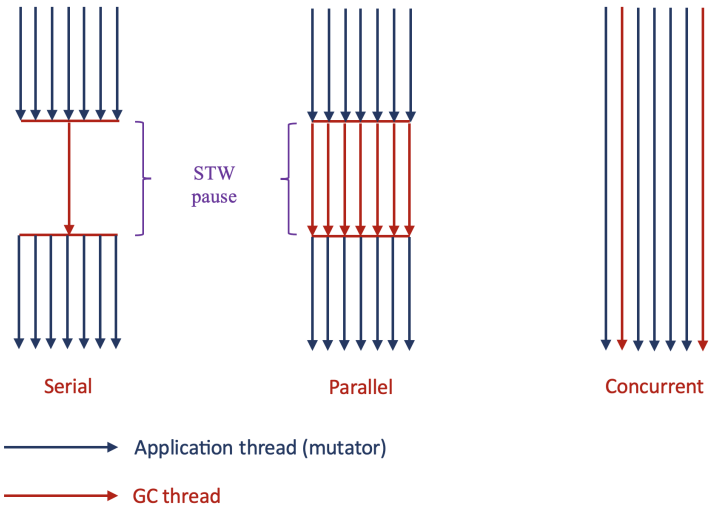


Figure 2.3: Serial and Parallel (STW GCs), and Concurrent GC working with mutators.

2.3.2 GC's Heap Layouts

GCs implement different strategies to manage the heap optimally and increase the effectiveness of garbage collection. Some algorithms choose to partition the heap to hold objects based on their lifetime (generations), and some may partition the heap into spaces (regions) with fixed sizes. These solutions allow holding the objects based on their size or other characteristics, and some may even use a mixed strategy.

Generational GCs Examining the entire heap in each GC cycle to determine the live objects imposes a significant overhead that affects different performance metrics, such as application throughput and GC pause time. However, the generational strategy follows the generational hypothesis [LH83] to make the collection process more efficient. The generational hypothesis is based on the observation that newly created objects by the application are more likely to die soon. Therefore, based on this hypothesis, GCs can hold the newly created objects in a different partition in the heap, known as the young generation, and examine this partition frequently through a minor collection. Following this approach, GCs are able to check a fraction of the heap, rather than the entire heap, seeking for garbage. Thus, they would spend less time in the collection process, i.e., shorter GC pause times, and therefore increase performance. Objects that survive several minor GC cycles are moved to the old generation. When a certain amount of the old generation is occupied, GC triggers a major collection to perform the collection in both the old and young generations.

Region-based GCs These GCs divide the heap into fixed-size regions and use these regions to store objects with different lifetimes. Different GCs may use various region sizes; they may split the heap into regions with equal or fixed different sizes. In this way, they can have good control over the heap. Dividing the heap into fixed-size regions reduces the appearance of small gaps between the objects and reduces the fragmentation in the heap (compared to non-region-based approaches). This strategy can help GCs target regions with more garbage instead of scanning the entire heap.

2.4 Overview of Existing JVM GCs

Different GCs have been developed for the JVM in the field of automatic memory management to effectively manage memory. This section aims to give an overview of the landscape of existing GCs, more specifically in OpenJDK. These GCs are the ones used in our research papers to address the research questions mentioned in Section 1.2.

2.4.1 Serial GC

The Serial GC [Ora22] has been available since the early versions of OpenJDK. As explained in Section 2.3.1, it uses one single thread to perform the garbage collection tasks. Serial GC is a STW GC that halts mutators while it is doing the garbage collection process. This GC is the primary choice on single-processor machines, yet it can be used on multiprocessor machines for applications with small data sets. Since it uses a single thread it eliminates all the overheads of thread communications and syncing needs in other multi-threaded GCs. Serial GC is a generational GC that uses mark-copy (identifying live objects and moving them to a new location in memory) in the young generation and a mark-sweep-compact in the old generation. However, using a single thread makes all the tasks

in the young and old generation serial which leads to longer GC pauses. Serial GC can be enabled by the command-line option `-XX:+UseSerialGC` available in JVM.

2.4.2 Parallel GC

The Parallel GC [Ora22] is the default collector for JDK8 and earlier[4] and utilizes multiple threads to perform the garbage collection tasks. It is suitable for multiprocessor machines to run applications with medium-sized to large-sized data sets. Parallel GC is also known as the throughput collector since it uses several threads for the GC while between the collection cycles, it does not use any resources for GC threads. Although using several threads speeds up the collection process compared to the serial GC, it is still a STW GC that pauses mutators to do the collection.

Parallel GC is also a generational GC that uses mark-copy in the young and mark-sweep-compact in the old generation and in a parallel manner. The number of threads used in GC can be defined with a command-line option `-XX:ParallelGCThreads=NNN`. The default number of threads in this GC is equal to the number of cores in the machine. If the Parallel GC spends an extended time in the garbage collection process while it recovers a small portion of the heap (less than 2%), it will throw an OOM error. The Parallel GC adjusts the sizes of the generations to meet its performance goals. These adjustments are made based on statistics and tests performed at the end of each collection. Parallel GC can be enabled using the `XX:+UseParallelGC` option.

2.4.3 Concurrent Mark and Sweep GC (CMS)

CMS [JHM16] was introduced in OpenJDK5 and was deprecated after OpenJDK version 13. The CMS GC is a generational GC that is designed to decrease the GC pause times (compared to STW GCs) by doing most of the garbage collection process concurrently with the mutators in the old generation. The collection process in the young generation includes a mark-copy task which is done while mutators are stopped (STW). However, in the old generation, the GC stops mutators shortly for an initial marking phase, and it continues marking the live objects concurrently with the mutators. CMS imposes another STW pause (longer than the first one in the old generation) to mark live objects that were missed during the concurrent marking phase (as shown in Figure 2.4). Once the marking phase is completed, CMS starts a concurrent sweeping phase to reclaim the memory of the dead objects.

Due to the concurrent phases in CMS, it reduces the GC pause times compared to the Serial and Parallel GCs. However, this concurrency comes with the cost of using more CPU resources because of the additional threads it utilizes. In addition, CMS causes floating garbage to appear in the heap. Floating garbage refers to those live objects that become unreachable in the concurrent phase of

[4] https://blogs.oracle.com/javamagazine/post/java-garbage-collectors-evolution

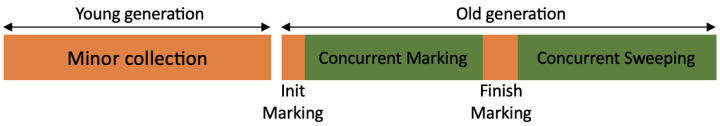


Figure 2.4: CMS GC's concurrent (green) and STW (orange) phases.

CMS since mutators are working simultaneously with the GC threads. These dead objects remain garbage in the heap waiting to be collected by the next GC cycle. Furthermore, because of the lack of a compaction process in CMS, removing dead objects results in fragmentation in the heap. The scattered heap increases memory usage since a contiguous space is not available to hold new large objects. The CMS GC is enabled with the command-line option `-XX:+UseConcMarkSweepGC`.

2.4.4 Garbage-First (G1) GC

G1 [Det+04] became the default GC since OpenJDK9. It is a generational and parallel GC that does most of the garbage collection tasks concurrently with the running application while including some STW pauses. G1 strives to make a balance between application throughput and GC pause time goals. It performs some garbage collection tasks in STW pauses but to keep these pauses short, it performs heap reclamation in steps and incrementally in the heap. To achieve this, it divides the heap into fixed-size regions. Upon startup, the JVM sets the region size (from 1 MB to 32 MB depending on the heap size). The goal is to have no more than 2048 regions. Instead of assigning portions of the contiguous heap to young and old generations, G1 assigns each of its regions to one generation when it is needed. It keeps track of the number of live objects in each region and uses this information to create a collection set of regions. As its name implies, *Garbage First*, it starts reclaiming the regions with the most garbage first.

G1 starts with a *young-only* phase in which it moves live objects that have survived multiple GC cycles in the young generation to the old generation. When the old generation is occupied to a certain threshold, G1 starts a *Concurrent Start Young Collection* phase. In this phase, G1 performs a normal young collection as well as marking all live objects in the old generation.

Marking the live objects in G1 includes two STW pauses:

- Remark: G1 completes the marking process, plus some additional cleanings like reclaiming empty regions. Also, it uses the Snapshot-At-The-Beginning (SATB) [Ora21] technique that takes a virtual "snapshot"[5] of the live objects at the moment; then, it uses the live object set captured in SATB to trace objects and choose the collection set. This provides an incremental and fast collection.

[5]A virtual snapshot is a representation or a reference to the current state of the heap, to track live objects during garbage collection without making a physical copy of the entire heap.

- Cleanup: It prepares the heap for the next phase (*space reclamation*) by counting the number of live objects in each region and then sorting the regions based on their live objects count. This helps G1 to perform efficiently; regions that are more likely to be collected (more garbage) and those that are better to be evacuated are prioritized.

Then, G1 proceeds with a space reclamation phase. In this phase, G1 performs a mixed collection both in young and old generations. It copies live objects from different regions to a new region while compacting. G1 also takes care of remembered sets [Det+04] which allow the collection in different regions.[6] In that case, if objects in one region are referenced by objects in other generations, they are considered live objects. Finally, parallel threads reclaim the memory for future use. It is worth mentioning that G1 utilizes *write barriers* to perform garbage collection. So, when an object reference is modified, the write barrier updates the necessary data structures, such as the remembered set, to maintain accurate information about which objects may have references to objects in other regions.

G1 is the default GC on most hardware and operating system configurations, however, it can be explicitly enabled using the JVM's command-line option `-XX:+UseG1GC`.

2.4.5 The Z GC (ZGC)

ZGC [CK22] is a single-generation (no concept of old and young objects) and low-latency GC designed to do heavy parts of garbage collection concurrently with the running application. ZGC aims to keep the GC pause times at some sub-milliseconds independent of the size of the heap, which can be between 8 MB to 16 TB; therefore, ZGC will have a minimum impact on a Java application's responsiveness. It is also a region-based GC that divides the heap into regions (also called *ZGC pages*) of three sizes: small (2 MB), medium (32 MB), and large (the size is configurable but should be more than 4 MB) [YÖW20b]. Small and medium regions can hold multiple objects, but large regions only accommodate a single large object [YW22].

Defining a good value for maximum heap size, which can be configured using the JVM option `-Xmx`, really matters while using ZGC. The more heap size available for the ZGC, the more headroom (space) it has to perform the collection tasks, and the better it accommodates the objects of an application. However, it should not come with the cost of wasting memory. ZGC was introduced as an experimental feature in OpenJDK11 and has become a production-ready feature since OpenJDK15.

ZGC utilizes *read barriers* to perform the garbage collection more accurately. A read barrier is invoked when GC reads a reference to an object and it makes sure that the GC has an up-to-date view of the object. Also, ZGC employs a data structure called *colored pointers* to manage objects in the heap. It uses four

[6] A remember set is a data structure that keeps track of the references from the old generation to the young generation.

unused bits of the 64-bit object pointer to store some metadata: Finalizable (F), Remapped (R), Marked1 (M1), and Marked0 (M0). When the bit *F* is set to 1, it indicates that an object is eligible for finalization, a process for cleaning up resources before it is deallocated (i.e., removed from memory). Having R bit 1 signifies that the object has been remapped during garbage collection. *M0* and *M1* are the bits used for marking objects during the garbage collection process (only one of them is used in each GC cycle). In ZGC there is a *good* color when one of R, M1, or M0 is 1 and the other three are zero. All the other combinations are considered as a *bad* color. ZGC uses one of the M0 and M1 at a time and decides the good color in each GC cycle. The distinction between good and bad colors helps to identify which objects need to be collected and which ones should be retained, improving the efficiency of memory management.

ZGC follows several phases to do the collection:

- STW Mark Start: A good color for the current cycle is chosen and all the root objects are marked as live objects.
- Marking and Remapping: GC threads mark live objects, and update the liveness information of pages (the total number and the total size of live objects on each memory page). Also, ZGC updates pointers to the objects that have been moved to the new locations through previous cycles.
- STW Mark End: ZGC checks if all the live objects have been marked.
- Reference Processing: ZGC ensures that references are properly managed and processed, then it does operations like clearing or updating references if necessary.
- Evacuation Candidates Set Selection: This phase involves adding sparse pages to the Evacuation Candidate (EC) set, removing empty pages, and sorting pages according to their number of live bytes. Based on a fragmentation limit, some pages will be removed from the EC set.
- STW Transitioning to Relocation: In this phase, a new good color with the R bit 1 is selected. Next, the roots are visited and all the objects they point to, while they are located on EC pages, are moved to non-EC pages and tinted with good color.
- Relocation: In this phase, GCs migrate all the objects in the EC sets. If all the objects in an EC page are evacuated, GC threads can reclaim the page. During this phase, ZGC compacts the evacuated objects to decrease fragmentation.

It should be noted that, at the time of writing this thesis, a generational version of ZGC had been released relatively recently with Java version 21. Generational ZGC [Kar21] will benefit from the current features of ZGC plus lower risks of allocation stalls, lower required heap memory overhead, and lower garbage collection CPU overhead. In addition, it provides applications with the

benefits of generational GCs that consider the short lifetime of newly created objects.

In order to enable ZGC, users should include the JVM option `-XX:+UseZGC`. Additionally, to activate ZGC's generational mode, the option `-XX:+ZGenerational` should be specified within the JVM configuration.

2.4.6 Shenandoah GC

Shenandoah [Flo+16] was developed by Red Hat and introduced as an experimental feature since OpenJDK11. Just like ZGC, Shenandoah also aims to reduce the GC pause times, regardless of the heap size, by performing most garbage collection tasks concurrently with the mutators. Shenandoah is a non-generational and region-based GC. It divides the heap into equal-size regions, similar to the heap layout in G1 (see Section 2.4.4).

Shenandoah's performance is dependent on the available heap size and live data set size. So, it is important to provide it with a reasonable heap size while being careful about memory resource wastage. Shenandoah uses various barriers to provide concurrency including write and read barriers which can induce throughput loss (up to 15% [Iri15]).

There are several phases in Shenandoah:

- STW Initial Mark: This phase prepares the heap for concurrent marking by scanning the root set. Therefore, the size of the root set directly affects the duration of the GC pauses in this phase.
- Concurrent Marking: GC threads traverse the object graph to mark the live objects while mutators are running.
- STW Final Mark: This phase finishes the marking phase and re-scans the root set. It also specifies the regions that need to be evacuated (evacuation collection set) and then evacuates some root objects to other regions (to reduce the amount of work required during the evacuation phase).
- Concurrent Cleanup: It reclaims regions with only garbage and no live objects as detected in the previous marking phases.
- Concurrent Evacuation: In this phase, GC threads evacuate (copy) live objects from the regions in the evacuation collection set to other regions. This phase is done concurrently with the mutators, which means that mutators may allocate new objects meanwhile. Shenandoah makes sure every read or write operation on an object is happening in the correct location of the objects in memory.
- STW Initial Update References: This short STW phase ensures that GC threads have finished the evacuation process.
- Concurrent Update References: This phase updates the references to those objects that were moved during the previous concurrent phase.

- STW Final Update References: It completes the previous updating phase and updates the root set.
- Concurrent Cleanup: It reclaims regions in the collection set as their objects have been moved to other regions.

In order to enable Shenandoah GC, users can include the JVM command-line option `-XX:+UseShenandoahGC`.

Also, a generational version of Shenandoah has been developed in collaboration between Amazon Corretto[7] and Red Hat teams. Compared to the single-generation Shenandoah, the generational version still maintains pause times below 10 milliseconds while using less heap. Also, it delivers a higher allocation rate for short-lived objects, in addition to decreasing the probability of incurring STW pauses during allocation spikes. Evaluation of generational Shenandoah showed that it has less than 5% reduction in overall application throughput compared to single generation Shenandoah [Coo+21].

2.5 Summary

This chapter provides an overview of important performance metrics related to the GC, core garbage collection concepts, methods, strategies, and existing GCs in OpenJDK. It emphasizes the intricate performance trade-offs in GC design and the core garbage collection algorithms. It highlights the significance of performance metrics such as application throughput, GC pause times, memory usage, CPU utilization, and application latency when discussing GCs. We also explore commonly employed reference tracing algorithms and heap layouts in GCs, while delving into various GC types, including stop-the-world (STW) and concurrent approaches.

Moreover, an overview of major production GCs in OpenJDK is also provided. The Serial collector uses a single thread with stop-the-world pauses. Parallel GC uses multiple threads to reduce pause times at the cost of more resource usage. CMS and G1 collectors perform parts of garbage collection concurrently to lower pauses. ZGC and Shenandoah target low pause times regardless of heap size by doing more tasks concurrently.

[7] https://docs.aws.amazon.com/corretto/

Chapter 3

Related Work

The importance of GCs in programming languages such as Java led to extensive studies on GC performance. Many studies compared existing GC algorithms, evaluated their performance, combined the features of existing GCs, or introduced new ones. In this section, we go over some of the studies tightly related to our work. Papers I, II, and III, provide more extensive details and insights regarding the related work.

3.1 Novel GC Strategies

Several studies have introduced novel GCs built upon existing collection algorithms. In the work by Ossia et al. [Oss+02], a parallel, incremental (which performs the collection in steps), and a predominantly concurrent GC was developed to achieve the objective of minimal GC pause times. This collector is specifically designed for shared-memory and multiprocessor servers, suitable for highly multi-threaded applications.

To address fragmentation concerns in multi-threaded applications, Pizlo et al.[Piz+07] proposed STOPLESS, a tracing collector supplemented with a compactor. Additionally, Pizlo et al.[PPS08] proposed CLOVER and CHICKEN as two solutions for concurrent real-time GCs, aiming to simplify the complexities associated with STOPLESS.

Frampton et al. [Fra+07] designed an incremental, young-generation STW GC for real-time systems. The collector uses both read and write barriers to track remembered set changes.

Tene et al. [TIW11] proposed C4, the Continuously Concurrent Compacting Collector. It supports concurrent compaction, and incremental update tracing through the use of a read barrier. C4 enables simultaneous-generational concurrency that allows different generations to be collected concurrently. It continuously performs concurrent young generation collections, even during long periods of concurrent full heap collection, maintaining high allocation rates and efficiency without sacrificing response times.

Wu et al. [Wu+20] introduces Platinum, a novel concurrent GC designed to reduce latency in interactive services. Platinum creates an isolated execution environment for concurrent mutators, improving application latency without restricting GC thread execution. Additionally, Platinum utilizes a new hardware feature to make access control to different regions of memory more efficient and therefore minimize software overhead present in previous concurrent collectors.

Bruno et al. [BF18] described how objects created by Big Data applications are kept in memory and surveyed the existing GCs for big data environments.

They also addressed scalability issues in classic garbage collection algorithms and analyzed several relevant systems that try to solve these scalability issues.

Xu et al. [Xu+19] used the Apache Spark [Fou18] application to analyze GCs like G1 and Parallel. They proposed strategies for designing GCs specified for Big Data.

Nguyen et al. [Ngu+16] also proposed a GC with low GC pause time and application high throughput based on the logical distinction between the data path and the control path. The data path consists of data manipulation functions, while the control path is responsible for cluster management, setting communication channels between nodes, and interacting with users. They argue these paths differ in heap usage and object creation patterns. So, based on the previously mentioned paths, they divide the heap into data and control spaces to reduce the objects managed by the GC. Not many objects are created in the control spaces, and they are subject to generation-based collection; the objects in the data space, which creates the most objects, are subject to region-based collections. However, the developer is responsible for marking the beginning and end points of the data path in the program.

Broom [Gog+15] and NG2C [BOF17] also proposed two GCs for Big Data systems. In Broom, the programmer has to create the regions in the heap explicitly; in NG2C, the programmer identifies the generation in which a new object should be allocated. These GCs require developer efforts and are prone to errors. Then, the authors of NG2C proposed POLM2 [BF17], an offline profiler that automatically infers generations for object allocation, and finally, ROLP [Bru+19], an online profiler to select an object generation with no user intervention.

However, the GCs mentioned above are not available in OpenJDK HotSpot, the most widely used JVM implementation, and therefore they have not been included in our analysis.

3.2 GC Comparative Analysis

Zhao et al. [ZBM22] introduced LXR to deliver low GC pause times and high application throughput. Their approach includes regular and brief STW collections to optimize responsiveness and an RC mechanism to deliver scalability and promptness. They evaluated and compared barrier overhead and GC pause time for GCs like G1, Shenandoah, and ZGC that we also used in this thesis; yet, note that RC-based GCs are not widely used in production and, therefore, they are out of the scope of this document.

Zhao et al. [ZB20] decomposed G1 into several key components to evaluate the impact of different algorithmic elements of G1 on performance. They first built a simple collector and incrementally added key elements of G1 and built 6 collectors to evaluate the G1 element's impact. Their evaluations revealed that G1's concurrent marking and generational collection reduce the 95^{th}percentile GC pauses by 64% and 93% respectively. Also, they also independently measured

the barriers used by G1 and found that they have an overhead of around 12% with respect to total performance.

Pufek et al. [PGM19] analyzed several garbage collectors like G1, Parallel, Serial, and CMS [PD00] (a generational GC with concurrent tracing in the old generation). The study provides a solid benchmark-based methodology for empirically comparing GC algorithms. It demonstrates the need for continuous evolution of GCs to improve performance as Java applications grow more complex. They used the DaCapo benchmark suite [Bla+06] to evaluate the GCs. They compare the number of algorithm iterations and the total GC pause time for applications running on top of JDK8 and JDK11. Also, they compared ZGC and Shenandoah, which were experimental GCs by then, with G1. Key findings showed that G1 greatly improved from Java 8 to Java 11, significantly reducing collection times and outperforming Parallel GC in several DaCapo benchmarks. This indicates the value of new enhancements to the G1 collector. Overall, they concluded that G1 and Parallel operated better than Serial and CMS regarding the overall duration of collections. They also showed that those experimental GCs (ZGC and Shenandoah) would contribute to overall system optimizations. Their results also emphasize the importance of evaluating ZGC and Shenandoah, as we do in this document.

Grgic et al. [GMR18] analyzed Serial, G1, Parallel, and CMS using the DaCapo benchmark suite in JDK version 9. They concluded that the G1 showed significant improvements compared to Parallel GC and CMS, especially for multi-threaded benchmarks. G1 required less collection time and fewer GC cycles than Parallel and CMS. However, for single-threaded environments, G1 did not show obvious advantages compared to other collectors. The authors suggest optimizations and tuning may be needed based on application characteristics to fully utilize the capabilities of G1. The results highlight the importance of GC selection and configuration based on factors like application behavior.

Lengauer et al. [Len+17] described commonly used benchmarks, including DaCapo, DaCapo Scala [Sew+11], and SPECjvm2008 [Sta08] in terms of memory and garbage collection behavior. They compared G1 and Parallel Old (parallel collection in the old generation) regarding the number of full collections (GC counts), the GC time relative to the total execution time of the application, and GC pause times. They concluded that G1 performs better than the Parallel Old concerning GC pause times by selecting different regions to collect.

Beronić et al. [Ber+22] investigated and compared memory issues, heap allocation, CPU usage, and duration of collection in three GCs: G1, ZGC, and Shenandoah. They found that GCs are sensitive to the heap size and that G1 uses less heap than the two other GCs. However, G1 is more CPU intensive since it occupies more OS threads necessary for scanning the live objects in a heap.

Cai et al. [Cai+22] introduced a new methodology that defines a practical lower bound on the costs of GCs for any given cost metric. They showed that GCs are sensitive to heap size and indicated that low GC pause times achieved by concurrent GCs do not translate into low application latency. To achieve this conclusion, the authors used latency-sensitive workloads from the DaCapo Chopin

benchmark suite. These latency-sensitive workloads serve requests arriving at a determined rate, and if they are not able to process a request instantly, they enqueue the request. Therefore, they used a metric, called metered latency, to show the delay both for the executing and the enqueued requests. Using this metric (metered latency), the authors show that the GCs, especially concurrent ones, cause delays for both executing and enqueued requests and therefore affect the latency.

While drawing insights from prior research, our approach stands out by comparing a collection of GCs that have not been directly compared together in previous studies. In addition, our main focus was on identifying the most suitable GC for specific categories of applications. Furthermore, we aimed to leverage these comparisons and GC evaluations, while also taking into account user preferences in the development of a software system (called BestGC). As it is described in Paper II, this tool automates the GC selection process which aligns with user preferences and the application's objectives.

3.3 Heap Sizing Algorithms for GCs

A number of studies have been conducted for STW collectors, aiming to improve execution time [Bre+01], avoid paging [Grz+07] or both [Yan+04]. Brecht et al. [Bre+01] proposed an adaptive technique for increasing the heap size with the goal of reducing execution time in a STW Boehm-Demers-Weiser GC [BW88]. The authors suggest increasing the heap size aggressively without collecting garbage if sufficient memory is available. Only when memory is scarce, garbage collection becomes more frequent, and the heap size stabilizes. This approach prioritizes reducing the GC calls in order to improve the throughput of the application.

White et al. [Whi+13] proposed an approach using control theory to minimize GC overhead (defined as the proportion of time spent in garbage collection rather than application execution) and make efficient use of memory by adaptively resizing the heap. They implemented a Proportional-Integral-Derivative (PID) controller that monitors GC overhead and adjusts the heap resize ratio to maintain a target GC overhead level set by the user. The GC overhead is the proportion of execution time spent doing garbage collection rather than application work. They utilized the Jikes Research Virtual Machine (RVM) along with the Memory Management Toolkit (MMTk) as the experimental platform and used the MarkSweep [1] collector within MMTK.

Yang et al. [Yan+04] introduced an analytical model to adjust the heap size in a multi-program environment. In the proposed approach, an operating system's virtual memory manager monitors an application's memory allocation and footprint. Then, it periodically changes the heap size to closely match the real amount of memory used by the application. The proposed technique uses a model that minimizes GC overhead (by giving it enough heap size) but also

[1] https://docs.mmtk.io/tutorial/mygc/ss/prefix.html

minimizes paging (by avoiding large heaps). The model is offered for semi-space[2] and Appel collectors [App89].

Zhang et al. [Zha+06] proposed a novel approach to memory management called Program-level Adaptive Memory Management (PAMM). PAMM uses the program's repetitive patterns (phases) information to manage memory adaptively. The authors highlight that the behavior of the phase instances is quite similar and repetitive, so they can represent the memory usage cycle in the application. PAMM monitors the program's current heap usage and the number of page faults to adjust a softbound as a GC threshold. When the threshold is reached, PAMM triggers the GC to collect and free unused memory. They evaluate PAMM with three STW and generational collectors within Jikes Research Virtual Machine (JikesRVM[3])(Mark-Sweep[4], CopyMS[5], and GenCopy[6] [CT21]). PAMM relies on a specific phase detection algorithm, which may not be applicable to all types of applications.

Grzegorczyk et al. [Grz+07] proposed the Isla Vista heap sizing strategy to avoid GC-induced paging. Their strategy is to grow the heap when more physical memory is available and shrink it by triggering the GC when physical memory overflows. Thus, it trades more garbage collection for less paging. This is done by communicating between the OS and Virtual Machine (VM) and triggering the heap sizing logic when allocation stalls occur during GC.

Bruno et al. [Bru+18] proposed a vertical memory scalability approach to scale JVM heap sizes dynamically. To do this, the authors introduce a new parameter: `CurrentMaxMemory`. Contrary to the static memory limit defined at launch time, `CurrentMaxMemory` can be re-defined at run-time. In addition to the new dynamic limit, this work also proposed an automatic trigger to start heap compaction whenever the amount of unused memory is large. This technique allows returning memory to the Operating System as soon as possible.

Immix [BM08] is a GC suitable for high-performance computing. Immix does not require the maximum heap size to be known in advance. It continuously monitors the amount of free memory available in the heap and adjusts memory allocation accordingly. When the amount of free memory falls below a certain threshold (which may vary between implementations), Immix triggers a GC cycle to reclaim unused memory. If the free space is still insufficient after collection, Immix may allocate additional memory blocks to meet the application's memory

[2] A copying collector that divides the heap into two halves (semi-spaces); when one half becomes full, it collects the heap by copying live objects to the other semi-space.

[3] https://www.jikesrvm.org/

[4] Mark-Sweep (MS) GC traverses the entire object graph, marks live objects and considers the unmarked objects as garbage. In the Jikes RVM objects are allocated in specific-sized blocks and managed in a free-list. Objects that are not marked are simply returned to the list [Zha+06].

[5] CopyMS GC utilizes two memory regions: new objects are sequentially allocated in the first region, a copying space. Once full, reachable objects are transferred to the second region, which is managed by Mark-Sweep. [Zha+06].

[6] GenCopy GC is a generational garbage collector. In its design, the young generation consists of two semi-spaces. When one semi-space becomes full, live objects are copied into the other semi-space. The old generation utilizes the Mark-Sweep algorithm to reclaim memory space [Zha+06].

needs. Immix also considers the rate of object allocation as a metric. If the allocation rate exceeds a certain threshold, it indicates a high memory consumption and the potential need for more memory to avoid out-of-memory errors. Moreover, Immix uses heuristics to estimate the size of the working set or the set of objects that are actively being used by the application. Since Immix is a STW collector, this dynamic heap resizing brings many disadvantages. For example, the application may experience brief pauses or slowdowns during the resizing process, which in turn makes it more difficult to reason about the memory usage and performance characteristics of an application.

Cheng et al.[CB01] introduced a parallel, concurrent, real-time GC for multi-processors. The work done by the GC is proportional to the allocation rate; so, it indirectly scales up and down with application CPU utilization. It aims to provide bounds on pause times for the GC while also scaling well across multiple processors. To evaluate their collector, they implemented several variants of their algorithm including a non-incremental collector. Using the concept of Minimum Mutator Utilisation (MMU), they capture the percentage of time in a given time window the mutators have access to the CPU. They showed that their proposed collector keeps higher MMU results compared to the non-incremental GC. However, they do not assess GC CPU or utilize MMU-based actions.

Degenbaev et al. [Deg+16] proposed scheduling the GC during detected idle periods in the application to reduce GC latency. It uses knowledge of idle times from Chrome's scheduler to opportunistically schedule different GC tasks like minor collections and incremental marking. This allows adapting the GC based on real-time application behavior and available idle cycles. While not directly adjusting heap size, scheduling GC during idle periods allows for reducing memory usage and footprint when the application becomes inactive and based on the real-time needs of the application.

The G1 [Det+04] (Garbage First) garbage collector requires knowledge of the maximum memory needed for an application in advance. If it is not explicitly provided, it uses a default value. G1 uses a dynamic heap size adjustment strategy to adjust the memory usage during runtime based on the current usage pattern of the application [Ora21]. G1 divides the heap into regions of equal size and groups them into two generations: young and old. When the young generation fills up, G1 performs a young collection, during which live objects are copied to a new region while unused regions are reclaimed. G1 also performs periodic concurrent marking of live objects in the old generation. When the old generation fills up, G1 performs a mixed collection, which collects both young and old regions that have been marked as garbage. During a mixed collection, G1 dynamically sizes the heap by using the occupancy of the old generation as a target and adjusts the heap size to meet that target.

The subject of adjusting heap size has been studied in existing works, however, they did not propose an approach that fits well concurrent GCs, especially for ones that are used in the well-known and mostly used OpenJDK Hotspot.

3.4 Discussion

Differently from previous works, this Ph.D thesis:

- studies and evaluates four widely used GCs, G1, Parallel, Shenandoah, and ZGC, using a broad collection of workloads; these workloads represent real-world applications to show GCs' overhead on the different performance metrics (such as application throughput, GC pause time, memory usage).
- implements a system (BestGC) to help the users overcome the complications of selecting a GC for their application (more details in Section 4.2); to the best of our knowledge, there is no such software tool that suggests the most proper GC solution for a user's application, as a defined goal, in this Ph.D. project.
- introduces an approach for helping users with limited/no expertise in GCs. So, by utilizing this approach, users can use a more understandable and better-known metric, CPU, to gain control over memory usage; this is done, without the user going through the complicated process of determining a good maximum heap size for their applications.

3.5 Summary

This chapter explores relevant literature on GCs, including novel GC solutions, comparative analysis of GCs, and heap-sizing algorithms for GCs. The goal is to introduce and discuss existing works that are relevant to our research that helped us to identify research gaps, and position our own research within the broader context.

Chapter 4

Architecture

In this chapter, we present the architecture of our contributions that form the backbone of this thesis. As already mentioned, the goal is to empower developers and users to overcome heap-related challenges including selecting a GC and the heap size for their applications. The overall architecture we follow in this thesis begins with a deep dive into the evaluation of GC algorithms to unravel the performance of the GCs across diverse application types. It continues with simplifying the often challenging task of GC and heap size selection through a system, called BestGC, by considering user preferences regarding two crucial performance metrics, application throughput and GC pause time. Following our goal of automating heap size selection, we pick one of the concurrent GCs that showed promising results in our evaluations and then introduce an approach to heap size adjustment, offering dynamic control over memory utilization through a GC CPU limit. Together, these approaches provide smoother and more productive ways for developers and users in the GC and heap size selection.

In the following, we aim to provide an overview of each paper's architecture and the methodology utilized to support it. For more details, the readers should refer to Paper I, Paper II, and Paper III.

4.1 Selecting a GC for Java Applications

The motivation behind this research stems from the critical role of automatic memory management, facilitated by GCs, in object-oriented programming languages like Java. With the growing prevalence of high-volume data handling in applications, such as Big Data and Cloud Services, efficient memory management becomes crucial. However, the selection of the most suitable GC solution for specific applications remains a challenge for users who are not experts in GCs, particularly when dealing with significant data volumes and resource demands.

In this paper, firstly, we demonstrate the performance differences in various GC solutions, in terms of performance metrics, more specifically application throughput, GC pause time, and memory usage. Secondly, we introduce BufferBench (B2), a benchmark we developed specifically to examine GC behavior and the associated costs related to read and write barriers employed by GCs. Thirdly, we propose a methodology that guides users in the GC selection process, taking into account application characteristics (CPU or I/O intensiveness) and prioritizing specific performance metrics such as application throughput, GC pause time, and memory usage. By offering a systematic approach, we aim to empower users without GC expertise to make informed decisions.

Methodology We utilize the well-known and widely used benchmark suites: DaCapo [Bla+06] and Renaissance [Pro+19]. Additionally, as mentioned above, we developed a custom workload, B2, designed to assess the read and write barriers of the GCs. B2 puts GCs under the pressure of handling a large number of requests. Furthermore, we also utilize the PetClinic application [Spr07] to investigate the effectiveness of our approach with another real-world application that uses an in-memory database, therefore, is a good choice to evaluate GCs.

To determine the category of the applications (CPU-intensive and I/O-intensive), we compare the average CPU usage percentage with the average I/O usage percentage. Applications with a higher percentage of average CPU usage are classified as CPU-intensive, while those with higher I/O usage are considered I/O-intensive.

Subsequently, we utilize the above-mentioned benchmarks to evaluate four GCs available in OpenJDK 13: *CMS*, *G1*, *Shenandoah*, and *ZGC*, in terms of application throughput, GC pause time, and memory usage. The performance metrics are collected during the execution of the applications. Because of the challenge of defining a universally applicable unit of operation across all benchmarks and considering that all benchmarks report execution times, we calculate throughput by averaging the execution times of the applications. GC pause times are extracted from the JVM log files, allowing us to calculate the 90^{th} percentile. Additionally, we measure the average heap memory usage before and after GC to evaluate the extent of memory reduction achieved by each collection. These metrics are thoroughly analyzed to assess the performance of each GC solution. Finally, based on the evaluation results, we determine the most effective GC for each category of applications.

4.2 BestGC: An Automatic GC Selector

In the first paper, we evaluated and showed how GCs impact performance metrics. Furthermore, we provide some high-level recommendations regarding choosing a GC for different categories of applications. However, the main goal of this paper is to propose a system, that automatically suggests the most suitable GC for users' applications guided by their performance preferences (regarding application throughput and GC pause time). The system aims to facilitate GC selection by allowing users to define weights for these performance metrics based on how they care about these metrics according to their applications' goals. Additionally, the work evaluates four widely used GCs in OpenJDK15 (a newer version of OpenJDK compared to the previous research): G1, Parallel, Shenandoah, and ZGC, utilizing Renaissance and DaCapo (Chopin version) benchmark suites. The evaluation considers application execution time (as a metric to report application throughput), GC pause time, and diverse heap sizes, achieved through experiments using different workloads that represent real-world applications.

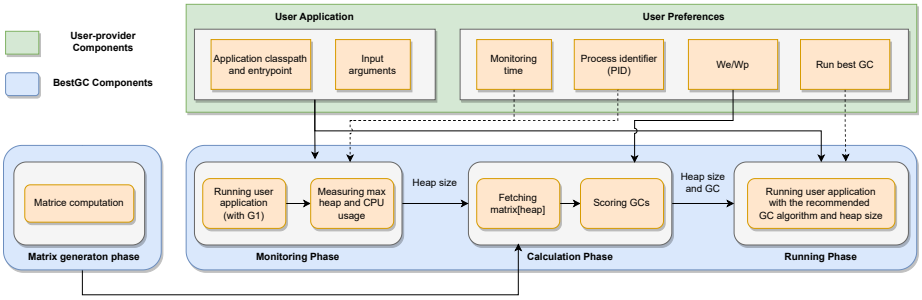


Figure 4.1: Architecture of BestGC: the phases that BestGC goes through to suggest the most suitable GC for the user's application, based on the user's inputs and preferences.

Methodology The methodology used in this Paper II is divided into two parts. First, we evaluate four GCs in OpenJDK15 (G1, Parallel, Shenandoah, and ZGC) in terms of performance metrics such as application throughput (execution time) and GC pause time. We utilize workloads from well-accepted benchmark suites (DaCapo and Renaissance). To assess GCs based on memory usage, we assign them the same fixed heap sizes, eliminating their dependency on the available heap memory of the machine. This approach prevents GCs from freely selecting heap sizes according to their individual policies and ensures a fair and consistent experimental setup for all GCs. In this way, we incorporate the impact of heap usage on the evaluations. The evaluation results are employed within BestGC to score and select an appropriate GC. The results are stored in the form of matrices, with one matrix dedicated to each evaluated heap size.

Second, we design and develop BestGC to automate the GC selection process for users' applications. An overview of the internal workings of BestGC is presented in Figure 4.1.

As shown in Figure 4.1, there are four phases in BestGC: Matrix Generation, Monitoring, Calculation, and Running. In the matrix generation phase, matrices are created based on the extensive measurement of the four above-mentioned GCs for each heap size. These matrices will be used in the calculation phase. The monitoring phase consists of running BestGC (as long as it is set using the monitoring time defined as a constant in BestGC or any time set by the user) to find an application's maximum heap size and CPU usage. In the calculation phase, GCs are scored based on the matrices generated in the first phase. Finally, in the running phase, the user's application is executed with the suggested GC and heap size.

Finally, the validation of BestGC using workloads from SPECjvm2008, another widely used benchmark suite, further confirms the effectiveness of the proposed system.

4.3 Heap Size Adjustment with CPU Control

As a result of the previously mentioned work, we observed that efficient management of memory resources is vital for software applications to achieve optimal performance. In fact, the GC relieves users from explicit memory management but leaves the challenge of selecting the maximum heap size (the maximum amount of memory the application can use while running in the JVM). Currently, JVM selects a maximum heap size for an application on startup (currently, OpenJDK selects 25% of the available RAM) or leaves it to the developer. However, a small heap size may result in frequent GCs and a large one may result in wasting memory resources.

In the previous paper, we introduced BestGC which in addition to suggesting a suitable GC for an application, also suggests a maximum heap size for the user application. In this paper (Paper III), we put a greater emphasis on the heap size. We propose an automated approach to adjust heap size (in concurrent collectors), removing the need for manual heap configuration by setting an upper bound for GC CPU usage. ZGC's highly concurrent design, strong performance based on our earlier analyses in our previous paper (BestGC), and integration with OpenJDK made it an ideal platform for building and validating our automatic heap adjustment approach. To this end, we aimed to provide users with a new "tuning knob" to control the GC frequency and therefore heap usage based on the CPU usage of the GC. In addition to being able to perform effectively regarding application execution time, latency, energy, and memory usage.

This technique has been developed under the direct supervision of the ZGC development team of Oracle Corporation and is going to be integrated into the new generational ZGC.

Methodology The methodology of this paper (Paper III) involves an approach to tuning heap size during application execution time and helping users overcome the complications of setting a proper maximum heap size. Unlike existing related works, the proposed approach allows developers to control memory by setting a GC target, which dictates the percentage of CPU spent on GC compared to GC spent on the application threads. The rationale behind this approach is the observed inverse correlation between memory and GC CPU utilization. A large heap size leads to infrequent garbage collections and low CPU time spent on GC, while a small heap size increases the frequency of garbage collections and corresponding CPU time. This trade-off between memory and CPU usage is particularly relevant for concurrent garbage collectors, as they work concurrently with the running applications. By determining the GC target (in terms of GC CPU usage), the heap size can dynamically be adjusted to match the application's memory usage behavior. The methodology involves iteratively adjusting the heap size until the GC CPU overhead meets the GC target set by the developer. If the actual GC CPU overhead exceeds the target, the heap size is increased to reduce the frequency of garbage collections and lower the CPU overhead. Conversely, if the actual GC CPU overhead is below the target, the heap size is decreased to trigger more collections and increase the GC CPU overhead. Fluctuations

in the heap sizes are mitigated by passing the GC overhead error (difference between current GC utilization and the target) through the Sigmoid [HM95] function, which smoothens the changes. This prevents sudden and huge changes in the heap size. This methodology also honors the maximum heap size (*Xmx*) option, if defined by the developer. In the absence of *Xmx*, the system sets it to a default value close to the maximum memory available on the machine while avoiding system instability (in our implementation in generational ZGC, we set it to 80% of the available RAM); this is due to the fact that the plan is to adjust the heap size based on the application memory usage pattern, regardless of the amount of available memory. This methodology can be applicable in concurrent GCs and in the following chapter, we show our implementation in generational ZGC.

4.4 Summary

In this chapter, we provide a summary of the core architecture and methodology in our proposed user-centric solutions for simplifying the process of selecting the right GC and heap size for user applications. More details can be found in Papers I, II, III. In the first paper, we categorize applications as I/O or CPU intensive and identify the best GC solutions, among G1, CMS, ZGC, and Shenandoah, for each category through extensive GC evaluations. In the second paper, we introduce BestGC, a system that suggests a maximum heap size to run the application and the most suitable GC solution based on user preferences regarding application throughput and GC pause time. Although we evaluate four GCs (G1, Parallel, ZGC, Shenandoah) in our study, the proposed architecture can be extended to work with other GCs and runtime environments. Lastly, in the third paper, to continue our investigation of dealing with heap size challenges for the users, we propose an approach where users can define the desired GC CPU utilization, allowing the GC to dynamically adjust the heap size based on application memory usage while respecting a user-defined upper bound for GC CPU utilization in concurrent GCs.

The methodology utilized to evaluate GCs and suggest a proper GC for diverse application types, as well as the way we automated this process by BestGC, is adaptable to various JVMs and can be extended to various GCs. Furthermore, the heap adjustment strategy can be implemented into the concurrent collectors that operate based on concepts such as a soft maximum heap size.[1].

[1]This is a soft limit on how large the Java heap can grow. ZGC will strive to not grow beyond this limit, but it allows the heap to expand beyond this limit without triggering out-of-memory errors.

Chapter 5

Implementation

This chapter explores the practical aspects of translating the architectural concepts outlined in the previous chapter into implementations. We presented an approach to categorizing applications based on their CPU and I/O characteristics and explored the performance characteristics of various GCs to recommend the most suitable GC for each application category. In this chapter, we explain how we implement these ideas. We delve into the specifics of BestGC, the tool that automates the selection of optimal heap size configurations and GC for applications. We elaborate on how BestGC measures key performance metrics, such as heap and CPU utilization, while also accommodating users' preferences in terms of application throughput and GC pause times. The aim is to ensure that the selected heap size and the GC configuration align with the specific needs of each application. Furthermore, we provide details on our heap adjustment approach within ZGC, a concurrent collector that demonstrated promising results during our previous evaluations. Specifically, we discuss our approach of using GC CPU utilization as a tuning parameter to dynamically adjust heap sizes with the real-time memory usage pattern of the application.

For a deeper understanding of the details involved in these implementations, we encourage readers to refer to Paper I, Paper II, and Paper III. These papers offer a more extensive exploration and context to the content presented in this chapter.

5.1 Selecting a GC for CPU or I/O-Intensive Applications

As discussed in Section 4.1, in order to recommend an ideal GC for specific application categories (either CPU-intensive or I/O-intensive), we follow a two-step approach. Initially, we categorize applications based on their CPU or I/O usage. Subsequently, we evaluate the performance of four GCs (CMS, G1, ZGC, and Shenandoah) available in OpenJDK version 13 utilizing DaCapo and Renaissance benchmark suites, the PetClinic application, and a self-developed benchmark (BufferBench, B2). All the implementations were done on a server running GNU/Linux, Ubuntu 16.04.4 LTS, running OpenJDK version 13.

5.1.1 Categorizing Applications

To categorize applications based on their CPU and I/O usage, we collected CPU and disk data using the *atop* Linux command at one-second intervals. From the obtained values, we calculate the average disk and CPU usage. Additionally, we monitored the number of running threads for each Java process using the *htop* command at one-second intervals to determine the average number of engaged

CPU cores. Dividing the average CPU utilization by the average CPU cores, we derived the application's average CPU utilization per core. If the average disk utilization exceeds the average CPU utilization, the application is categorized as I/O-intensive; otherwise, it is considered CPU-intensive.

5.1.2 BufferBench (B2)

B2 is a Java application designed to perform extensive read/write operations in memory. Its purpose is to facilitate a comprehensive evaluation of the behavior of various GCs and to analyze the performance implications imposed by the read and write barriers they employ. B2 requires three input values to run: data size, number of operations, and the read percentage. Based on the data size a hash map data structure is created to store the objects. The number of operations represents the total count of both read and write operations. The read percentage specifies the proportion of operations that involve reading objects from the heap, with the remaining percentage allocated to write operations and generating new objects.

5.2 BestGC: An Automatic GC Selector

We implement BestGC using Java and OpenJDK version 15[1] on a machine running GNU/Linux, Ubuntu 20.04.5 LTS. BestGC encompasses four distinct phases (as presented in Figure 4.1): matrix generation, monitoring, calculation, and running. The matrix generation phase utilizes the evaluation results of the GCs in the form of matrices in BestGC. The remaining three phases are executed for each user's application running, enabling BestGC to recommend the most suitable GC solution based on the user's input values (for application throughput and GC pause time). BestGC measures the heap usage of the user's application to determine the relevant matrix and then assigns scores to different GCs. Finally, it suggests the GC with the lowest score as the optimal choice.

5.2.1 Input Options for BestGC

BestGC requires a minimal set of input parameters to initiate its operation. Users, who utilize BestGC, need to provide two essential inputs. Firstly, it necessitates the absolute path to the user application's JAR file, along with all the application-specific input arguments. Secondly, users must define at least one of the mandatory weights (ranging from 0 to 1) that signifies weights for application execution time (w_e) or GC pause time (w_p). The weights reflect the user's preferences for these two critical performance metrics, where a value of 1 signifies the highest importance. It is essential to note that the sum of these weights must equal 1, indicating a consideration of both aspects.

[1]We used a newer version of OpenJDK compared to our previous research. In this version of OpenJDK, ZGC was declared "Production Ready", while CMS was deprecated.

$$\begin{array}{c} \\ G1 \\ Parallel \\ Shenandoah \\ ZGC \end{array} \begin{array}{c} \begin{array}{cc} ExecutionTime & PauseTime \end{array} \\ \begin{bmatrix} 1.0 & 1.0 \\ 0.954 & 1.448 \\ 1.093 & 0.144 \\ 1.098 & 0.044 \end{bmatrix} \end{array}$$

Figure 5.1: Example of a matrix for the heap size of 8192 MB.

BestGC also provides additional optional switches, allowing users to configure various parameters, such as the *monitoring-time*, which allows users to control the monitoring time interval in which BestGC collects crucial data on heap and CPU usage from the user's application.

5.2.2 Heap Size Variety

GCs may use different heuristics to choose heap size based on their requirement. So, to keep the evaluation setup the same for all the GCs, we use the fixed heap sizes of 256, 512, 1024, 2048, 4096, and 8192 MB in all the evaluations to generate the corresponding matrices. The heap sizes in powers of two are commonly used by the users [Eva20]; also, these include the required heap sizes to run all workloads in our evaluations.

5.2.3 Matrices

In the matrix generation phase, there is a collection of matrices that have been created through experimental evaluations performed on G1, Parallel, Shenandoah, and ZGC for each heap configuration (256, 512, 1024, 2048, 4096, and 8192 MB). As Figure 5.1 shows, in each matrix, every row represents a specific GC and contains two columns: the average application execution time (application throughput) and the average 90^{th} percentile of GC pause times. These values are normalized to the corresponding values of G1 (which is OpenJDK's default GC). These matrices will be utilized in subsequent phases.

5.2.4 Measuring Heap and CPU Utilization

During the monitoring phase, BestGC runs a user's Java application with the default GC (G1) of the available default JDK installed on the user's machine. Therefore, both the application's maximum heap memory and CPU usage are measured by BestGC during *monitoring-time*, which by default is set to 30 seconds if the user does not change it.

To capture the maximum heap usage, every second, BestGC invokes the *jstat*[2] command with the *gc* option, which represents the behavior of the garbage-collected heap, for the user's application using its Process ID (PID). BestGC considers 20% extra headroom for the application (i.e., $max\_heap \times 1.2$) and picks the closest bigger heap configuration among 256, 512, 1024, 2048, 4096,

[2]https://docs.oracle.com/en/java/javase/15/docs/specs/man/jstat.html

and 8192 MB. Based on the suggested heap size, the corresponding matrix is also used in the other phases.

For accurate CPU utilization measurements, BestGC must run alone on the user's machine to avoid interference from other applications that may impact CPU measurements during the monitoring phase. BestGC captures the amount of CPU utilization of the user's application by averaging the recorded total CPU usage every second using the *top*[3] command during the *monitoring-time* interval. Moreover, since not all the applications utilize all the CPU cores in the machine, BestGC also calculates the number of engaged CPU cores for the user's application. It achieves this by utilizing the *proc/stat*[4] command to compute the CPU usage for each core at every second during the *monitoring-time*. If the usage exceeds 50% for a core, BestGC increments the count of the engaged CPU cores per second by one. Subsequently, the average number of engaged cores is calculated at the end of the *monitoring-time*. Finally, it divides the average total CPU utilization of the application by the average number of engaged CPU cores. Should the results be over 90%,[5] the application is reported as a CPU-intensive application; otherwise, it is considered non-CPU-intensive.

5.2.5 Scoring GCs

To execute BestGC, the user needs to pass a weight for the execution time (application throughput) (w_e) or GC pause time (w_p), as already pointed. Having w_p (or w_e), and the most appropriate matrix (selected based on the detected maximum used heap size), BestGC calculates a formula (Equation (5.1)) to score each GC during BestGC's calculation phase:

$$\begin{aligned} score_{gc} = & w_e \times matrix[heap]_{<gc,ExecutionTime>} \\ & + w_p \times matrix[heap]_{<gc,PauseTime>} \end{aligned} \tag{5.1}$$

$$gc \in \{G1, Parallel, Shenandoah, ZGC\}$$
$$heap \in \{256, 512, 1024, 2048, 4096, 8192\}$$

5.2.6 Executing the User's Application

Finally, in the running phase, BestGC runs the user's application with the minimum-scored GC (the best one) and suggested heap size using the following command:

```
java -Xmx<max_heap>*1.2m -XX:+Use_<best_gc> -jar user_application
```

However, if the user has previously deactivated the auto-execution feature (*run-best-gc=false*), BestGC will simply print out the command above.

[3] https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html

[4] https://manpages.ubuntu.com/manpages/xenial/man5/proc.5.html

[5] We used the considerations in the Linux command *atop*, which defines the CPU usage to be critical when the total CPU usage percentage is 90% and above; https://manpages.ubuntu.com/manpages/bionic/man1/atop.1.html

5.3 Heap Size Adjustment with CPU Control

We implemented our heap size adjustment strategy by modifying the generational ZGC available in OpenJDK version 21 (which includes the new generational ZGC). The implementation has been done on a machine running Oracle Linux Server 8.4. More details can be seen in Paper III.

We used *soft max heap size* concept available in ZGC. It is a limit on the size of the heap beyond which ZGC tries not to grow. Unlike *Xmx*, which is a JVM option to set a hard limit on the heap size, exceeding the soft max heap size will not result in an OOM error. We set the *Xmx* to 80% of the machine's available RAM, although we honor the *Xmx* value if set by the user. When approaching the soft max heap, ZGC triggers garbage collection to bring the heap size below the soft max heap size. So, we used the soft max heap size to implement adaptive heap sizing by scaling it up and down based on the GC CPU usage.

5.3.1 GC CPU Overhead

We need to scale the soft maximum heap size when the GC CPU usage reaches a user-defined limit. As it is shown in Equation (5.2), we define the GC CPU overhead (henceforth denoted GC_{CPU}) as the ratio of time spent doing GC (T_{GC}) to the time spent in the entire application (T_{APP}).

$$GC_{CPU} = \frac{T_{GC}}{T_{APP}}. \tag{5.2}$$

For simplicity and to avoid adding logic contributing to GC CPU overhead, we use existing telemetries in ZGC. So, we used wall-clock time measurements for GC time (W_{GC}), which is a good proxy for GC CPU time, and CPU time measurements for application time (C_{APP}). Also, to mitigate fluctuations if one GC cycle has high CPU activity and the other low, W_{GC} should be calculated using average times for the last n collections (in our implementation, we pick $n = 3$, as it could effectively mitigate sudden fluctuations in W_{GC}). The W_{GC} is calculated as the sum of time spent on young W_{young} and old collections W_{old}, plus the time spent by mutators in the slow path of barriers (B):[6]

$$W_{GC} = W_{young} + W_{old} + B \tag{5.3}$$

When the mutator enters the slow path, it ensures that the pointer is valid by looking up the new canonical address of the object from a forwarding table. Mutator may involve copying the object elsewhere and writing the new address in the forwarding table. Moreover, the application's average time is the sum of the scheduled time of all threads spawned by the process (i.e., a CPU time measurement) between two collections in the same generation:

$$T_{APP} = C_{GC_i} - C_{GC_{i+1}} \tag{5.4}$$

[6]A slow path is a code path that is executed when specific conditions or less common cases occur, which can result in slower performance due to additional checks, synchronization, or more complex operations.

5.3.2 Adjusting the Heap Size

At the end of each GC cycle, we compare the GC CPU overhead to the user-defined GC target to calculate $overhead\_error_{CPU}$ which we use to adjust the heap size:

$$overhead\_error_{CPU} = GC_{CPU} - \frac{Target\_GC_{CPU}}{100} \tag{5.5}$$

Large error numbers can cause fluctuations in the heap sizes and lead to memory being claimed that is not required in the long run. To mitigate this, we pass the $overhead\_error_{CPU}$ through the Sigmoid function [HM95] to calculate the *Sigmoid overhead error*. It maps input values to a range between 0 and 1, and thus, smoothens changes in heap sizes:

$$S(overhead\_error) = \frac{1}{1 + e^{-overhead\_error_{CPU}}}. \tag{5.6}$$

Then, we use this result to calculate an *adjustment factor* that limits the changes of the heap size within a range of 0.5 to 1.5:

$$adjustment\_factor = S(overhead\_error) + 0.5 \tag{5.7}$$

An $S(overhead\_error) < 0.5$ means that the actual GC_{CPU} has exceeded the $Target\_GC_{CPU}$, so the $adjustment\_factor$ would be less than 1 and will reduce the heap size, leading to more GC cycles. When the actual GC_{CPU} is below $Target\_GC_{CPU}$, $S(overhead\_error) > 0.5$, i.e., $adjustment\_factor > 1$ will increase the heap size. The heap size will never change more than 50% of the current size (in any direction). Finally, we compute the new heap size as follows:

$$new\_size = current\_size \times adjustment\_factor \tag{5.8}$$

5.4 Summary

In this chapter, we provide an overview of the practical aspects of our research papers. We implemented the core ideas of our proposed architecture and approach described in Chapter 4. We begin by explaining the way we categorize applications into CPU-intensive and I/O-intensive categories based on their resource usage. Notably, we introduce the BufferBench (B2) benchmark, a tool developed for comprehensive GC behavior analysis. Next, we present the implementation of "BestGC", which is a solution for selecting the most suitable GC and heap size for a given application. BestGC utilizes matrices generated from experimental evaluations of multiple GCs in OpenJDK15. It then measures the user application heap and CPU utilization, scores GCs, and recommends a heap size and the optimal GC based on user-defined priorities. Finally, we detail our strategy for adjusting heap sizes in response to GC CPU usage limits, primarily implemented in the generational ZGC of OpenJDK version 21. This adaptive approach, driven by a user-defined GC CPU utilization target, helps optimize memory management and aligns with an application's real-time memory usage pattern. For in-depth technical information, please refer to the research papers I, II, and III.

Chapter 6

Evaluation

In this chapter, we provide an overview of the evaluation details, mostly the benchmarks utilized in the research papers and the obtained results. The benchmark selection and performance metric choices were made to provide a comprehensive evaluation of our proposed approaches. These benchmarks resemble real-world applications and the performance metrics we evaluate are crucial in GCs' performance realm. We aimed to replicate real-world conditions, challenge GCs, and assess their performance across various dimensions, ultimately contributing valuable insights to the field of JVM optimization. These benchmarks serve the purpose of identifying the most suitable GC solution for distinct application categories, as well as evaluating the effectiveness of GCs for integration into the BestGC software. Additionally, we discuss benchmarks employed to introduce a JVM option that dynamically adjusts heap usage based on the CPU utilization of the GC. Then, we go over the performance metrics we measured in our research. Finally, we present the results derived from our contributions by evaluating our proposed approaches in the research papers. More evaluation details can be found in Paper I, Paper II, and Paper III.

6.1 Benchmarks

DaCapo Benchmark Suite The DaCapo benchmark suite [Bla+06] is a widely used set of various real-world applications, designed to evaluate the performance of GCs, including web services, XML processing, interpreters, databases, etc. The regular branch of the DaCapo benchmark suite comprises popular programs that provide insights into the performance characteristics of GCs under real-world workloads. These workloads cover a diverse range of application categories including I/O-intensive and CPU-intensive, therefore they enabled us to evaluate GCs' behavior and performance for each category.

In addition to the regular branch, this benchmark suite includes the Chopin branch.[1] DaCapo Chopin includes latency-sensitive workloads that provide insights into GCs' ability regarding responsiveness. These latency-sensitive workloads handle incoming requests arriving at a determined rate, and if they are not able to process a request instantly, they enqueue the request. These workloads measure latency in two ways: simple latency and metered latency. Simple latency disregards queuing effects. On the other hand, metered latency considers delay both for the executing and the enqueued requests [Cai+22]. It makes the metered latency a good choice to compare the performance of GCs.

We utilized the regular DaCapo benchmark suite in Paper I, and the Chopin version in Paper II and Paper II.

[1] https://github.com/dacapobench/dacapobench/tree/dev-chopin

There are several input options available for the workloads in DaCapo including the input size and the number of iterations. During our evaluation, we prioritized using a *large* input size whenever it was available for the workloads to put more load on the application and stress the GC more. Otherwise, we used the *default* input size. Also, using the switch `-no-pre-iteration-gc`, we disabled explicit GC calls in the workload's code. DaCapo provides execution time measurements for its applications, like other benchmarks utilized in our research. Therefore, we use the same metric to report average application throughput.

Renaissance Benchmark Suite The Renaissance benchmark suite [Pro+19] is a collection of applications designed specifically to evaluate the performance of GCs. The benchmarks in Renaissance cover a wide range of application domains, including web services, databases, and more. The applications are representative of real-world applications and are designed to stress different aspects of GCs.

As for the DaCapo, the number of iterations specifies how many times the workloads are executed to gather performance data. Therefore, to execute workloads, we set the number of iterations to a value that gave us stable execution time results from then on. Also, we set the input size to *large*, if available, otherwise we set it to the *default*. Disabling explicit GC calls (*System.gc()*) existing in the source code was done using the switch `-no-forced-gc` while running the workloads.

We utilized both the DaCapo and Renaissance benchmark suites for precise comparisons of different GCs and to draw conclusions. These benchmarks are widely used and reliable to evaluate GCs' performance. The combined use of DaCapo and Renaissance benchmarks ensured a comprehensive and thorough evaluation of the GCs, strengthening the validity and reliability of our research findings. Additionally, to provide a broader evaluation of our proposed solutions, we utilized other benchmark suites, which are discussed next.

Buffer Bench (B2) We developed BufferBench (B2), a software designed to perform read and write operations in memory. B2 is a Java-based tool explicitly created to assess the impact of read and write barriers on GCs' performance metrics. BufferBench relies on three input parameters: data size, the number of operations, and read percentage. Based on the data size, a hash map is created to keep the objects. For our study, we chose data sizes of 1 or 2 million objects, providing the basis for our evaluations. The number of operations represents the total count of both read and write operations, set at 100 million for our experiments. These quantities, both for the number of objects and operations, were chosen as they effectively trigger multiple GC cycles to measure GC performance metrics accurately. The read percentage parameter indicates the proportion of operations involving reading objects from the heap, with the remaining percentage devoted to write operations generating new objects in the heap. For our study, we explored read percentages of 25%, 50%, 75%, and 100%,

allowing us to analyze GCs' behavior concerning their performance metrics by varying the read and write operation proportions.

Petclinic While comparing different GCs for different categories of applications (CPU-intensive and I/O-intensive), to provide real-world scenarios for different application categories, we utilized the PetClininc application [Spr07]. It is a web-based application that typically models a simple veterinary clinic management system, where you can perform tasks like adding and managing veterinarians, owners, and their pets. To conduct these evaluations, we employed Apache JMeter[2] (version 5.4). Using JMeter, we simulated the usage of the PetClinic application with 100 threads. Input data files, containing records for various entities within the application, were used for testing. Additionally, we configured each thread group to run 10 iterations. These parameters have been fine-tuned through extensive testing to represent typical usage of the application, minimize errors arising from simultaneous thread execution, induce multiple GC cycles, and effectively utilize server resources.

SPECjvm2008 The SPECjvm 2008 benchmark [Sta08] emulates a range of typical real-world general-purpose applications. This design aims to ensure the benchmark's relevance in assessing Java performance across diverse client and server systems. We use the `Lagom` switch (available in SPECjvm2008); Lagom provides a fixed-size workload for the benchmarks, i.e., it does a fixed number of operations in each benchmark (just like DaCapo and Renaissance benchmark suites). Then, we use the corresponding application execution time as a metric for throughput. We utilized workloads in SPECjvm2008, as any application a user may run, for performance check of our GC selector system, BestGC. This validation process proved that using BestGC shows performance benefits in a highly optimized environment like OpenJDK.

Hazelcast In order to obtain a more comprehensive understanding of our prototype while proposing a new JVM option based on the GC CPU usage, we also include the Hazelcast benchmark [Gen+21]. Hazelcast was chosen since most of the latency-sensitive workloads in DaCapo Chopin were excluded for different reasons (as it is noted by the DaCapo's maintainers[3] at the time of writing Paper III). As low latency is the main goal of a concurrent collector, we wanted to study more such workloads. Hazelcast is designed to provide distributed and scalable in-memory data storage and processing, which can help reduce data access and processing latency. We used all the suggested configuration parameters [Top20] and reported the results of 3 different configurations of Hazelcast (fixed workload with key-set sizes: 400 000, 250 000, 100 000). Using three configurations offers a good enough representation of various workloads and is practical for evaluating GCs under different conditions and data sizes.

[2] https://jmeter.apache.org/

[3] https://github.com/dacapobench/dacapobench/blob/dev-chopin/benchmarks/status.md

We executed the benchmarks until a steady state was reached in execution times. Then, we utilized the recorded data to report the performance metrics detailed in the next section.

6.2 Performance Metrics

To evaluate our proposed solutions, we measured several performance metrics crucial in evaluating GCs: application throughput, GC pause time, memory usage, and latency (delay in responsiveness).

Although our benchmarking methodology varied across our research papers, a common aspect was the execution of benchmarks until a steady state was reached in execution times. Reaching this stable state allowed us to extract and report desired performance metrics to draw conclusions.

All the benchmarks used in the evaluations provide execution time data, which we utilized to report the average application throughput. The reported application throughput is calculated as the average execution time of running workloads over multiple iterations. The number of iterations set in the evaluations is determined by our observation that it provides stable results for execution times. This chosen number of iterations for the workloads ensures more accurate results through repeated executions of the same workload. During the execution of each workload with each of the studied GCs at a time, we enabled the JVM option `-Xlog:gc*` to record GC dump data. Subsequently, from the recorded GC pause times, we calculated and reported the 90^{th} percentile (as it is used in most SLAs) of pause time to assess GC pause times. In addition, we extracted heap-related data, from the recorded GC logs, to determine the average heap usage before/after garbage collection. Furthermore, for latency-sensitive workloads, we employed metered latency values and reported the 99^{th} percentile, which is a common practice in latency analysis.

6.3 Results

We conducted evaluations of the proposed approaches to achieve the objectives outlined in this thesis using the aforementioned benchmarks, which are representative of real-world applications. Next, we present a summary of the results presented in our research papers.

6.3.1 Selecting a GC for Java Applications

We categorized applications from the DaCapo (version 9.12) and Renaissance (version 0.11.0) benchmark suites, as well as the B2 benchmark and the PetClinic application, into CPU-intensive and I/O-intensive categories. All the evaluations have been done on a machine running GNU/Linux, Ubuntu 16.04.4 LTS, with an x86_64 Intel(R) Xeon(R) 4-core CPU E5506 @2.13GHz, and 16GB RAM. We evaluated four GCs in OpenJDK13: CMS, G1, ZGC, and Shenandoah. For all the evaluations, we set the maximum heap to 4 gigabytes, using *–Xmx4g* and

used the *-Xlog:gc** to enable the printing of each GC's activities. The evaluation focused on application throughput, GC pause times, and memory usage before and after GC. The following results were obtained:

- In approximately 57% of CPU-intensive applications, CMS performed better than other GCs in terms of application throughput; ZGC, G1, and Shenandoah demonstrated acceptable throughput in around 20%, 13%, and 10% of these applications, respectively; for I/O-intensive applications, using CMS delivered the best throughput performance in almost 86% of cases, while the remaining 14% got better results with ZGC.

- In roughly 97% of CPU-intensive and all I/O-intensive applications, ZGC achieved the shortest 90^{th} percentile of pause times.

- In terms of average memory utilization before garbage collection, among the CPU-intensive applications, CMS emerged as the top GC choice in 80% of the cases; also, in the I/O-intensive category, CMS and Shenandoah were equally selected as the leading GCs.

- Concerning heap usage reduction after garbage collection, findings highlight G1 as the optimal solution for average memory reduction in approximately 57% of CPU-intensive applications; in the I/O-intensive category, both G1 and CMS were effective in reducing heap usage in about 43% of benchmarks; these results align with one of the main design principles of G1, which focuses on reclaiming heap spaces with the most garbage.

In summary, the evaluation of the mentioned GCs provided valuable insights for users in the process of selecting the most suitable GC for their applications. Using CMS consistently delivered the best application throughput, while ZGC excelled in minimizing GC pause times, and these findings held true for both CPU-intensive and I/O-intensive application categories. When it came to average heap usage before garbage collection, CMS emerged as the preferred GC for both application categories. Shenandoah also demonstrated as good results as CMS, particularly for I/O-intensive applications. As expected, G1 proved to be an excellent choice for reducing heap usage after GC, showcasing its effectiveness in this regard. Additionally, CMS was found to be a possible option for I/O-intensive applications when considering heap reduction. More details on the evaluations are available in Paper I, Section I.5.

6.3.2 BestGC: An Automatic GC Selector

The results are obtained from two phases of evaluation. Firstly, we evaluated the performance of four commonly used GCs within OpenJDK15, Parallel, G1, ZGC, and Shenandoah. As previously mentioned, this evaluation was conducted using the DaCapo Chopin and Renaissance (version 0.11.0) benchmark suites. Secondly, we conducted an evaluation of BestGC's performance, while pretending to be a user, using the SPECjvm2008 benchmark suite. All the evaluations have

been done on a machine running GNU/Linux, Ubuntu 16.04.4 LTS, with an x86_64 Intel(R) Xeon(R) 4-core CPU E5506 @2.13GHz, and 16GB RAM.

The results obtained from the evaluation of the mentioned GCs regarding the 90^{th} of the GC pause times and the application execution time (throughput) across various heap sizes (256, 512, 1024, 2048, 4096, and 8192 MB), reveal the following findings:

- Parallel GC by maintaining nearly constant application execution times consistently outperforms other GCs across various heap sizes because it adjusts the sizes of generations to optimize throughput. Following Parallel, G1 ranks second in terms of application execution time. In contrast, Shenandoah and ZGC exhibit worse results and show different behaviors as heap sizes change. Even with an 8192 MB heap size, Shenandoah and ZGC, both exhibit longer application execution times compared to G1 and Parallel. Notably, Shenandoah and ZGC demonstrate similar execution times at an 8192 MB heap size, but their performance deteriorates as heap sizes decrease. ZGC performs the poorest with a 256 MB heap size compared to other GCs. The difference in performance between ZGC and Shenandoah, both of which are concurrent GCs, is attributed to ZGC's prioritization of concurrency, which requires more heap space for object allocations during garbage collection [CK22]. This, in turn, has a negative effect on application execution times. Whereas, Parallel and G1 are both generational collectors, focusing on the frequent garbage collection that occurs in the young generation due to the generational hypothesis [LH83]. Consequently, these two GCs (Parallel and G1) offer better application execution times (throughput) compared to ZGC and Shenandoah.

- Unlike the application execution time results, Parallel performs poorly regarding the GC pause time. Parallel tries to change one generation size at a time (the generation with the more significant GC pause time [Ora20]). However, for heap sizes exceeding 2048 MB, Parallel's generation sizes become large enough to avoid frequent collections, resulting in shorter GC pause times. Concurrent GCs, particularly ZGC, exhibit significantly shorter GC pause times compared to generational G1 and Parallel GCs. This improvement is attributed to the use of concurrent tracing and copying mechanisms in ZGC and Shenandoah. While Shenandoah consistently keeps GC pause times very small, it does not exhibit a predictable pattern with different heap sizes. In contrast, ZGC maintains a nearly constant average GC pause time across all heap sizes.

In the second step, we assess BestGC. The goal is to validate BestGC using workloads from the SPECjvm2008 benchmark suite and evaluate its effectiveness in recommending the most suitable GC for various applications. The validation process involves three key phases: 1) empirically determining the most suitable GC and heap size by manually measuring application execution time and GC pause time for SPECjvm2008 workloads, 2) employing BestGC to obtain heap and GC suggestions for the SPECjvm2008 workloads, and 3) comparing the

results obtained from the first two phases to assess the accuracy and performance of BestGC's recommendations. The validation process regarding heap sizes, which are obtained empirically and through BestGC for the SPECjvm2008 workloads indicates BestGC's reliability in this aspect.

The evaluation of the accuracy of BestGC's GC suggestions considering user preferences regarding application throughput and GC pause time indicates that BestGC suggests the most suitable GC on average 51.24% of the time and recommends the correct category of the GC (concurrent/non-concurrent) in about 85.95% of cases. When BestGC fails to provide an exact GC or GC category, using its recommended GC still results in about a 1.75% improvement in GC performance compared to the default OpenJDK GC (G1). On average, BestGC offers an overall performance benefit of approximately 36.75% when considering both successful and unsuccessful recommendations, highlighting its potential to significantly enhance application performance, especially in highly optimized environments like OpenJDK. More details on the evaluations are available in Paper II, Section II.6.

6.3.3 Heap Size Adjustment with CPU Control

We utilize the DaCapo Chopin and Hazelcast benchmarks which include latency-sensitive applications to evaluate the adjustment strategy implemented in ZGC (the GC showcased very good results among the concurrent collectors in previous papers). Since the proper configuration of a concurrent collector should avoid allocation stalls, we decided to adopt a manual heap size adjustment strategy for our baseline (vanilla ZGC), where we pick the smallest power-of-two heap size with which the application runs reliably without stalling. This way we compared our modified ZGC with vanilla ZGC. Then, we investigated the implications of our proposed design with varying percentages of GC CPU target overheads: 5%, 10%, 15%, and 20%. The evaluations have been done on an Intel Xeon server machine with 30GB RAM and 32 identical CPUs E5-2680, running Oracle Linux Server 8.4. Throughout the experiments, we closely examined various metrics, including memory usage, application execution time (throughput), application latency, and energy consumption. The following results were obtained:

- Memory usage mostly decreased when the GC target was higher than the application default GC CPU overhead. We even observed a reduction of 96 % for an application (Sunflow) with 15 % and 20 % GC targets. The reduction in memory usage correlates with a higher number of minor and major collections (collections in the young and old generations, respectively), which simply means that GC works more to keep a tighter heap. As expected, in terms of reducing memory footprint, the GC CPU target of 20 % leads to the smallest heap size across almost all the benchmarks.

- The results show that adjusting the heap size dynamically with 15 % and 10 % GC targets had a minimal negative impact on execution time (except for 2 out of 13 workloads). We even obtained a reduction in execution time

in some applications. This improvement can be attributed to the collector compacting live objects close together, improving cache locality [YÖW20a], and making memory accesses easier to prefetch.

- We expected energy changes to exhibit an opposite trend to memory. We anticipated that if a benchmark consumed more CPU during GC than the baseline, then we would see a decrease in memory usage and an increase in energy consumption. This is because CPU usage incurs higher energy costs than DRAM [Hor14]. As expected, the 20 % GC target showed on average worse energy results compared to 10 % and 15 % GC targets. However, it was apparent that the relationship between reduced memory and increased energy is not always linear.

- Our adaptive approach demonstrated that it had no adverse impact on the 99^{th} percentile latency; in fact, it could even lead to latency reduction. For CPU-intensive workloads, where there is intense competition for CPU resources between collector and mutator threads, lower GC targets (e.g., 10 %) resulted in increased memory usage. This positively influenced latency by allowing GC to run less frequently, thereby reducing its impact on mutator performance. While increasing *Xmx* could achieve a similar effect, our approach reduces latency while also decreasing memory usage. By maintaining stable GC CPU overhead around a specified GC target, our technique ensures that GC does not take up a lot of space, allowing mutators to deliver stable low latency without frequent drops. Higher GC targets in CPU-intensive workloads could reduce latency by mitigating contention between GC and mutator threads. However, due to the limited number of available latency-sensitive workloads assessed, we could not draw definitive conclusions regarding the positive impact of our technique on latency. Nevertheless, our results indicated that it did not exhibit a statistically significant adverse effect on latency.

It can be observed that different GC targets yielded varying outcomes for different optimization objectives. The highest GC target, 20 %, proved optimal for memory, while energy optimization favored the lowest GC target. Meanwhile, too many or too few GC cycles can harm performance. Thus, choosing the best GC target for each application may require manual selection. However, an examination of the benchmarks collectively revealed that a 15 % GC target achieved a 51 % reduction in memory usage with only a 3 % increase in execution time and energy consumption (calculated as the geometric mean across all benchmarks[FW86]). Hence, a 15 % GC target may serve as a suitable default choice for optimizing the trade-off between memory usage, execution time, and energy consumption. More details are provided in Paper III, Section III.5.

6.4 Summary

To evaluate our proposed solutions, we utilize several benchmarks and evaluate various metrics. We utilize DaCapo, Renaissance, B2, SPECjvm2008, and

Hazelcast benchmark suites. These benchmarks are chosen for their ability to stress different aspects of GCs and provide insights into GC performance.

Furthermore, to evaluate the results obtained from our solutions, we measure important performance metrics. These metrics include application throughput, GC pause times (reported as the 90^{th} percentile), memory usage, and latency (reported as the 99^{th} percentile). We utilize execution time data to report throughput and analyze GC log files to extract the necessary information for calculating other performance metrics.

The combination of benchmark suites and performance metrics allows us to comprehensively evaluate and compare the performance of different GCs, ensuring the validity and reliability of our research findings. Our findings emphasize the effectiveness of our proposed approaches. They show the impact of various GCs for different application categories. Also, the results underscore the benefits derived from automating the GC selection process, aligning with user preferences for application throughput and pause time. Additionally, results reveal the efficacy of optimizing memory usage by imposing an upper limit on GC CPU usage, a strategy implemented in ZGC which was proven as a superior concurrent GC through prior results.

Chapter 7

Conclusion and FutureWork

In this final chapter, we bring our exploration to a close by encapsulating the contributions of our research journey. We provide a summary of the insights gained from our evaluation of various GCs, the development of BestGC, and the introduction of an approach to heap size adjustment in concurrent GCs. This chapter also looks ahead to the future. We outline promising directions for further research, presenting opportunities for advancing the field of JVM GCs. From a focus on benchmarking and latency-sensitive applications to the potential of machine learning-driven enhancements, we set the stage for what lies ahead.

7.1 Conclusion

Java is one of the leading programming languages and development platforms, with millions of developers running over 60 billion JVMs worldwide[1]. GCs play a pivotal role in managing the heap automatically within the JVM. While a default GC is available, alternative GCs have been developed with different performance goals and trade-offs. Changing the default GC based on an application's requirements and functionalities can potentially improve application performance, offering benefits such as lower latency, increased throughput, and reduced memory usage that can affect, for example, cost savings when using cloud resources. However, the process of selecting the most suitable GC for a specific application is far from straightforward. Users face numerous complications and challenges when evaluating different GCs and their associated parameters. These complications can include understanding the trade-offs between various application's and GC's performance metrics and considering the compatibility of GC algorithms with the application's characteristics and requirements.

As already mentioned in Chapter 1.1, the main motivation of this thesis is to address these complications and empower users to make informed decisions when selecting the best GC solution and a heap size for their Java applications. With a user-centric approach, this research aims to enable users to navigate the complexities of GC and heap size selection effectively. Through a comprehensive analysis of various GCs, their performance characteristics, and the factors influencing their suitability for specific applications, this thesis offers guidance and tools that facilitate the GC and heap size selection process.

We started our work by conducting a comprehensive analysis of various GCs, classifying applications into two categories: CPU-intensive and I/O-intensive, and then, suggesting the most suitable GC for each category. Next, we embarked on the development of a software tool called BestGC. This tool automates

[1] https://www.oracle.com/java/

the GC selection process, allowing users to input their preferences (application throughput and GC pause time) in accordance with their application's goals. BestGC leverages our extensive GC evaluations to provide GC recommendations for the user application. By automating this traditionally complex and time-consuming task, BestGC empowers users to make informed decisions regarding the GC and heap size they use for executing their applications. Finally, by recognizing the complexity and importance of selecting an appropriate heap size for GCs, particularly in concurrent GCs, we directed our attention to this aspect. The complexity surrounding heap size selection for users motivated us to propose an approach that helps users determine the optimal heap size for concurrent collectors. We introduced a new tuning knob, GC CPU utilization target, which allows users to set a limit on the CPU utilization of the GC. This method provides users with a practical and intuitive means to indirectly control the maximum heap size, addressing the challenges associated with this process. We implemented this strategy in ZGC as a concurrent GC that showcased promising results in our prior evaluation.

Importantly, the approaches and methodologies employed throughout these projects can be adapted to other runtime systems. Furthermore, our research findings and tools can be extended to encompass other GCs, broadening their applicability and utility beyond the specific scope of this thesis.

7.2 Future Work

This section highlights several promising directions for future research, which can serve as the foundation for PhD and Masters research proposals.

Given the continuous evolution of GCs and the expanding complexity of contemporary applications, several areas exist for exploration. Future research efforts could delve into conducting thorough evaluations involving a broader array of GCs, by focusing on comparing GCs (including newer versions of powerful GCs like generational versions of ZGC and Shenandoah) and considering various performance metrics. The increasing availability of benchmarks that report detailed data, such as latency (as seen in benchmark suites like DaCapo's Chopin version), provides an excellent opportunity to evaluate GCs with a strong emphasis on latency. Given the low-latency performance of concurrent GCs, such evaluations could offer valuable insights into these GCs' behavior. This, in turn, increases the demand for reliable latency-sensitive benchmarks and opens up opportunities for the development of more applications that facilitate GC evaluations. Therefore, the scope of our research conducted on finding a proper GC solution for CPU-intensive and I/O-intensive applications can be extended to encompass other application classes, e.g., latency-sensitive applications. Additionally, this work can be extended to encompass evaluating other relevant metrics for GCs, such as energy usage. It can explore the energy usage of GCs across various application categories or can address the selection of the most energy-efficient GC for user applications, particularly when energy costs are a significant consideration. Moreover, it can conduct more comprehensive

energy assessments in connection with GC CPU usage when adjusting the heap size during application runtime.

Building upon the foundation of BestGC, future research can explore the integration of machine learning techniques to enhance the precision of GC suggestions for users' applications. The inherent adaptability of the BestGC approach, which allows for training with a diverse range of GCs and performance metrics, and even runtime environments beyond the JVM, offers the possibility of machine learning-driven enhancements. This line of research could ultimately lead to the development of an advanced enterprise-level system for optimizing GC selection across diverse application scenarios.

The research on dynamically adjusting heap sizes based on GC CPU utilization represents an approach to memory management. Future investigations could build upon this strategy by evaluating its effectiveness across a broader range of benchmark classes. Exploring its applicability in different concurrent GCs can provide valuable insights into its extensibility and potential areas for optimization.

In summary, the evolving landscape of GC technologies and the diverse requirements of modern applications offer diverse research opportunities. This thesis provides a roadmap for future research works that can make a significant impact on the field of JVM GCs. These efforts will contribute to creating more efficient, user-friendly, and adaptable solutions that can better help both users and developers in this domain.

Bibliography

[App89] Appel, A. W. “Simple Generational Garbage Collection and Fast Allocation”. In: *Softw. Pract. Exper.* vol. 19, no. 2 (Feb. 1989), pp. 171–183.

[Ber+22] Beronić, D. et al. “Assessing Contemporary Automated Memory Management in Java–Garbage First, Shenandoah, and Z Garbage Collectors Comparison”. In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1495–1500.

[BF17] Bruno, R. and Ferreira, P. “POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications”. In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 147–160.

[BF18] Bruno, R. and Ferreira, P. “A study on garbage collection algorithms for big data environments”. In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. “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*. 2006, pp. 169–190.

[BM03] Blackburn, S. M. and McKinley, K. S. “Ulterior reference counting: Fast garbage collection without a long wait”. In: (2003), pp. 344–358.

[BM08] Blackburn, S. M. and McKinley, K. S. “Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 22–32.

[BOF17] Bruno, R., Oliveira, L. P., and Ferreira, P. “NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications”. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 2017, pp. 2–13.

[Bre+01] Brecht, T. et al. “Controlling garbage collection and heap growth to reduce the execution time of Java applications”. In: *ACM Sigplan Notices* vol. 36, no. 11 (2001), pp. 353–366.

[Bru+18] Bruno, R. et al. “Dynamic vertical memory scalability for OpenJDK cloud applications”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, pp. 59–70.

[Bru+19] Bruno, R. et al. “Runtime object lifetime profiler for latency sensitive big data applications”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[BW88] Boehm, H.-J. and Weiser, M. “Garbage collection in an uncooperative environment”. In: *Software: Practice and Experience* vol. 18, no. 9 (1988), pp. 807–820.

[Cai+22] Cai, Z. et al. “Distilling the real cost of production garbage collectors”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 46–57.

[CB01] Cheng, P. and Blelloch, G. E. “A parallel, real-time garbage collector”. In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 2001, pp. 125–136.

[CK22] Clark, I. and Karlsson, S. *Z Garbage Collector*. Sept. 2022. URL: https://wiki.openjdk.org/display/zgc/Main.

[Coo+21] Cook, M. et al. *Announcing preview release for the generational mode to the Shenandoah GC*. Oct. 2021. URL: https://aws.amazon.com/blogs/developer/announcing-preview-release-for-the-generational-mode-to-the-shenandoah-gc/.

[CT21] Contributors, J. R. and Team, C. *Jikes RVM Garbage Collectors*. 2021. URL: https://www.jikesrvm.org/UserGuide/ConfiguringJikesRVM/index.html.

[Deg+16] Degenbaev, U. et al. “Idle time garbage collection scheduling”. In: *ACM SIGPLAN Notices* vol. 51, no. 6 (2016), pp. 570–583.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Eva20] Evans, B. *What Tens of Millions of VMs Reveal about the State of Java*. Mar. 2020. URL: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[Fou18] Foundation, T. A. S. *Apache spark™ - unified engine for large-scale data analytics*. 2018.

[Fra+07] Frampton, D. et al. “Generational real-time garbage collection”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 101–125.

[FW86] Fleming, P. J. and Wallace, J. J. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Commun. ACM* vol. 29 (1986), pp. 218–221.

[Gen+21] Gencer, C. et al. “Hazelcast jet: Low-latency stream processing at the 99.99^{th} percentile”. In: vol. 14, no. 12 (2021), pp. 3110–3121.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. “Comparison of garbage collectors in Java programming language”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gog+15] Gog, I. et al. “Broom: Sweeping out garbage collection from big data systems”. In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[Grz+07] Grzegorczyk, C. et al. “Isla vista heap sizing: Using feedback to avoid paging”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE. 2007, pp. 325–340.

[HM95] Han, J. and Moraga, C. “The influence of the sigmoid function parameters on the speed of backpropagation learning”. In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[Hor14] Horowitz, M. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[Iri15] Iris Clark, A. S. *Shenandoah GC*. Sept. 2015. URL: https://wiki.openjdk.org/display/shenandoah/Main.

[JHM16] Jones, R., Hosking, A., and Moss, E. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[Kar21] Karlsson, S. *JEP 439: Generational ZGC*. Aug. 2021. URL: https://openjdk.org/jeps/439.

[Len+17] Lengauer, P. et al. “A comprehensive java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo scala, and SPECjvm2008”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 3–14.

[LH83] Lieberman, H. and Hewitt, C. “A real-time garbage collector based on the lifetimes of objects”. In: *Communications of the ACM* vol. 26, no. 6 (1983), pp. 419–429.

[McC60] McCarthy, J. “Recursive functions of symbolic expressions and their computation by machine, Part I”. In: *Communications of the ACM* vol. 3, no. 4 (1960), pp. 184–195.

[Ngu+16] Nguyen, K. et al. “Yak: A High-Performance Big-Data-Friendly Garbage Collector”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 349–365.

[Ora15] Oracle. *Concurrent Mark Sweep (CMS) Collector*. 2015. URL: https://docs.oracle.com/en/java/javase/11/%20gctuning/concurrent-mark-sweep-cms-collector.html.

[Ora20] Oracle. *HotSpot Virtual Machine Garbage Collection Tuning Guide, JDK15*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf.

[Ora21] Oracle. *Garbage-First Garbage Collector Tuning*. 2021. URL: https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-garbage-collector-tuning.html#GUID-3D3E4662-1E89-42EE-96FA-836C0E7C97AA.

[Ora22] Oracle. *HotSpot Virtual Machine Garbage Collection Tuning Guide*. May 2022. URL: https://docs.oracle.com/en/java/javase/17/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf.

[Oss+02] Ossia, Y. et al. “A parallel, incremental and concurrent GC for servers”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[PD00] Printezis, T. and Detlefs, D. “A generational mostly-concurrent garbage collector”. In: *Proceedings of the 2nd international symposium on Memory management*. 2000, pp. 143–154.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. “Analysis of garbage collection algorithms and memory management in Java”. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. “Stopless: a real-time garbage collector for multi-processors”. In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. “A study of concurrent real-time garbage collectors”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sew+11] Sewe, A. et al. “Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine”. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, pp. 657–676.

[Spr07] Spring PetClinic. *Spring PetClinic Project*. 2007. URL: https://spring-petclinic.github.io/.

[Sta08] Standard Performance Evaluation Corporation. *SPECjvm2008*. 2008. URL: http://www.spec.org/jvm2008/index.html.

[TIW11] Tene, G., Iyengar, B., and Wolf, M. "C4: the continuously concurrent compacting collector". In: *International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*. 2011.

[Top20] Topolnik, M. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*. June 2020. URL: https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch.

[Whi+13] White, D. R. et al. "Control theory for principled heap sizing". In: *ACM SIGPLAN Notices* vol. 48, no. 11 (2013), pp. 27–38.

[Wu+20] Wu, M. et al. "Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services". In: *USENIX Annual Technical Conference*. 2020.

[Xu+19] Xu, L. et al. "An experimental evaluation of garbage collectors on big data applications". In: *The 45th International Conference on Very Large Data Bases (VLDB'19)*. 2019.

[Yan+04] Yang, T. et al. "Automatic heap sizing: Taking real memory into account". In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 61–72.

[YÖW20a] Yang, A. M., Österlund, E., and Wrigstad, T. "Improving Program Locality in the GC Using Hotness". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[YÖW20b] Yang, A. M., Österlund, E., and Wrigstad, T. "Improving program locality in the GC using hotness". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 301–313.

[YW22] Yang, A. M. and Wrigstad, T. "Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 44, no. 4 (2022), pp. 1–34.

[ZB20] Zhao, W. and Blackburn, S. M. "Deconstructing the garbage-first collector". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

[ZBM22] Zhao, W., Blackburn, S. M., and McKinley, K. S. "Low-latency, high-throughput garbage collection". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 76–91.

[Zha+06] Zhang, C. et al. “Program-level adaptive memory management”. In: *Proceedings of the 5th international symposium on Memory management*. 2006, pp. 174–183.

Papers

Paper I

Selecting a GC for Java Applications

I

Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira

Published in *2021 Norsk IKT-konferanse for forskning og utdanning Conference*, November 2021, Trondheim, Norway, pp. 2–15.

Abstract

Nowadays, there are several Garbage Collector (GC) solutions that can be used in an application. Such GCs behave differently regarding several performance metrics, in particular throughput, pause time, and memory usage. Thus, choosing the correct GC is far from trivial due to the impact that different GCs have on several performance metrics. This problem is particularly evident in applications that process high volumes of data/transactions especially, potentially leading to missed Service Level Agreements (SLAs) or high cloud hosting costs.

In this paper, we present: i) a thorough evaluation of several of the most widely known and available GCs for Java in OpenJDK HotSpot using different applications, and ii) a method to easily pick the best one. Choosing the best GC is done while taking into account the kind of application that is being considered (CPU or I/O intensive) and the performance metrics that one may want to consider: throughput, pause time, or memory usage.

Contents

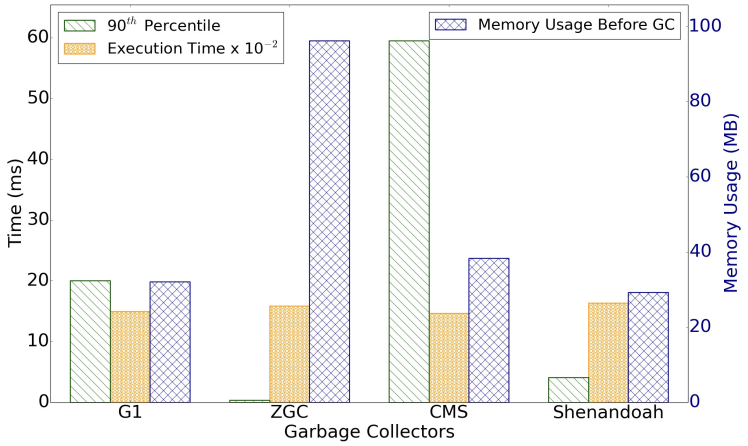


Figure I.1: Throughput, 90$^{\text{th}}$ percentile of pause times, and memory usage before GC, in the Luindex benchmark.

I.1 Introduction

In object-oriented programming languages, e. g. Java, automatic memory management regulates all the objects' allocations and deallocations in memory using Garbage Collection algorithms. A Garbage Collector (GC) in Java, which is responsible to free objects that are no longer referenced by any part of running applications and processes, is even more important when applications are dealing with high volumes of data (i.e., Big Data and/or Cloud Services [CML14]).

Existing garbage collection solutions apply a trade-off between different performance metrics based on how the GC algorithm is designed. Taking Luindex benchmark (included in DaCapo benchmark suites [Bla+06]) as an example, 90$^{\text{th}}$ percentile of pause times along with execution time and memory usage for four different GCs are depicted in Figure I.1. We can conclude that ZGC [Per18] has almost 12x and 57x less pause time compared to Shenandoah [Flo+16] and G1 [Det+04] GCs, respectively. regarding pause time, CMS [Jon96] would be the last choice when compared to the other GCs.

As Figure I.1 shows, the minimum execution time for Luindex is obtained when CMS is running; also, there is a slight difference between the execution time in CMS and G1 (we consider an application's execution time as the throughput; for more detail see Sections I.3.3 and I.4). Furthermore, Shenandoah followed by the G1 uses the heap memory better than the two other GCs before garbage collection. Based on the results, ZGC uses about 3x more memory than Shenandoah. For this particular benchmark, although ZGC is the best option regarding pause time, in terms of memory usage before GC it is the worst choice.

The existence of several GCs with diverse functionality and purposes makes it hard for a developer to choose the most suitable GC solution for a given application. This happens especially when the application deals with a large amount of data and demands a significant amount of resources like CPU

or I/O. To select the most suitable GC, a user must know whether a given application is CPU-intensive or I/O-intensive; then, the user only requires to decide what GC performance metric (throughput, pause time, and memory usage) he wants to prioritize. In this paper, we evaluate how several widely known GCs behave regarding the above mentioned performance metrics, then we propose a methodology to choose the best GC for a given Java application running on the most widely used Java Virtual Machine (JVM) implementation, OpenJDK HotSpot JVM.

Several relevant works studied and compared various performance metrics in different GC solutions using existing benchmarks or real applications. However, to the best of our knowledge, selecting the best GC solution for a given Java application, given its characteristics (CPU or I/O intensive), and regarding some GC performance metrics, is not covered in previous studies.

In this work, we use both Renaissance [Pro+19] and DaCapo [Bla+06] benchmark suites that represent a large set of applications. We also developed an application, called BufferBench (B2), that does read/write operations in memory to better study the GCs' behavior and the cost associated with the read and write barriers that some GCs use. Furthermore, we use Spring Boot based PetClinic application [Spr07] to investigate our methodology using a real-world application. Using these applications, while applications are classified as being CPU-intensive or I/O-intensive, we investigate four well-known GCs available in the JVM. G1, which is the default GC since JDK 9, ZGC, Shenandoah, and CMS to evaluate their performance metrics. According to the results, we recommend the best GC solution for a specific application to fulfill its requirements: maximize throughput, minimize pause time, or minimize memory usage.

In the following section, we present related work, while providing some basic information about the key concepts of GCs. In Section I.3, we describe the architecture of our solution with a focus on its most relevant technical aspects. In Section I.4, we give some implementation details, and in Section I.5 we describe the experiments and the results obtained. We conclude this paper with the conclusions in Section I.6.

I.2 Related Work

This section starts with a summary of background work regarding basics aspects of garbage collection, with a focus on some relevant characteristics of GCs that are addressed in the remaining of this article. Then, it addresses some existing work that can be compared to ours.

I.2.1 Background

Garbage collection is an essential part of automatic memory management and aims at allowing the reuse of the memory occupied by unused objects. This leads to organizing objects in two main groups: i) dead objects, i.e., those that are not being referenced by any running application, and ii) live objects, i.e., those that are still referred from an application.

In Java run-time systems, several GC solutions were developed to automatically manage heap memory that stores objects created by applications on the JVM. Although GCs employ different approaches to optimize their target performance metrics, they tend to make a trade-off between: **throughput** (number of operations done by an application), **pause time** (time that an application is forced to stop to let the GC execute), and **memory usage**(the amount of memory used in a process).

There are several designs and implementations for GC solutions [JHM11]. Some GCs partition the heap into multiple partitions or sub-heaps [UJ88]. This is the case of generational GCs in which the partitioned heap holds objects that are segregated based on their lifetime. In this case, a newly created object is placed in the young generation partition; should it survive several GC cycles, it will be transferred to the old generation partition. In serial GCs [Ora16] garbage collection is done serially using just one single thread in both generations, or multiple threads can be used by GC in the young generation (parallel GC) or in both young and old generations (Parallel Old GC) [Ora16]. In the GCs to organize the free spaces in the heap, compaction and copying algorithms may be used. Through the compaction, GC moves all the live objects to the beginning of the memory segment, whilst through copying, a GC groups the live objects by moving them from multiple memory segments into a single one [BF18].

Moreover, several GCs are designed to work concurrently with a running application; multiple threads are responsible for running the application and the GC simultaneously. In contrast, some GCs employ the stop-the-world (STW) technique, in which the GC stops an application while running.

Also, GC algorithms use read and/or write barriers. Read barrier (also known as load barrier) code is run whenever an application thread loads a reference from the heap. A write barrier is also called by the compiler just before any write operation to some object occurs.

There are multiple GC algorithms available for the JVM. **Concurrent Mark/ Sweep collector (CMS)** [Ora15] is a tracing collector (i. e. attempts to identify all the reachable objects in the heap by tracing the root objects) and generational GC including young and old generations. CMS employs an STW mark and copy technique in the young generation while there is a concurrent mark and sweep collector in the old generation to collect the marked unreachable objects. Also, It uses a write barrier that is run every time a reference in an object is updated. Since CMS does not have the compacting step; this may lead to heap fragmentation [Xu+19].

Garbage first (G1) [Det+04] is the default GC since JDK9. G1 is a generational GC with fixed-size regions in the heap that uses a compacting algorithm. In STW young generation GC phase, G1 starts to traverse the object graph to find the live objects and copies them to the old regions. The old generation GC employs write barriers to mark live objects concurrently with the running application [ZB20]. In fact, the main strategy in G1 is to reclaim regions with the least live objects, i.e., most garbage first (as its name suggests).

The main goal in **Shenandoah** [Flo+16] GC is having short pause times regardless of the heap size. It maintains the heap as a collection of regions

and uses both concurrent copy and compaction techniques. To achieve object relocation concurrently with the application threads, Shenandoah uses a data structure that needs an additional field per object which points back to the object itself, when initializing the object and, with using the write barriers, to the object's new location once it is moved. It also uses read barriers to read objects from the exact current location in the heap memory.

ZGC [Per18] is a non-generational GC that divides the heap into regions of different sizes (small, medium, and huge) to hold the objects based on their size. To relocate the live objects concurrently, it uses read barriers and *colored pointers*, and stores some important metadata to hold information about an object itself and marking and relocation-related information [PGM19]. ZGC applies STW pauses to mark live objects and mark the regions for compaction.

I.2.2 Existing Work

Several studies have been conducted over the past few years regarding GC designs, methodology, and performance in Java.

Ossia et el. [Oss+02] designed an incremental, and concurrent collector, with threads running in parallel, to ensure low pause time; Also, Pizlo et al. [Piz+07] propose a GC, STOPLESS, that uses a mark and sweep collector and a compactor to avoid fragmentation, for multi-processors running multithreaded applications. Pizlo et al. [PPS08], also propose two other solutions for concurrent real-time GCs, CLOVER and CHICKEN, to improve the complexity of STOPLESS. Detlefs et al. [Det+04] propose a GC that attempts to achieve high throughput for the applications running on multi-processor systems while they have a high allocation rate and need large heaps. Printezis et al. [PD00], propose a GC algorithm that works mostly concurrent with the application and attempts to reduce the worst-case pause time in generational memory systems.

Zhao et el. [ZB20] decompose G1 into several key components and reimplement it. Also, they produce six collectors to evaluate different algorithmic elements of G1 using benchmarks including the DaCapo benchmark suite (also used by us as shown in Section I.5). They developed a concurrent, region-based moving collector to find the shared elements in G1, ZGC, C4 [TIW11], and Shenandoah. Pufek et al. [PGM19] used selected benchmarks of the DaCapo suite and compared the results of G1, Parallel, Serial, and CMS GCs regarding the time spent in the garbage collection process as well as the number of collections in JDK versions 8, 11, and 12. Grgic et al. [GMR18] analyzed the same GCs as Pufek's also using DaCapo benchmarks. They concluded that G1 performed better than CMS and Parallel GC regarding the total number of garbage collections and CPU utilization in multi-threaded applications. Prokopec et al. [Pa19] describe Renaissance benchmark suite and explain the focus of each benchmarks. They also provide information about different metrics, including average CPU utilization, during a single steady-state execution of Renaissance and DaCapo benchmark suites.

Lengauer et al. [Len+17] provides a description of commonly used benchmarks,

including DaCapo, DaCapo Scala [Sew+11], and SPECjvm2008,[1] in terms of memory and garbage collection behavior. They compared G1 and Parallel Old (parallel old generation) regarding GC counts, the GC time relative to total run time, and the total pause time. They concluded that G1 performs better than the Parallel Old regarding pause time as it can select which regions to collect.

There are several works that studied GCs in Big Data systems. Bruno et al. [BF18] describes existing GC solutions for Big Data environments and the scalability of some memory management algorithms in terms of throughput and pause time. Nguyen et al. [Ngu+16] propose a GC for Big Data systems with low pause time and high throughput To reduce the objects managed by the GC, the GC divides the heap into data space and control space and uses generation-based and region-based algorithms in the spaces respectively. However, the developer has to mark the beginning and end points of the data path in the program. Broom [Gog+15] and NG2C [BOF17] propose two GCs for Big Data systems. In Broom the regions in the heap are explicitly created by the programmer; in NG2C, the programmer identifies the generation where a new object should be allocated

To the best of our knowledge, there are no studies that compare all the GCs we consider (G1, CMS, ZGC, and Shenandoah) regarding throughput, pause time, and memory usage. Also, none of the previous work proposes a methodology to pick the best GC both for CPU-intensive and I/O-intensive applications.

I.3 Architecture

We start this section by first describing how a user can pick the best GC for his application; then, we describe if a certain application (see Section I.5) is CPU or I/O intensive; finally, we present some main aspects of the system.

I.3.1 User Level Overview

When a user wants to choose the best GC solution for a given application, he must: i) characterize the application as being CPU-intensive or I/O-intensive; and ii) decide what GC metric he wants to prioritize among the following three: throughput, pause time, or memory usage.

The first item mentioned above is simply done (by the user) by running the application and using the Linux command *atop* (more details are given in see Section I.4) to categorize an application as being CPU-intensive or I/O-intensive.

Then, based on the GC metrics that the user wants to make better, the best GC regarding throughput is CMS in both CPU and I/O-intensive applications. Also, ZGC minimizes the pause time in almost all the CPU and I/O-intensive applications; and CMS has acceptable results regarding using memory before garbage collection, especially in CPU-intensive applications, while CMS can be replaced with Shenandoah in I/O-intensive applications; G1 is a good choice for

[1] https://www.spec.org/jvm2008/

Table I.1: Categorizing benchmarks as CPU or I/O intensive based on average CPU usage per core (%) and average I/O usage (%)

a) Renaissance benchmark suite

Benchmark	Avg CPU	Avg I/O	Category
Akka Uct	89.25	58	CPU-intensive
Als	82.25	93	I/O-intensive
Chi Square	102	39	CPU-intensive
Dec Tree	88.66	60	CPU-intensive
Fj Kmeans	86.28	21	CPU-intensive
Future Genetic	88	12	CPU-intensive
Gauss Mix	99.5	29	CPU-intensive
Mnemonics	104	12	CPU-intensive
Movie Lens	96.33	98	I/O-intensive
Naive Bayes	79.62	79	CPU-intensive
Neo4j Analytics	87.33	38	CPU-intensive
Page Rank	72.37	56	CPU-intensive
Par Mnemonics	129	13	CPU-intensive
Philosophers	92.14	12	CPU-intensive
Reactors	77	17	CPU-intensive
Rx Scrabble	95	32	CPU-intensive
Scala Doku	109	14	CPU-intensive
Scala Kmeans	106	27	CPU-intensive
Scrabble	80.66	14	CPU-intensive

b) DaCapo benchmark suite

Benchmark	Avg CPU	Avg I/O	Category
Avrora	66.24	92	I/O-intensive
Fop	58.81	80	I/O-intensive
H2	76.15	10	CPU-intensive
Jython	122.24	15	CPU-intensive
Luindex	62.09	99	I/O-intensive
Lusearch-fix	88.84	99	I/O-intensive
PMD	76.71	43	CPU-intensive
Sunflow	92.65	23	CPU-intensive
Xalan	96.01	100	I/O-intensive

c) B2

Benchmark	Avg CPU	Avg I/O	Category
1M-25% read	66.00	46.34	CPU-intensive
1M-50% read	69.73	27.18	CPU-intensive
1M-75% read	83.87	21.07	CPU-intensive
1M-100% read	101.03	11.32	CPU-intensive
2M-25% read	58.19	49.68	CPU-intensive
2M-50% read	56.16	49.85	CPU-intensive
2M-75% read	53.75	40.30	CPU-intensive
2M-100% read	66.87	14.58	CPU-intensive

d) PetClinic Application

Benchmark	Avg CPU	Avg I/O	Category
PetClinic	88.25	17	CPU-intensive

reducing memory usage after GC in CPU-intensive applications, CMS works as well as G1 regarding this metric in I/O-intensive applications (see Section I.5).

I.3.2 Application Classification

In this section, we focus on classifying applications based on their CPU and I/O usage. As described in Section I.4, all the process mentioned above is done with G1 activated as a default GC in the OpenJDK and by employing representative applications available in well-known benchmark suites: Renaissance [Pro+19] and DaCapo [Bla+06]. These two Java-based benchmark suites allow us to disable/enable multiple features (e.g., disabling garbage collection before each iteration in benchmarks, or change input parameters like input size), and are representative of most existing applications.

We also developed B2 to test and evaluate the aforementioned GCs so that we can broaden our study experiments. In particular, B2 makes read and write operations from/to the heap while the percentage of the read and write operations can be changed. This leads to triggering GC, affects the GC's behavior and indeed the performance metrics.

Table I.1 shows the relative average CPU consumption per core and average I/O usage of each application used in the evaluation (for more details, see Section I.5). We tag an application as CPU-intensive if the percentage of average CPU usage is bigger than the average I/O usage; otherwise, we consider the application to be I/O-intensive.

Among Renaissance benchmarks, as shown in Table I.1.a, only ALS and Movie Lens benchmarks have a greater percentage of I/O usage than the percentage of CPU consumption; thus, these applications are considered to be I/O-intensive. All the other applications in the Renaissance benchmark suite are considered to be CPU-Intensive.

In the DaCapo benchmark suite (Table I.1.b) more than 50% of the benchmarks are I/O-intensive; H2, Jython, PMD, and Sunflow are the only CPU-intensive applications.

In B2 (Table I.1.c) all running cases are as CPU-intensive (in fact, the CPU is the resource that is most used). And, finally, Table I.1.d shows that there is a significant difference between average CPU usage and average I/O usage in the PetClinic application; thus, we conclude that PetClinic is CPU-intensive.

I.3.3 Main Aspects

The GC solution especially when there are huge amounts of objects like in Big Data and Cloud environments, has a great impact on different performance metrics. Therefore, by knowing a given application's requirements regarding throughput, pause time, or memory usage, while identifying the resources an application needs mostly (CPU or I/O), we can find the most suitable GC.

On one hand, using Renaissance and DaCapo benchmarks allows us to evaluate throughput, pause time, and memory usage and examine how they change by employing different GC solutions. On the other hand, using the two benchmarks suites while knowing if the benchmarks included are CPU or I/O-intensive, aids to figure out which GC solution is the best to a specific application.

We consider an application's execution time as a measure of its throughput, rather than the (most usual) number of operations in a time interval. The reason is to make the different benchmarks comparable since they are in different contexts and software designs; thus, it is impossible to define a common definition of what an operation is. The execution time includes all the time that is needed for the GC process and the time for the execution of the application itself. Since the amount of work is the same for each benchmark, and this is the GC that is changed, less execution time means a better throughput.

Moreover, recording the logs generated by the GCs, about heap usage changes during GC's phases, provides useful information regarding GC's pause times and memory usage. We extract the records related to pause times to calculate the desired percentile of such metric. The average amount of memory usage before and after a garbage collection process indicates how a GC is successful to reduce memory usage, deplete the dead objects from the heap, and prepare the heap to host new objects. An ideal GC manages memory usage properly before garbage collection and frees up significant memory by performing garbage collection.

Knowing how each GC performs regarding throughput, pause time, and memory usage in different applications, while an application is CPU or I/O intensive, leads to picking the best GC solution. The user who knows the application's requirements can then decide on the best GC that fulfills them more accurately.

I.4 Implementation

We developed B2 on a server running GNU/Linux, Ubuntu 16.04.4 LTS, with four *Intel(R) Xeon(R) E5506 @ 2.13GHz* CPU cores and *16 GB* of RAM memory. We employed OpenJDK 13 and set the heap memory size to *4 GB*.

Moreover, we create a configuration file for each garbage collector. The file consists of JAVA_HOME, JAVA_EXE, and JAVA_OPTS that indicate the address of the Java folder on the machine, Java execution file location, and the desired options to run the Java run-time with them, respectively. We added the relevant Java options for each GC in the related configuration file to replace the default GC. For G1, we added *–XX:+UseG1GC*, for CMS *–XX:+Use ConcMarkSweepGC*, for Shenandoah *–XX:+UseShenandoahGC*, and for ZGC *–XX:+UseZGC*. Also, we set the maximum heap to 4 gigabytes, using *–Xmx4g* and added the *-Xlog:gc** to enable the printing of each GC's detailed messages.

B2 is a simple application written in Java with a focus on evaluating the impact of read and write barriers on garbage collection performance metrics. This application needs three input values: data size, number of operations, and read percentage. Based on the data size, a hash map is created to keep the objects. In our study, data size is either 1 or 2 million objects. The number of operations indicates the total number of both read and write operations; this is set to 100 million in our study. The results showed that this amount is big enough to trigger several GC cycles to measure the GC's performance metrics. Finally, the read percentage indicates how much of the operations are reading objects from the heap; the remaining percentage refers to write operations that create new objects in the heap. We chose 25%, 50%, 75%, and 100% as read percentages in our study. We change the percentage of read (write) operations to analyze GC's behavior regarding their performance metrics. Also, we wrote a simple script that evaluates the GCs with the defined data size, the number of operations, and read percentage using B2 JAR file. So, there is a triple nested loop; each iteration is executed with one of the four evaluated GCs at a time. To run the B2 application using each configuration file, the B2 JAR file is executed with the three inputs using the configuration file of each GC as the selected Java executor. At the end of each iteration, the log file of the running GC is copied to the desired file. We also record the exact time of the start and the end of the B2 application to calculate the execution time (i.e., its throughput).

We also wrote a script to execute each of the benchmarks. For DaCapo, the script runs each benchmark 10 times using each GC configuration file and the DaCapo JAR file. The DaCapo JAR file needs three input variables: benchmark name, number of iterations (using switch -n), and the input size (using switch -s). The input size is set to *large* for all benchmarks, except for *fop* and *luindex* that were set to *default*, since *large* was not available for them. We recorded the DaCapo output log that shows the execution time for each iteration.

We followed the same process to write an executable script for the Renaissance. The difference is that there is no input size for the benchmarks in Renaissance. We used *–repetitions* to set the number of iterations to 10. Also, we used *–no-forced-gc*, to disable the garbage collection that would be forced by the original

Renaissance benchmark suite before each iteration.

As we mentioned in Section I.3.1, to categorize an application into CPU-intensive or I/O-intensive, we use *atop* command to obtain useful information regarding CPU and disk usage of any application. A user should activate the *storage accounting* feature in the OS (using the command *sudo /usr/sbin/accton on*) as it shows the accumulated read and write throughput percentage on disk. Then, a user just has to log the *disk* column values per second and calculate the average percentage of disk utilization. In addition, with the *hyperthreading* feature deactivated, a user collects the values in the *CPU* column for the Java process; this indicates the CPU consumption per second. The hyperthreading feature specifies the number of threads per core and can be deactivated both in BIOS or by changing the simultaneous multithreading (SMT) state to *off*. Then, it is just a matter of calculating the average amount of CPU consumption and divide it by the number of cores that are being used by the running Java process. We just do this in our study to analyze the CPU usage of the benchmarks and to categorize them as CPU or I/O intensive.

There are 8 cores available on our server, and to obtain the number of cores engaged for each application, a user simply determines the average number of running threads of the corresponding Java process by executing the Linux command *htop*. Then, a user uses this number and divides the average CPU utilization by it to obtain a relative average CPU usage per core. There may be results bigger than 100% because of the relative numbers of engaged cores.

We capture the *atop* data every second during the execution of an application using a script. At the same time, we extract the values for running threads from *htop* and store them in a file. Then, the script executes the applications, as explained before, just with the G1 configuration file since we just need to check the CPU and disk utilization of applications. In the end, we just read the *atop* output file and extract the data related to disk and CPU utilization and print them into separate files.

All the software and the related scripts are open source and are available on https://anonymous.4open.science/r/BestGC-6A71.

I.5 Evaluation

In this section we start by presenting the methodology used to obtain the results concerning the GC metrics, i.e., throughput, pause time, and memory usage for each one of the GCs evaluated (CMS, G1, ZGC, and Shenandoah).

I.5.1 Methodology

All the experiments were done on a server running GNU/Linux, Ubuntu 16.04.4 LTS, with four *Intel(R) Xeon(R) E5506 @ 2.13GHz* CPU cores and *16 GB* of RAM memory. We employ OpenJDK 13 and set the heap memory size in JVM to *4 GB* to exercise Java-based applications (it was large enough to run all workloads for most of the applications). To run the applications with the desired

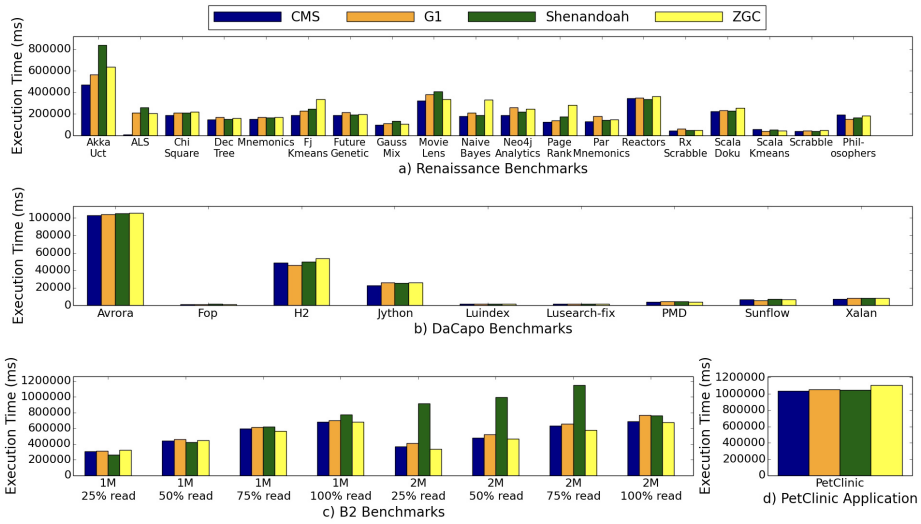


Figure I.2: Average execution time in applications using different GCs.

GCs, we made a separate configuration file for each GC containing different JVM options for each one of them.

To evaluate CMS, G1, ZGC, and Shenandoah in JVM, we use Renaissance (version 0.11.0) and DaCapo (version 9.12) benchmark suites that include several applications (see Section I.3.2).

When using these benchmark suites, after several experimental evaluations, we set the number of iterations for each benchmark to 10 (as we obtained stable results from then on and adequate data to study). Also, during the study, we excluded particular benchmarks (e.g., Batik, Tradebeans, and Tradesoap in the Dacapo benchmark suite, and Db-shootout, Dotty, and Finagle-chirper in the Renaissance benchmark suite) from the experiments if one of the GCs was not able to do garbage collection and takes an excessively large amount of time while the application stops doing any progress waiting for more memory.

In B2, which is developed with a focus on the performance of GC read and write barriers, we consider 100 million read/write operations. We change the read percentage to 25 (i. e. 25% of 100 million operations are allocated to read the objects from the memory, and 75% of them are related to generate new objects in the heap), 50, 75, or 100, while there are 1 or 2 million objects created in the heap.

Moreover, to evaluate the GC performance metrics with a real-world application we examine the GCs using the PetClinic Spring Boot application. First, we investigate if the application is CPU or I/O intensive, then we evaluate for each GC, the throughput, pause time, and memory usage. To test the PetClinic application we use Apache JMeter [Apa] (version 5.4). Using JMeter, the APIs in the Petclinic application were called using 100 threads and data files containing records as input values for different entities in the application. We

also set the number of iterations to 10 for each thread group. Choosing these numbers after several evaluations, revealed a typical usage of the application, reduced the errors resulting from using simultaneous threads, triggered several GC cycles and used resources significantly in the server.

For all the applications, to evaluate throughput, pause time, and memory usage we mainly use the log files obtained from their execution. Regarding throughput, we used the average execution time of the application. Then, using the JVM log files, we find all the records related to pause times and calculate the 90^{th} *percentile*. We also calculate the average heap memory usage before and after GC , and then evaluate the percentage of memory reduction achieved by the GC (see Section I.3.3). In the following section, we present the results of the evaluations.

I.5.2 Results

In this section, we present the results of the experiments regarding throughput, pause time, and memory usage, while different CPU-intensive or I/O-intensive applications are being executed. Then, we present a discussion of the results.

I.5.2.1 Throughput

Figure I.2 shows the results regarding average throughput (execution time) for both the Renaissance and DaCapo benchmark suites as well as for B2 and PetClinic.

Figure I.2.a reveals that, in 16 out of 19 benchmarks, on average, CMS performs better. G1 comes after CMS with around a 16% difference in average execution time (the lower the execution time the better the throughput). ZGC is the worst option in all of the Renaissance benchmarks.

Figure I.2.b shows the average execution time for DaCapo benchmarks. In 7 out of 9 benchmarks, CMS has the minimum execution time. While for H2 and Sunflow, G1 is the preferred GC. On average, there is less than a 1% difference between CMS and G1 execution times in the benchmarks included in DaCapo. Shenandoah and ZGC, with an average difference of about 5% and 3% compared to CMS, are the last option for all benchmarks in DaCapo.

In Figure I.2.c, unlike the previous applications, ZGC has the lowest execution time in 75% of the cases. CMS, with around a 3% difference when compared to ZGC in the average value of all the execution times, is the second choice for B2.

In the PetClinic, the lowest average execution time (the best throughput) is obtained when using CMS. After CMS, Shenandoah can be selected; however, there is a very small difference between GCs' execution times in PetClinic.

I.5.2.2 Pause Time

Table I.2 shows the 90^{th} percentile of all the pause times obtained from all the executions for each application (because of the very small values obtained using ZGC and a huge difference between the CMS and ZGC results, we preferred to

Table I.2: 90^{th} Percentile of pause times (ms).

a) Renaissance benchmark suite

Benchmark	CMS	G1	Shenandoah	ZGC
Akka Uct	139.14	102.03	24.00	**1.07**
als	36.08	70.11	2.89	**1.18**
Chi Square	17.22	27.65	2.81	**0.63**
Dec Tree	28.94	27.34	2.99	**0.69**
Fj Kmeans	15.95	7.84	1.50	**0.47**
Future Genetic	19.22	9.70	1.50	**0.33**
Gauss Mix	17.20	15.41	2.35	**0.92**
mnemonics	53.50	19.62	1.68	**0.53**
Movie Lens	21.00	60.10	7.21	**0.80**
Naive Bayes	59.74	18.97	2.50	**0.45**
Neo4j Analytics	102.21	120.34	3.05	**0.57**
Page Rank	156.24	128.73	2.90	**0.62**
Par Mnemonics	52.05	22.53	1.74	**0.55**
philosophers	17.58	6.72	1.17	**0.32**
Reactors	346.11	185.85	2.22	**0.44**
Rx Scrabble	20.23	38.89	2.22	**0.61**
Scala Doku	53.92	55.48	1.73	**0.53**
Scala Kmeans	110.39	41.36	1.94	**0.41**
Scrabble	33.60	26.62	2.48	**0.41**

b) DaCapo benchmark suite

Benchmark	CMS	G1	Shenandoah	ZGC
Avrora	229.61	12.32	2.02	**0.31**
Fop	107.25	22.61	2.61	**0.34**
H2	942.84	313.25	2.41	**0.46**
Jython	17.36	79.21	2.51	**0.67**
Luindex	59.50	20.02	4.06	**0.35**
Lusearch-fix	53.09	32.19	2.94	**0.34**
Pmd	131.17	20.95	3.47	**0.48**
Sunflow	17.39	19.52	1.58	**0.49**
Xalan	15.08	24.08	1.42	**0.41**

c) B2

Benchmark	CMS	G1	Shenandoah	ZGC
1M-25% read	298.66	111.14	2.48	**0.40**
1M-50% read	204.92	106.42	2.48	**0.39**
1M-75% read	123.46	104.25	2.50	**0.37**
1M-100% read	34.50	7.99	1.57	**0.35**
2M-25% read	283.94	127.15	2.35	**0.33**
2M-50% read	244.49	115.75	2.29	**0.36**
2M-75% read	144.32	97.54	2.32	**0.36**
2M-100% read	33.62	4.92	1.59	**0.35**

d) PetClinic Application

Benchmark	CMS	G1	Shenandoah	ZGC
PetClinic	124.20	**116.67**	129.23	120.68

show the results in a table). ZGC has a pause time of less than 1 millisecond in most of the applications and has the minimum pause time in comparison with other GCs. Although Shenandoah has on average a pause time about 6x more than ZGC, it performs significantly better than CMS and G1. Based on the values in Tables I.2.a, I.2.b , and I.2.c, CMS is not able to manage the pause time as well as other GCs. In addition, in both DaCapo benchmarks and B2, CMS has a pause time of about 400x more than ZGC; in Renaissance benchmarks, CMS has a pause time of around 100x more than ZGC on average. Also, Table I.2.d shows that G1 performed slightly better than the other three GCs regarding pause time in PetClinic.

I.5.2.3 Memory Usage

There are two main aspects to consider regarding memory usage: i) the size of the heap space occupied by objects before a garbage collection, and ii) the size of the heap space occupied by objects after a garbage collection has been done.

Average memory utilization before garbage collection As Figure I.3.a shows, in 15 out of 19 benchmarks of the Renaissance benchmark suite, CMS uses less memory than the other three GCs. For Akka Uct, ALS, Page Rank, and Reactors, G1 is the superior GC. G1, ZGC, and Shenandoah use respectively, on average, around 27%, 43%, and 46% more memory than CMS.

In the DaCapo benchmark suite, as shown in Figure I.3.b, although Shenandoah beats CMS in Avrora, Fop, and Luindex, CMS is the GC that manages heap memory before garbage collection the best for the rest of the benchmarks. Figure I.3.c reveals that in B2, G1 works better just for two cases (with 1M and 2M objects) with the lowest read percentage; otherwise, CMS is the preferred choice. G1, ZGC, and then Shenandoah use on average around 20%, 34%, and 46% more memory than CMS before garbage collection, respectively. Based on the Figure I.3.d, although there is a small difference between the

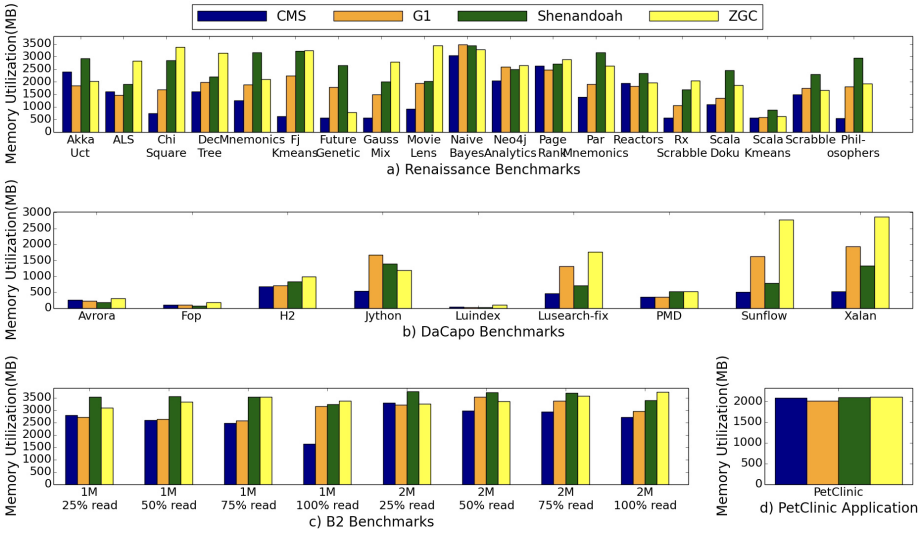


Figure I.3: Average memory usage before garbage collection.

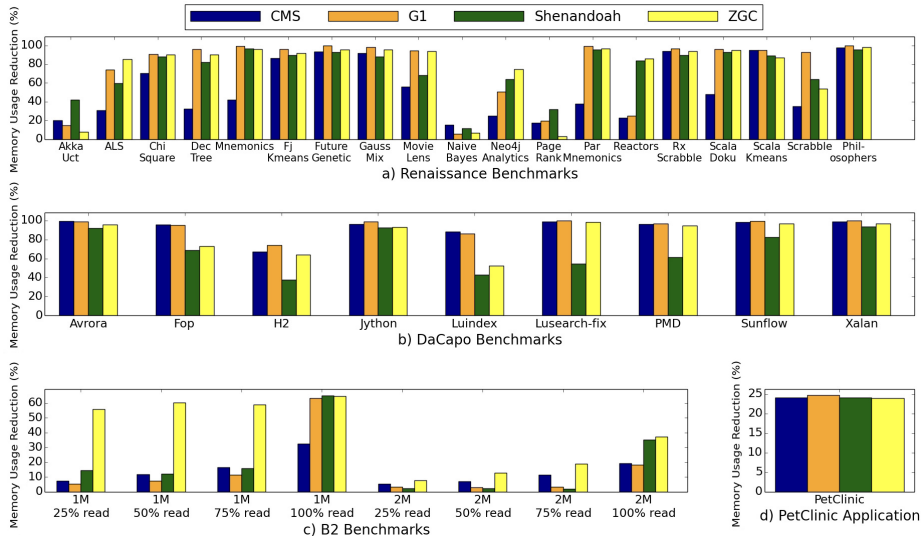


Figure I.4: Memory usage reduction (%) after GC.

memory usage in the four GCs, we can see that G1 uses the minimum amount of memory. After G1, CMS performs the best followed by Shenandoah and ZGC.

Average memory utilization after garbage collection As shown in Figure I.4.a, in the Renaissance benchmark suite, G1 can reduce the average memory consumption the best after garbage collection in most of the benchmarks. Moreover, ZGC and Shenandoah have a very similar average memory usage reduction percentage and much smaller when compared to CMS.

In DaCapo benchmarks (Figure I.4.b) the results of CMS and G1 are close (they differ in the average reduction of memory consumption by 1.1%). Although CMS has the best memory usage reduction in Avrora, Fop, and Luindex, for the rest of the benchmarks, G1 works better. Shenandoah reduces the memory usage by an average of about 35% less than G1; this value is 11% for ZGC.

In B2, memory reduced not more than 65% after garbage collection by the GCs (since most of the objects, held in a hashmap data structure, are still alive). However, ZGC is the preferred solution in almost 90% of experiments in B2. Finally, in the PetClinic application (Figure I.4.d), all the GCs reduced the memory usage by around 24%. However, G1 is the GC that performed better.

I.5.3 Discussion

We use the data of previous experiments regarding GC performance metrics to find the best GC solution for applications categorized as being CPU-intensive or I/O-intensive.

I.5.3.1 Throughput

In about 57% of CPU-intensive applications, CMS is the preferred GC regarding throughput. ZGC, G1, and Shenandoah have an acceptable throughput in about 20%, 13%, and 10% of the applications, respectively. For I/O-intensive applications, CMS has the best performance regarding throughput in almost 86% of applications; the remaining 14% are best served with ZGC.

I.5.3.2 Pause-time

In about 97% of CPU-intensive and in all I/O-intensive applications, ZGC achieves the minimum pause time.

I.5.3.3 Memory Usage

We evaluate memory usage before and after garbage collection. In 24 out of 30 CPU-intensive applications, CMS is the best GC, and for the I/O-intensive category, CMS and Shenandoah are equally selected as the leading GCs regarding average memory utilization before garbage collection.

Regarding the heap usage reduction after garbage collection, in the CPU-intensive category, G1 is the leading GC in both DaCapo and Renaissance benchmark suites. However, ZGC is the GC that reduces most the amount of heap usage in B2. Also, as shown in Figure I.4, for the PetClinic application, G1 reduced the heap utilization the best. According to the results, G1 is the best solution regarding average memory reduction for about 57% of applications in the CPU-intensive category. In the I/O-intensive category, G1 and CMS could reduce the heap usage by around 43% of benchmarks. The results match one of the main design principles in G1. G1 attempts to reclaim the heap spaces with the most garbage [Det+04].

Table I.3: Summary: The best GC solutions regarding performance metrics for CPU-intensive and I/O-intensive categories.

Category	Throughput	Pause Time	Heap Usage Before GC	Heap Usage Reduction (after GC)
CPU-intensive	CMS	ZGC	CMS	G1
I/O-intensive	CMS	ZGC	CMS / Shenandoah	G1 / CMS

I.5.3.4 Best GC

Table I.3 shows the best GC solution selected based on all evaluation results regarding each GC performance metric both for CPU-intensive and I/O-intensive applications.

I.6 Conclusion

Big data and Cloud services require a high amount of memory. A GC is responsible for allocating and releasing the memory used by such applications automatically. Although there is a default GC available in the Java run-time system, changing the default GC based on the application's requirements leads to better performance.

In this work, we perform a study on CMS, G1, Shenandoah, and ZGC GCs using: i) the Renaissance and DaCapo benchmark suites, ii) an application that we developed (called B2) that examines GCs cost with a focus on read and write barriers, and iv) a real-world Spring Boot-based application called PetClinic.

Having the results of GCs' behavior regarding the performance metrics we considered (throughput, pause time, and memory usage), while taking into account an application category (CPU-intensive or I/O-intensive), we indicate the best GC solution as well as a methodology for any user to maximize throughput, minimize pause time, or minimize memory usage in any application.

References

[Apa] Apache JMeter. URL: https://jmeter.apache.org/.

[BF18] Bruno, R. and Ferreira, P. "A study on garbage collection algorithms for big data environments". In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. "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*. 2006, pp. 169–190.

[BOF17] Bruno, R., Oliveira, L. P., and Ferreira, P. "NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 2017, pp. 2–13.

[CML14] Chen, M., Mao, S., and Liu, Y. "Big data: A survey". In: *Mobile networks and applications* vol. 19, no. 2 (2014), pp. 171–209.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. “Comparison of garbage collectors in Java programming language”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gog+15] Gog, I. et al. “Broom: Sweeping out garbage collection from big data systems”. In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[JHM11] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011.

[Jon96] Jones, R. E. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[Len+17] Lengauer, P. et al. “A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 3–14.

[Ngu+16] Nguyen, K. et al. “Yak: A high-performance big-data-friendly garbage collector”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation*. 2016, pp. 349–365.

[Ora15] Oracle. *Concurrent Mark Sweep (CMS) Collector*. 2015. URL: https://docs.oracle.com/en/java/javase/11/%20gctuning/concurrent-mark-sweep-cms-collector.html.

[Ora16] Oracle. *Java Garbage Collection Basics*. 2016. URL: https://www.oracle.com/webfolder/technetwork%20/tutorials/obe/java/gc01/index.html.

[Oss+02] Ossia, Y. et al. “A parallel, incremental and concurrent GC for servers”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[Pa19] Prokopec, A. and al., et. “On Evaluating the Renaissance Benchmarking Suite: Variety, Performance, and Complexity”. In: *arXiv preprint arXiv:1903.10267* (2019).

[PD00] Printezis, T. and Detlefs, D. “A generational mostly-concurrent garbage collector”. In: *Proceedings of the 2nd international symposium on Memory management*. 2000, pp. 143–154.

[Per18] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. “Analysis of garbage collection algorithms and memory management in Java”. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. “Stopless: A Real-Time Garbage Collector for Multiprocessors”. In: *Proceedings of the 6th International Symposium on Memory Management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. “A study of concurrent real-time garbage collectors”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sew+11] Sewe, A. et al. “Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine”. In: *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 657–676.

[Spr07] Spring PetClinic. *Spring PetClinic Project*. 2007. URL: https://spring-petclinic.github.io/.

[TIW11] Tene, G., Iyengar, B., and Wolf, M. “C4: The continuously concurrent compacting collector”. In: *Proceedings of the international symposium on Memory management*. 2011, pp. 79–88.

[UJ88] Ungar, D. and Jackson, F. “Tenuring policies for generation-based storage reclamation”. In: *ACM SIGPLAN Notices* vol. 23, no. 11 (1988), pp. 1–17.

[Xu+19] Xu, L. et al. “An Experimental Evaluation of Garbage Collectors on Big Data Applications”. In: *Proc. VLDB Endow.* vol. 12, no. 5 (2019), pp. 570–583.

[ZB20] Zhao, W. and Blackburn, S. M. “Deconstructing the garbage-first collector”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

Paper II

BestGC: An Automatic GC Selector

Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira

Published in *IEEE Access Journal*, vol. 11, pp. 72357-72373, 11 July 2023, doi: 10.1109/ACCESS.2023.3294398.

Abstract

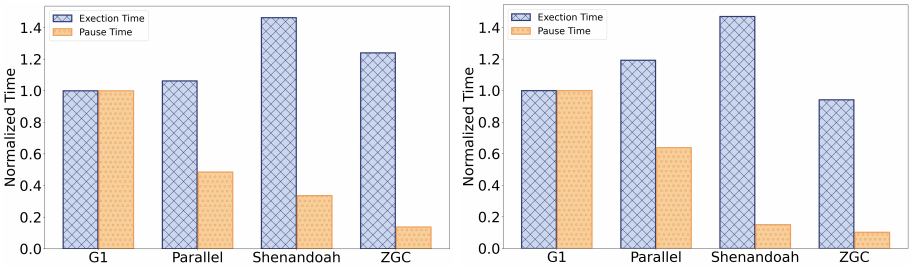
Garbage collection algorithms are widely used in programming languages like Java. However, selecting the most suitable garbage collection (GC) algorithm for an application is a complex task since they behave differently regarding crucial performance metrics such as garbage collection pause time, application throughput, and memory usage. This challenge is particularly more complicated as there is currently no available tool to assist users/developers in this critical decision-making process. In this paper, we address this pressing need by conducting an extensive evaluation of four widely used GCs (G1, Parallel, Shenandoah, and ZGC) in OpenJDK, considering application throughput, GC pause time, and various heap sizes. Building upon this evaluation, we present BestGC, a novel system that suggests the most suitable GC solution based on user-defined performance goals in terms of application throughput and GC pause time. Our evaluation of BestGC using multiple workloads demonstrates its effectiveness in suggesting the most suitable GC category (concurrent or generational/non-fully concurrent GC) in approximately 86% of the experiments on average. Additionally, BestGC accurately identifies the best GC in approximately 52% of the cases on average. Even in situations where BestGC failed to suggest the exact best GC or GC category, the suggested GC still outperforms the default GC (G1) in the JDK, exhibiting an average improvement of 1.75%. Notably, BestGC is designed to be easily extensible, facilitating its compatibility with other JDK versions, as well as new GCs and heap sizes. By addressing the lack of a practical tool to aid in GC selection, our research makes a significant contribution to the field of performance optimization in Java applications.

Contents

II.1 Introduction

A significant portion of today's applications use managed runtime languages like Java, and therefore, take advantage of automatic memory management, also commonly known as Garbage Collection. It is a crucial component in managed runtimes as it eliminates developers' effort to manually allocate and deallocate objects in memory, thus improving developer productivity.

Several Garbage Collectors (GCs) are available in the Java Virtual Machine (JVM), attempting to improve performance metrics like application throughput, GC pause time (the time application threads stop to let the GC execute), and memory footprint. For example, Figures II.1a and II.1b show the total application execution time and the 90^{th} percentile pause time (both normalized to G1) for the Philosopher workload (from the Renaissance [Pro+19] benchmark suite) with heap sizes of 256 MB and 4096 MB. Results show that GCs behave differently regarding application execution time and GC pause time. ZGC performs significantly better than other GCs regarding pause time in both heap sizes, while it sacrifices application execution time when the heap size is reduced to 256 MB. G1 outperformed other GCs considering application execution time with a 256 MB heap. However, ZGC has the best execution time with 4096 MB of heap. Note that, as in the rest of the paper, we use the total execution time of an application instead of the number of operations being done per second (throughput) since the concept of an operation is widely distinct for the workloads we use. In addition, for GC pause time, the usual 90^{th} percentile is used.



(a) Heap Size = 256 MB (b) Heap Size = 4096 MB

Figure II.1: Normalized application execution time and 90^{th} percentile of GC pause time for Philosopher workload (from Renaissance benchmark suite). Lower is better.

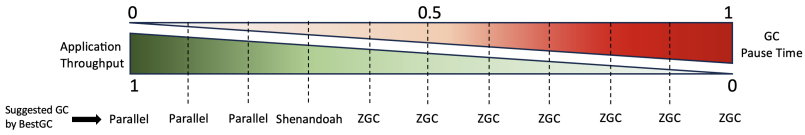


Figure II.2: Suggested GCs by BestGC for varying user preferences in terms of application throughput and GC pause time, for the Compress workload (included in SPECjvm2008 benchmark suite) and with a heap size of 512 MB. The suggested GC algorithm is Parallel when prioritizing application throughput, whereas the suggestion changes to ZGC when GC pause time becomes a more crucial factor.

Selecting the appropriate GC for an application is a challenging task due to the significant impact GCs can have on application performance goals, specifically GC pause time and application throughput. For applications handling large data sets or experiencing high transaction rates, the choice of GC algorithm becomes even more critical. However, determining the optimal GC to meet specific performance demands, such as minimizing GC pause time, poses a significant difficulty for users and developers who may not have expertise in GC optimization. Consequently, without a dedicated tool to aid in GC selection, developers and users are left to navigate this complex process with limited guidance, to choose a suitable GC solution with trial and error. Therefore, addressing the lack of a tool to assist in GC selection is important to alleviate the burden on users and developers and enable them to make well-informed decisions regarding the selection of the most suitable GC algorithm for their applications.

Thus, this work aims to propose a system, BestGC, that automatically suggests a GC that *best* suits the users' preferences regarding GC pause time and application throughput and uses the GC right away to run the user's application. The two performance metrics, application throughput and GC pause time, are widely used by users due to the desire to have more resources and time spent in the user's application and the maximum responsiveness possible. BestGC makes it possible for the users to simply apply their requirements to these performance metrics by defining a weight for each while running BestGC (Figure II.2 shows an example of suggested GCs by BestGC for different weights of application throughput and GC pause time). Given that memory footprint is another important performance metric to evaluate GCs, BestGC employs a strategy to include this metric in its GC suggestion process. Since we evaluate GCs from two generational and non-generational categories, and each of them follows different policies to pick the default maximum heap size, we use several heap sizes to investigate GCs' behavior. Providing GCs with fixed heap size prevents them from using their policy (which varies in GCs) to choose the maximum heap size and make the comparison between GC fare by working with the same memory budget.

We extensively evaluate four widely used GCs (G1, Parallel, Shenandoah, and

ZGC) in OpenJDK version 15. This evaluation considers application throughput (in fact, total application execution time), GC pause time, and diverse heap sizes. We do so while aiming at the following sub-goals:

- evaluating four well-known GCs (G1, Parallel, Shenandoah, and ZGC) in a production JVM implementation (OpenJDK) considering application execution time and GC pause time, with different amounts of memory available (256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, or 8192 MB);

- conducting the above mentioned experiments based on a set of workloads from standard benchmark suites (DaCapo [Bla+06] and Renaissance [Pro+19]) that represent existing real-world applications to reveal the costs of the GCs; and

- proposing an approach to suggest a suitable GC for a user's application while considering how the user cares about the GC pause time and application throughput.

Regarding the requirements of the final system, BestGC, the most fundamental is the impact that the system has on the given applications. This should be minimized as much as possible. In particular, the time it takes for BestGC to run should be short when compared to the time interval taken to run a user application. In addition, BestGC should have some flexibility regarding this aspect. Also, BestGC should be able to take into account the kind of performance the user is interested in optimizing regarding her/his application.

We evaluate G1 [Det+04], Parallel [Ora20c], Shenandoah [Flo+16], and ZGC [Per18], the most relevant GC implementations available in OpenJDK 15. We use workloads from two widely-used benchmark suites, DaCapo [Bla+06] and Renaissance [Pro+19]. These two benchmarks include diverse workloads that mostly need CPU (CPU-intensive), which is also a critical resource for GC. Using a forthcoming release of DaCapo, called DaCapo Chopin,[1] we have several latency-sensitive workloads. These workloads enqueue requests if they can not be processed immediately; this causes requests to be served after some time. This latency can be affected by a GC and results in longer application execution times (i.e., less throughput). Therefore, using these workloads, we can consider the concurrency overhead of the GCs on our studied performance metrics [Cai+22].

We measure application execution time (as a metric to report application throughput) and GC pause time, due to their undeniable importance, while providing different heap sizes for the GCs. Modifying the heap size allows us to investigate GCs' behavior when they have to scan a large heap area to find used objects; or when they are trying to manage a small heap to deliver high throughput (minimum application execution time) while imposing GC low pause times. Thus, the contributions of this work are as follows:

[1] https://github.com/dacapobench/dacapobench/tree/dev-chopin

- extensive measurement of four GCs (G1, Parallel, Shenandoah, and ZGC) using two widely accepted benchmark suites (DaCapo and Renaissance);

- developing a system, BestGC (by using a set of matrices we created from the results obtained from our extensive measurements done in the previous step) that frees the user from the complicated process of selecting a GC that fits a user's application performance goals; and

- validation of BestGC with the workloads from another widely used benchmark (which is SPECjvm2008 [Sta08]).

Due to the importance of the GCs, extensive research analyzed, characterized, and proposed new GCs to either improve GCs' capabilities or fix current GC issues [Fra+07; Oss+02; Piz+07; PPS08]. Several studies also compared GC solutions regarding performance metrics like application throughput, GC pause time, and memory usage [Ber+22; GMR18; JHM16; PGM19; Tav20; ZB20; ZBM22]. These performance metrics are the most common, and we also consider them the most critical metrics that highly affect users' applications. However, we believe that there is a lack of a system that suggests the best GC, among those existing powerful and in-production GCs, that matches a user's application performance goals. We believe BestGC is the first system that allows the user/developer to overcome the challenges of selecting a suitable GC that meets the application's performance requirements. BestGC is extendable to work with new JDK versions and new GCs. Besides, the methodology used in BestGC could be used in other runtimes in which there are GCs available to manage the heap.

In this document, we put our focus on the user side, assuming she/he has not enough knowledge to choose a GC that fits the user application's requirements. We selected two generational (non-fully concurrent) and two (mostly) concurrent GCs that have different main goals: i) providing maximum throughput, i.e., minimum application execution time (Parallel), ii) balancing the GC pause time and application throughput (G1), and iii) keeping the GC pause times to a minimum (Shenandoah and ZGC). We evaluated these GCs regarding application throughput/execution time and GC pause time. We added a third important performance metric in our evaluations, memory usage, by changing the available heap for the GCs; this way, we investigated its impact on GCs. Then, we build up BestGC based on the results conducted in our evaluations that suggests the most proper GC for a user's application.

The rest of this paper is organized as follows. The next section presents some background important to understand the GCs studied. Then, in Section II.3, we present some related work. In Section II.4 and Section II.5 we describe the BestGC architecture and implementation, respectively. Finally, we present evaluation results in Section III.5, conclude the paper in Section II.7, and present some directions for the future in Section II.8.

II.2 Background

This section briefly presents GCs' key terminology and algorithms (for more detailed background, see Jones and Lins's book [JHM16]). Then, we give an overview of the GCs evaluated in this paper.

II.2.1 Garbage Collection Algorithms

The Java HotSpot VM [20] provides different GCs to satisfy various performance requirements of Java applications. These GCs provide dynamic memory management that facilitates allocating memory to objects, managing the heap, and reclaiming unused memory for future use. There are two types of garbage collection algorithms: reference counting [Col60], and reference tracing [Ora15].

A reference counting (RC) algorithm counts the number of references (e.g., pointers) to an object, keeps that number updated, and removes the objects when their counter falls to zero (providing immediate memory reuse). However, it comes with the cost of counters' space and updating overhead, in addition to skipping cyclic structures with destroyed references. Tracing collectors identify objects in use by the running applications (also called live objects) by tracing the object graph, starting from a set of roots (registers, stacks, and global variables), and moving live objects to another space or sweeping them [BCM04]. Although it eliminates the additional field in RC, it requires tracing the whole object graph and needs mechanisms to deal with application threads' (mutator) activities. GCs we evaluate in this work follow the reference tracing approach.

To optimize the collection, GCs divide the heap into generations or regions. Generational GCs divide the heap into two age groups. Newly created objects are placed in the young generation, and those that survive multiple collection cycles are moved into the old generation. As the generational hypothesis [LH83] states, most objects die (i.e., become unreferenced) after a short period of time. Therefore, the young generation often fills up with dead objects (objects that are not being referenced by the mutator threads); consequently, minor collections occur more frequently and move live objects to the old generation (and implicitly remove unused objects). A major collection runs in the old generation if it fills up. Conversely, some GCs maintain the heap as a single generation. However, GCs may use mechanisms to divide the heap into regions to make the collection and improve their performance goals (for example, to evacuate the regions with the most unused objects first [Det+04]).

To monitor the object graph, keep track of references in generations, and provide concurrency in the collectors, GCs employ read and write barriers [ZBM22]. A read barrier (also known as a load barrier) is executed when loading an object reference from the heap [PGM19]. In contrast, a write barrier's code snippet is invoked before any write operations in the heap [PGM19]. Stop-the-world (STW) collectors pause mutator threads as long as the garbage collection runs, then use write barriers to update pointers to the evacuated objects. However, concurrent GCs do not pause the mutator as STW GCs. Concurrent GCs use write barriers to mark live objects and also may employ

read barriers to do evacuation and reclamation simultaneously with the running mutator threads. These barriers impose performance costs on the GCs [Yan+12].

II.2.2 Evaluated GCs

In this work, as already mentioned, we evaluate four production GCs in OpenJDK 15:[2] G1 [Det+04], Parallel [Ora20b], Shenandoah [Flo+16], and ZGC [Per18]. In this section, we highlight the GCs' key features that provide the background to understand the results obtained in the evaluation (see Section III.5) that are used in BestGC (see Sections II.4 and II.5).

Parallel (also known as throughput collector [Ora20b]) is a STW generational collector. It performs garbage collection in its generations using multiple threads in parallel. So, it is well suited for applications that can tolerate long application pauses (as it is not a concurrent GC).

Garbage First (G1) has become the default collector of OpenJDK since version 9. It tries to keep a balance between pause time and throughput [Ora20b]. G1 divides the heap into fixed-size regions, and, as its name implies, it targets the regions containing the most existing garbage (dead objects) to start the collection from them. It is a generational GC that uses a concurrent tracing mechanism to mark live objects, yet it performs a STW collection to evacuate objects. It also uses write barriers to make sure that all the live objects remain alive during the concurrent tracing phase.

Moving from STW and generational to mostly-concurrent collectors, GC pause times are significantly reduced. Concurrent collectors such as Shenandoah and ZGC, two of the mostly-used production concurrent collectors, keep the pause time to a minimum regardless of the heap size. Shenandoah GC (available from JDK version 12) is a single-generation (a generational version is available as an experimental GC) and region-based collector that does both the tracing and evacuation concurrently.

Z Garbage Collector, ZGC (available from JDK version 15), tries to keep the GC pause times below 10 ms. It is a single-generation collector (a generational version is under development) that divides the heap into regions of different sizes. It uses part of an object's reference (called colored bits) to keep marking and relocation-related information [PGM19]. A read barrier checks on the colored bits once a reference is loaded to take action for the object.

II.3 Related Work

The importance of GCs in programming languages such as Java led to extensive studies on GC performance. Many studies compared existing GC algorithms, evaluated their performance, combined the features of existing GCs, or introduced new ones. In this section, we go over some of the studies tightly related to our work.

[2] https://openjdk.org/projects/jdk/15

II.3.1 Novel GC Strategies

Many studies introduce new GCs created over existing collection algorithms. Ossia et al. [Oss+02] designed a parallel, incremental (which performs the collection in steps), and mostly concurrent GC to meet the low GC pause time goal. The collector is suitable for shared-memory and multiprocessor servers and supports highly multi-threaded applications. Pizlo et al. [Piz+07] propose STOPLESS, which uses a tracing collector and a compactor to control fragmentation to support multi-threaded applications. Pizlo et al. [PPS08] also propose two other solutions for concurrent real-time GCs, CLOVER, and CHICKEN, to reduce the complexity of STOPLESS. Frampton et al. [Fra+07] designed an incremental, young-generation STW GC for real-time systems. The collector uses both read and write barriers to track remembered set changes.

Tene et al. [TIW11] propose C4, the Continuously Concurrent Compacting Collector. It supports concurrent compaction, and incremental update tracing through the use of a read barrier. C4 enables simultaneous-generational concurrency that allows different generations to be collected concurrently. It continuously performs concurrent young generation collections, even during long periods of concurrent full heap collection, maintaining high allocation rates and efficiency without sacrificing response times. Wu et al. [Wu+20] introduces Platinum, a novel concurrent GC designed to reduce latency in interactive services. Platinum creates an isolated execution environment for concurrent mutators, improving application latency without restricting GC thread execution. Additionally, Platinum utilizes a new hardware feature to make access control to different regions of memory more efficient and therefore minimize software overhead present in previous concurrent collectors.

However, the above mentioned GCs are not very popular because they are not available in most mainstream JVMs.

Bruno et al. [BF18] described how objects created by Big Data applications are kept in memory and surveyed the existing GCs for big data environments. They also addressed scalability issues in classic garbage collection algorithms and analyzed several relevant systems that try to solve these scalability issues. Xu et al. [Xu+19] used the Apache Spark [Fou18] application to analyze GCs like G1 and Parallel. They proposed strategies for designing GCs specified for Big Data. Nguyen et al. [Ngu+16] also propose a GC with low pause time and application high throughput based on the logical distinction between the data path and the control path. The data path consists of data manipulation functions, while the control path is responsible for cluster management, setting communication channels between nodes, and interacting with users. They believe these paths differ in heap usage and object creation patterns. So, based on the previously mentioned paths, they divide the heap into data and control spaces to reduce the objects managed by the GC. Not many objects are created in the control spaces, and they are subject to generation-based collection, while the objects in the data space, which creates the most objects, are subject to region-based collections. However, the developer is responsible for marking the beginning and end points of the data path in the program. Broom [Gog+15] and NG2C [BOF17] also

propose two GCs for Big Data systems. However, in Broom, the programmer has to create the regions in the heap explicitly; in NG2C, the programmer identifies the generation in which a new object should be allocated. These GCs require developer efforts and are prone to errors. Then, the authors of NG2C proposed POLM2 [BF17], an offline profiler that automatically infers generations for object allocation, and finally, ROLP [Bru+19], an online profiler to select an object generation with no user intervention.

II.3.2 GC Comparative Analysis

Zhao et al. [ZBM22] introduce LXR to deliver low GC pause times and high application throughput. Their approach includes STW collections to increase responsiveness and an RC mechanism to deliver scalability and promptness. They evaluated and compared barrier overhead and pause time for GCs like G1, Shenandoah, and ZGC that we also used in this paper; yet, note that RC-based GCs are not widely used in production and, therefore, they are out of the scope of this document. Zhao et al. [ZB20] decomposed G1 into several key components to evaluate the impact of different algorithmic elements of G1 on performance. Pufek et al. [PGM19] analyzed several garbage collectors like G1, Parallel, Serial (a GC that uses the same thread as the mutator [Ora20a]), and CMS [PD00] (a generational GC with a concurrent tracing in the old generation).[3] They used the DaCapo benchmark suite [Bla+06] to evaluate the GCs. They compare the number of algorithm iterations and the total GC pause time for applications running on top of JDK 8 and JDK 11. Also, they compared ZGC and Shenandoah, which were experimental GCs by then, with G1. They concluded that G1 and Parallel operated better than Serial and CMS regarding the overall duration of collections. They also showed that those experimental GCs would contribute to overall system optimizations. Their results also emphasize the importance of evaluating ZGC and Shenandoah, as we do in this paper.

Grgic et al. [GMR18] analyzed Serial, G1, Parallel, and CMS using the DaCapo benchmark suite in JDK version 9. They concluded that G1 is a better choice regarding the total number of garbage collections and CPU utilization for multi-threaded environments (but not for single-threaded environments).

Lengauer et al. [Len+17] describes commonly used benchmarks, including DaCapo, DaCapo Scala [Sew+11], and SPECjvm2008 [Sta08] in terms of memory and garbage collection behavior. They compared G1 and Parallel Old (parallel collection in the old generation) regarding the number of full collections (GC counts), the GC time relative to the total execution time of the application, and GC pause time. They concluded that G1 performs better than the Parallel Old concerning GC pause time by selecting different regions to collect.

Beronić et al. [Ber+22] investigated and compared memory issues, heap allocation, CPU usage, and duration of collection in three GCs: G1, ZGC, and Shenandoah. They found that GCs are sensitive to the heap size and that G1 uses less heap than the two other GCs. However, G1 is more CPU intensive

[3] CMS is deprecated in JDK version 15, so we do not evaluate it in our study.

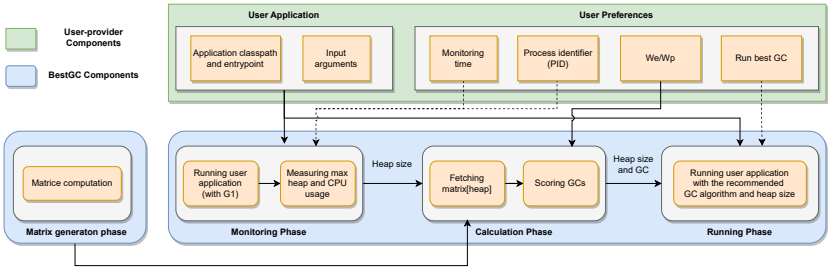


Figure II.3: Overall architecture of BestGC. Solid arrows for mandatory inputs and dotted arrows for optional inputs.

since it occupies more OS threads necessary for scanning the live objects in a heap.

Cai et al. [Cai+22] introduced a new methodology that defines a practical lower bound on the costs of GCs for any given cost metric. They showed that GCs are sensitive to heap size and indicated that low GC pauses achieved by concurrent GCs do not translate into low application latency. To achieve the conclusions, the authors used latency-sensitive workloads from the DaCapo Chopin benchmark suite. These latency-sensitive workloads serve requests arriving at a determined rate, and if they are not able to process a request instantly, they enqueue the request. Therefore, they used a metric, called metered latency, to show the delay both for the executing and the enqueued requests. Using this metric (metered latency), the authors show that the GCs, especially concurrent ones, cause delays for both executing and enqueued requests and therefore affect the latency. In this document, we do not measure such metric, but we include latency-sensitive workloads in our evaluations; in fact, we do so because it impacts the application's total execution time, the metric we use to report the throughput.

Differently from previous works, in this study, we evaluate four widely used G1, Parallel, Shenandoah, and ZGC for application throughput/execution time and GC pause time using a broad collection of workloads from DaCapo-Chopin (in the rest of the paper, when we refer to DaCapo-Chopin, we will use the simpler term DaCapo) and Renaissance benchmark suites (more details in Sections II.4 and II.5). These workloads simulate real-world applications to show GCs' overhead on the performance metrics mentioned above (i.e., application throughput/execution time and GC pause time). We use the results from such experiments in BestGC. BestGC is a system that runs any Java application with the best GC, from the four above-mentioned GCs, and a recommended heap size while considering the user's preferences regarding application throughput and GC pause time. To the best of our knowledge, BestGC is the first system to automate GC selection, while it would be easily extensible to work with a new JDK version, new heap sizes, and new GCs. Furthermore, although we developed BestGC for the JVM, its algorithm is totally agnostic (regarding the JVM), as it could be used for other runtimes and languages.

II.4 BestGC

This section addresses the architecture of BestGC, which aims to suggest the most suitable GC for a user's application. As Figure II.3 illustrates, there are four phases to consider: Matrices Generation, Monitoring, Calculation, and Running. First, in the matrices generation phase, matrices are created based on the extensive measurement of the four GCs under consideration (this is done only once, i.e., when building BestGC). These matrices will be used in the calculation phase (see Section II.4.4). When using BestGC, a user needs to pass a set of input arguments to BestGC and run it (see Section II.4.2). The monitoring phase (see Section II.4.3) consists of running BestGC to find an application's maximum heap size and CPU usage. In the calculation phase (see Section II.4.4), GCs are scored based on the matrices generated in the first phase. The running phase (see Section II.4.5) corresponds to the running of the Java application with the suggested GC available immediately after the monitoring and calculation phases. Thus, regarding the four phases mentioned above, the first one runs only once; the others run whenever BestGC is executed.

II.4.1 Matrices Generation Phase

A set of matrices for different heap sizes is included in BestGC. They hold the manually evaluated application execution time and pause time for four GCs (G1, Parallel, Shenandoah, and ZGC). Such evaluated results are obtained by measuring the average application execution time and GC pause time (with different heap sizes) of the workloads in DaCapo [Bla+06] and Renaissance [Pro+19] in OpenJDK version 15 (more details in Section III.5). These matrices are generated only once and inserted as constants in BestGC. BestGC fetches the matrix corresponding to the measured heap size to score GCs, suggests the best of the GCs (see Section II.4.4), and runs the user/developer application with it.

Note that the workloads from the DaCapo and Renaissance benchmark suites report application execution time for a fixed input size, i.e., a fixed amount of work done by the workloads. So, we do not report the throughput with its common definition (number of requests per time unit), yet we report application execution time as a metric for throughput.

In short, we extensively evaluate G1, Parallel, Shenandoah, and ZGC, regarding the application execution time and GC pause time for distinct heap configurations (256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, and 8192 MB). Based on our evaluations, the heap sizes set includes those that are big enough to give headroom for the application to work with no limitation. Also, small heap sizes put GCs under pressure to provide the heap needed by the application.

Figure II.4 shows such a matrix (as an example). As can be seen, the matrix shows the application execution time and GC pause time when the heap size is 8192 MB normalized to G1. For each evaluated GC and heap size, the first column contains the average application execution time of all the workloads; the

Table II.1: Available switches for BestGC.

Switch	Value	Optional	Comments
user-app	Absolute path to user's application jar file + input arguments.	No	A space-separated string.
we	[0,1]	Yes*	Weight for application execution time (throughput) (*If w_p is set, w_e is optional).
wp	[0,1]	Yes*	Weight for GC pause time (*If w_e is set, w_p is optional).
monitoring-time	Time in seconds; default value is 30 seconds.	Yes	Time interval to monitor the user's application.
pid	Process ID.	Yes	PID of the running user's application.
run-best-gc	Boolean (true / false); default is true.	Yes	Autorun of the user's application with the recommended GC.

second column contains the average 90^{th} percentile of the GC pause times (values are normalized to G1) also for all workloads. This is detailed in Section II.4.4.

$$\begin{array}{c} \\ G1 \\ Parallel \\ Shenandoah \\ ZGC \end{array} \begin{array}{c} \begin{array}{cc} ExecutionTime & PauseTime \end{array} \\ \begin{bmatrix} 1.0 & 1.0 \\ 0.954 & 1.448 \\ 1.093 & 0.144 \\ 1.098 & 0.044 \end{bmatrix} \end{array}$$

Figure II.4: Example of a matrix for the heap size of 8192 MB.

II.4.2 Input Options for BestGC

BestGC requires a few simple inputs to run.[4] The user (who is defined as she/he who wants to run a Java application with the best performance possible) needs to pass two inputs (see Table II.1): first, the mandatory absolute path to the application's jar file, plus all the application-specific input arguments (if the user's application is already running, the user must pass its process id to BestGC using the *pid* command option); second, a mandatory weight (between 0 and 1)

[4]A simple command to run BestGC with, in this case, 40 seconds of *monitoring-time*: java -jar BestGC.jar --user-app="*path to the user's application's jar file + its input options*" --monitoring-time=40 --wp="*weight for pause time*"

Algorithm 1 Calculation of maximum heap usage.

```
1: t ← 0
2: pid ← user_app_pid
3: while t ≤ monitoringTime do
4:     heap ← 0
5:     jstat ← jstat(pid)
6:     heap ← jstat.s1u + jstat.eu + jstat.ou + jstat.ccsu
7:     heap_usage_list.add(heap)
8:     t++
9: end while
10: max_heap ← heap_usage_list.max()
```

for application execution time (w_e) and/or GC pause time (w_p) to show how the user cares for these two performance metrics (1 for the highest importance) in such a way that $w_p + w_e = 1$. Users have to specify at least one of these weights to run BestGC.

There are also several optional switches to set different parameters in BestGC (as shown in Table II.1). For example, when using *monitoring-time*, the user determines the monitoring time interval during which BestGC captures heap and CPU usage of the user's application (more details in the upcoming section). BestGC runs the user's application with the suggested GC in its final phase (see Section II.4.5); however, the user can deactivate the auto-execution feature by setting *run-best-gc* to *false*.

II.4.3 Monitoring Phase

During this phase, BestGC runs a user's Java application with the default GC (G1) of the available default JDK installed on the user's machine. Also, both the application's maximum heap memory and CPU usage are obtained by BestGC. BestGC measures heap and CPU usage during a time interval called *monitoring-time*. As previously mentioned, this time interval, by default, is set to 30 seconds in BestGC. Based on our evaluations, the 30 seconds *monitoring-time* is long enough to measure the heap and CPU usage. However, the user can change it by specifying the desired value; for example, a longer *monitoring-time* is recommended if there is an initialization phase in the application. Also, if the user's application execution time is really short, obviously, 30 seconds should be reduced as needed. In other words, using BestGC may not be practical for an application with a short execution time that runs only once.

Based on the recorded maximum heap usage in this phase, BestGC offers a reasonable heap configuration for future executions of the user's application. As already mentioned, such maximum heap values can be 256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, or 8192 MB. We now describe how a Java application's heap usage is obtained by BestGC. The pseudocode in Algorithm 1 shows how BestGC measures the heap usage of a Java application. To capture the maximum

Algorithm 2 Calculating the number of total engaged cores.

```
1: t ← 0
2: pid ← user_app_pid
3: engaged_cores ← 0
4: while t ≤ monitoringTime do
5:    cpu_by_core_list.add(read(/proc/stat))
6:    for all core_usage in cpu_by_core_list do
7:       new_core_cpu_usage ← core_usage.all_usage_values
8:       new_core_cpu_idle ← core_usage.idle_value
9:       delta_core_cpu_usage ←
10:                new_core_cpu_usage − last_core_cpu_usage
11:      delta_core_cpu_idle ←
12:                last_core_cpu_idle − last_core_cpu_idle
13:      core_cpu_usage ←
14:                100 × (delta_core_cpu_usage−
15:                delta_core_cpu_idle)/delta_core_cpu_usage
16:      if core_cpu_usage ≥ 50 then
17:         engaged_cores_per_second + +
18:      end if
19:   end for
20:   engaged_core_list.add(engaged_cores)
21:   t + +
22: end while
23: engaged_cores ← engaged_core_list.average()
```

heap usage, every second, BestGC invokes the *jstat*[5] command with the *gc* option for the user's application using its Process ID (line 5). This option displays statistics about the heap behavior. BestGC considers all the metrics that report the capacity of the different parts of the whole heap (line 6). Considering G1, which BestGC uses to run a user's application in this phase, these metrics are: *S1U* (survivor space 1 utilization), *EU* (Eden space utilization), *OU* (old space utilization), and *CCSC* (compressed class space used). Finally, the maximum heap usage is detected by BestGC at the end of the *monitoring-time* (line 10).

BestGC also reports if a user's application is CPU-intensive or not. BestGC obtains a Java application's CPU usage through two steps; first, it calculates the number of engaged cores while running the Java application (Algorithm 2), and then the average CPU usage per engaged core (Algorithm 3). When running a CPU-intensive application, the choice of GC is even more important (than running a non-CPU-intensive application) given that the CPU is a shared resource between the application and the GC. This feature is also included in BestGC for adding future GC selection factors in the system. For best results, BestGC should run alone on the user's machine to avoid interference from other applications that may impact CPU measurements during the monitoring phase.

[5]https://docs.oracle.com/en/java/javase/15/docs/specs/man/jstat.html

Algorithm 3 Calculating average CPU usage per engaged core.

```
1: t ← 0
2: pid ← user_app_pid
3: while t ≤ monitoringTime do
4:     cpu_usage ← top(pid)
5:     cpu_usage_list.add(cpu_usage)
6:     t++
7: end while
8: average_cpu ← cpu_usage_list.average()
9: average_cpu_per_core ← average_cpu/engaged_cores
```

BestGC captures the amount of CPU the user's application uses every second during the *monitoring-time* interval (Algorithm 2, line 4). Moreover, since not all the applications utilize all the CPU cores in the machine, there is a function (pseudocode presented in Algorithm 2) to calculate the number of engaged CPU cores for the user's application. The *proc/stat*[6] command reports different metrics for each CPU core in Linux. BestGC sums all consumed values (line 6) and the CPU idle time (line 7) for each CPU core. Then, to calculate the core CPU usage at the current second relative to the last second, BestGC calculates the difference between the new and last CPU usage (also the CPU idle time) for the selected core. Finally, it computes the CPU usage for the core at the current second (lines 8-10). Should the value be above 50% for a core, BestGC increases the number of engaged CPU cores per second by one. Then, the average number of engaged cores is calculated at the end of the *monitoring-time*. Also, the average total CPU utilization (Algorithm 3) is computed by averaging the recorded total CPU usage every second using the *top*[7] command and dividing it by the average number of engaged CPU cores. Should the results be over 90%,[8] the application is CPU-intensive; otherwise, it is considered non-CPU-intensive (Algorithm 3, lines 4-8).

Thus, recording the CPU usage for the engaged CPU cores reports the CPU consumption intensity of the user application, which may affect the GC performance (see Section II.6.3 for more details). Note that in the monitoring phase, the length of which is specified by *monitoring-time*, BestGC runs the user's application with the user inputs using the default GC of the default JDK available on his/her machine while it sets no other limitations. Then, in the calculation phase (see Section II.4.4), BestGC uses the weights (w_e or w_p), provided by the user, in addition to the pre-calculated application execution time and GC pause time results obtained from the various evaluations, to score G1, Parallel GC, Shenandoah, and ZGC.

[6]https://manpages.ubuntu.com/manpages/xenial/man5/proc.5.html

[7]https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html

[8]We used the considerations in the Linux command *atop*, which assumes the CPU usage to be critical when the total CPU usage percentage is 90% and above; https://manpages.ubuntu.com/manpages/bionic/man1/atop.1.html

II.4.4 Calculation Phase

BestGC uses the matrices previously created (described in Section II.4.1) to score each GC for the user application. The GC with the lowest score is the winner, i.e., it will be the GC solution that will be used to run the user's application in the running phase of BestGC. For example, the matrix shown in Figure II.4 illustrates the application execution time and GC pause time results when the heap size is 8192 MB.

BestGC selects the suitable matrix among the matrices available for all six heap sizes depending on the maximum heap size used by the user's application (see Section II.4.3). BestGC considers 20% headroom for the application (i.e., $max\_heap \times 1.2$) and picks the closest bigger heap configuration's matrix available. Due to our evaluation, 20% additional heap space is big enough to let GCs manage the heap. For example, during a 30-second *monitoring-time*, BestGC reports that the maximum heap memory used by *Tomcat* is 308.1 MB. It simply computes the required heap, $308.1 \times 1.2 = 369.72$, and looks for the closest next heap size in its heap sizes set, which is 512 MB. Then, it uses matrix_512 for further calculations.

Furthermore, BestGC needs to know the user's performance goals regarding the application's throughput and the GC pause time. Accordingly, as stated in Section II.4.2, the user needs to pass a weight for the application throughput (w_e) or GC pause time (w_p) to execute BestGC ($[w_p + w_e = 1]$ with $w_p \in [0,1], w_e \in [0,1]$). Having w_p (or w_e), and the most appropriate matrix (for the detected maximum used heap size), BestGC calculates a formula (Equation II.1) to score each GC:

$$\begin{aligned} score_{gc} =& w_e \times matrix[heap]_{<gc,ExecutionTime>} \\ &+ w_p \times matrix[heap]_{<gc,PauseTime>} \\ & \\ & gc \in \{G1, Parallel, Shenandoah, ZGC\} \\ & heap \in \{256, 512, 1024, 2048, 4096, 8192\} \end{aligned} \tag{II.1}$$

Consequently, using the formula above, BestGC scores the GCs. Finally, it selects the minimum-scored GC along with the recommended heap size and passes it to the last phase.

II.4.5 Running Phase

In this phase, BestGC utilizes the maximum heap size and the most suitable GC suggested, both from the previous phase. It executes the user's application with the following command:

```
java -Xmx <max_heap>*1.2m -XX:+Use_<best_gc> -jar
    path_to_run_user_application
```

However, if the user has previously deactivated the auto-execution feature (*run-best-gc=false*), BestGC will simply print out the command above.

II.5 Implementation

This section describes the implementation steps we took to develop BestGC. The system is available for download and testing at https://github.com/SaTaSo/BestGC-Software.

II.5.1 BestGC

We implemented BestGC using Java and OpenJDK version 15 on a machine running GNU/Linux, Ubuntu 16.04.4 LTS, with a x86_64 Intel(R) Xeon(R) 4-core CPU E5506 @2.13GHz. Running BestGC is quite straightforward since it only asks the user to pass a few simple inputs (input options are available in Table II.1). In the monitoring phase, BestGC looks for the JAVA_HOME environment variable on the local machine and executes the given application on a new JVM instance while setting no heap limitations. The user should not run any other application on the machine to let the application run in isolation for best results, as interference might impact CPU and memory utilization tracking.

JDK provides tools like *jcmd*[9] and *jstat* to measure the heap usage. Using *jcmd* to monitor heap usage requires enabling the Native Memory Tracking feature (`-XX:NativeMemoryTracking=[summary | detail]`) when starting the JVM. However, Native Memory Tracking will cause a 5-10% performance overhead to JVM [Ora01] which is not the case with *jstat*. Therefore, we decided to use *jstat* in BestGC. *Jstat* provides different output options to display various statistics. We use the *gc* option, which represents the behavior of the garbage-collected heap. Computing the utilized capacities reported for separated parts of the heap reveals the heap consumption at the moment in which *jstat* is invoked. Since BestGC sets no JVM options while executing the user's application, it employs the JVM default GC (G1 since JDK version 9). Therefore, *jstat* reports heap consumption, including both young and old generations statistics. Recording the total heap usage every second lets BestGC find the adequate heap size to decide on the most suitable GC for the application.

II.5.2 Heap Size Variety

GCs are sensitive to the heap size [Bla+06]. The way GCs manage the heap is one of their critical performance features. Large heap memory size shows GCs' behavior when there exists much memory to assign the new objects and GCs have to manage a large memory area. However, small heap memory causes frequent garbage collections to free more memory; this has obvious impacts on the GCs' pause time and may lead to GCs' failure in the worst case. This research also involved heap size in GC evaluations to monitor its impact on the other performance metrics, such as application execution time and GC pause time. As already mentioned, we set the heap size in our evaluations to 256, 512, 1024, 2048, 4096, and 8192 MB. First, heap sizes in powers of two are commonly

[9] https://docs.oracle.com/en/java/javase/15/docs/specs/man/jcmd.html

used by the users [Eva20], also, these include the required heap sizes to run all workloads in our evaluations. If the maximum heap usage by the application is higher than 8192 MB, BestGC runs the application with the actual maximum heap usage (plus extra headroom for running); however, it chooses the GC based on the GC scores for 8192 MB heap size (conversely, the same process applies for heap usage bellow 256 MB).

II.6 Evaluation

First, we describe some details regarding the benchmark suites we use: DaCapo and Renaissance. Then, we present the application execution time and GC pause time results (per each heap size) for the selected workloads. As already mentioned, these benchmark workloads are representative of real-world applications. Finally, we validate BestGC using workloads from a third benchmark suite, SPECjvm2008 [Sta08]; BestGC runs the workloads from SPECjvm2008 as any other application a user may run. All the results were obtained on a server running GNU/Linux, Ubuntu 16.04.4 LTS, with eight *Intel(R) Xeon(R) E5506 @ 2.13GHz* CPUs and *16 GB* of RAM.

II.6.1 Benchmarks

For both benchmark suites DaCapo and Renaissance, we varied the number of iterations for each workload to achieve stable execution times then we used the reported results for the last iteration.

We used the latest version of the DaCapo benchmark suite, the Chopin branch.[10] DaCapo is a widely used benchmark suite composed of Java CPU-intensive along with, in the Chopin version, latency-sensitive workloads. Since there are different sizes available for the workloads' inputs in DaCapo, we use *large* if available (to increase the number of live objects in the heap and make GCs work frequently); otherwise, we set the input size (using switch `-s`) to *default* (for Zxing, Fop, and Luindex workloads). Using the switch `-no-pre-iteration-gc`, we disable explicit GC calls in the workload's code.

We also use the Renaissance benchmark suite (version gpl-0.11.0). It includes several Java-based workloads representing a large collection of existing applications such as big data, machine learning, and functional programming. As with DaCapo, we set the input size to *large*, if available, otherwise we set it to *default*. Also, disabling explicit GC calls (*System.gc()*) existing in the source code is done using the switch `-no-forced-gc` while running the workloads.

Table II.2 shows the workloads in DaCapo and Renaissance benchmark suites we used in this work. Some of the workloads in DaCapo and Renaissance benchmark suites fail due to incompatibility with JDK version 15 (e.g., Cassandra in DaCapo or Db Shootout in Renaissance) or other errors like database connection failures (in Tradesoap and Tradebeans workloads). Also, the strategy we employ to evaluate GCs provides GCs with the same fixed heap sizes and

[10] https://github.com/dacapobench/dacapobench/tree/dev-chopin

Table II.2: Workloads from DaCapo and Renaissance benchmark suites used in the experiments.

Benchmark Suite	Workloads
DaCapo	Avrora, Fop, Jme, Luindex, Lusearch, Tomcat, Xalan, Zxing
Renaissance	Dotty, Finagle-Chirper, Finagle-HTTP, Fj-Kmeans, Future-Genetic, Mnemonics, Par-Mnemonics, Philosophers, Rx-Scrabble, Scala-Doku, Scala-Kmeans, Scrabble

Table II.3: Number of garbage-collected workloads per GC with different heap configurations for: a) 24 workloads in DaCapo benchmark suite, and b) 16 workloads in Renaissance benchmark suite.

GC	a) DaCapo						b) Renaissance					
	Heap Size (MB)						Heap Size (MB)					
	256	**512**	**1024**	**2048**	**4096**	**8192**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
G1	14	20	22	24	24	24	10	13	16	16	16	16
Parallel	12	20	21	23	24	24	9	12	15	16	16	16
Shenandoah	14	20	22	24	24	24	10	12	15	16	16	16
ZGC	13	19	22	23	24	24	8	12	15	16	16	16

eliminates the dependence of the GCs on the available heap memory on the machine; also, it prevents GCs from freely choosing heap sizes by their different policies. Since they have to manage the same heap size, it makes their abilities comparable. This heap selection strategy in BestGC results in facing failures while running some workloads due to an *Out of Memory* error. Thus, we omit all the non-running workloads from the evaluations with all the heap sizes. This narrows our set of workloads down but allows us to make the GC comparison fair since they are working on the same workloads with the same heap availability.

Tables II.3.a and II.3.b show the number of workloads for which the GCs could do the garbage collection for the two benchmark suites (maximum is 24 for DaCapo and 16 for Renaissance). For workloads in the DaCapo benchmark suite (Table II.3.a), all four GCs manage the heap with 4096 and 8192 MB heap sizes. Parallel and ZGC fail to execute when the heap size is decreased to 2048 MB. As the table shows, by reducing the heap size, GCs start to fail, and with the smallest heap size (256 MB), Parallel could pass 12 out of 24 workloads, while G1 and Shenandoah outperformed the other two GCs and performed well in 14 workloads. With 2048, 4096, and 8192 MB of the heap, all the GCs perform well while running workloads from the Renaissance benchmark suite (Table II.3.b). Although Parallel still manages the heap for all the workloads, G1, Shenandoah, and ZGC fail in one workload with a 1024 MB heap size. GCs fail in more workloads when the heap size is set to 512 MB. ZGC is the GC with the worst performance with a 256 MB heap.

Table II.4: Average (the arithmetic mean is calculated for the set of all the workloads in each benchmark suite) of the application execution time for the GCs with different heap sizes. Values are normalized to G1. Lower is better. The line highlighted in green (light gray) is the one with the best results.

GC	a) DaCapo						b) Renaissance					
	Heap Size (MB)						Heap Size (MB)					
	256	**512**	**1024**	**2048**	**4096**	**8192**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Parallel	0.951	0.940	0.934	0.958	0.961	0.976	0.929	0.887	0.900	0.932	0.953	0.939
Shenandoah	1.035	1.009	1.001	1.043	1.018	1.032	1.692	1.364	1.254	1.143	1.178	1.133
ZGC	1.032	1.007	0.986	0.995	1.032	1.037	2.381	1.552	1.324	1.288	1.210	1.138

II.6.2 Application Execution Time and GC Pause Time

This section shows the results obtained regarding application execution time and GC pause time when running a workload from benchmarks DaCapo and Renaissance, with each one of the GCs (G1, Parallel, Shenandoah, and ZGC) for various heap sizes (256, 512, 1024, 2048, 4096, and 8192 MB). As previously mentioned, these results are embedded into BestGC in the form of matrices (as detailed in Section II.4.1).

II.6.2.1 Application Execution Time

Table II.4 shows the arithmetic mean of the DaCapo and Renaissance workloads' application execution time with different heap sizes. Since all the execution times are normalized to G1, the values for G1 are all 1 (so, we do not show these values in the table). Also, since we use application execution time to report throughput, a lower value is better. The table shows that Parallel outperforms other GCs in DaCapo and Renaissance benchmark suites for all the heap sizes. The results clearly confirm that Parallel GC is optimized for high throughput. It is, on average, about 5% better than G1 (value 1) and almost 7% and 6% better than Shenandoah and ZGC in DaCapo. Moreover, there is no considerable difference between the results when the heap size decreases for Parallel. After Parallel, G1 (value 1) has a better execution time than Shenandoah and ZGC with most of the heap sizes.

In the Renaissance, Parallel outperforms G1 (value 1), Shenandoah, and ZGC, respectively, by about 8%, 40%, and 60% on average. Parallel can keep the workloads' execution time almost the same for all the heap sizes, while application execution time in Shenandoah and ZGC worsens when decreasing the heap size. For example, with a 256 MB heap size, the application execution time of the workloads is 2.5× and 1.8× better with Parallel compared to ZGC and Shenandoah, respectively. This shows that the workloads in Renaissance are more aggressive regarding memory usage than DaCapo; in fact, using concurrent GCs, which work concurrently with the application threads, significantly affects the application execution time of these workloads. This difference is not happening in G1 (value 1), even with a 256 MB heap size. Unlike ZGC and Shenandoah, G1

Table II.5: Average (the arithmetic mean is calculated for the set of all the workloads in each benchmark suite) of the 90th percentile of GC pause times for the GCs with different heap sizes. Values are normalized to G1. Lower is better. The line highlighted in green (light gray) is the one with the best results.

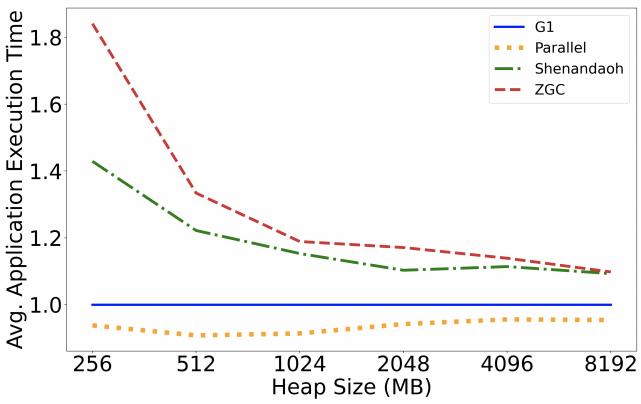
GC	a) DaCapo						b) Renaissance					
	Heap Size (MB)						Heap Size (MB)					
	256	**512**	**1024**	**2048**	**4096**	**8192**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
Parallel	2.298	2.208	2.485	2.188	1.174	0.945	1.060	1.050	1.143	1.543	0.959	1.996
Shenandoah	0.596	0.645	0.550	0.737	0.366	0.410	0.146	0.173	0.254	0.259	0.057	0.140
ZGC	0.149	0.136	0.128	0.128	0.132	0.127	0.069	0.053	0.111	0.112	0.054	0.021

is a generational collector that keeps a balance between throughput (application execution time in our case) and GC pause time. Also, since the minor collection happens more frequently, G1 is able to manage the heap and, consequently, the overall application execution time better.

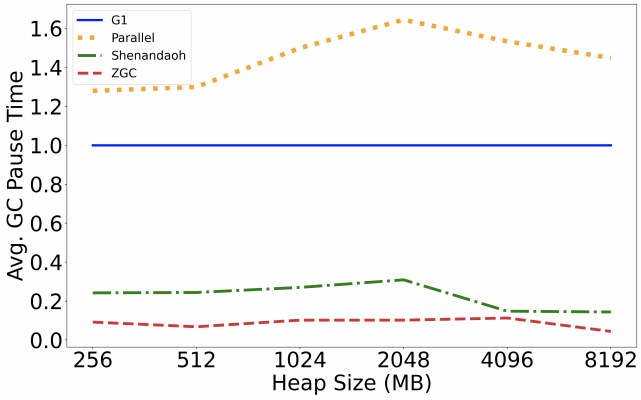
II.6.2.2 GC Pause Time

To extract GC pause times, we use the JVM log files corresponding to executing each workload for each GC. As previously mentioned, for each workload, after achieving stable results from several iterations, we use the results of the last iteration. To obtain the relevant GC pause times of the last iteration, we use the records of the JVM logfile from the exact time at which warm ups finish to the time the last iteration ends; then, we calculate the 90th percentile of the GC pause times (as it is used in most SLAs) that happened in the chunked log.

The average 90th percentile of GC pause times for both DaCapo and Renaissance are shown in Table II.5. For each GC, the 90th percentile of the GC pause times obtained from the workload's last iteration is normalized to G1. Since all the GC pause times are normalized to G1, the values for G1 are all 1 (so we do not show these values in the table). The arithmetic mean is calculated for the set of all the workloads' results (the results highlighted in green/light gray are the best). ZGC achieved the best results for all the heap sizes in DaCapo and Renaissance. On average, in DaCapo, ZGC has 7.5×, 14×, and 4× lower GC pause time than G1 (value 1), Parallel, and Shenandoah, respectively. Also, in Renaissance, ZGC outperformed G1 (value 1), Parallel, and Shenandoah by having the results 19.5×, 28×, and 3× lower than G1, Parallel, and Shenandoah on average. The workloads in Renaissance are memory demanding and sensitive to changing the heap size; so, with larger heap sizes, there is enough room for the concurrent GCs, especially ZGC, to spend less time in the collection process than Parallel and G1. By decreasing the heap size, concurrent GCs need more time to manage the heap.Parallel is the worst choice considering GC pause time due to the STW pauses and lack of concurrency. G1 (value 1), using its concurrent evacuation phase, performs better than Parallel. After ZGC, Shenandoah manages the GC pause time better than G1 (value 1)



(a) application execution time



(b) GC pause time

Figure II.5: Normalized (to G1) average application execution time and the normalized average of 90$^{\text{th}}$ percentile of GC pause time for all the DaCapo and Renaissance workloads.

and Parallel; it obtained results closer to ZGC than G1 (value 1) and Parallel. However, because of the overheads of the different barriers [Flo+16] it uses to provide concurrent collection, it manages the GC pause time worse than ZGC.

II.6.2.3 Discussion

Figures II.5a and II.5b show application execution time and GC pause times over all the workloads from both benchmark suites (values are normalized against G1, also shown). First, we provide each workload with a large heap size (letting GCs freely assign memory to new objects). Then, we gradually decrease the heap size and make the GCs do their best to manage the heap.

As Figure II.5a shows, Parallel outperforms other GCs for all heap sizes. It keeps application execution times almost the same while changing the heap size.

Parallel, which is a throughput-oriented GC, adjusts the generations' sizes to reach the best throughput (application execution time in our case). G1 comes after Parallel regarding the time it takes to run a workload. Shenandoah and ZGC have worse results and show different behavior with different heap sizes. Even with an 8192 MB heap size, they both have higher application execution times than G1 and Parallel. These two GCs' application execution times (Shenandoah and ZGC) are almost the same for 8192 MB; their execution times worsen when the heap size gets smaller. In fact, reducing the heap size from 1024 MB to 512 MB results in a significant application execution time degradation in Shenandoah and especially in ZGC. With a 256 MB heap, ZGC shows the worst application execution time among the other GCs. Compared to Shenandoah, which is also a concurrent GC, ZGC sacrifices more of the workloads' application execution time to provide concurrency since it requires more room in the heap to allow object allocations while the GC is running [CK22]. Parallel and G1 are generational collectors. As already mentioned, since newly created objects tend to die in a short time (generational hypothesis), garbage collection happens most frequently in the young generation. So, in each collection, a GC detects a noticeable amount of dead objects by tracing a portion of the heap (not the entire heap). Therefore, as Figure II.5a shows, these two GCs (Parallel and G1) deliver better application execution time, in our study, than ZGC and Shenandoah.

As Figure II.5b shows, unlike the application execution time results, Parallel performs poorly regarding the GC pause time. Parallel tries to change one generation size at a time (the generation with the more significant GC pause time [Ora20b]). With a 2048 MB heap capacity, it experiences the highest GC pause time to meet its throughput (application execution time) goal. While more heap capacity is available for Parallel, it may increase the young generation size; also, the objects created by the workloads fill this capacity, resulting in longer GC pause times for the frequent collections in the young generation with 2048 MB heap. For heap sizes above 2048 MB, generation sizes created by Parallel are large enough not to invoke a collection repeatedly. GC pause times for concurrent GCs, especially ZGC, are shorter than generational G1 and Parallel, as Figure II.5b illustrates. In other words, utilizing concurrent tracing and copying mechanisms, ZGC and Shenandoah results are significantly better than G1 and Parallel regarding the GC pause time. Although Shenandoah keeps the GC pause time very small, it shows no predictable pattern for different heap sizes. However, ZGC maintains almost a steady average GC pause time for all the heap sizes because its main goal is keeping GC pause times small, regardless of the heap size [Per18].

II.6.3 CPU Usage and GC

This section addresses the correlation between CPU usage and GCs. CPU is a resource shared between GC and application threads. So, the amount of CPU an application use affects GC performance and vice versa. We evaluated several workloads from DaCapo and Renaissance benchmark suites to investigate the correlation between CPU usage and GC. These benchmark suites mostly contain

Table II.6: Average performance benefit of the selected GC for each workload with different heap sizes.

Workloads	Benchmark Suite	Type	AVG Perf. Benefit(%) (comparing to G1) for Each Heap Size (MB)					
			256	512	1024	2048	4096	8192
Finagle-Chirper	Renaissance	CPU-Intensive	38.6	44.2	46.0	48.1	51.3	48.5
Xalan	DaCapo	CPU-Intensive	39.3	48.9	47.9	50.0	49.7	48.7
Avrora	DaCapo	Non-CPU-Intensive	50.1	48.6	48.5	49.1	49.5	50.5
Jme	DaCapo	Non-CPU-Intensive	49.3	49.4	49.4	49.1	48.1	47.5

workloads with high CPU consumption, yet, we selected some workloads with low and very high CPU usage per engaged cores (as discussed in Section II.4.3).

We run each workload with one GC (G1, Parallel, Shenandoah, ZGC) at a time with different heap sizes. Then, we empirically (i.e., manually) scored each GC, with different weights for application throughput/execution time (w_p) and GC pause time (w_e), using the same formula we used to score the GCs in BestGC (see Section II.4.4).

Next, we normalized the scores for each GC (G1, Parallel, Shenandoah, ZGC) to G1 and selected the GC with the minimum score. Then, for each pair of w_p and w_e, we calculated the performance benefit of the selected GC (i.e., the difference between the score of the best GC and G1). This metric shows how the selected GC is superior to G1 (when the application runs with the default GC). Finally, we computed the average of these differences in every heap size.

Table II.6 shows an example of the average performance benefit of the selected GCs, when compared to the default GC (G1), for two highly CPU-intensive and two non-CPU-intensive workloads. In CPU-intensive workloads, we can see that changing the heap size affects the performance benefit of the selected GC. In particular, for both Finagle-Chirper and Xalan, with the 256 MB heap, the performance benefit drops compared to larger heap sizes. In non-CPU-intensive workloads, Avrora and Jme, the results show that the performance benefit of the selected GC does not fluctuate with different heap sizes. In both workloads, the maximum and minimum performance benefits differ by approximately 4% in different heap sizes.

Based on the results obtained, we can conclude that changing the default GC (G1) has a positive performance effect. However, with different heap sizes available for the application, the performance benefit is lower in CPU-intensive applications compared to non-CPU-intensive applications. In fact, in applications with high CPU demands, GCs are restricted by available CPU resources, and their performance is affected consequently; this is depicted in Table II.6 that the performance benefit of the selected GC decreases as the heap size decreases. Due to this, in our system, BestGC, we report if a user's application is CPU-intensive (in the BestGC's output log) to inform the user that the performance benefit of the suggested GC by BestGC may be affected and restricted by the CPU usage of the application.

Table II.7: Workloads from SPECjvm2008 benchmark suite used in the validation.

Benchmark Suite	Workloads
SPECjvm2008	Compress, MPEGaudio,Crypto.rsa, Crypto.aes, Crypto.signverify, Sunflow, Scimark.large, Scimark.small, Serial, XML, Derby

II.6.4 Validation of BestGC Results

As already mentioned, BestGC suggests a GC based on the results obtained from the evaluation of workloads in DaCapo and Renaissance benchmark suites (in the form of matrices as described in Section II.4.1). In this section, we validate BestGC with workloads from the SPECjvm2008 benchmark suite. Our objective is to verify the degree of correspondence between the GC suggested by BestGC and the empirically determined GC for various applications. For this purpose, we empirically (i.e., manually) do all the steps and measurements we performed in BestGC (regarding the workload's execution time and GC pause time) for the SPECjvm2008 workloads.Thus, we have the following three phases to consider: 1) empirically (i.e., manually) finding the most proper GC by measuring the application execution time and GC pause time for SPECjvm2008 workloads, 2) running BestGC to get the suggested GC for the workloads in SPECjvm2008, and 3) comparing the results of the previous two steps. In other words, we validate BestGC using workloads from SPECjvm2008 as if these were any other application a user may have. For the first step:

- We use the `Lagom` switch (available in SPECjvm2008) to run each workload. Lagom provides a fixed-size workload for the benchmarks, i.e., it does a fixed number of operations in each benchmark (just like DaCapo and Renaissance benchmark suites). Then, we use the corresponding application execution time as a metric for throughput.

- We select eleven workloads (see Table II.7) from SPECjvm2008. We excluded the Startup workload since it has a very short execution time (less than one second), as well as the Compiler workload since it is not compatible with JDK version 15.

- Just as with the workloads from DaCapo and Renaissance benchmark suites shown in the previous sections, we invoke all the workloads of the SPECjvm2088 several times to the point the execution times remain stable, employing one of four GCs (G1, Parallel, Shenandoah, ZGC) at a time. Then, we use the application execution time and GC pause times of the last iteration.

To empirically obtain the application execution time and GC pause time, and consequently, to find the GC that fits best a SPECjvm2008 workload, we applied the following:

Table II.8: Heap size selected with empirical measurements for each one of the SPECjvm2008 workloads and the heap size suggested by BestGC (for all the w_e and w_p).

Workloads	real max heap usage × 1.2 (MB)	selected heap size by empirical measurements (MB)	heap size determined by BestGC (MB)
Compress	259.5	512	512
MPEGaudio	187.2	256	256
Crypto.rsa	184.6	256	256
Crypto.aes	866.9	1024	1024
Crypto.signverify	326.8	512	512
Sunflow	777.6	1024	1024
Scimark.large	1532.0	2048	2048
Scimark.small	314.5	512	512
Serial	654.6	1024	1024
XML	1359.9	2048	2048
Derby	1626.9	2048	2048

- We record the output of the *jstat -gc* command every second while running each workload; at the end of its execution, we find the maximum heap used by it. We increase the maximum heap usage by 20% (maximum-used-heap ×1.2), the same way as we did for BestGC, and select the next available heap size from the heap sizes set (256, 512, 1024, 2048, 4096, 8192 MB).
- We calculate the average application execution time, and the 90^{th} GC pause time normalized to G1 for the selected heap size, for each SPECjvm2008 workload. Then, we use the same formula as we used for BestGC to score the GCs (see Section II.4.4).
- Finally, we mark the GC with the minimum score for different application execution time and GC pause time weights as the most proper GC.

In the second step, we follow the same approach as BestGC to obtain the max heap used by the application and find the most proper GC. Then, in the last step, we compare the results from what we have previously obtained empirically to those obtained with BestGC.

The heap sizes obtained from the empirical maximum heap size measurement and the one obtained with BestGC are shown in Table II.8. In the second column, the real max heap size utilized by each SPECjvm2008's workload is multiplied by 1.2. We consider 20% extra headroom for GC (see Section II.5) as we did with BestGC. In the third and fourth columns, it can be seen that the measurement of the heap in BestGC is consistent with the measurements we took empirically.

Table II.9 shows the most appropriate heap size (empirically obtained), the suggested GC by BestGC (for different w_e and w_p values), and the most proper GC, all for the Compress workload. Note that the last row is the GC with the minimum score obtained empirically (for the Compress workload) corresponding to the heap used. As the table shows, for all the w_p larger than 0.3, the

Table II.9: Comparing the GCs suggested by BestGC and the GC with the minimum score (empirically obtained) for the Compress workload.

w_e	**0.0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
w_p	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**	**0.5**	**0.4**	**0.3**	**0.2**	**0.1**	**0.0**
Most proper heap size in MB (empirically obtained)	512	512	512	512	512	512	512	512	512	512	512
Suggested GC (by BestGC)	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	Shenandoah	Parallel	Parallel	Parallel
Empirically obtained GC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	ZGC	G1

suggested GC by BestGC and the GC obtained empirically are exactly the same. With $w_p = 0.3$ and $w_e = 0.7$, although the GC suggested by BestGC and the GC empirically obtained are not exactly the same, they are both in the fully-concurrent GC category. For other values of w_p and w_e, the suggested GC by BestGC and the GC empirically obtained are neither the same nor of the same category. In these cases, we think that such a mismatch may happen because BestGC did not use even more workloads. In the future, as mentioned in Section II.8, we plan to include further evaluations of other classes of workloads that rely on resources other than CPU, such as I/O. This will allow BestGC to have even better results.

By using a similar evaluation (like Compress) for other workloads in SPECjvm2008, we evaluate the accuracy of the suggested GC (by BestGC) for different w_e/w_p while $w_t + w_p = 1$ (see Table II.10):

- **Exact GC**: indicates the percentage of cases in which BestGC suggests a GC that exactly matches the GC empirically chosen for the workload.
- **Exact Category**: indicates the percentage of cases in which BestGC successfully suggests a GC with the same category (concurrent or generational (non-fully concurrent)) as the GC empirically chosen. Note that BestGC is not offering the exact GC in this case.
- **Worst-case Performance Benefit (comparing to G1)**: in the cases in which BestGC fails to offer both the exact GC and the exact category, this metric shows the difference between the score of the GC suggested by BestGC and the default GC, which is G1 (when BestGC is not used and the JVM runs with the default GC).
- **Overall Performance Benefit (comparing to G1)**: shows the average difference between the scores of suggested GC by BestGC and G1 with respect to all the BestGC's failures and successes. In other words, it shows how the suggested GC performs compared to the default GC (G1) overall.

Table II.10: Validation of BestGC using SPECjvm2008 workloads. N/A when the fail percentage is 0. The average (arithmetic mean) is calculated for each metric.

Workloads	Exact GC (%)	Exact Category (%)	Worst-case Perf. Benefit (comparing to G1) (%)	Overall Perf. Benefit (comparing to G1) (%)
Compress	63.64	81.82	4.18	37.53
MPEGaudio	36.36	81.82	0	34.16
Crypto.rsa	36.36	81.82	1.49	33.46
Crypto.aes	18.18	100	N/A	37.42
Crypto.signverify	63.64	72.73	2.8	36.98
Sunflow	90.91	100	N/A	39.79
Scimark.large	9.09	81.82	-0.55	29.21
Scimark.small	18.18	72.73	11.07	34.90
Serial	100	100	N/A	41.10
XML	81.82	90.91	0	45.57
Derby	45.45	81.82	-4.97	34.22

Table II.10 shows the BestGC's results in terms of the metrics defined above for the eleven SPECjvm2088 workloads (shown in Table II.7). For each workload, there is a percentage for *exact GC*, *exact category* detection by BestGC, in addition to the average *worst-case performance benefit (comparing to G1)* and average *overall performance benefit (comparing to G1)*.

The table indicates, for instance, that BestGC's exact GC suggestion percentage is 63.64% for the Compress workload. At the same time, it offers a GC with the exact category in 81.82% of the cases. BestGC failed to suggest the exact GC category in 18.18% of the cases (worst-cases) in Compress; however, the suggested GC by BestGC still causes a 4.18% improvement (worst-case performance benefit) in GC score compared to running the user application with the default GC (G1). In addition, for this workload, it is shown in the table that using the suggested GC (with respect to all failures and successes of BestGC for all the w_e and w_p) makes, on average, a 37.53% improvement compared to default GC (G1) when the user does not use BestGC. The worst-case performance benefit for MPEGaudio and XML workloads are 0. It shows that for these two workloads, BestGC suggested G1, where it failed to detect both the exact GC and the exact category; thus, there is 0% improvement between the suggested GC and the default GC (G1). Also, according to the table, BestGC suggests the exact GC for all the application execution time and GC pause time weights in the Serial workload. For Sunflow and Crypto.aes workloads, although the *exact GC* metric for BestGC is not 100%, it could suggest a GC with the correct GC category (Figures II.5a and II.5b demonstrates that most of the time, GCs with the same category have close scores). In Scimark.large and Derby workloads, the worst-case performance benefit is 0.55% and 4.97%, respectively. For these workloads, using BestGC still results in significant overall performance benefits compared to G1. The average of all the worst-case performance benefits for the workloads in SPECjvm2008 is 1.75%. It shows that using the suggested GC

by BestGC has a 1.75% performance benefit compared to the default GC (G1). BestGC suggests the exact GC for the workloads in 51.24%, while it suggests the best GC category in 85.95%, on average. BestGC's average overall performance benefit is reported in the last column in Table II.10. It indicates that using BestGC's suggested GC results in an average improvement of 36.75% (including all the failures and successes of BestGC) compared to the situation in which applications are run with the default GC (G1).

II.7 Conclusion

In this paper, we proposed BestGC, a system that automatically runs a user application with the suggested GC, considering user preferences regarding application throughput and GC pause time. Although GCs used in production may have different objectives, the end-user may not be familiar with their specific characteristics. Users may prioritize throughput to a certain extent or may have the primary concern of achieving an acceptable level of GC pause time for their application, as deviations from this may negatively impact their goals.

To do that, we evaluated four widely used production GCs (G1, Parallel, Shenandoah, and ZGC) available in OpenJDK version 15 regarding their most critical performance metrics: application throughput/execution time and GC pause time, while changing the available heap sizes. BestGC respects all the requirements we set at the beginning of this work (see Section II.1). It provides a flexible *monitoring-time* and allows a user/developer to indicate what performance metrics of her/his application (application throughput or GC pause time) should be considered.

The results show that G1 and Parallel perform better than (mostly) concurrent GCs regarding application execution time, especially when decreasing the heap size. With an 8192 MB heap size, the application execution time for concurrent GCs is about 15% more than Parallel GC; however, it rises about 43% for Shenandoah and about 84% for ZGC when the heap size is decreased to 256 MB. Considering GC pause time, ZGC outperforms all the GCs, followed by Shenandoah, while there is a huge difference between these two (mostly) concurrent GCs and G1 and Parallel. In the worst case, for Parallel with 2048 MB heap, ZGC achieves about 16× smaller GC pause times than Parallel.

We also evaluated BestGC using SPECjvm2008 workloads in which each workload is used as any user application. On average, BestGC suggests the most proper GC for about 51.24% of the time; also, it suggests a GC with the best category (concurrent/non-concurrent) on average about 85.95% of the time. When BestGC fails, still using the suggested GC by BestGC results in about a 1.75% improvement in GC score compared to using the default OpenJDK GC (G1). This improvement for a highly optimized environment like OpenJDK would greatly affect the performance of the users' applications, especially for big data and cloud applications. Using BestGC results in having a GC with an average of 36.75% overall performance benefit, considering both BestGC's failures and successes, compared to running the user's application with the

default GC (when not using the BestGC).

We also investigated the correlation between CPU usage and the suggested GC by BestGC. The user should be aware that although the suggested GC by BestGC improves performance compared to default GC (G1), this performance benefit in CPU-intensive applications may be lower with different heap sizes.

II.8 Future Work

The extensibility of BestGC is not limited to the scope of this specific work but has broader applicability. By leveraging the underlying principles and methodologies utilized in the development of BestGC, it becomes feasible to adapt and integrate the tool with other JDK versions, GCs, and heap configurations. Therefore, the next version of BestGC will be able to re-run all benchmarks and re-generate all the performance matrices to accommodate new JDK versions and/or new GCs.

We will also include further evaluations of other classes of workloads that rely on resources other than CPU, such as I/O. This will depict how the GCs impact this class of applications and leads to expanding the BestGC data set.

While we focused on three crucial performance metrics, memory usage, application throughput, and GC pause time, it is worth noting that other metrics, such as latency, could potentially be added to our scoring formula. There are some latency-sensitive workloads available in DaCapo; however, limitations existed in our evaluation set due to the chosen heap sizes, resulting in excluding certain latency-sensitive workloads (Section II.6.1). Additionally, there are issues with CPU utilization in some of these latency-sensitive workloads (noted by the DaCapo maintainers[11]), e.g. Jme, Kafka, and Lusearch, that will directly affect the latency results and make them unreliable. So, future work also can encompass evaluating a sufficient number of latency-sensitive workloads to better score the GCs.

Furthermore, in our future work, we also plan on exploring Machine Learning techniques [Sin+07] to achieve a more accurate model and replace the formula we used in BestGC. It has the potential to enhance the accuracy of the GC offered by BestGC.

References

[20] *Java Hotspot VM*. 2020. URL: https://docs.oracle.com/javase/specs/jvms/se15/html/.

[BCM04] Blackburn, S. M., Cheng, P., and McKinley, K. S. "Myths and realities: The performance impact of garbage collection". In: *ACM SIGMETRICS Performance Evaluation Review* vol. 32, no. 1 (2004), pp. 25–36.

[11] https://github.com/dacapobench/dacapobench/blob/dev-chopin/benchmarks/status.md

[Ber+22] Beronić, D. et al. “Assessing Contemporary Automated Memory Management in Java–Garbage First, Shenandoah, and Z Garbage Collectors Comparison”. In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1495–1500.

[BF17] Bruno, R. and Ferreira, P. “POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications”. In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 147–160.

[BF18] Bruno, R. and Ferreira, P. “A study on garbage collection algorithms for big data environments”. In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. “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*. 2006, pp. 169–190.

[BOF17] Bruno, R., Oliveira, L. P., and Ferreira, P. “NG2C: pretenuring garbage collection with dynamic generations for HotSpot big data applications”. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 2017, pp. 2–13.

[Bru+19] Bruno, R. et al. “Runtime object lifetime profiler for latency sensitive big data applications”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[Cai+22] Cai, Z. et al. “Distilling the real cost of production garbage collectors”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 46–57.

[CK22] Clark, I. and Karlsson, S. *Z Garbage Collector*. Sept. 2022. URL: https://wiki.openjdk.org/display/zgc/Main.

[Col60] Collins, G. E. “A method for overlapping and erasure of lists”. In: *Communications of the ACM* vol. 3, no. 12 (1960), pp. 655–657.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Eva20] Evans, B. *What Tens of Millions of VMs Reveal about the State of Java*. Mar. 2020. URL: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[Fou18] Foundation, T. A. S. *Apache spark™ - unified engine for large-scale data analytics*. 2018.

[Fra+07] Frampton, D. et al. "Generational real-time garbage collection". In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 101–125.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. "Comparison of garbage collectors in Java programming language". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gog+15] Gog, I. et al. "Broom: Sweeping out garbage collection from big data systems". In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. 2015.

[JHM16] Jones, R., Hosking, A., and Moss, E. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[Len+17] Lengauer, P. et al. "A comprehensive java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo scala, and SPECjvm2008". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 3–14.

[LH83] Lieberman, H. and Hewitt, C. "A real-time garbage collector based on the lifetimes of objects". In: *Communications of the ACM* vol. 26, no. 6 (1983), pp. 419–429.

[Ngu+16] Nguyen, K. et al. "Yak: A High-Performance Big-Data-Friendly Garbage Collector". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 349–365.

[Ora01] Oracle. *Native Memory Tracking*. 2001. URL: https://docs.oracle.com/en/java/javase/15/vm/native-memory-tracking.html.

[Ora15] Oracle. *Concurrent Mark Sweep (CMS) Collector*. 2015. URL: https://docs.oracle.com/en/java/javase/11/%20gctuning/concurrent-mark-sweep-cms-collector.html.

[Ora20a] Oracle. *Available Collectors in JDK 15*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/available-collectors.html#GUID-45794DA6-AB96-4856-A96D-FDE5F7DEE498.

[Ora20b] Oracle. *HotSpot Virtual Machine Garbage Collection Tuning Guide, JDK15*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf.

[Ora20c] Oracle. *The Parallel Collector*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/parallel-collector1.html#GUID-74BE3BC9-C7ED-4AF8-A202-793255C864C4.

[Oss+02] Ossia, Y. et al. “A parallel, incremental and concurrent GC for servers”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[PD00] Printezis, T. and Detlefs, D. “A generational mostly-concurrent garbage collector”. In: *Proceedings of the 2nd international symposium on Memory management*. 2000, pp. 143–154.

[Per18] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. “Analysis of garbage collection algorithms and memory management in Java”. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. “Stopless: a real-time garbage collector for multiprocessors”. In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. “A study of concurrent real-time garbage collectors”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sew+11] Sewe, A. et al. “Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine”. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, pp. 657–676.

[Sin+07] Singer, J. et al. “Intelligent selection of application-specific garbage collectors”. In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 91–102.

[Sta08] Standard Performance Evaluation Corporation. *SPECjvm2008*. 2008. URL: http://www.spec.org/jvm2008/index.html.

[Tav20] Tavakolisomeh, S. “Selecting a JVM Garbage Collector for Big Data and Cloud Services”. In: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*. 2020, pp. 22–25.

[TIW11] Tene, G., Iyengar, B., and Wolf, M. “C4: the continuously concurrent compacting collector”. In: *International Symposium on Mathematical Morphology and Its Application to Signal and Image Processing*. 2011.

[Wu+20] Wu, M. et al. “Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services”. In: *USENIX Annual Technical Conference*. 2020.

[Xu+19] Xu, L. et al. “An experimental evaluation of garbage collectors on big data applications”. In: *The 45th International Conference on Very Large Data Bases (VLDB’19)*. 2019.

[Yan+12] Yang, X. et al. “Barriers reconsidered, friendlier still!” In: *ACM SIGPLAN Notices* vol. 47, no. 11 (2012), pp. 37–48.

[ZB20] Zhao, W. and Blackburn, S. M. “Deconstructing the garbage-first collector”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

[ZBM22] Zhao, W., Blackburn, S. M., and McKinley, K. S. “Low-latency, high-throughput garbage collection”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 76–91.

Paper III

Heap Size Adjustment with CPU Control

Sanaz Tavakolisomeh, Marina Shimchenko, Eric Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad

Published in *20th International Conference on Managed Programming Languages and Runtimes (MPLR) 2023*, Lisbon, Portugal, doi: 10.1145/3617651.3622988

Abstract

This paper explores automatic heap sizing where developers let the frequency of GC expressed as a target overhead of the application's CPU utilization, control the size of the heap, as opposed to the other way around. Given enough headroom and spare CPU, a concurrent garbage collector should be able to keep up with the application's allocation rate, and neither the frequency nor duration of GC should impact throughput and latency. Because of the inverse relationship between time spent performing garbage collection and the minimal size of the heap, this enables trading memory for computation and conversely, neutral to an application's performance.

We describe our proposal for automatically adjusting the size of a program's heap based on the CPU overhead of GC. We show how our idea can be relatively easily integrated into ZGC, a concurrent collector in OpenJDK, and study the impact of our approach on memory requirements, throughput, latency, and energy.

Contents

Table III.1: Smallest heap sizes in MB without any allocation stalls for multiple benchmarks across multiple machines. Machines are listed in ascending order based on the number of cores and memory capacity. The lower half of the table displays architectural details for each machine. Machine number 3 is listed three times, indicating configurations with 8, 16, and 24 cores, where the core count was controlled using the `taskset` command. Note that machine #1 could not run Batik without experiencing stalls due to a combination of insufficient memory and inadequate hardware resources for running GC effectively.

	Machines	#1	#2	#3a	#4	#3b	#3c	#5	#6
Heap Sizes	Tomcat	256	512	512	1024	1024	2048	2048	4096
	Spring	4096	4096	4096	8192	4096	16384	4096	32768
	Batik	–	32768	65536	16384	32768	32768	16384	16384
	Luindex	512	512	512	256	512	512	256	256
Hardware Specs	Cores	8	8	8	16	16	24	32	44
	Hardware threads	8	16	16	24	32	48	64	88
	Memory (GB)	16	64	256	128	256	256	16	64
	Core freq (GHz)	2.13	2.3	3.7	3.2/2.4	3.7	3.7	2.7	2.2
	L1 (bytes)	512 KB	64 KB	1 MB	1.5 MB	1 MB	1 MB	64 KB	64 KB
	L3 (bytes)	8 MB	16 MB	128 MB	30 MB	128 MB	128 MB	20 MB	55 MB

III.1 Introduction

Garbage collection (GC) offers significant benefits to applications and developers. By abstracting away memory management, code is not tied to a specific strategy for managing heap memory, allowing programs to switch easily between different GC implementations with different properties, e.g., by a command-line argument.

Managed programming languages use various approaches for controlling an application's footprint. Some languages include strategies that automatically reduce the heap size based on memory usage or other metrics. Although the programmer can influence this behavior to some extent by ensuring that objects become garbage, the system may not detect it immediately. Other languages allow the programmer to set an upper bound for the heap size and then manage it relative to that limit. Regardless of how language runtimes manage memory, collecting memory inherently impacts performance in an indirect and hard-to-predict manner.

In OpenJDK HotSpot JVM (OpenJDK for short) a maximum heap size is set on startup, to a user-defined value using the *–Xmx* command-line flag, or in its absence, by picking a default value based on the available memory of the machine (at the time of writing, OpenJDK sets *Xmx* to 25% of the machine's RAM). This decision is made before the program is started, so unless care is taken to explicitly control the maximum heap size, simple programs and complex enterprise applications will share the same memory constraints.

The size of the heap affects the performance differently depending on the GC algorithm. Stop-the-world GC's typically optimize for high throughput and are only able to deliver low latency if the working sets are small enough. In contrast, concurrent collectors typically are not able to achieve as high throughput, but by allowing program activities to continue while GC is running, the frequency or duration of GC has very little impact on a program's performance, as long as the GC is able to collect memory at the same rate as the application is allocating. When memory is abundant or allocation rate is low, infrequent GC can materialize through worse spatial locality in both types of collectors, which may negatively affect performance and/or latency [YÖW20a].

A common approach to picking a heap size for an application is by trial-and-error: run the program multiple times with representative load across different JVM instances with varying maximum heap limits and measure its performance until a suitable heap size is identified. A heap size may not be portable across machines and may have to be reevaluated after changes are made to the software or after a switch to a new JVM. This approach may be time-consuming and may not account for variations in the program's memory use during execution. If the maximum heap size is invariant throughout the entire program duration (as in OpenJDK), the entire heap may be used for allocating objects even when memory pressure is low(er), which defers GC and may not be optimal for a program's performance. In conclusion, determining an appropriate heap size for a given application is a complex task that necessitates consideration of various factors. These factors include the hardware configuration of the machine running the program and software-related details such as memory usage patterns.

Automatic heap size adjustment aims to free developers from the need to manually set a heap size, which has proven to be complicated. Instead, developers will be given a sensible default parameter for effective resource management, which should also be intuitive to change. In this work, we explore automatic heap size adjustment in the context of concurrent collectors, where the heap size is controlled by how often we trigger GC, instead of the other way around. As a result, developers can launch Java applications (servers, GUI programs, command line tools, etc.) without having to worry about estimating their memory requirements or worrying that Java's default values might result in these processes ballooning to impractical proportions, thereby disrupting other programs or affecting the application's performance negatively. Our proposal distinguishes itself from previous proposals for automatically adjusting the heap size (e.g., Bruno et al. [Bru+18], Grzegorczyk et al. [Grz+07], and Yang et al. [Yan+04], and White et al. [Whi+13], cf. Section III.7) by utilizing a different "tuning knob" for concurrent collectors. Instead of letting developers control performance through an upper bound on the heap size, we let developers control how much CPU they are willing to spend on GC, expressed as a proportion of the CPU usage of the application. Our strategy is thus more directly tied to performance than heap size, and the heap size becomes a consequence of the GC CPU overhead budget (we call it GC target henceforth). As a result of our choice of tuning knob, the job of picking a reasonable default is easier (or dare we say possible!) than picking a default maximum heap size.

Our contributions can be summarized as:

- **Highlighting heap size variability**: We reveal significant variability of heap sizes across different benchmarks and hardware configurations, emphasizing the impracticality of a one-size-fits-all default heap size. This finding underscores the need for more adaptive approaches. (Section III.2)
- **Exploring automatic heap sizing in a concurrent collector:** We target concurrent collectors whose CPU-intensive activities are not on the critical path of the program's performance. This allows us to dynamically change the heap size to match the program's current behavior and allows developers to trade CPU for memory (and conversely) with minimal impact on performance. (Section III.3)
- **Application in ZGC**: We specifically showcase the implementation and application of our proposed technique on ZGC, a fully concurrent garbage collector. By doing so, we demonstrate its feasibility in a real-world example. (Section III.4)
- **Performance evaluation**: We conduct a comprehensive evaluation demonstrating that adopting our heap size adjustment does not compromise performance or introduce latency issues. Furthermore, we establish that it is possible to determine a sensible default value for CPU overhead that can effectively cater to a variety of applications. (Section III.6)
- **Energy efficiency considerations**: In addition to performance optimization, we illustrate how the concept of CPU overhead can be harnessed as a powerful tool for adjusting the energy spent by an application. (Section III.6)

III.2 The Perils of Manual Heap Size Picking

When it comes to finding heap sizes, people use multiple rules of thumb, for example, setting heap limits to some multiple of a live set [HB05]. The challenge becomes even more intricate when considering multiple applications running simultaneously. Kirisame et al. [KSP22] introduced a framework to compare different practices people used for setting up a heap size and derived an optimal "square-root" heap limit rule, which minimizes total memory usage for all applications running together. However, it is still a static heap limit, which might not be optimal on a machine with another architecture, as we demonstrate below.

To study the challenges of manually picking a heap size, we conducted an experiment across multiple machines to find heap sizes for a number of benchmarks. Table III.1 shows heap sizes for 4 benchmarks from the DaCapo suite running with the ZGC collector across a range of different machines. Following best practices for tuning heap sizes for concurrent collectors, we tried to find the smallest heap size—expressed as a power of two— that does not produce an allocation stall, a relocation stall, or an OOM (Out of Memory)

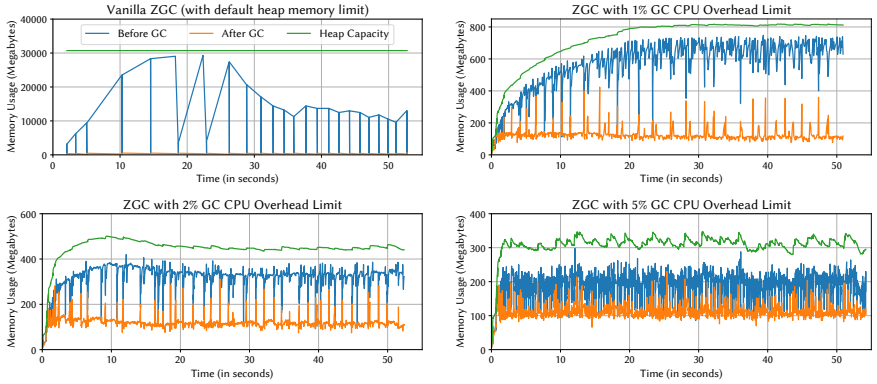


Figure III.1: Memory usage of vanilla (unmodified, by default uses 25% of the available RAM) ZGC (22 cycles) and ZGC with 1% (856 cycles), 2% (1506 cycles), and 5% (3182 cycles) GC CPU overhead limits. For this run, we used 12 application threads on a 16-core machine, leaving a 4-core headroom. For each, we measure the following: maximum heap size, memory usage before GC, and memory usage after GC. Note that the y-axis for vanilla ZGC is two orders of magnitude higher. The differences in the x-axes demonstrates the impact of GC on throughput. An artifact of the current ZGC design where each GC cycle forces mutators to take a slow path in the load barrier the first time each reference is loaded. Thus, very frequent GC (, i.e., 5%) can materialize as a throughput regression.

error for each benchmark on each machine. Ensuring the absence of stalls is of paramount importance when utilizing fully concurrent collectors, given their low-latency nature. Stalls not only lead to performance degradation, as GC becomes critical, but they also undermine the predictability of GC, thereby posing a risk to meeting server-level agreements (SLAs) and latency requirements. Maintaining a consistent and predictable latency profile is essential to uphold performance standards and guarantee uninterrupted service delivery. The reason why we limit ourselves to powers of two is twofold: first, developers have a preference for selecting heap sizes in powers of two [Eva20], and second, finding a stall-free heap size in a reasonable time requires increasing the heap in some increments. In our case, we started at 16MB, doubled the heap size on a stall, and continued our search at the higher heap size. Nevertheless, due to Java's inherited variance, we adopt a stability-oriented approach in which we consider a heap size to be a successful candidate only if three consecutive runs with the same heap size yield no stalls or OOM errors. If stalls or OOM errors do occur, we increment the heap size and repeat the evaluation process.

As it is clear from this experiment, heap sizes vary between the machines without a discernible pattern[1], such as being a function of the number of cores.

[1]One anonymous MPLR reviewer suggested that heap sizes might be contingent on both

In addition, we experimented with the same machine, tagged as #3 in the table, with different numbers of cores controlled by taskset: 8, 16, and 24. For Tomcat and Spring, the heap size changes by 4×. So, even within the same machine, modifying core configurations can often require substantial changes in heap sizes. Thus, application deployment across different configurations requires heap sizing to be repeated for each configuration.

III.3 Heap Size Adjustment with CPU Control

Similar to [Whi+13], but in a concurrent setting, we explore an approach where developers directly control memory by setting a *GC target* dictating how much CPU should be spent on GC, expressed as a percentage of the total CPU utilization of the program. Our insight is that memory and GC CPU utilization are inversely correlated. Let's consider a program with a constant allocation rate. When the heap is large, GC occurs infrequently, resulting in low CPU time spent doing GC; conversely, when the heap is small, GC occurs more often, causing a corresponding increase in the CPU time spent doing GC. In a concurrent garbage collector, this kind of trading memory for CPU, or the other way around, should be largely (at least ideally) orthogonal to the program's performance since the program will not block on GC. Furthermore, the program's CPU usage can be considered a proxy for its allocation rate and, by extension, its need for GC. By expressing the GC target in terms of the program's CPU usage, increased program activity immediately translates to increased CPU headroom for GC in absolute numbers. Understanding and controlling the scalability and CPU utilization of a program is a more direct task compared to comprehending its live set, which encompasses all objects contributing to memory pressure.

We define the GC overhead (henceforth denoted GC_{CPU}) as the ratio of time spent doing GC (henceforth T_{GC}) to time spent in the entire application (henceforth T_{APP}):

$$GC_{CPU} = \frac{T_{GC}}{T_{APP}}. \tag{III.1}$$

These time measurements are the main inputs to our algorithm for determining the new heap size. To mitigate fluctuations, T_{GC} should be calculated using average times for the last n collections (in our implementation, we pick $n = 3$). For instance, if one GC cycle has high CPU activity when the previous cycles did not, it might be too hasty to change the heap size. Thus, the heap size varies "slowly," preventing committing memory that is not needed in the long run.

The core idea of our proposal is to iteratively adjust the heap size until the GC overhead,, i.e., GC_{CPU}, meets the target set by the developer, $Target\_GC_{CPU}$. Note that the value is a *target*, not an upper bound. Thus, if $GC_{CPU} > Target\_GC_{CPU}$, we increase the heap size to lower the GC frequency and thereby

the count of CPU cores and the number of mutators. We sadly currently lack data on the number of mutators, but this aspect presents an intriguing avenue for future exploration.

lower the GC CPU overhead. Conversely, when $GC_{CPU} < Target\_GC_{CPU}$, we decrease the heap size to trigger more collections, to increase the GC CPU overhead.

To showcase the impact of target GC CPU overhead, we run the Xalan benchmark from the DaCapo suite. It was run four times with vanilla ZGC on machine #3a from Table III.1. Additionally, the benchmark was executed with GC targets: 1%, 2%, and 5%. By default, vanilla ZGC uses a high heap memory size (25% of RAM) that is significantly reduced when higher GC target values are used. Figure III.1 depicts the results of these runs.

In our approach, during periods of lower CPU activity in the application, a collector will work less, as it is proportional to the application's CPU usage. This results in fewer allocations and overall less pressure on both the allocator and memory manager. Conversely, spikes in the application's activity translate into a higher CPU budget for the GC threads. While it may seem logical to run GC during low CPU activity to utilize available CPU resources, the effectiveness may be limited if there is less memory to free.

At the end of each GC cycle, we compare the GC CPU overhead to the user-defined GC target to calculate $overhead\_error_{CPU}$ which we use to adjust the heap size:

$$overhead\_error_{CPU} = GC_{CPU} - \frac{Target\_GC_{CPU}}{100} \tag{III.2}$$

We aim to prevent sudden and sharp heap size changes. Therefore, in addition to smoothing out fluctuations in the T_{GC} by considering the average over the last three collections, we avoid using $overhead\_error_{CPU}$ directly to modify the heap size, as large error numbers can cause fluctuations in the heap sizes. To mitigate this, we pass the $overhead\_error_{CPU}$ through the Sigmoid function [HM95] to smoothen changes in heap sizes[2]. The Sigmoid function is a mathematical function that is commonly used to model non-linear relationships between variables in statistical models. It maps input values to a range between 0 and 1. Thus, using the Sigmoid function prevents aggressive changes in the heap size. We pass the $overhead\_error_{CPU}$ to the Sigmoid function S to calculate "Sigmoid overhead error":

$$[h]S(overhead\_error) = \frac{1}{1 + e^{-overhead\_error_{CPU}}}. \tag{III.3}$$

We use this result to calculate an *adjustment factor* that limits the changes to the heap size to within a range of 0.5 to 1.5:

$$[h]adjustment\_factor = S(overhead\_error) + 0.5 \tag{III.4}$$

[2]We explored two variations of a step function as well. The first adjusted the heap size proportional to the disparities between CPU overhead and the target. The second involved increasing the soft limit by 50% in either direction if the CPU overhead was above or below the target. Each function led to distinct rates of adaptation and total memory usage. We did not directly compare these three functions against one another; instead, we somewhat arbitrarily opted for the Sigmoid function in our approach.

When $overhead\_error_{CPU}$ is zero, i.e., actual GC CPU overhead equals GC target, the Sigmoid function returns 0.5. Therefore, the *adjustment_factor* becomes 1 and the heap size remains unchanged.

An $S(overhead\_error) < 0.5$ means that the actual GC_{CPU} has exceeded the $Target\_GC_{CPU}$, so the *adjustment_factor* would be less than 1 and will reduce the heap size, leading to more GC cycles. When the actual GC_{CPU} is below $Target\_GC_{CPU}$, $S(overhead\_error) > 0.5$,, i.e., $adjustment\_factor > 1$ will increase the heap size. The heap size will never change more than 50% of the current size (in any direction). Finally, we compute the new heap size as follows:

$$new\_size = current\_size \times adjustment\_factor \tag{III.5}$$

Our approach can be used in combination with an upper bound on the heap size—e.g., *Xmx*—to trigger an OOM error. However, setting this upper limit may prevent the application from reaching the target GC CPU utilization rate ($Target\_GC_{CPU}$). If an upper limit is not specified, the system sets it to a default value, which should be close to the maximum memory available on the machine, but not set to 100% to prevent system instability and swapping. Note, that previously it was 25% of the machine.

III.4 Prototype Implementation in ZGC

Adjusting the heap size based on GC CPU overhead is suitable for concurrent GCs that do not interfere with the application's critical path. In this section, we implement a prototype on ZGC, a concurrent collector in OpenJDK, to demonstrate the effectiveness of this approach. The prototype follows the ideas presented in the previous section.

III.4.1 Background on ZGC

The Z Garbage Collector [**ZGC-main**] (ZGC) is designed for low latency, offering sub-millisecond pause times invariant of the heap size. GC activity in ZGC occurs concurrently with "mutators" (application threads) by relying on barriers that trap object accesses and coordinate accesses to objects from mutators and GC worker threads. A barrier is essentially some additional logic triggered (in the case of generational ZGC) when a reference is read from a field and placed on the stack or when a reference is loaded from a field. The barrier logic branches on metadata bits embedded in pointers [JHM12]. For example, in the case of a load barrier, if the metadata shows that the pointer is valid, we enter the fast path in which the overhead of the barrier is simply shifting off the metadata bits from the address. Otherwise, we enter the slow path, where we ensure that the pointer is valid by looking up the new canonical address of the object from a forwarding table. This last step may involve copying the object elsewhere and writing to the forwarding table ourselves. ZGC is a multi-phase collector with separate mark and evacuation phases. Its overall design was described by Yang et al. [YW22].

Single-generation ZGC uses load barriers to synchronize GC activities with mutators. Generational ZGC [Kar23] instead uses write barriers in addition to load barriers. It maintains a remembered set of references from the old generation to young objects, serving as additional roots during GC in the young generation only. Such a design favors generational workloads where objects are more likely to die young (following the weak generational hypothesis [LH83]) by supporting a more aggressive collection of the young generation without having to do repeated work on long-living objects. From a resource perspective, generational ZGC requires less CPU and memory usage than single-generation ZGC. In this paper, when we refer to ZGC, we are specifically discussing the generational version of ZGC.

Memory in OpenJDK and ZGC In addition to *Xmx*, ZGC introduced a new JVM option in OpenJDK 13 called "soft max heap size", and subsequently adopted by G1. A soft max heap size is a limit on the size of the heap, beyond which ZGC *strives* not to grow. Unlike *Xmx*, exceeding the soft max heap size will not result in an OOM error (unless the limit is equal to *Xmx*). When approaching the soft max heap, ZGC triggers GC to bring the heap size below the soft max heap size. If it fails to do so, it will grow the heap instead of going into an allocation stall. The soft max heap size value thus serves as a guiding parameter for GC to balance heap size and allocation rate and has a direct impact on GC activity and frequency. If the value is too small, ZGC might end up doing back-to-back collections. If the value is too large, it can lead to inflated memory costs, floating garbage, heap fragmentation, and poor spatial locality; especially under a low allocation rate.

The relations between different memory parameters in ZGC are shown in Figure III.2. Used memory refers to the occupied memory by both live and dead objects (that have not yet been collected). Maximum capacity or committed memory represents the amount of memory requested by OpenJDK from the Operating System, which is always higher than used memory. In practice, committed memory is often significantly higher: bursts of allocation immediately drive the committed memory up, and to avoid requesting memory from the OS—which may cause delay, or worse, fail—OpenJDK will not return committed memory unless several minutes have elapsed since it was needed (lower bounded by *Xms*, the flag is used to set the minimum and initial heap sizes).

ZGC Heuristics Heuristics control when to start a GC cycle to avoid running OOM and also how many threads to use for each cycle. In addition, a GC may also be triggered due to other reasons such as a high allocation rate, high heap usage, or if no collection has been triggered for 5 minutes. Collecting the old generation can also be performed occasionally if not triggered by other reasons. These heuristics consider the available free memory and the time remaining before an OOM error occurs based on the average allocation rate and unforeseen circumstances. To determine the number of GC workers required to prevent OOM, ZGC analyzes the duration of previous GC cycles and adjusts the worker

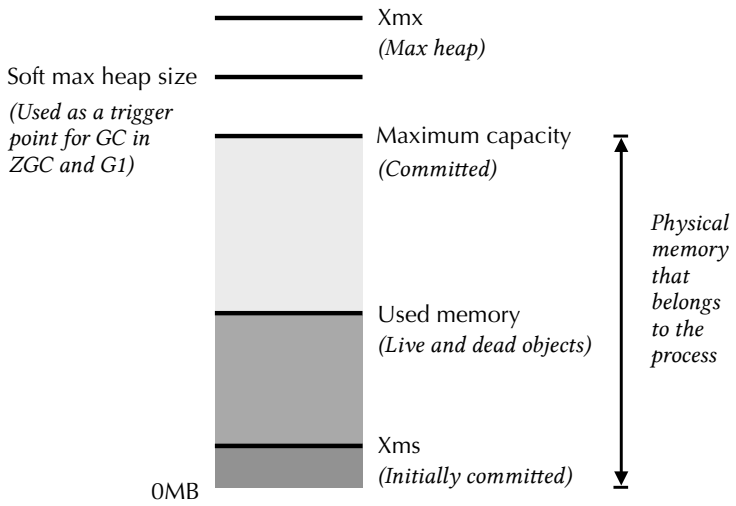


Figure III.2: The different heap parameters. *Xmx*: absolute maximum memory for an application. *Xms*: minimum and initial heap. *Maximum capacity*: currently committed memory (above or equal *Xms*, below or equal *Xmx*). *Used memory*: memory occupied by all the objects (above or equal *Xms*, below or equal *Maximum capacity*). *Soft max heap*: point below *Xmx* is used to trigger a GC but will not generate a stall if exceeded, up to *Xmx*.

count according to hardware limitations. Finally, ZGC predicts the duration of the next GC cycle based on the number of GC workers and calculates the start time for the next cycle.

III.4.2 Heap Size Adjustment with CPU control in ZGC

We take advantage of the aforementioned soft max heap size limit as it has the characteristics we require: it triggers GC but does not stall. To prevent exceeding the machine's heap capacity unintentionally (e.g., due to a too low target), we set *Xmx* to 80% of the available RAM[3] (unless the user has explicitly set *Xmx*). This ensures that the adaptive heap size remains within an upper limit.

Thus, the heap size in our prototype implementation is ZGC's soft max heap, and our technique ultimately results in adjusting the soft heap max up and down at the end of each GC cycle to meet the GC CPU overhead target set by the programmer. The amount of memory committed from the OS by OpenJDK is limited (as usual) by *Xmx* and will only grow in tandem with the soft max heap.

[3]This number reflects a pragmatic choice motivated by wanting to keep some spare memory for remaining programs running on the machine and also to leave space for ZGC's forwarding tables which are allocated off-heap and may grow very large under certain circumstances [NÖW22].

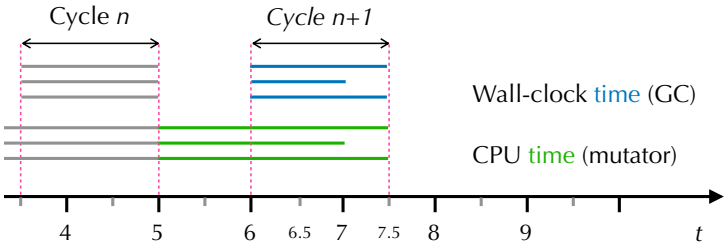


Figure III.3: Concrete measurements of GC and application time in our implementation. At the end of the GC cycle $n + 1$ ($t = 7.5$), we consider the time spent in GC threads (blue) and the time spent in mutators (green). Gray lines denote time measured at the end of GC cycle n. We only include the time when mutators were *scheduled*, meaning $C_{APP} = 2.5 + 2 + 2.5 = 7$. In the case of W_{GC}, we measure from the start to the finish of the GC cycle. Thus, $W_{GC} = 3 \times 1.5 = 4.5$, even though the 2nd GC thread was not scheduled after $t = 7$. Thus, $GC_{CPU} = \frac{4.5}{7} \approx 64\%$. (This example omits barriers, read more about them in Section III.4.3)

III.4.3 Obtaining T_{GC}

We calculate T_{GC} as the sum of time spent on young (W_{young}) and old collections (W_{old}) plus an estimate of the time mutators spent in the slow path of barriers (B). For simplicity and to avoid adding logic contributing to GC overhead, we use existing telemetry in ZGC. Thus, W_{young} and W_{old} are wall-clock time measurements. For traceability, we prefix wall-clock time measurements by W and CPU time measurements by C below. Thus, we will henceforth write W_{GC} instead of T_{GC} to highlight that the time measurement is a wall-clock time. To address potential inaccuracies in individual measurements, we calculate W_{young} and W_{old} using the average times for the last 3 collections (as we described in Section III.3). For uniformity, we use a single formula (Equation (III.6)) to describe W_{GC} and, in a minor collection, set W_{old} to 0.

$$W_{GC} = W_{young} + W_{old} + B \tag{III.6}$$

As already mentioned, B is the mutator time spent in the slow paths of barriers. When mutators hit slow paths in barriers, they do GC work, either remapping an old address to a forwarding address or performing relocation. We measure the wall-clock time of barriers using sampling: we record the time once for every 1024 slow paths taken, calculate the average time spent in slow paths, and multiply that with the number of slow paths taken.

ZGC calculates GC time separately for each generation by adding the times for the serial and parallel work. The serial time is the wall-clock time spent on non-parallel tasks like relocation set selection after marking, while the parallel time is the sum of the wall-clock time spent by worker threads on parallelizable tasks.

$$W_{young} = W_{serial\_young} + W_{parallel\_young} \tag{III.7}$$

Similarly, for activity in the old generation:

$$W_{old} = W_{serial\_old} + W_{parallel\_old} \tag{III.8}$$

III.4.4 Obtaining T_{APP}

The application's average time is the sum of the scheduled time of all threads spawned by the process (, i.e., a CPU time measurement) between two collections in the same generation. Thus, we write C_{APP} henceforth to clarify the nature of T_{APP} in our implementation:

$$C_{APP} = C_{GC_i} - C_{GC_{i+1}} \tag{III.9}$$

Similarly to GC time, we reuse existing GC telemetry to capture application time to avoid additional measurement overheads. Application time is obtained by measuring CPU time (see Figure III.3 for an overview). Listing III.1 shows the code for measuring the CPU time of the process.

Listing III.1: Code that calculates the process CPU time at moment of the call.

```
1  double ZAdaptiveHeap::process_cpu_time() {
2    timespec tp;
3    int status =
4      clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tp);
5    if (status != 0) {
6      return -1.0;
7    } else {
8      return double(tp.tv_sec) +
9           double(tp.tv_nsec) / NANOSECS_PER_SEC;
10   }
11 }
```

The function `clock_gettime` measures the CPU time consumed by a process, meaning that it includes the CPU time consumed by all threads in the process, including application threads, GC threads, compiler threads, etc. To measure the CPU time between two moments in time, we cache the last result and subtract it from the result of the subsequent call.

III.4.5 Calculating a Suggested Heap size

Most of our modifications to ZGC are located in its heap sizing mechanism: class `ZAdaptiveHeap`. The main logic is captured in the method `ZAdaptiveHeap::adapt` (see Listing III.2), which performs the calculations outlined in Sections III.3 to III.4. For clarity, we add comments with the labels from the equations to aid in mapping the C++ code to the descriptions above. The method is called at the end of each GC activity in both major (young + old) and minor (only young) collections.

Listing III.2: The modified `adapt` method that recalculates heap limits in ZGC. (For simplicity, we only show the logic for major GC and remove one lock to reduce clutter.)

```
1  void ZAdaptiveHeap::adapt(ZGenerationId generation,
2   ZStatCycleStats stats) {
3    ZGenerationData& generation_data =
4      _generation_data[(int)generation]; // holds the historical
5
6    double time_last = generation_data._last_cpu_time;
7    double time_now = process_cpu_time(); // see listing 1
8    generation_data._last_cpu_time = time_now;
9
10   // calculate C_APP as in (9)
11   double total_time = time_now - time_last;
12   // record C_APP to calculate averages
13   generation_data._process_cpu_time.add(total_time);
14
15   // Obtain the number of barriers triggered
16   size_t barriers =
17     Atomic::xchg(&_barrier_slow_paths, (size_t)0u);
18   // Obtain average barrier time
19   double barrier_slow_path_time=_barrier_cpu_time.davg();
20   // Calculate B in (6)
21   double avg_barrier_time =
22     barriers * barrier_slow_path_time;
23   double avg_gc_time = stats. // (7) or (8)
24     _avg_serial_time + stats._avg_parallelizable_time;
25   // recalculate C_APP using historical data to smoothen the curve
26   double avg_total_time =
27     generation_data._process_cpu_time.davg();
28
29   double avg_generation_cpu_overhead =
30     (avg_gc_time + avg_barrier_time) / avg_total_time;
31   Atomic::store(&generation_data._generation_cpu_overhead
32     , avg_generation_cpu_overhead);
33
34   double young_cpu_overhead =
35     Atomic::load(&young_data()._generation_cpu_overhead);
36   double old_cpu_overhead =
37     Atomic::load(&old_data()._generation_cpu_overhead);
38   double cpu_overhead = // Calculate W_GC as in (6)
39     young_cpu_overhead + old_cpu_ov or noterhead;
40   double cpu_overhead_error =
41     cpu_overhead - (ZCPUOverheadPercent / 100.0); // (2)
42   double cpu_overhead_sigmoid_error =
43     sigmoid_function(cpu_overhead_error); // (3)
44   double correction_factor =
45     cpu_overhead_sigmoid_error + 0.5; // (4)
46
47   if (is_enabled()){
48     // Call into ZGC to resize the heap, c.f. §III.4.6
49     ZHeap::heap()->resize_heap(correction_factor);
50   }
51 }
```

III.4.6 Bounding the Heap Size

After calculating the new suggested heap size, the listing below depicts how we ensure that the result of the calculation falls within a given range. If it does, the

function returns `suggested_heap_size`. If it is less than `lower_bound`, then the function returns `lower_bound`. Also, if it is greater than `upper_bound`, the function returns `upper_bound`.

```
1    const size_t upper_bound = // select smallest of two
2      MIN2(soft_max_capacity, current_max_capacity);
3    const size_t lower_bound = // select smallest of two
4      MIN2(1.1 * used(), upper_bound); // 10% extra headroom
5
6    const size_t selected_capacity =
7      clamp(suggested_capacity, lower_bound, upper_bound);
```

We establish a lower bound for the suggested heap by using the amount of *used* memory. This is because we aim to avoid triggering GC more often than necessary. If we set the suggested heap size below the *used* value, we risk triggering GC when there are no objects to clean. Although concurrent GC does not interfere with the application's critical path (as it is running concurrently and does not force the application to stop) and therefore it might have a negligible impact on performance or latency, the additional GC work can have a negative impact on energy consumption.

III.4.7 Initial and Adapted Heap Sizes

We set the initial heap size to 16 MB, in terms of the soft limit. This is an unlikely heap size for most programs and will trigger GC as the limit is approached or exceeded which will cause GC to adapt the heap size and (most likely) increase the soft limit (by at most 50% each time). Figure III.1 shows the frequent increases of the soft limit in green. (The top-left sub-figure shows Vanilla ZGC where the soft limit is equal to *Xmx* and never exceeded.) If the soft limit is not exceeded, and the GC overhead is below the target, we will decrease the soft limit to trigger GC more often. This is clearly visible in the two bottom subfigures of Figure III.1.

III.5 Evaluation

In this section, we are going to answer the following question: How effective is our automated heap sizing strategy, based on CPU usage as a tuning knob, compared to vanilla ZGC which relies on setting a maximum heap size? We now explain our experimental setup and benchmarking methodology.

III.5.1 Hardware and Software

We evaluate our work by comparing our modified ZGC with its unmodified base also referred to as vanilla (generational ZGC in OpenJDK version 21). We used an Intel Xeon *SandyBridge* EN/EP server machine (machine #5 in Table III.1) running Oracle Linux Server 8.4. The machine has 32 identical CPUs, which we configured as a single NUMA-node to avoid NUMA effects. The CPU model is Intel® Xeon® CPU E5-2680 with 64KB L1 cache, 256KB L2 cache, a shared

20MB L3 cache, and 30GB RAM. The configuration allows us to obtain energy consumption statistics.

III.5.2 Benchmarks

We use the DaCapo benchmark suite (Chopin branch), which includes a variety of microbenchmarks and real-world applications that stress the JVM and the garbage collector. The suite includes several latency-sensitive applications that require low-latency response times. These benchmarks measure metered latency, including request serving time, queuing delays, and interruptions like GC. By using these benchmarks, GC performance can be evaluated in terms of both throughput and responsiveness. We excluded the benchmarks Kafka and JME due to a low CPU utilization issue, as well as Lusearch due to a high CPU utilization variability, making it hard to draw any meaningful conclusions (noted by the benchmark maintainers). We also excluded H2 due to a reproducible memory leak across multiple machines and garbage collectors. When referring to DaCapo in this paper, we specifically mean the DaCapo Chopin benchmark suite. We included all throughput-oriented benchmarks except Cassandra, which is incompatible since OpenJDK 16.

In order to obtain a more comprehensive understanding of our prototype, we also include the Hazelcast benchmark [Gen+21]. Hazelcast was chosen since most of the latency-sensitive workloads in DaCapo were excluded for the aforementioned reasons. As low latency is the main goal of a concurrent collector, we wanted to study more such workloads. Hazelcast is designed to provide distributed and scalable in-memory data storage and processing, which can help reduce data access and processing latency.

DaCapo We use a commit (number 300acaa7) that includes latency-oriented benchmarks as evaluating latency is crucial for fully concurrent collectors. We conducted the benchmarks using the *large* size for all applicable tests. For the remaining benchmarks (Fop, Zxing, Xalan), we used the *default* size.

Hazelcast Hazelcast performs real-time stream processing. We used all the suggested configuration parameters [Top20]. It has a fixed workload, set by its key-set size. We experiment with multiple key-set sizes: 400 000, 250 000, 100 000. Thus, we report the results of those 3 different configurations.

III.5.3 Benchmarking Methodology

We run each benchmark using 5 JVM instances, which lets us identify performance anomalies and outliers that might not have been discernible using a single JVM instance. Note that the variation between JVM instances is within the variation between the last 5 stable iterations of a single JVM. Inside each JVM instance, each benchmark repeats multiple iterations (varies across benchmarks to reach a coefficient of variation (CV) for the last 5 iterations below 5% with respect to execution time), which is necessary to avoid impact from warmup and JIT

compilation. Notably, our approach takes time to adjust from the initial heap size before stabilizing around a GC target.

Once we reached a steady state, we calculated the arithmetic mean of the last 5 iterations to remove noise from the environment. However, in cases where a steady state could not be reached, we used all recorded values for the last 5 iterations per JVM instead of computing the arithmetic mean. This is because taking a mean could hide outliers, and we do not know the shape of the data distribution.

In summary, we compute either one arithmetic mean per JVM instance (resulting in 5 data points in the final set) or all values from each JVM (resulting in 25 data points in the final set). We use the same approach for both adaptive ZGC and vanilla ZGC and to compare them, we perform statistical analysis on the final data sets. The final results reported in Table III.2 were calculated using an arithmetic mean of the final set.

III.5.4 Statistical Analysis

We used different tests to verify the validity and reliability of the results. We perform statistical analysis on the final data sets to draw our conclusions. We employed Welch's t-test [Wel38], Grubb's outlier test [Gru69], and Yuen's t-test [Yue74] to determine whether the differences between the means of the compared results from vanilla and adaptive ZGC are statistically significant. Welch's t-test and Yuen's t-test are particularly useful in cases where we can not make assumptions about the shape of data distribution and the variances of the compared groups are not equal. We believe these tests are safer to use instead of relying on a non-verifiable assumption about the normality of our data distribution.

We used Grubb's outlier test to check if the data set has statistical outliers. If so, we use Yuen's t-test instead of Welch's t-test. Yuen's test involves trimming a fixed proportion of the extreme values from each data set, we used 10%, to reduce the influence of outliers. To determine whether the results exhibit significant differences, we used the p-value obtained from Welch's t-test (with a significance level of 0.05). If the resulting p-value is greater than 0.05, we conclude that the data sets do not exhibit significant differences.

To help provide an overview, we color code the results if statistical significance was found. Red means the adaptive approach is worse than the vanilla ZGC; green means the opposite. White indicates the results are statistically the same. We also highlight a bigger than 5 % negative impact of our approach with a darker shade of red (Table III.2, Table III.3).

III.5.5 Energy Measurements

Energy consumption was measured using the Running Average Power Limit (RAPL) [Int09] interface available on recent Intel architectures. This interface allows machine-specific registers (MSRs) to be read, which contain energy scores. To calculate the final energy score, we report the sum of the package and

DRAM domains, following the method used by Shimchenko et al. [SPW22]. Our approach for measuring energy consumption is similar to that used for measuring throughput and latency. For DaCapo benchmarks, warmup iterations were excluded, and statistics were aggregated across 5 JVM instances for the last 5 iterations in each run. For the Hazelcast benchmark, we report energy consumption for the entire run, as it is a longer-running benchmark where the warmup period is a small fraction of the total run time.

III.5.6 Baseline Heap Sizes

If the *Xmx* option is not specified by the user when starting the JVM, the JVM will default the maximum heap size to 25% of the physical memory available on the system.[4] Research papers that involve measurements across multiple garbage collectors use other collectors like G1 [Cai+22] or Serial GC to pick the minimum heap size [Sah+16] and then employ a scaling factor to provide additional headroom (additional memory space) for other collectors. However, it is not at all clear if such an approach reflects the actual heap sizes chosen by developers for production systems. For example, developers often tend to choose heap sizes that are powers of two [Eva20].

Proper configuration of a concurrent collector should avoid allocation stalls as these introduce jitter and hurt latency. Thus, we decided to adopt a manual heap size adjustment strategy for our baseline (vanilla ZGC), where we pick the smallest power-of-two heap size with which the application runs reliably without stalling. We use this value for each benchmark as *Xmx* for a baseline configuration in vanilla ZGC; also, we explicitly set *Xms* to 16MB, which is the same as its default value according to the ZGC codebase. Finally, we had to manually pick heap sizes as there is no "best option". We pick baseline values not in order to "beat" something but explain the behavior of our system.

III.5.7 GC Targets

To investigate the implications of our proposed design, we studied the impact of GC targets on latency and throughput using varying percentages of GC CPU target overhead. Specifically, we examined the following GC targets: 5%, 10%, 15%, and 20%. To assess if the picked list of GC targets is representative, we found the actual GC CPU Overheads without the heap size adjustment for memories picked according to Table III.2. Looking at Table III.2's GC CPU overheads, the actual GC CPU overheads have a big variation from less than 1 % (Sunflow) to 23 % (Fop). Given that having a closer GC target to the actual GC overhead might better reveal the effect of our adaptive solution, we only evaluated our strategy with a 5% GC target for the benchmarks with actual GC overheads below 5%. This required running additional iterations to reach a steady state, adding time to benchmarking.

Methodologically, testing very small GC targets on short-running benchmarks is challenging since it takes time to grow the heap from the initial 16MB to a

[4] https://docs.oracle.com/en/java/javase/19/gctuning/ergonomics.html

size that sustains the required GC target. If this time exceeds the benchmark's run-time, it never reaches a steady state, causing the results inconclusive. Very high GC targets do not represent a real deployment. This said we believe that a picked range of GC targets is sufficient to demonstrate how our system behaves and showcase main trends.

III.6 Results

We now compare the performance of running the vanilla ZGC with manually selected heap sizes against our adaptive technique, which leverages different values of GC CPU overhead. Throughout the experiments, we closely examined various metrics, including memory usage, execution time, latency, and energy consumption. We sought to identify the advantages and drawbacks of each approach. Additionally, we propose an optimal default value for the GC CPU overhead that strikes a balance between efficient resource utilization and overall system performance.

Prior to presenting our results, we would like to address the absence of 3 benchmarks, Batik, Jython, and Pmd, from our study. These benchmarks have actual GC targets of 80 %, 76 %, and 170 %, respectively, using the maximum memory available on the SandyBridge machine. Therefore, we were unable to allocate additional memory to lower the GC targets for these benchmarks. Nevertheless, our methodology remains valid, and we were able to obtain results for these benchmarks by running them with the maximum available memory on the machine. As a result, the GC CPU overhead of these benchmarks remained similar to their actual values. Note that failing to attain a CPU target does not result in the failure of benchmark execution. The observed outcome is merely a disparity in real CPU overhead when compared to the requested target. If the target is set lower than the actual value and insufficient memory is available to elevate it, the application will persist in running without reaching the target. This situation remains unchanged unless the entire machine's memory suffices to prevent OOM issues, a scenario shared by Vanilla ZGC. Conversely, when the target surpasses the real CPU overhead and reducing memory fails to rectify it, this signifies an absence of substantial GC work. Irrespective of these scenarios, the application continues to function without interruption.

Memory Usage Memory usage for different GC targets is presented in Table III.2, normalized to the vanilla ZGC with the chosen heap size as described in Section III.5.6. Memory represents the average used memory before a GC for the last 5 stable iterations. Despite comparing memory maximums, normalization yielded similar results. Results show that overall memory usage decreases if the tested GC target is higher than the default GC CPU overhead. For instance, Hazelcast_100 has, by default, a GC CPU overhead of 21 %. Therefore, the memory used grows with 5 %, 10 %, and 15 % GC targets but is on par for a 20 % GC target. The biggest observed reduction is 96 % for Sunflow with 15 % and 20 % GC targets.

Table III.2: Execution time, memory, energy (all three normalized), the number of minor and major collections as well as GC CPU overheads in vanilla ZGC and adaptive ZGC for various benchmarks (BMs). Heap size (MB) (Z) is the minimum stall-free heap size for each benchmark. CPU Utilised shows the number of CPU cores used by the application (out of 32 cores available on SandyBridge). White cells show no statistical significance according to the methodology explained in Section III.5.4. Different shades of red represent highlights where the adaptive approach is worse than the default. Darker red indicated the CV above 5 %. We write (Z) for vanilla ZGC and (A) for our adaptive approach.

	GC targets	Avrora	Biojava	Graphchi	Hazelcast_100	Hazelcast_250	Hazelcast_400	Luindex	Spring	Sunflow	Tomact	Xalan	Fop	Zxing
Memory	5	0.76	0.8	0.22	1.51	1.65	1.9	0.86	0.52	0.09	0.72	0.66	1.41	0.6
	10	0.67	0.82	0.2	1.69	1.18	1.73	0.74	0.15	0.05	0.37	0.33	0.93	0.56
	15	0.66	0.43	0.19	1.36	0.93	1.15	0.61	0.14	0.04	0.44	0.32	0.92	0.49
	20	0.65	0.18	0.18	1.03	0.82	0.9	0.54	0.13	0.04	0.48	0.31	0.86	0.43
Execution Time	5	1.01	1.01	1.04	1.01	0.98	0.96	2.05	1.05	1.17	1.0	1.09	1.04	1.02
	10	0.99	1.01	1.02	1	1	0.96	1.02	1.05	1.15	1.02	1.06	0.61	1.01
	15	0.97	1.01	1	1	0.99	0.97	1.01	1.05	1.16	1.04	1.15	0.43	1.01
	20	0.95	1.02	1	1	0.99	1.01	1.02	1.05	1.15	1.06	1.15	0.44	1.02
Energy	5	0.99	0.62	1.01	0.95	0.95	0.95	1.84	0.99	1.12	1.0	1.08	1.35	1.02
	10	0.98	0.83	1	0.95	0.99	0.96	1.12	1.09	1.14	1.02	1.08	0.74	1.02
	15	0.96	0.82	1	0.93	1.01	0.97	1.09	1.13	1.15	1.03	1.15	0.64	1.02
	20	0.93	0.8	1.01	0.95	1.02	1.01	1.15	1.19	1.14	1.05	1.17	0.66	1.03
Minor (Z)		0	142	11	1213	443	798	4296	356	38	315	146	41	9
Minor (A)	5	82	174	450	151	170	195	12412	1187	38	482	676	51	18
	10	84	175	657	136	192	239	16134	6503	1269	3783	949	50	24
	15	146	428	1649	202	437	493	18979	9730	1529	6394	1335	62	38
	20	221	3338	2790	560	782	885	25941	14239	1661	10308	1649	82	44
Major (Z)		44	20	90	108	49	57	2952	16	7	64	9	7	3
Major (A)	5	86	25	126	38	35	31	3931	36	7	67	89	23	3
	10	88	23	240	33	37	34	5987	213	90	350	117	23	7
	15	95	27	330	49	55	48	6813	375	157	576	167	26	8
	20	91	45	383	86	69	68	9804	572	202	932	215	30	11
GC CPU OH (%) (Z)		1	3	1	21	12	17	4	4	0.2	3	3	9	2
GC CPU OH (%) (A)	5	5	5	3	10	9	10	5	5	5	5	5	5	6
	10	11	11	6	11	10	12	10	10	10	11	10	11	10
	15	16	13	9	15	15	15	17	15	14	15	15	16	17
	20	21	19	11	20	21	20	22	20	18	21	20	20	18
Heap Size (MB) (Z)		1024	8192	16384	2048	4096	4096	256	4096	16384	2048	1024	256	2048
CPU Utilised/32		0.9	1.07	5.45	15	19.2	27	1.25	12.32	30.22	21.06	21	2.5	21

Moreover, the reduction in memory usage correlates with a higher number of minor and major collections, which simply means that GC works more to keep a tighter heap. As expected, in terms of reducing memory footprint, 20 % leads to the smallest heap size across all the benchmarks.

Execution Time The results show that adjusting the heap size dynamically with 15 % and 10 % GC targets had a minimal negative impact on execution time, except for Xalan and Sunflow. However, Avrora, Hazelcast_400, and Fop have a reduction in execution time. For instance, Avrora showed a 3 % and 5 % improvement in execution time for 15 % and 20 % GC targets, respectively. This improvement can be attributed to the collector compacting live objects close together, improving cache locality [YÖW20b], and making memory accesses

Table III.3: The 99th-percentile metered latency from the adaptive approach normalized to vanilla ZGC. The color coding is the same as in Table III.2. H is for Hazelcast.

	Target	Tomcat	Spring	H_100	H_250	H_400
latency	10	0.68	1	0.4	0.91	0.69
	15	0.84	1.06	0.47	0.96	0.74
	20	0.96	1.18	0.73	1.14	1.16

easier to prefetch. However, Sunflow experienced a significant 15 % degradation in execution time. Additional profiling revealed more stalling in the instruction pipeline backend, which is often an indication of memory stalls [Int20]. It is possible that the 96 % memory reduction resulted in too many GC cycles, which interfered with the mutator accesses. To improve our technique in the future, we will consider the cache effects of too many GC cycles. From prior work, we also know that Sunflow is very sensitive to keeping allocation order during relocation and it is possible that this order is kept less well with so frequent GC cycles.

Energy As per our initial hypothesis (Table III.2), we expected energy changes to exhibit an opposite trend to memory. We anticipated that if a benchmark consumed more CPU during GC than the baseline, then we would see a decrease in memory usage and an increase in energy consumption. This is because CPU usage incurs higher energy costs than DRAM [Hor14]. As expected, the 20 % GC target yields on average worse energy results compared to 10 % and 15 % GC targets. However, it is apparent that the relationship between reduced memory and increased energy is not always linear. For instance, the Graphchi benchmark with 20 % GC target has a 82 % reduction in memory usage but only 1 % increase in energy consumption. At a single-program granularity, opting for high GC targets has an increased energy cost. However, in a cloud setting, where CPU is typically highly overcommited [Ope22] and memory is the limiting factor for consolidating virtual machines and containers, significant memory reductions lead to fewer physical nodes and ultimately lower energy consumption [Bas+21].

Latency In our evaluation, latency results were available only for a subset of benchmarks, which we report in Table III.3. Our adaptive approach has no negative impact on 99th-percentile latency and can even reduce it. For instance, in CPU-intensive workloads such as Tomcat and Hazelcast_400, where there is high competition for CPU resources between the collector and mutator threads, lower GC targets (, i.e., 10 %) lead to using more memory and positively affect latency by allowing GC to run less frequently, thereby reducing the impact on mutator performance. While increasing *Xmx* could achieve a similar effect, our approach reduces latency while also decreasing memory usage. Because, with the fixed memory level, GC CPU overhead can vary drastically throughout execution, leading to high numbers. Our technique keeps the GC CPU overhead more stable, aiming to fluctuate around a certain GC target. It ensures that GC

does not take up a lot of space, allowing mutators to deliver stable low latency without frequent drops. With a 20 % GC target and the smallest heap size, Spring showed a notable increase in latency. However, it is important to note that this benchmark has less than half of the capacity of the machine's average CPU utilization, but at times it spikes quite high, becoming CPU intensive. Higher GC targets in CPU-intensive workloads can reduce latency by mitigating contention between GC and mutator threads, as explained above. However, due to the limited number of latency-oriented workloads tested and the high variance in DaCapo benchmarks, we cannot make definitive conclusions about the positive impact of our technique on latency. Nonetheless, our findings suggest that it does not have a statistically significant negative effect.

Picking the Default GC Target Different GC targets can yield opposite trends for different optimization goals. While the highest GC target of 20 % provided the best results for memory, energy optimization requires the lowest GC target. Meanwhile, too many or too few GC cycles can harm performance. Thus, choosing the best GC target for each program may require manual selection. However, upon examining the benchmarks as a whole, we found that a 15 % GC target achieved a 51 % memory reduction, with only a 3 % execution time degradation and a 3 % increase in energy (calculated as the geometric mean across all benchmarks, following [FW86]). Therefore, a 15 % GC target may be a good default choice for optimizing the trade-off between memory usage, execution time, and energy consumption.

III.7 Related Work

Language runtimes that host managed languages—such as Java, Python, and JavaScript—maintain a garbage collected heap to manage live application objects (unreachable objects are collected by the garbage collector). Determining the heap size is challenging as it involves a tradeoff between application pause time, GC CPU, and memory utilization. Various heuristics have been proposed to achieve this goal by minimizing pause time, GC utilization, and memory usage.

III.7.1 Heap Size Adjustment Algorithms

A number of studies have been conducted for STW collectors, aiming to improve execution time [Bre+01], avoid paging [Grz+07] or both [Yan+04]. Brecht et al. [Bre+01] propose an adaptive technique to increase the heap size aiming to reduce execution time in the STW Boehm-Demers-Weiser GC [BW88]. The authors suggest increasing the heap size aggressively without collecting garbage if sufficient memory is available. Only when memory is scarce GC becomes more frequent, and the heap size stabilizes. This approach prioritizes reducing the GC target overhead to improve the application throughput.

Yang et al. [Yan+04] introduced an analytical model to adjust the heap size in the multi-program environment. In their approach, an operating system's virtual

memory manager monitors an application's memory allocation and footprint. Then, it periodically changes the heap size to closely match the real amount of memory used by the application. A model is used to minimize GC overhead by giving it enough heap size but also to minimize paging by avoiding large heaps. The model is offered for Appel and semi-space collectors [App89]. Zhang et al. [Zha+06] propose a novel approach to memory management called Program-level Adaptive Memory Management (PAMM). PAMM uses the program's repetitive patterns (phases) information to manage memory adaptively. The authors believe the behavior of the phase instances is quite similar and repetitive, so they can represent the memory usage cycle in the application. PAMM monitors the program's current heap usage and the number of page faults to adjust a softbound as a GC threshold. When the threshold is reached, PAMM triggers GC to collect and free unused memory. They evaluate PAMM with three STW and generational collectors (Mark-Sweep, CopyMS, and GenCopy). PAMM relies on a specific phase detection algorithm, which may not be applicable to all types of programs.

Grzegorczyk et al. [Grz+07] propose the Isla Vista heap size adjustment strategy to avoid GC-induced paging. Their strategy is to grow the heap when more physical memory is available and shrink it by triggering GC when there is not enough physical memory. Thus, it trades more GC for less paging by communicating between OS and VM and triggering the heap size adjustment logic on relocation stalls.

Controlling the ratio of GC time to overall execution time also has been addressed within HotSpot's collectors, using `-XX:GCTimeLimit`[5]. However, this strategy may not be suitable for concurrent collectors, as they are designed to operate concurrently with the mutator and outside of the program's critical path.

In a closely aligned study, White et al. [Whi+13] propose a PID (Proportional-Integral-Derivative) controller that monitors GC overhead (the percentage of total execution time spent on GC) and adjusts the heap resize ratio to maintain a target GC overhead level set by the user. They utilize the Jikes Research Virtual Machine (RVM), the Memory Management Toolkit (MMTk) as the experimental platform, and the FastAdaptiveMarkSweep collector.

More recently, Bruno et al. [Bru+18] propose a vertical memory scalability approach to scale JVM heap sizes dynamically. To do this, the authors introduce a new parameter: `CurrentMaxMemory`. Contrary to the static memory limit defined at launch time, `CurrentMaxMemory` can be re-defined at run-time, similar to our soft max capacity. In addition to the new dynamic limit, this work also proposed an automatic trigger to start heap compaction whenever the amount of unused memory is large. This technique allows returning memory to the Operating System as soon as possible.

[5] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gc-ergonomics.html

III.7.2 Heap Size Adjustment in State-of-the-Art GC

Immix [BM08] is a collector suitable for high-performance computing. Immix does not require the maximum heap size to be known in advance. It continuously monitors the amount of free memory available in the heap and adjusts memory allocation accordingly. When the amount of free memory falls below a certain threshold (which may vary between implementations), Immix triggers a GC cycle to reclaim unused memory. If the free space is still insufficient after collection, Immix may allocate additional memory blocks to meet the application's memory needs. Immix also considers the rate of object allocation as a metric. If the allocation rate exceeds a certain threshold, it indicates a high memory consumption and the potential need for more memory to avoid out-of-memory errors. Immix also uses heuristics to estimate the size of the working set or the set of objects that are actively being used by the application. Since Immix is a STW collector, this dynamic heap resizing brings many disadvantages. For example, the application may experience brief pauses or slowdowns during the resizing process, which in turn makes it more difficult to reason about the memory usage and performance characteristics of an application.

Cheng et al.[CB01] introduce a parallel, concurrent, real-time garbage collector for multi-processors. GC work is proportional to the allocation rate, so it indirectly scales up and down with program CPU utilisation. It aims to provide bounds on pause times for GC while also scaling well across multiple processors. Using the concept of Minimum Mutator Utilisation (MMU), they capture the percentage of time in a given time window the mutators have access to the CPU. They showed that their proposed collector keeps higher MMU results compared to non-incremental GC. However, they do not assess GC CPU or utilize MMU-based actions.

Degenbaev et al. [Deg+16] propose scheduling GC during detected idle periods in the application to reduce GC latency. It uses knowledge of idle times from Chrome's scheduler to opportunistically schedule different GC tasks like minor collections and incremental marking. This allows adapting GC based on real-time application behavior and available idle cycles. While not directly adjusting heap size, scheduling GC during idle periods allows for reducing memory usage and footprint when the application becomes inactive and based on the real-time needs of the application.

The G1 [Det+04] (Garbage First) garbage collector requires knowledge of the maximum memory needed for an application in advance. If it is not explicitly provided, it uses a default value. G1 uses a dynamic heap size adjustment strategy to adjust the memory usage during runtime based on the current usage pattern of the application [Ora21]. G1 divides the heap into regions of equal size and groups them into two generations: young and old. When the young generation fills up, G1 performs a young collection, during which live objects are copied to a new region while unused regions are reclaimed. G1 also performs periodic concurrent marking of live objects in the old generation. When the old generation fills up, G1 performs a mixed collection, which collects both young and old regions that have been marked as garbage. During a mixed collection,

G1 dynamically sizes the heap by using the occupancy of the old generation as a target and adjusts the heap size to meet that target.

Heap size adjustment in .NET is difficult because of the prevalence of object pinning which can make it impossible to uncommit memory. .NET offers a ConserveMemory interface to the garbage collector that allows "conserving memory at the expense of more frequent garbage collections and possibly longer pause times" [War+23]. This setting works by controlling the fragmentation tolerance in old generations, before triggering a full, compacting GC cycle.

III.7.3 Discussion

Previous works, adjust the amount of heap size by estimating the amount of memory that is necessary to keep the application running without incurring high latency and CPU overheads. Instead of estimating the amount of memory needed by the application, we adjust the heap size to meet a specific GC target. Our CPU-driven heap size adjustment is particularly important for concurrent collectors like ZGC, which compete with the mutator for CPU resources to collect memory, unlike the STW collectors used in prior studies. Dissimilar to fixed-size heap headroom used in STW collectors, a concurrent GC requires variable headroom depending on the available CPU for collection. If the mutator consumes most of the CPU, a large headroom is necessary for a concurrent collector, while a small headroom suffices for collection when the mutator has minimal CPU usage. In sum, rather than directly controlling the heap headroom as in previous works for STW collectors, we specify the desired GC target and adjust the heap headroom accordingly.

III.8 Conclusion

This paper explores an adaptive approach for automatically adjusting heap size based on CPU overhead for GC work as a tuning knob. Our evaluation demonstrates that this technique does not negatively impact latency, which is the main goal of fully concurrent collectors. In addition, we offer insights into optimizing energy and performance by tuning GC targets. Our ongoing work focuses on seamlessly integrating and refining this approach within the ZGC framework to unlock its full potential in real-world applications.

References

[07] *OpenJDk*. 2007. URL: https://openjdk.java.net/.

[App89] Appel, A. W. "Simple Generational Garbage Collection and Fast Allocation". In: *Softw. Pract. Exper.* vol. 19, no. 2 (Feb. 1989), pp. 171–183.

[Bas+21] Bashir, N. et al. "Take it to the limit: peak prediction-driven resource overcommitment in datacenters". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 556–573.

[Ber+22] Beronić, D. et al. “Assessing Contemporary Automated Memory Management in Java–Garbage First, Shenandoah, and Z Garbage Collectors Comparison”. In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1495–1500.

[BF18] Bruno, R. and Ferreira, P. “A study on garbage collection algorithms for big data environments”. In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. “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*. 2006, pp. 169–190.

[BM08] Blackburn, S. M. and McKinley, K. S. “Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 22–32.

[Bre+01] Brecht, T. et al. “Controlling garbage collection and heap growth to reduce the execution time of Java applications”. In: *ACM Sigplan Notices* vol. 36, no. 11 (2001), pp. 353–366.

[Bru+18] Bruno, R. et al. “Dynamic vertical memory scalability for OpenJDK cloud applications”. In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, pp. 59–70.

[BW88] Boehm, H.-J. and Weiser, M. “Garbage collection in an uncooperative environment”. In: *Software: Practice and Experience* vol. 18, no. 9 (1988), pp. 807–820.

[Cai+22] Cai, Z. et al. “Distilling the real cost of production garbage collectors”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 46–57.

[CB01] Cheng, P. and Blelloch, G. E. “A parallel, real-time garbage collector”. In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 2001, pp. 125–136.

[CML14] Chen, M., Mao, S., and Liu, Y. “Big data: A survey”. In: *Mobile networks and applications* vol. 19, no. 2 (2014), pp. 171–209.

[Deg+16] Degenbaev, U. et al. “Idle time garbage collection scheduling”. In: *ACM SIGPLAN Notices* vol. 51, no. 6 (2016), pp. 570–583.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Eva20] Evans, B. *What Tens of Millions of VMs Reveal about the State of Java*. Mar. 2020. URL: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[Fra+07] Frampton, D. et al. “Generational real-time garbage collection”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 101–125.

[FW86] Fleming, P. J. and Wallace, J. J. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Commun. ACM* vol. 29 (1986), pp. 218–221.

[Gen+21] Gencer, C. et al. “Hazelcast jet: Low-latency stream processing at the 99.99^{th} percentile”. In: vol. 14, no. 12 (2021), pp. 3110–3121.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. “Comparison of garbage collectors in Java programming language”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gru69] Grubbs, F. E. “Procedures for detecting outlying observations in samples”. In: *Technometrics* vol. 11, no. 1 (1969), pp. 1–21.

[Grz+07] Grzegorczyk, C. et al. “Isla vista heap sizing: Using feedback to avoid paging”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE. 2007, pp. 325–340.

[HB05] Hertz, M. and Berger, E. D. “Quantifying the performance of garbage collection vs. explicit memory management”. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 313–326.

[HM95] Han, J. and Moraga, C. “The influence of the sigmoid function parameters on the speed of backpropagation learning”. In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[Hor14] Horowitz, M. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[Int09] Intel. *Intel Architecture Software Developer’s Manual*. Vol. Volume 3: System Programming Guide. 2009.

[Int20] Intel. *Intel VTune Profiler Performance Analysis Cookbook: Top-down Microarchitecture Analysis Method*. Dec. 2020. URL: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[JHM12] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Jan. 2012.

[Jon96] Jones, R. E. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[Kar23] Karlsson, S. *JEP draft: Generational ZGC*. Jan. 2023. URL: https://openjdk.org/jeps/8272979.

[KSP22] Kirisame, M., Shenoy, P., and Panchekha, P. "Optimal heap limits for reducing browser memory use". In: *Proceedings of the ACM on Programming Languages* vol. 6, no. OOPSLA2 (2022), pp. 986–1006.

[LH83] Lieberman, H. and Hewitt, C. "A real-time garbage collector based on the lifetimes of objects". In: *Communications of the ACM* vol. 26, no. 6 (1983), pp. 419–429.

[NÖW22] Norlinder, J., Österlund, E., and Wrigstad, T. "Compressed Forwarding Tables Reconsidered". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. 2022, pp. 45–63.

[Ope22] OpenStack. *Overcommitting CPU and RAM*. Sept. 2022. URL: https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.

[Ora20] Oracle. *The Parallel Collector*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/parallel-collector1.html#GUID-74BE3BC9-C7ED-4AF8-A202-793255C864C4.

[Ora21] Oracle. *Garbage-First Garbage Collector Tuning*. 2021. URL: https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-garbage-collector-tuning.html#GUID-3D3E4662-1E89-42EE-96FA-836C0E7C97AA.

[Oss+02] Ossia, Y. et al. "A parallel, incremental and concurrent GC for servers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[Per18a] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[Per18b] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. "Analysis of garbage collection algorithms and memory management in Java". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. “Stopless: a real-time garbage collector for multi-processors”. In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. “A study of concurrent real-time garbage collectors”. In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. “Renaissance: benchmarking suite for parallel applications on the JVM”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sah+16] Sahin, S. et al. “Jvm configuration management and its performance impact for big data applications”. In: *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE. 2016, pp. 410–417.

[SPW22] Shimchenko, M., Popov, M., and Wrigstad, T. “Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK”. In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR ’22. Brussels, Belgium: Association for Computing Machinery, 2022, pp. 3–15.

[Tav20] Tavakolisomeh, S. “Selecting a JVM Garbage Collector for Big Data and Cloud Services”. In: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*. 2020, pp. 22–25.

[Top20] Topolnik, M. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*. June 2020. URL: https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch.

[UJ88] Ungar, D. and Jackson, F. “Tenuring policies for generation-based storage reclamation”. In: *ACM SIGPLAN Notices* vol. 23, no. 11 (1988), pp. 1–17.

[War+23] Warren, G. et al. *Runtime configuration options for garbage collection*. Feb. 2023. URL: https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector#conserve-memory.

[Wel38] Welch, B. L. “The significance of the difference between two means when the population variances are unequal”. In: *Biometrika* vol. 29, no. 3/4 (1938), pp. 350–362.

[Whi+13] White, D. R. et al. “Control theory for principled heap sizing”. In: *ACM SIGPLAN Notices* vol. 48, no. 11 (2013), pp. 27–38.

[Yan+04] Yang, T. et al. “Automatic heap sizing: Taking real memory into account”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 61–72.

[YÖW20a] Yang, A. M., Österlund, E., and Wrigstad, T. “Improving Program Locality in the GC Using Hotness”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[YÖW20b] Yang, A. M., Österlund, E., and Wrigstad, T. “Improving Program Locality in the GC Using Hotness”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[Yue74] Yuen, K. K. “The two-sample trimmed t for unequal population variances”. In: *Biometrika* vol. 61, no. 1 (1974), pp. 165–170.

[YW22] Yang, A. M. and Wrigstad, T. “Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 44, no. 4 (2022), pp. 1–34.

[ZB20] Zhao, W. and Blackburn, S. M. “Deconstructing the garbage-first collector”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

[ZBM22] Zhao, W., Blackburn, S. M., and McKinley, K. S. “Low-latency, high-throughput garbage collection”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 76–91.

[Zha+06] Zhang, C. et al. “Program-level adaptive memory management”. In: *Proceedings of the 5th international symposium on Memory management*. 2006, pp. 174–183.

Other Publications

Other Publications I

Selecting a JVM Garbage Collector for Big Data and Cloud Services

I

Sanaz Tavakolisomeh

Published in *Proceedings of the 21st International Middleware Conference Doctoral Symposium*, pp. 22–25, December 2020, Virtual Event / Delft, The Netherlands.

Abstract

Memory management is responsible for allocating and releasing the memory used by applications (e.g., in the Java Virtual Machine-JVM). There are several garbage collectors (GCs) each designed to target different performance metrics, making it very hard for developers to decide which GC to use for a particular application. We start with a review of existing GC algorithms. Then, we intend to evaluate throughput, pause time, and memory usage in existing JVM GCs using benchmark suites like DaCapo and Renaissance. The goal is to find the trade-offs between the above mentioned performance metrics to have a better understanding of which GC helps fulfill certain application requirements.

Contents

I.1 Introduction

An important feature of memory management is the GC, which is responsible for automatically collecting unused objects in memory. GC plays an important role especially when applications process, store, and analyze massive amounts of data (i.e., Big Data and/or Cloud Services [CML14]).

Several GC algorithms try to optimize a combination of different performance metrics: throughput, pause time, and memory usage. This makes it hard to choose the best GC for a particular application. In this paper, we investigate

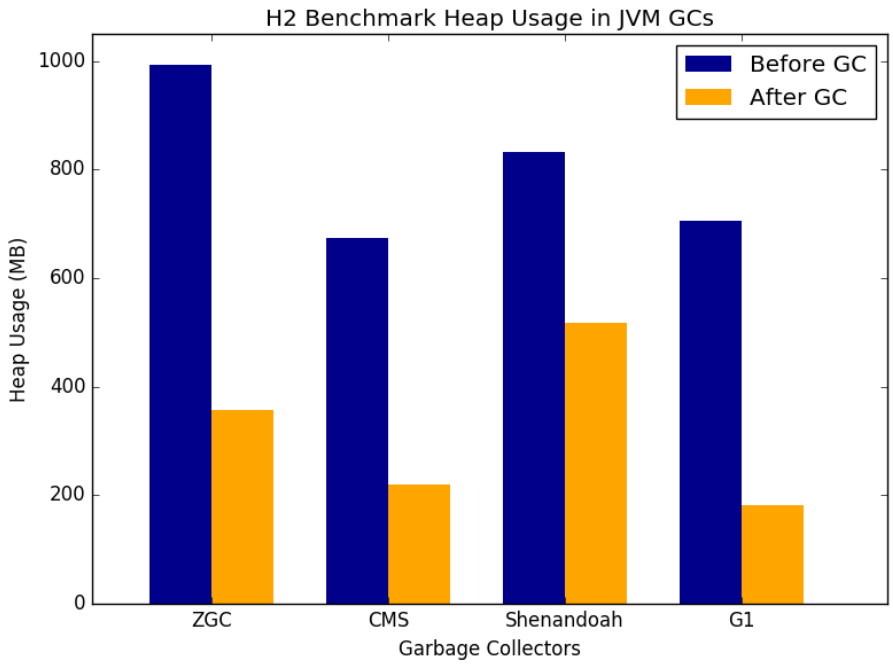


Figure I.1: Average Memory usage before and after GC in H2 benchmark

performance metrics in the most relevant GCs provided in OpenJDK HotSpot JVM [07], the most widely used JVM implementation. As depicted in Figure I.1, choosing different GC algorithms can have a significant impact on memory usage for the same exact application with the same heap limits. In this particular example, H2 benchmark (included in DaCapo benchmark suite) leads to more than 2x increased memory utilization after GC when Shenandoah [Flo+16] is utilized compared to G1 [Det+04] or CMS [Jon96].

Figure I.2 represents how different GCs perform in terms of pause time in the akka-uct benchmark (included in Renaissance benchmark suit). While ZGC [Per18a] handled pause times significantly better than other GCs, Shenandoah decreased pause time more than 3x in comparison with G1 and CMS.

We use benchmark suites (DaCapo [Bla+06] and Renaissance [Pro+19]) which provide various programs in different contexts. All these help us to provide further information on which GC should be selected if a particular performance metric is targeted and the trade-offs involved.

The rest of the paper presents a description of the GC algorithms we use (Section I.2), the architecture of our solution (Section I.3) and finally a conclusion (Section I.4).

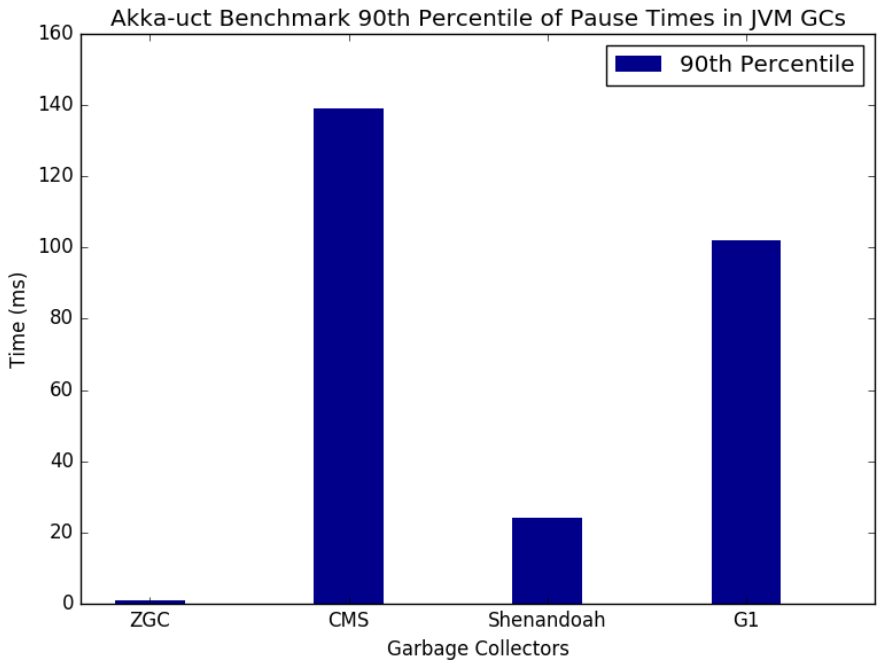


Figure I.2: 90^{th} percentile of pause times in akka-uct benchmark

I.2 Related Work

There are many GC algorithms, with different designs and implementations. All the GCs consider **throughput**, **pause time**, and **memory usage** as crucial performance metrics. Throughput indicates the proportion of time or memory spent in the application process, not the garbage collection. Pause time is the time an application must stop to let the GC execute. Finally, the amount of memory used in a process, in which GC is responsible for evacuating dead objects, indicates memory usage.

GC algorithms use different approaches to collect dead objects and manage the heap. To do so, the heap is partitioned into multiple partitions or sub-heaps [UJ88]. In the most well-known partitioning technique, objects are segregated based on their estimated lifetime[UJ88]. This results in generational GCs that can have a young generation containing newly created objects and an old generation that contains objects that survived previous garbage collections.

In the garbage collection process, some GCs use the stop-the-word (STW) technique. This technique stops the application process, collects the dead objects, and evacuates live objects to other parts of the heap space. Another approach is a concurrent GC in which there are multiple threads to run the application and GC code. So, garbage collection happens while the application is running and creating new objects in the heap.

The aforementioned approaches try to find live objects in an object tree (starting from the root) and do the evacuation of live objects either through copy or compaction. Copying allows an application to group live objects by moving them from multiple memory segments into a single segment. Thus, live objects will not be scattered across the heap. Compaction moves all live objects to the start of the memory segment when the cost of copying objects to another segment is high for large amounts of live objects [BF18].

There are many GC algorithms that can be used in the JVM, ranging from the most commonly used older GCs, including Concurrent Mark and Sweep (CMS) GC, and Garbage-First (G1) GC, up to the latest experimental ones like ZGC, and Shenandoah GC [PGM19]. In the following, we briefly explain some existing GC solutions.

The **Concurrent Mark/Sweep collector (CMS)** is a generational GC. It uses a monolithic STW copying collector to manage all the objects in the young generation. The old generation is managed by a mostly concurrent mark and sweep collector without compaction. When objects can no longer be promoted to the old generation or concurrent marking fails, it falls back to a monolithic STW compaction of the old generation. CMS can cause memory fragmentation and use more processing power than any other GCs [GMR18].

The **Garbage-First (G1)** is a fixed-sized, concurrent, and generational GC, that groups all objects whose life cycles are similar in separate regions. It uses a STW copying approach to manage the young generation. For the old generation, it uses a mostly concurrent marker, while applying STW pauses to catch up on application changes and reference processing [ZB20].

The **Z Garbage Collector (ZGC)** is an experimental scalable low-latency GC. Its main goal is that its pause time does not increase with the heap or live-set size. It handles heaps ranging from relatively small to huge multi-terabyte sizes. There is no generational separation of the heap; the heap is internally divided into many small regions, and we can choose to compact a subset of those, typically one that contains the most garbage. ZGC uses "colored pointers" to store some important metadata to hold information about the object itself or about the object that it points to, and marking and relocation-related information [PGM19].

Every object has an additional reference field (a forward pointer) that points to the object itself, or, as soon as the object gets moved to a new location, to that new location [BF18].

I.3 Architecture of the Solution

Choosing the best GC solution is an essential part of memory management, mainly when dealing with massive amounts of objects in memory (i.e., Big Data environments and Cloud Services). So, we must know application requirements regarding throughput, latency, and memory usage.

We intend to use OpenJDK 13 to evaluate the G1, Shenandoah, ZGC, and CMS using DaCapo and Renaissance benchmark suites. In addition, we will

also develop a small application to investigate performance metrics in GCs. We run our application and benchmarks included in both DaCapo and Renaissance on the same machine with the same configuration. Our application occupies the heap with different amounts of read and write operations and applies one of the previously mentioned GCs at a time for each instance; meanwhile, records the GC logs and the throughput of the application. Furthermore, the result set from several executions of benchmarks in DaCapo and Renaissance using their maximum input size plus the GCs' log files helps us to calculate the desired performance metrics.

To define throughput, we use the execution time to make it comparable with two other benchmarks which do not provide us with other metrics like the number of operations during the application's execution time. In the case of pause time, it matters how long it takes for an application to get input and respond to it. Using different GCs because of the various approaches they apply for garbage collection, directly affects the pause time. We extract pause times from GCs' log files and calculate the 90^{th} percentile of them. This will represent the frequent pause time that happened during garbage collection. Regarding memory usage, it is important to consider the amount of memory an application is allowed to use and shall not exceed that amount. We measure the average heap usage before and after garbage collection happens. Then, we measure how much a specific GC aims to reduce heap usage.

Specifying the characteristics of a given application is another significant step in choosing the best GC solution. For example, we must determine whether an application is CPU-intensive or I/O intensive (also including networking) and which of the throughput, pause time, or memory usage are more essential to consider accordingly. Moreover, the main requirement of an application is an important factor that directly prioritizes GC solutions, e.g., if high responsiveness is expected for an application, a GC that performs better in terms of pause time should be selected. The results obtained will allow us to find the best-matched GC solution that makes a trade-off between the desired performance metrics and specific application requirements.

Once we have done the evaluation of the GCs, we intend to develop an application that, based on application requirements and characteristics, will help the user to decide which GC to use. Then, we also plan to evaluate the possibility of automatizing this process requiring no intervention from the user.

I.4 Conclusion

Due to recent developments, new GC algorithms providing efficient memory utilization, ultra-low latency, and high throughput have become more complex. Thus, selecting the correct GC for a particular application, especially in big data and cloud environments, is not trivial since each GC provides different trade-offs. In our work, we study how current GCs perform regarding specific metrics, in particular, pause time, throughput, and memory usage, to better understand their trade-offs. We use widely used benchmarks like DaCapo and

Renaissance benchmark suites to measure performance metrics in GC solutions. In addition, we also developed an application to examine different GCs in JVM and investigate the desired metrics. Knowing the characteristics of a given application alongside the result from our experiences help us to propose the best GC solution for an application.

Acknowledgments

Thanks to my supervisors, Prof. Paulo Ferreira (University of Oslo) and Dr. Rodrigo Bruno (Oracle Labs Zurich) for their great support, advice, and encouragement during this project.

References

[07] *OpenJDk*. 2007. URL: https://openjdk.java.net/.

[App89] Appel, A. W. "Simple Generational Garbage Collection and Fast Allocation". In: *Softw. Pract. Exper.* vol. 19, no. 2 (Feb. 1989), pp. 171–183.

[Bas+21] Bashir, N. et al. "Take it to the limit: peak prediction-driven resource overcommitment in datacenters". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 556–573.

[Ber+22] Beronić, D. et al. "Assessing Contemporary Automated Memory Management in Java–Garbage First, Shenandoah, and Z Garbage Collectors Comparison". In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1495–1500.

[BF18] Bruno, R. and Ferreira, P. "A study on garbage collection algorithms for big data environments". In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. "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*. 2006, pp. 169–190.

[BM08] Blackburn, S. M. and McKinley, K. S. "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance". In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 22–32.

[Bre+01] Brecht, T. et al. "Controlling garbage collection and heap growth to reduce the execution time of Java applications". In: *ACM Sigplan Notices* vol. 36, no. 11 (2001), pp. 353–366.

[Bru+18] Bruno, R. et al. "Dynamic vertical memory scalability for OpenJDK cloud applications". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, pp. 59–70.

[BW88] Boehm, H.-J. and Weiser, M. “Garbage collection in an uncooperative environment”. In: *Software: Practice and Experience* vol. 18, no. 9 (1988), pp. 807–820.

[Cai+22] Cai, Z. et al. “Distilling the real cost of production garbage collectors”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 46–57.

[CB01] Cheng, P. and Blelloch, G. E. “A parallel, real-time garbage collector”. In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 2001, pp. 125–136.

[CML14] Chen, M., Mao, S., and Liu, Y. “Big data: A survey”. In: *Mobile networks and applications* vol. 19, no. 2 (2014), pp. 171–209.

[Deg+16] Degenbaev, U. et al. “Idle time garbage collection scheduling”. In: *ACM SIGPLAN Notices* vol. 51, no. 6 (2016), pp. 570–583.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Eva20] Evans, B. *What Tens of Millions of VMs Reveal about the State of Java*. Mar. 2020. URL: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[Fra+07] Frampton, D. et al. “Generational real-time garbage collection”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 101–125.

[FW86] Fleming, P. J. and Wallace, J. J. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Commun. ACM* vol. 29 (1986), pp. 218–221.

[Gen+21] Gencer, C. et al. “Hazelcast jet: Low-latency stream processing at the 99.99^{th} percentile”. In: vol. 14, no. 12 (2021), pp. 3110–3121.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. “Comparison of garbage collectors in Java programming language”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gru69] Grubbs, F. E. “Procedures for detecting outlying observations in samples”. In: *Technometrics* vol. 11, no. 1 (1969), pp. 1–21.

[Grz+07] Grzegorczyk, C. et al. “Isla vista heap sizing: Using feedback to avoid paging”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE. 2007, pp. 325–340.

[HB05] Hertz, M. and Berger, E. D. “Quantifying the performance of garbage collection vs. explicit memory management”. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 313–326.

[HM95] Han, J. and Moraga, C. “The influence of the sigmoid function parameters on the speed of backpropagation learning”. In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[Hor14] Horowitz, M. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[Int09] Intel. *Intel Architecture Software Developer’s Manual*. Vol. Volume 3: System Programming Guide. 2009.

[Int20] Intel. *Intel VTune Profiler Performance Analysis Cookbook: Top-down Microarchitecture Analysis Method*. Dec. 2020. URL: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[JHM12] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Jan. 2012.

[Jon96] Jones, R. E. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[Kar23] Karlsson, S. *JEP draft: Generational ZGC*. Jan. 2023. URL: https://openjdk.org/jeps/8272979.

[KSP22] Kirisame, M., Shenoy, P., and Panchekha, P. “Optimal heap limits for reducing browser memory use”. In: *Proceedings of the ACM on Programming Languages* vol. 6, no. OOPSLA2 (2022), pp. 986–1006.

[LH83] Lieberman, H. and Hewitt, C. “A real-time garbage collector based on the lifetimes of objects”. In: *Communications of the ACM* vol. 26, no. 6 (1983), pp. 419–429.

[NÖW22] Norlinder, J., Österlund, E., and Wrigstad, T. “Compressed Forwarding Tables Reconsidered”. In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. 2022, pp. 45–63.

[Ope22] OpenStack. *Overcommitting CPU and RAM*. Sept. 2022. URL: https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.

[Ora20] Oracle. *The Parallel Collector*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/parallel-collector1.html#GUID-74BE3BC9-C7ED-4AF8-A202-793255C864C4.

[Ora21] Oracle. *Garbage-First Garbage Collector Tuning*. 2021. URL: https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-garbage-collector-tuning.html#GUID-3D3E4662-1E89-42EE-96FA-836C0E7C97AA.

[Oss+02] Ossia, Y. et al. "A parallel, incremental and concurrent GC for servers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[Per18a] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[Per18b] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. "Analysis of garbage collection algorithms and memory management in Java". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. "Stopless: a real-time garbage collector for multi-processors". In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. "A study of concurrent real-time garbage collectors". In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. "Renaissance: benchmarking suite for parallel applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sah+16] Sahin, S. et al. "Jvm configuration management and its performance impact for big data applications". In: *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE. 2016, pp. 410–417.

[SPW22] Shimchenko, M., Popov, M., and Wrigstad, T. "Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR '22. Brussels, Belgium: Association for Computing Machinery, 2022, pp. 3–15.

[Tav20] Tavakolisomeh, S. "Selecting a JVM Garbage Collector for Big Data and Cloud Services". In: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*. 2020, pp. 22–25.

[Top20] Topolnik, M. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*. June 2020. URL: https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch.

[UJ88] Ungar, D. and Jackson, F. "Tenuring policies for generation-based storage reclamation". In: *ACM SIGPLAN Notices* vol. 23, no. 11 (1988), pp. 1–17.

[War+23] Warren, G. et al. *Runtime configuration options for garbage collection*. Feb. 2023. URL: https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector#conserve-memory.

[Wel38] Welch, B. L. "The significance of the difference between two means when the population variances are unequal". In: *Biometrika* vol. 29, no. 3/4 (1938), pp. 350–362.

[Whi+13] White, D. R. et al. "Control theory for principled heap sizing". In: *ACM SIGPLAN Notices* vol. 48, no. 11 (2013), pp. 27–38.

[Yan+04] Yang, T. et al. "Automatic heap sizing: Taking real memory into account". In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 61–72.

[YÖW20a] Yang, A. M., Österlund, E., and Wrigstad, T. "Improving Program Locality in the GC Using Hotness". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[YÖW20b] Yang, A. M., Österlund, E., and Wrigstad, T. "Improving Program Locality in the GC Using Hotness". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[Yue74] Yuen, K. K. "The two-sample trimmed t for unequal population variances". In: *Biometrika* vol. 61, no. 1 (1974), pp. 165–170.

[YW22] Yang, A. M. and Wrigstad, T. "Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 44, no. 4 (2022), pp. 1–34.

[ZB20] Zhao, W. and Blackburn, S. M. "Deconstructing the garbage-first collector". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

[ZBM22] Zhao, W., Blackburn, S. M., and McKinley, K. S. "Low-latency, high-throughput garbage collection". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 76–91.

[Zha+06] Zhang, C. et al. “Program-level adaptive memory management”. In: *Proceedings of the 5th international symposium on Memory management*. 2006, pp. 174–183.

Other Publications II

BestGC: An Automatic GC Selector Software

Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira

Published in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR)*, pp. 144–146, September 2022, Brussels, Belgium.

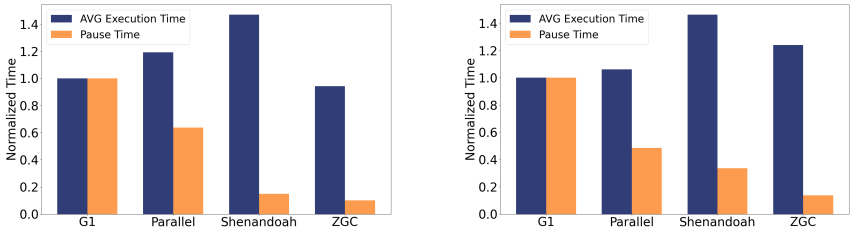
Abstract

Garbage collection (GC) solutions are widely used in programming languages like Java. Such GC solutions follow prioritized goals and behave differently regarding crucial performance metrics like pause time, throughput, and memory usage. Consequently, running an application using each GC would have an evident impact on the application's performance, especially on those dealing with massive data and tons of transactions. Nevertheless, it is challenging for a user/developer to pick a GC solution that fits an application's performance goals. However, surprisingly, there is no tool to help a user/developer on this matter. In this study, we follow an effective methodology to build a heuristic combining throughput and pause time, with diverse heap sizes available, to score four production GCs (G1, Parallel, Shenandoah, and ZGC). Then we propose a system, BestGC, which offers the most suitable GC solution for a user application taking into account the performance goals of the user application.

Contents

II.1 Introduction

A significant portion of today's applications use managed runtime languages like Java and, therefore, take advantage of automatic memory management, also commonly known as Garbage Collection (GC). GC is a key attribute in managed runtimes as it eliminates developers' effort to manually allocate and deallocate the objects in memory, thus improving developer productivity.



(a) heap size = 4096 MB (b) heap size = 256 MB

Figure II.1: Normalized average execution time and normalized average 90$^{\text{th}}$ percentile of GC pause times for Philosopher workload. Lower is better.

Several GC algorithms are available in different Java versions attempting to improve performance metrics like throughput, pause time, and memory footprint. Figures II.1a and II.1b show the average execution time (which we use to report throughput) and 90th percentile of pause times (normalized by G1) for the Philosopher workload (from Renaissance [Pro+19] benchmark suite) with heap sizes of 4096MB and 256MB. Results demonstrate that GC algorithms behave differently considering throughput and pause time. ZGC performs significantly better than other garbage collectors regarding pause time in both heap sizes, while it sacrifices throughput when heap size is reduced to 256MB. However, G1 outperformed other collectors considering execution time with a 256MB heap.

As such, GC algorithms may significantly affect application performance goals, especially those dealing with large data (which may focus on throughput) and substantial transaction rates (which may focus on pause time). However, choosing an algorithm that best suits an application's performance demands is hard for a user/developer who might not be a GC expert.

Due to its importance, extensive research analyzed, characterized, and proposed new GC algorithms [Fra+07; Oss+02; Piz+07; PPS08]. Several studies also compared GC solutions regarding performance metrics like throughput, pause time, and memory usage [Ber+22; GMR18; Jon96; PGM19; Tav20; ZB20; ZBM22]. As these performance metrics are the most common, we consider them to be the most critical metrics that highly affect the user's applications. However, there is still no tool to help users and developers to overcome the challenges of selecting a suitable GC solution for their applications. Also, to the best of our knowledge, other studies have not proposed a performance-based approach to score recent collectors and, accordingly, a system to offer a proper GC.

Thus, the goal of this work is to propose a solution and a system that automatically determines the most suitable GC according to users' preferences (regarding pause time and throughput) and use it right away to run the user's application.

This study follows a simple yet effective methodology to compare and analyze recent collectors' throughput, pause time, and memory usage. We do so while

aiming at the following goals:

- evaluating four well-known collectors in a production JVM implementation considering execution time and pause time, while the memory available to collectors changes;
- conducting experiments and drawing results based on a set of workloads, including CPU-intensive and latency-sensitive ones, that represent existing real-world applications to reveal the costs of the collectors; and
- proposing an approach to recommend a suitable collector for a user's application while letting the user decide on his performance priority.

We evaluate G1 [Det+04], ZGC [Per18b], Parallel [Ora20], and Shenandoah [Flo+16], the most relevant GC implementations available in OpenJDK 15. We use workloads from two widely-used benchmark suites, DaCapo [Bla+06] and Renaissance [Pro+19]. These two benchmarks include diverse workloads that mostly need CPU, which is also a critical resource for GC. Moreover, using a specific version of DaCapo (DaCapo-Chopin) provides us with latency-sensitive workloads in which requests will be queued when the latency increases. Thus, it allows us to measure also the concurrency overhead of ZGC and Shenandoah on our performance metrics [Cai+22].

We measure throughput and pause time due to their undeniable importance while providing different heap sizes for the collectors. Modifying the heap size allows us to investigate collectors' behavior when they have to scan a large heap area to find the unused objects; or when they are trying to manage a small heap to deliver high throughput while imposing low pause times.

Thus, our main contribution is developing a system, BestGC, using the results obtained from our extensive evaluations; it frees the user from the complicated process of selecting a GC that fits his application's performance goals. BestGC asks for the user's preferences for throughput (w_t) and pause time (w_p) and afterward runs the user's application with the collector that best matches the desired performance goals:

$$w_p \in [0, 1]$$
$$w_t \in [0, 1]$$
$$w_p + w_t = 1$$

BestGC captures the max heap size used by the application and accordingly scores the four collectors. It uses results it maintains for each heap configuration, based on the experiments, in matrices. Each matrix holds the values corresponding to the GCs in the rows and throughput and pause time in the columns. Therefore, the calculation is done using a formula:

$$\begin{aligned} score_{gc} = w_p \times matrix[heap]_{<gc,PauseTime>} \\ + w_t \times matrix[heap]_{<gc,Throughput>} \end{aligned} \tag{II.1}$$

$$heap \in \{8192, 4096, 2048, 1024, 512, 256\}$$

$$gc \in \{G1, Parallel, Shenandoah, ZGC\}$$

Finally, BestGC picks the GC with the best score (a lower score is better) and executes the user's application with the selected GC and suggested heap size.

References

[07] *OpenJDk*. 2007. URL: https://openjdk.java.net/.

[App89] Appel, A. W. "Simple Generational Garbage Collection and Fast Allocation". In: *Softw. Pract. Exper.* vol. 19, no. 2 (Feb. 1989), pp. 171–183.

[Bas+21] Bashir, N. et al. "Take it to the limit: peak prediction-driven resource overcommitment in datacenters". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 556–573.

[Ber+22] Beronić, D. et al. "Assessing Contemporary Automated Memory Management in Java–Garbage First, Shenandoah, and Z Garbage Collectors Comparison". In: *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2022, pp. 1495–1500.

[BF18] Bruno, R. and Ferreira, P. "A study on garbage collection algorithms for big data environments". In: *ACM Computing Surveys (CSUR)* vol. 51, no. 1 (2018), pp. 1–35.

[Bla+06] Blackburn, S. M. et al. "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*. 2006, pp. 169–190.

[BM08] Blackburn, S. M. and McKinley, K. S. "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance". In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 22–32.

[Bre+01] Brecht, T. et al. "Controlling garbage collection and heap growth to reduce the execution time of Java applications". In: *ACM Sigplan Notices* vol. 36, no. 11 (2001), pp. 353–366.

[Bru+18] Bruno, R. et al. "Dynamic vertical memory scalability for OpenJDK cloud applications". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*. 2018, pp. 59–70.

[BW88] Boehm, H.-J. and Weiser, M. "Garbage collection in an uncooperative environment". In: *Software: Practice and Experience* vol. 18, no. 9 (1988), pp. 807–820.

[Cai+22] Cai, Z. et al. “Distilling the real cost of production garbage collectors”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 46–57.

[CB01] Cheng, P. and Blelloch, G. E. “A parallel, real-time garbage collector”. In: *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 2001, pp. 125–136.

[CML14] Chen, M., Mao, S., and Liu, Y. “Big data: A survey”. In: *Mobile networks and applications* vol. 19, no. 2 (2014), pp. 171–209.

[Deg+16] Degenbaev, U. et al. “Idle time garbage collection scheduling”. In: *ACM SIGPLAN Notices* vol. 51, no. 6 (2016), pp. 570–583.

[Det+04] Detlefs, D. et al. “Garbage-first garbage collection”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[Eva20] Evans, B. *What Tens of Millions of VMs Reveal about the State of Java*. Mar. 2020. URL: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[Flo+16] Flood, C. H. et al. “Shenandoah: An open-source concurrent compacting garbage collector for openjdk”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016, pp. 1–9.

[Fra+07] Frampton, D. et al. “Generational real-time garbage collection”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 101–125.

[FW86] Fleming, P. J. and Wallace, J. J. “How not to lie with statistics: the correct way to summarize benchmark results”. In: *Commun. ACM* vol. 29 (1986), pp. 218–221.

[Gen+21] Gencer, C. et al. “Hazelcast jet: Low-latency stream processing at the 99.99^{th} percentile”. In: vol. 14, no. 12 (2021), pp. 3110–3121.

[GMR18] Grgic, H., Mihaljević, B., and Radovan, A. “Comparison of garbage collectors in Java programming language”. In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2018, pp. 1539–1544.

[Gru69] Grubbs, F. E. “Procedures for detecting outlying observations in samples”. In: *Technometrics* vol. 11, no. 1 (1969), pp. 1–21.

[Grz+07] Grzegorczyk, C. et al. “Isla vista heap sizing: Using feedback to avoid paging”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE. 2007, pp. 325–340.

[HB05] Hertz, M. and Berger, E. D. "Quantifying the performance of garbage collection vs. explicit memory management". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 313–326.

[HM95] Han, J. and Moraga, C. "The influence of the sigmoid function parameters on the speed of backpropagation learning". In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201.

[Hor14] Horowitz, M. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[Int09] Intel. *Intel Architecture Software Developer's Manual*. Vol. Volume 3: System Programming Guide. 2009.

[Int20] Intel. *Intel VTune Profiler Performance Analysis Cookbook: Top-down Microarchitecture Analysis Method*. Dec. 2020. URL: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[JHM12] Jones, R., Hosking, A., and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Jan. 2012.

[Jon96] Jones, R. E. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[Kar23] Karlsson, S. *JEP draft: Generational ZGC*. Jan. 2023. URL: https://openjdk.org/jeps/8272979.

[KSP22] Kirisame, M., Shenoy, P., and Panchekha, P. "Optimal heap limits for reducing browser memory use". In: *Proceedings of the ACM on Programming Languages* vol. 6, no. OOPSLA2 (2022), pp. 986–1006.

[LH83] Lieberman, H. and Hewitt, C. "A real-time garbage collector based on the lifetimes of objects". In: *Communications of the ACM* vol. 26, no. 6 (1983), pp. 419–429.

[NÖW22] Norlinder, J., Österlund, E., and Wrigstad, T. "Compressed Forwarding Tables Reconsidered". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. 2022, pp. 45–63.

[Ope22] OpenStack. *Overcommitting CPU and RAM*. Sept. 2022. URL: https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.

[Ora20] Oracle. *The Parallel Collector*. 2020. URL: https://docs.oracle.com/en/java/javase/15/gctuning/parallel-collector1.html#GUID-74BE3BC9-C7ED-4AF8-A202-793255C864C4.

[Ora21] Oracle. *Garbage-First Garbage Collector Tuning*. 2021. URL: https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-garbage-collector-tuning.html#GUID-3D3E4662-1E89-42EE-96FA-836C0E7C97AA.

[Oss+02] Ossia, Y. et al. "A parallel, incremental and concurrent GC for servers". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 129–140.

[Per18a] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[Per18b] Per Liden, S. K. *ZGC: A Scalable Low-Latency Garbage Collector*. 2018. URL: https://openjdk.java.net/jeps/333.

[PGM19] Pufek, P., Grgić, H., and Mihaljević, B. "Analysis of garbage collection algorithms and memory management in Java". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1677–1682.

[Piz+07] Pizlo, F. et al. "Stopless: a real-time garbage collector for multi-processors". In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 159–172.

[PPS08] Pizlo, F., Petrank, E., and Steensgaard, B. "A study of concurrent real-time garbage collectors". In: *ACM SIGPLAN Notices* vol. 43, no. 6 (2008), pp. 33–44.

[Pro+19] Prokopec, A. et al. "Renaissance: benchmarking suite for parallel applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 31–47.

[Sah+16] Sahin, S. et al. "Jvm configuration management and its performance impact for big data applications". In: *2016 IEEE International Congress on Big Data (BigData Congress)*. IEEE. 2016, pp. 410–417.

[SPW22] Shimchenko, M., Popov, M., and Wrigstad, T. "Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR '22. Brussels, Belgium: Association for Computing Machinery, 2022, pp. 3–15.

[Tav20] Tavakolisomeh, S. "Selecting a JVM Garbage Collector for Big Data and Cloud Services". In: *Proceedings of the 21st International Middleware Conference Doctoral Symposium*. 2020, pp. 22–25.

[Top20] Topolnik, M. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*. June 2020. URL: https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch.

[UJ88] Ungar, D. and Jackson, F. “Tenuring policies for generation-based storage reclamation”. In: *ACM SIGPLAN Notices* vol. 23, no. 11 (1988), pp. 1–17.

[War+23] Warren, G. et al. *Runtime configuration options for garbage collection*. Feb. 2023. URL: https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector#conserve-memory.

[Wel38] Welch, B. L. “The significance of the difference between two means when the population variances are unequal”. In: *Biometrika* vol. 29, no. 3/4 (1938), pp. 350–362.

[Whi+13] White, D. R. et al. “Control theory for principled heap sizing”. In: *ACM SIGPLAN Notices* vol. 48, no. 11 (2013), pp. 27–38.

[Yan+04] Yang, T. et al. “Automatic heap sizing: Taking real memory into account”. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 61–72.

[YÖW20a] Yang, A. M., Österlund, E., and Wrigstad, T. “Improving Program Locality in the GC Using Hotness”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[YÖW20b] Yang, A. M., Österlund, E., and Wrigstad, T. “Improving Program Locality in the GC Using Hotness”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313.

[Yue74] Yuen, K. K. “The two-sample trimmed t for unequal population variances”. In: *Biometrika* vol. 61, no. 1 (1974), pp. 165–170.

[YW22] Yang, A. M. and Wrigstad, T. “Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 44, no. 4 (2022), pp. 1–34.

[ZB20] Zhao, W. and Blackburn, S. M. “Deconstructing the garbage-first collector”. In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 15–29.

[ZBM22] Zhao, W., Blackburn, S. M., and McKinley, K. S. “Low-latency, high-throughput garbage collection”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 76–91.

[Zha+06] Zhang, C. et al. “Program-level adaptive memory management”. In: *Proceedings of the 5th international symposium on Memory management*. 2006, pp. 174–183.