TÉCNICO LISBOA

BestGC++: Optimizing Garbage Collection Selection Through Benchmarking

Guilherme Luís Francisco Soares

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Prof. Paulo Jorge Pires Ferreira

Examination Committee

Chairperson: Prof. Nuno Miguel Carvalho dos Santos
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Rodrigo Fraga Barcelos Paulus Bruno

November 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I would like to thank Prof.Paulo Ferreira, for the opportunity to go develop my thesis abroad under his guidance. The encouragement and insights provided were instrumental to conclude this research.

I would also like to express my gratitude to Prof.Luís Veiga for the guidance and advice provided throughout the development of my thesis, since the first PIC draft. His attention to detail and diverse perspectives were essential in achieving the final goals of this work.

A special thanks goes to the University of Oslo for all the help provided during my time abroad. It really doesn't feel cold when people are always ready to help.

I would like to thank my family for their never-ending support throughout my academic journey, and honestly my entire life! And my friends-those who have been with me since the beginning, those I made over the past five years, and those I met during my Erasmus experience. Thank you for all the laughs, memories, and support. Without you, this journey would not have been nearly as meaningful or even possible.

Lastly, and most importantly, to my girlfriend: thank you for your unwavering support through all my ups and downs. Your love and encouragement have been essential in shaping the person I am today. You are an example to follow, and every day I strive to be more like you.

Abstract

The rapid adoption of cloud computing has transformed technology infrastructure, enabling service models such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and serverless Function-as-a-Service (FaaS). Alongside this shift, microservices architecture has become widely adopted, allowing applications to be built from independent services. Java is a popular language for developing these microservices, yet their distributed systems nature presents challenges such as communication complexity and system failures, requiring optimized runtime environments and efficient Garbage Collection (GC) strategies.

In order to overcome these challenges, we will explore microservices architecture and review old and modern GC algorithms like G1, Shenandoah, and ZGC, suitable for low latency environments, and able to meet Service Level Agreements (SLAs). We also explore BestGC, a Java profiling tool that selects the best GC for an application, which serves as a foundation for the solution developed in this research. Building on this background, the thesis introduces two profiling tools: BenchmarkGC and BestGC++. BenchmarkGC facilitates Java workload benchmarking, while BestGC++ refines the original BestGC into a web service, allowing users to run their application with the optimal GC with minimal effort. Experiments on GraalVM and HotSpot runtimes validated their effectiveness in identifying performance bottlenecks and selecting the best GC based on workload characteristics, making it a valuable tool for production environments.

Keywords

Garbage Collection, Benchmarks, Java Runtimes, Microservices

Resumo

A rápida adoção da computação em nuvem transformou a infraestrutura tecnológica, permitindo modelos de serviço como Infrastructure as a Service (IaaS), Platform as a Service (PaaS) e Serverless Function as a Service (FaaS). Juntamente com essa mudança, a arquitetura de microsserviços foi amplamente adotada, permitindo que aplicações sejam construídas a partir de serviços independentes. Java é uma linguagem popular para desenvolver esses microsserviços; no entanto, a natureza dos sistemas distribuídos apresenta desafios como a complexidade de comunicação e falhas de sistema, exigindo ambientes de execução otimizados e estratégias eficientes de Garbage Collection (GC).

Para superar esses desafios, exploraremos a arquitetura de microsserviços e algoritmos de GC - antigos e modernos. Exemplos de modernos são G1, Shenandoah e ZGC, adequados para ambientes de baixa latência e capazes de atender aos Service Level Agreements (SLAs). Também exploramos o BestGC, uma ferramenta de profiling em Java, que seleciona o melhor GC para uma aplicação, que serve como base para a solução desenvolvida nesta pesquisa. Com base nesse contexto, a tese apresenta duas ferramentas de profiling: BenchmarkGC e BestGC++. BenchmarkGC facilita a avaliação de Java workloads, enquanto BestGC++ refina o BestGC original em um serviço web, permitindo que os utilizadores executem aplicações com o GC ideal e um mínimo de esforço. Testes com GraalVM e HotSpot validaram a sua eficácia na identificação de descidas de desempenho e na seleção do melhor GC com base nas características da aplicação, tornando-o uma ferramenta valiosa para ambientes de produção.

Palavras Chave

Garbage Collection, Benchmarks, Ambientes de Execução Java, Microsserviços

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

- **AOT** - Ahead of time
- **EMM** - Elastic Memory Management
- **E2E** - End-to-End
- **G1** - Garbage First
- **GC** - Garbage Collector
- **iGC** - Incremental Garbage Collection
- **LC** - Latency Critical
- **NAPS** - NUMA-Aware Parallel Scavenge
- **NUMA** - Non-Uniform Memory Access
- **SLA** - Service Level Agreements
- **TLABs** - Thread Local Allocation Buffers
- **ZGC** - Z Garbage Collector

1

Introduction

Contents

Within the domain of modern technology, the adoption of cloud computing as the main paradigm instead of traditional on-premise infrastructures has changed the way individuals, businesses, and organizations interact with information technology. The concept of cloud computing is a model where computing resources, storage, and services are delivered over the Internet. This can have many advantages, such as the access to computing resources on demand and the absence of resource allocation needed to manage infrastructure.

The three main models of cloud computing are [1]:

1. Infrastructure as a Service (IaaS): The consumer has control over the software being run, the operating system, and the storage used but not the underlying infrastructure.

2. Platform as a Service (PaaS): The consumer deploys its application developed with specific libraries/tools given by the provider, having no control over the software, operating system, and storage used by the cloud provider.

3. Software as a Service (SaaS): The consumer has the ability to use the provider's application, having no control over the application or underlying cloud infrastructure.

1.1 Serverless Computing

In the past decade, Serverless Computing, a cloud computing model, has gained increased popularity. This model was made popular by Amazon with AWS Lambda [1] in 2014 and completely eliminates the need for the users to manage the underlying architecture where their code is deployed. Usually, this service is provided using a Function-as-a-Service model, although Container-as-a-Service is also becoming popular nowadays. To use FaaS, the user develops a set of discrete functions on a specific runtime language (most popular languages are supported), which will later be executed in response to events or triggers. Users are charged only by the time spent in code execution, which doesn't happen in previously mentioned models. Despite the advantages of flexibility, ease of use, and cost, there are still some challenges related to FaaS services, that will be discussed later.

1.2 Microservices

In this day and age, big corporations are either migrating or have already migrated from their monolith architecture - where the application is developed as a single unit - to a microservices architecture, where the application is divided into a set of small independent services, all communicating through a well-defined API. This methodology allows each service to be developed and scaled independently based on demand. It is worth mentioning that these services can have a serverless nature leveraging the FaaS model. Netflix is one of the most well-known examples of a company with this kind of infrastructure, having started their migration to microservices in 2008.[2]

One of the languages most widely used for microservices architectures is Java due to its stability, ease of development as a Garbage-collected language, and platform independence originated from the "write once run anywhere" [3] principle of the Java Virtual Machine (JVM). There are various JVM implementations used, with the Hotspot JVM and GraalVM being among the most popular.

[1] Introducing aws lambda (https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/), accessed: 07/01/2023.

[2] Completing the netflix cloud migration (https://about.netflix.com/en/news/completing-the-netflix-cloud-migration), accessed: 07/01/2023

[3] Sun Microsystems slogan for the Java platform

1.3 Shortcomings

As mentioned earlier, challenges associated with microservices can stem from the chosen architecture or the specific software used, the most common ones being:

Cold Starts: This phenomenon is present in serverless solutions. It's the delay that occurs between the initial function trigger and the infrastructure initialization/set-up to handle the first request to a function that has been idle or hasn't been recently executed.

GC delays: This occurs when microservices runtimes use a Garbage Collected language e.g., Java. Due to Garbage Collection cycles, applications may experience higher latency when handling multiple requests than usual, and, in extreme cases, may become totally unresponsive for some time. This is due to the computation resources being allocated to the garbage collection process.

Communication between microservices: As the number of microservices rises, the network complexity also rises. So it is possible to have simple requests being bottlenecked by a specific service within the network.

1.4 Goals

The exploration of microservices and their integration within modern software ecosystems serves as a foundation for identifying optimizations and innovative solutions. Our goals centre around optimizing these integrations, primarily through the utilization of one of the most prevalent microservices platforms, the JVM (Java Virtual Machine). This pursuit aligns with:

- Improving language runtimes with the intention of mitigating cold starts.
- Optimize the Garbage Collection process by means of improving the Garbage Collector algorithm; manipulating GC heuristics or even introducing software middleware to decrease resource usage and improve system performance.

1.5 Organization of the Document

Chapter 1 introduces the current technological landscape, discussing its limitations, advantages, and setting forth our research objectives. Following this, Chapter 2 delves into the foundational research on Microservices, Garbage Collectors, and a Garbage Collector selection tool which supports the development of our solution. The review of contemporary research continues Chapter 3, where we explore current approaches to address the challenges mentioned earlier. Building on this foundation, Chapter 4 details the two GC profiling tools developed during this work, explaining their architecture and functionality. Next, Chapter 5 presents the evaluation of these tools, along with an analysis of whether

their performance aligns with the expectations set out in previous chapters. The thesis concludes with Chapter 6, offering a summary of the work and proposing potential avenues for future research.

2

Background

Contents

In the upcoming subsections, our aim is to provide a comprehensive overview of the current research landscape related to Microservices and Garbage Collectors, providing insights into key developments and challenges in these fields.

2.1 Microservices

The appearance of the microservice architecture didn't appear as a completely new concept, with some authors defending that many ideas are related to an older concept - Service Oriented Architecture (SOA) - being characterized as "SOA done right" [2]. One of the earliest definitions of microservices architecture was made by Lewis and Fowler [1] in the year 2014 and since then its popularity has been increasing rapidly (Figure 2.1). They characterize this architecture as a way to modularize a single application into

[1] Microservices (https://martinfowler.com/articles/microservices.html), accessed: 07/01/2023.

Figure 2.1: Google search for ”microservice” [2]

multiple services, all of them easily deployable with lightweight communication mechanisms between them.

The increasing popularity of microservices can be related to many factors such as:

- Famous companies like Amazon, Netflix, and Uber are actively sharing their successful migration stories.
- More flexibility, autonomy, and overall speed of development due to the ability to assign different teams the responsibility for the development and deployment of each service.
- The increased number of tools enabling easier implementation of microservices and a better development experience e.g., DevOps tools, Chaos Engineering, Serverless Computing [3].
- Easier scaling due to its distributed nature.

Associated with the popularity increase many companies and organizations started to migrate their infrastructure giving rise to many articles providing case studies and surveys about microservices migration.

On one of these studies, *M. Villamizar et al.* worked with a company to develop an application in both monolith and microservice infrastructure, allowing them to arrive at multiple insights [4]. The application was designed to only implement the two most popular services of this company for simplicity and deployed on AWS. Looking at the architecture in Figure 2.2 we can see that migrating to a microservice infrastructure leads to some additional complexity due to the increase of distribution in the system, which is corroborated by the authors' findings: *“(...) we validated that microservices introduce many problems of distributed systems (failures, timeouts, distributed transactions, data federation, responsibility assignments, etc.)(...)”* [4]. On the other hand, the microservice approach offers more granular scaling, with

each WebServer being able to scale individually both horizontally and vertically. Despite not making use of the scaling possibilities the authors conclude that this would be a benefit, exploiting the pay-per-use payment of the AWS cloud model. The latency introduced by the increase of intra-service communication was not very significant, allowing for the microservice approach to achieve the service requirements with the added benefit of 17% less cost. Despite these positive results, the authors concluded that unless the number of users expected is in the order of hundreds of thousands, you shouldn't make a transition. This is mostly related to the complexity of the new cultural and development policies that the company would need to establish for a smooth deployment, scaling, and continuous delivery system.

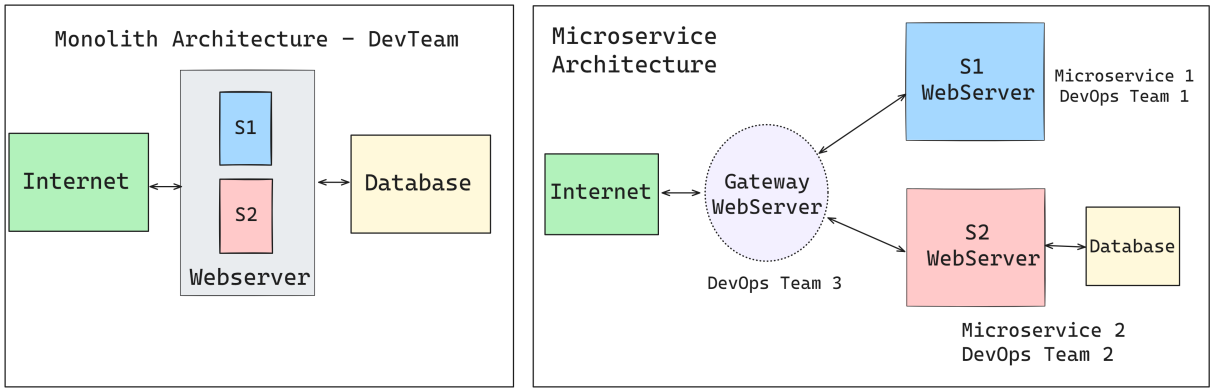

Figure 2.2: Monolith and Microservice architecture

Simultaneously, other studies are being made comparing a simple microservice architecture approach to a serverless architecture. *Shrestha and B. Nisha* did a case study implementing an Image Processing Application in both architectures using AWS resources [5]. The results demonstrated that the serverless architecture leveraging a Function-as-a-Service model had slightly worse performance (less than 10% discrepancy) in situations where the workload is constant or random. However, when the workload was incremental or spiked the serverless approach had the advantage due to faster scaling. The microservice architecture showed less memory consumption, while the serverless approach incurred 3.5 times less cost[3]. *M. Chadha et al.* also did a migration case study, however using different runtimes: Google Kubernetes Engine (GKE); Apache OpenWhisk on top of GKE; GCR (Google Cloud Run) [6]. In this instance, the findings were mostly in favor of the microservices architecture, which is, according to the authors, due to the workload being composed of simple API GET requests to a database.

Overall, studies on serverless architecture report a reduction w.r.t cost, however, they also mention the typical serverless problem - *cold starts* - showing multiple instances where it has negative effects with potential to break Service level agreements [5–8].

[2] https://www.google.com/trends/
[3] Extra care is needed when allocating serverless memory, due to being proportional to the cost

In summary, given the intricate nature of microservices infrastructure and its widespread adoption in the industry, it is logical to explore and create solutions tailored to these architectural paradigms.

2.2 Garbage Collectors

Before Garbage Collectors emerged, programmers had to manually manage their memory allocation and deallocation. This process led to multiple kinds of bugs including dangling pointers, double-free bugs, memory leaks, etc.

Nowadays, many of the most popular programming languages make use of some kind of Garbage Collector e.g., Java, C#, Go, Javascript, OCaml. The use of a language with automatic memory management removes the burden of handling the application memory from the programmer to the language runtime, simplifying the development process and reducing overall memory bugs. However, this also introduces runtime overhead, which is why languages like C/C++ still have their use case and continue to have static analysis tools being developed to remove previously mentioned bugs [9].

Garbage Collector algorithms can be divided into two main categories [10]:

- Tracing Algorithms
- Reference Counting Algorithms

Tracing Algorithms

Tracing Algorithms' main characteristic is the fact they determine the live objects of the application, classifying the remaining ones as garbage. The first Garbage Collector was a tracing algorithm developed for the Lisp language in 1959 by *Mccarthy* [11]. The algorithm is commonly known as Mark & Sweep and justifies its title by having two distinct phases:

Mark: This is the first phase of the algorithm, and it is the reason for its tracing classification. As mentioned before, in this phase all live objects are calculated, by starting at a known set of root objects and *marking* all accessed objects from there.

Sweep: In this phase, all objects are accessed and checked if they are marked or not. If they are not marked it means that it wasn't possible to access them from the root objects i.e., they are safe to be collected/deallocated.

Despite its age, as you will later verify (see chapter 3), this algorithm remains a fundamental stepping stone for current GCs, and so many of its issues like cache locality, lack of concurrency (application isn't modified while the Garbage cycle is running - Stop-the-World algorithm), etc. are still relevant and tackled. In particular, past work has addressed locality in object placement in the heap [12], and locality in code paths where objects are allocated w.r.t. their influence in object lifetime [13].

Referencing Counting Algorithms

In the same year 1960, *Collins* also developed the first reference counting garbage collector algorithm [14]. The main concept of this algorithm is the observation that an object is live if it has one or more incoming references. To track this data, a simple field on each object can be used, tracking the reference count as objects are created and deleted (incrementing and decreasing the ref. count respectively). This way, we can free the object immediately and concurrently to the application as soon as it becomes unavailable.

Additionally, a write barrier is needed, which consists of instructions injected by the compiler on all application writes, to remove concurrency problems e.g., multiple threads manipulating the same object. This is also a common concept in more recent research (see chapter 3).

Two of the main problems of referencing counting GCs are:

Barrier Overhead: Despite necessary, barriers introduce overhead to the application mutator.

Impossible to delete cyclic data structures: There isn't a way of deleting objects in a cyclic graph.

2.3 BestGC

BestGC [15] is a tool that aims to select the most appropriate garbage collector for a user-provided Java application. It does so by previously analyzing and benchmarking multiple applications with the list of available garbage collectors on Java's runtime across various heap sizes. The collected metrics are application throughput and garbage collector pause time, which are used to score the Garbage Collectors.

Afterwards, the user-provided application is monitored to collect metrics that will be used to choose the most suitable GC and finally run the application.

Best GC Phases

In more detail, the Best GC's phases could be outlined as follows:

Matrices Generation: This phase is only executed once, before even running BestGC. As mentioned before, its main purpose is to profile and benchmark the available Garbage Collectors with benchmark applications, in this case, the DaCapo[4] and Renaissance suites[5] were chosen. The benchmarks are run varying the heap size and upon completion, the metrics are compiled into a matrix where each Garbage Collector gets a score assigned concerning their pause time and throughput achieved. In this case, JDK 15 was being used, so Garbage First (G1), Parallel GC, Shenandoah,

[4]https://www.dacapobench.org/
[5]https://renaissance.dev/

and ZGC were the GCs used. Their score was normalized to the default Java GC collector, i.e., the Garbage First Collector.

Run BestGC: To run BestGC the user provides the compiled java application, meaning the JAR file, and a weight $\in [0, 1]$ to throughput (or pause time). At least one needs to be provided, and their relation is described by the equation: $1 = throughput\_weight + pause\_time\_weight$. The user can also choose to run the application after the benchmark, which is the default behaviour, and specify how long the next phase (**monitoring phase**) will be.

Monitoring Phase: So that BestGC can select the best Garbage Collector for the application, it needs to monitor the application first. This process occurs using the Garbage First GC and has two main purposes: collect the amount of heap size in use and CPU usage. The heap in use is calculated using the *jstat*[6] tool, by providing the *process id* we can extract the survivor/eden/old and compressed class space used by the Garbage First GC. The CPU usage is calculated using *proc/stat*[7] and *top*[8] command, and if it reaches an average above 90% the application is classified as **CPU intensive** which can be leveraged so to better understand the relation between CPU and Garbage Collector.

Calculation Phase: Based on the recorded maximum heap size, BestGC increases its value by 20%, i.e., $max\_heap = max\_heap * 1.2$, and then adjusts it upward to the nearest heap size used in the **Matrices Generation Phase** [256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, or 8192 MB]. With the *max heap* calculated we can already select the corresponding matrix i.e., if we got a max heap size of 4096MB the corresponding matrix will be the one where all benchmark applications were executed with a heap size equal to 4096MB. Furthermore, given the previously provided throughput (or pause time) weight, we calculate the Best GC by calculating the following equation for every GC in the matrix:

$$Score = throughput_{weight} * throughput_{score} + pause\_time_{weight} * pause\_time_{score}$$

. The lower the *Score* the better the GC, so the GC with minimum score is selected.

Results

After running the **Matrix Generation Phase** it was possible to observe that both suites had the **Parallel GC** has better GC in terms of throughput time (which can be expected due to being a Stop the World GC), while in relation to pause time, *ZGC* was the most suited, due to its concurrent nature. Analyzing the CPU influence we can infer that the performance gained by using the BestGC was similar in non-

[6] https://docs.oracle.com/en/java/javase/11/tools/jstat.html
[7] https://man7.org/linux/man-pages/man5/proc.5.html
[8] https://man7.org/linux/man-pages/man1/top.1.html

CPU-intensive and CPU-intensive applications, which will serve as a basis for one of our parameter decisions in 4.2.1. Finally, analyzing the overall quality of BestGC garbage collector selection, it reported on average a 51.24% selection of the best possible GC, and an 85.95% GC with the correct category i.e., the garbage collector selected belonged to the same category (concurrent/non-concurrent) as the one with the best performance. Also, when failing it still registered a 1.75% improvement compared with G1 and overall an average of 36.75% performance benefit.

3

Related Work

Contents

In this section, we summarize multiple research contributions and categorize them into three main subsections: Garbage Collector Algorithms, GCs - Studies and analysis, and Runtime Optimizations.

3.1 Garbage Collector Algorithms

3.1.1 Garbage First

The default Oracle HotSpot Java Virtual Machine Garbage Collector is G1 [16–18], a GC (Garbage Collector) targeted for multiprocessor systems with high memory scalability. Its main characteristics are: generational; parallel; mostly concurrent; Stop-The-World and evacuating. It aims to achieve the best tradeoff between latency and throughput for applications with large numbers of allocations, however tends to achieve worse throughput compared with throughput-oriented GCs. Below we will explain its main characteristics.

Generational

The G1 algorithm distinguishes the allocated objects in young and old generations. This is done due to the observation that recently allocated objects have a higher probability of being removed from the application, while objects that are kept on the application for a longer period, have a lower probability of future removal. So it is more efficient to run more frequent garbage collection cycles on objects with lower lifetime [19].

The young generation is divided into eden and survivor regions. The eden region contains the newly allocated objects, while the survivor region contains the young objects that already survived a collection cycle, and will be promoted to old objects if they survive another one (Figure 3.1).

Memory Layout

The heap is divided into regions of equal size. Mutator threads allocate thread-local allocation buffers (TLABS) directly on each region using a Compare and Swap (CAS) operation, which prevents memory allocation contention because each thread is responsible for different memory regions. After a successful CAS operation, the thread can allocate the objects on its TLAB.

Each region x has a remembered set known as a card table (Figure 3.2), which tracks all old regions that might contain pointers to live objects within the x region. This allows us to know which regions contain references to the selected collection set at collection time. The only references that remembered sets keep track of are old-to-young and old-to-old references because young-to-young and young-to-old references are not needed (young regions are always collected).

Marking

G1 uses a concurrent marking algorithm called Snapshot-at-the-Beginning (SATB), meaning that it does a heap snapshot and marks the objects as garbage concurrently using that snapshot. We can use a heap snapshot instead of a Stop-The-World method because of the correct assumption that objects that are garbage will remain garbage. Due to the nature of the snapshot mechanism, objects that are created after the heap snapshot will be classified as live objects, and objects that become garbage after the heap snapshot will become floating garbage having to wait for another marking phase to be correctly classified.

During the concurrent marking phase, the G1 algorithm has to insert an SATB barrier when writing to non-null references, because a concurrent write could violate the SATB assumption. When writing to a non-null reference, this barrier will push the reference to a buffer to be later processed, keeping the heap snapshot consistent.

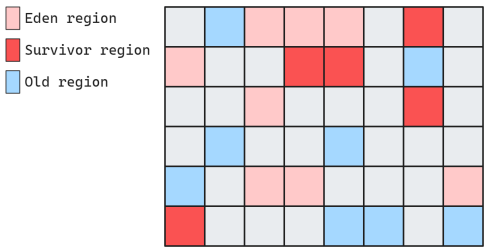

Figure 3.1: G1 Heap Layout

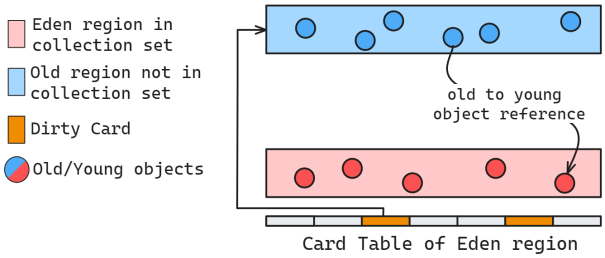

Figure 3.2: Card Table Example

Garbage Collection Cycle

The GC Collection Cycle is divided into two phases: The Young-Only phase and the Space Reclamation phase. The Young-Only phase contains:

Normal Young Collections: Are triggered when the ratio of $\#eden\ regions + \#survivor\ regions >= newSizeRatio$ [17], then all objects of the collection set that are still reachable from the root objects and remembered sets are evacuated.

Concurrent Young Collection: starts when the occupancy of the old generation reaches the Initiating Heap threshold. It starts a marking phase to determine all live objects in the old generation (while this process doesn't finish Normal Young Collections can occur). The process ends with two Stop-The-World pauses: **Remark phase** that finalizes the marking process and selects regions with low occupancy to be prepared concurrently before the **Cleanup phase**, responsible for sorting the prepared regions according to efficiency and deciding if the Space-Reclamation phase will occur.

After the Concurrent Young collection, G1 enters the Space-Reclamation phase where it does numerous mixed collections (both young and old generation). This phase ends when the benefit of evacuating more old-generation objects doesn't outweigh the overhead.

3.1.2 Shenandoah

Shenandoah GC is a garbage collector from Red Hat for the JVM platform [20]. It was developed with large heap applications in mind and its characteristics are region-based, low-pause, parallel, and concurrent GC.[1]

The Shenandoah GC main contributions are lower pause times due to concurrent compaction and an architectural neutral algorithm, only needing a reliable Compare and Swap (CAS) operation for its implementation.

[1] There are already experimental Shenandoah generational builds see https://aws.amazon.com/blogs/developer/announcing-preview-release-for-the-generational-mode-to-the-shenandoah-gc/ and https://openjdk.org/jeps/404

Concurrent compaction raises many issues due to moving objects while possibly using them and having to update all the references to that same object in an atomic fashion. To solve this, Shenandoah makes use of a forwarding pointer, forcing all operations to that object to use the forwarding pointer. This means that when moving an object the references can be updated incrementally. When multiple threads are trying to update the object it will be done with a CAS operation and only one will succeed. It is worth noting that this mechanism doesn't create a new race condition when one thread is reading an object while another is writing, but increases the window for the race condition to happen.

Memory Layout

To introduce a forwarding pointer means that each object will have an additional header word alongside the word for the object class and the mark word (used for forwarding pointers, age bits, locking, and hashing).

The heap is divided into regions of equal size. Both long-lived and newly allocated objects can coexist in these regions.

Garbage Collection Cycle

The Shenandoah Garbage cycle has four phases, two Stop-the-World (STW) phases, and two concurrent. They are:

Initial Marking (STW): Phase where the root set is scanned.

Concurrent Marking: Similar to Section 3.1.1, uses Snapshot at the Beginning algorithm (SATB). Each thread keeps the total live data for each heap region, to be later combined. It also updates references to regions evacuated in a previous cycle.

Final Marking (STW): Phase where remaining marking/update queues tasks are processed and the root set is re-scanned. It also creates the collection with the regions that contain the least amount of live data.

Concurrent Compaction: To evacuate the live objects in the collection set the GC threads use a speculative copy protocol. They first create a speculative local copy of the object. Next, they try to ensure the forwarding pointer references their local object using a CAS operation. Failing the CAS operation doesn't raise any problems because it means other thread successfully copied the object.

Barriers

Shenandoah has multiple barriers to ensure correctness while on the concurrent GC phases.

- SATB Write Barrier

- Read barrier: Single operation that dereferences the forwarding pointer.
- Evacuation Write Barrier: Ensures that objects in the collection set are evacuated before a write.
- Compare Barrier: While comparing two objects, you can be comparing two distinct references to the same object (from-space and to-space references). To correct this issue, on a failed comparison the two objects are read again using the read barrier. If the reference matches, they are the same.

All these barriers introduce overhead and diminish throughput however, they are the cost to have diminished GC pauses and better latency.

3.1.3 ZGC

The Z Garbage Collector (ZGC) [21] is a Garbage Collector targeted for a large range of heap sizes (up to 16TB), optimized for low latency. Its main characteristics are being a non-generational, region-based, mostly concurrent, parallel, and mark-evacuate garbage collection algorithm.[2]. To achieve concurrency, ZGC introduces two main novelties: colored pointers and load barriers.

Colored Pointers

ZGC uses 64-bit pointers (20 bits for metadata + 44 bits for object address), with currently only 4 bits of metadata in use. These 4 bits of metadata are:

- **Finalizable (F)**: The object is only reachable from a finalizer.
- **Remapped (R)**: Reference is up to date and points to the correct object location.
- **Marked0 (M0) and Marked1 (M1)**: If the object is marked.

The conjunction of these bits determines the color of the object. There are 3 possible good colors: only M0 is set; only M1 is set; and only R is set. The color can be ”good” or ”bad” depending on which phase of the Garbage Collection cycle ZGC is currently on (see Figure 3.3).

Load Barriers

To interpret the color pointers, ZGC uses a load barrier. The load barrier will be inserted by the Just In Time compiler (JIT) when an object is loaded from the heap. After that, the load barrier examines the colored pointer and determines if the color is ”bad” (slow path) or ”good” (fast path). If it has a bad

[2]Generational ZGC was introduced recently for JDK 21 see https://openjdk.org/projects/jdk/21/ and https://openjdk.org/jeps/439

color, it can be self-healed by either updating the pointer or relocating the object and then updating its pointer. The pointer update is done using a Compare and Swap (CAS) operation to prevent concurrency problems.

Memory Layout

ZGC divides its heap into memory regions (small, medium, large) that can be dynamically resized during runtime. Furthermore, it maintains 3 virtual views of the same physical memory, each view corresponding to one good color. This way, the pointers with good color can be dereferenced directly without the need for bit masking operations.

Garbage Collection Cycle

The ZGC cycle consists of three Stop-the-World (STW) pauses and four concurrent phases (Figure 3.3):

STW Pause 1: Threads agree on the current good color; Pages allocated before the current cycle are selected for collection; Roots are healed and pushed to the mark stack.

Marking/Remapping: Transverse, mark, and self-heal every object reached starting from root nodes. Both mutator threads and GC threads can mark the objects, using a mark barrier and a load barrier respectively. The GC threads will always hit the slow path even if the color is good to maximize the number of objects in the local mark stack.

STW Pause 2: Check if all the marking stacks are empty. This can impact the throughput if the mutator threads are all paused so, to avoid entering this phase early, local thread-handshaking is performed with each mutator thread to check for local mark objects.[3]

Reference Processing: Handle references edge cases.

Selection of Evacuation Candidates(EC): Previous ECs are cleared, and forwarding tables are dropped. Pages with no objects are reclaimed right away, and the remaining ones are added to the Evacuation Candidates sorted by size.

STW Pause 3: Relocates all root set objects updating their references and good color.

Relocation: Relocates all objects in the EC, updating their address and the forwarding table.[4] Objects that are not updated in this phase, will be updated on the next marking phase or by previous load barriers.

[3] JEP 312: Thread-Local Handshakes: Introduce a way to execute a callback on threads without performing a global VM safepoint. Make it both possible and cheap to stop individual threads and not just all threads or none.

[4] Table used to map pre-relocation addresses to post-relocation addresses

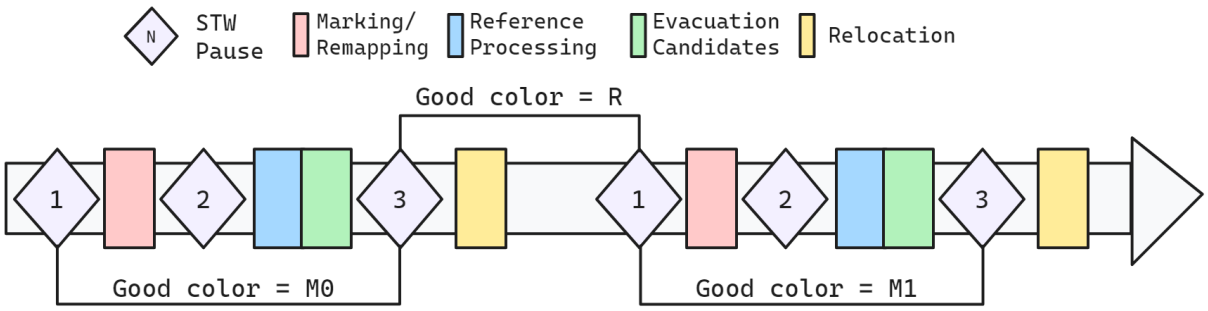

Figure 3.3: ZGC Garbage Collection Cycle: The cycle keeps repeating alternating the first good color between M0 and M1

3.2 GCs - Studies and analysis

3.2.1 NAPS

For throughput-oriented applications, Stop-the-World Garbage Collectors (GCs) have shown to be effective, on machines with a small number of cores. However, when using large core machines, with a NUMA architecture, running OpenJDK 7 with Parallel Scavenge as the default GC, the performance drastically decreases when the number of cores used is above 8.

NUMA (Non-Uniform Memory Access) architecture machines consist of multiple nodes each one having various processing units and one local memory. To communicate between nodes a *interconnected bus* is used.

The reasons for this performance decrease are:

Lack of NUMA-awareness: Leads to objects being allocated on a single node, overloading it.

Lock in Parallel Scavenge phase: Reduces parallelism and increases synchronization overhead.

To solve these issues NAPS [22] was developed, based on Parallel Scavenge and Numa-Aware. Its main objectives are balancing the live objects between nodes and avoid locking in parallel phases.

Parallel Scavenge

First, let's describe the workings of the Parallel Scavenge algorithm. As mentioned before, it is a Stop-The-World (stops the application when running the GC cycle), parallel, and generational Garbage Collector. To allocate objects each mutator thread uses a thread local allocation buffers (TLABs), avoiding synchronization costs. The young generation contains three sections: eden-space (recently allocated objects), from-space (objects that survived a collection), and the to-space. When there is a young collection all the eden-space objects are copied to to-space, and the from-space objects are copied to the old generation. This means that the from-space and to-space regions are swapped.

Young collection Phases: They are started when there isn't space on the current TLAB and it is impossible to fetch another.

- **Initialization:** All the mutator threads are paused, and the VM thread initializes a queue of GC tasks. After it wakes all the GC threads.
- **Parallel Phase:** The GC threads fetch tasks using a monitor. There are:
  - Root task: Provide an entry point to the object graph. While copying objects, the GC thread stores a forwarding pointer on the old object header using a Compare-and-Swap (CAS) operation to prevent other threads from copying the same object. All references reached from that object are pushed to a Breadth-First Traversal (BFT) lock-free bounded queue. When the BFT queue is empty, the GC thread fetches another root task, or a steal task if all root tasks were already fetched.
  - Steal task: The object graphs can be unbalanced so there is a need to distribute the workload to prevent some GC threads from being idle. When fetching a steal task, the GC thread will steal an object reference from a random BFT queue, popping it from the tail to avoid synchronization problems. When the thread can't steal a task it enters a termination protocol incrementing a counter. Then it pools the counter and peeks into the other threads' local queues. If all the queues are empty or the global counter is equal to $\#GCthreads$ then the thread leaves the termination protocol. If it finds more tasks, the thread decrements the counter and continues the stealing process.
  - Final task: When a thread leaves the termination protocol it pops a task called final task. The thread that has this task is the leader responsible for waking up the VM thread.
- **Final Synchronization Phase:** The VM thread resizes the different heap regions. In the end, it wakes up the mutator threads.

Old Collection Phases: When an object promotion from the from-space to the old generation fails the GC thread sets a flag. If the VM thread sees it, a two-phase mark-compact algorithm is started. Both phases are parallel and use the same mechanism as the Young Generation collection.

- **First Phase:** Live objects are marked in parallel.
- **Second Phase:** The GC threads compact in parallel live objects by moving them into free spaces.

Solution Design

After analyzing the Parallel Scavenge algorithm execution on multicore machines, the authors made the following design decisions:

- **Michael-Scoot lock-free queue:** In order to reduce lock contention when multiple threads were trying to dequeue the task queue.
- **Removal of GC monitors lock:** When terminating the GC threads would try to access the GC lock at almost the same time, leading to contention. This lock was removed and the termination protocol was changed to use a *futex wait* operation (compare and sleep) and an atomic variable with a timestamp to prevent race conditions e.g., a GC thread is interrupted before suspending itself by the VM thread, and when resuming it will execute the *futex wait* operation and verify that the timestamp variable was changed by the VM thread, avoiding the deadlock.
- **Fragmented spaces:** Used in young generation. This divides the space into multiple fragments, each with a virtual address range from a single physical node. When a thread allocates a TLAB it will access its node. This way, locality improves because mutator threads tend to access recently allocated objects, and thread migration is mitigated. When collecting, the GC threads will copy to their node too, ultimately, leading to a balanced memory occupation between nodes due to stealing tasks.
- **Interleaved spaces:** Used in old generation. Due to single-thread initialization, the objects tend to be allocated on the same physical node. To mitigate this issue, whose pages are mapped from different nodes with a round-robin policy. This will balance memory allocation and access.

With all these changes, after multiple benchmarks, it is possible to conclude that stop-the-world GCs, in this case NAPS, can scale up to 48 GC threads, provided it has enough memory to collect, reducing the application pause time.

3.2.2 NumaGiC

On cache-coherent Non-Uniform Access Memory architectures, applications with a large memory footprint suffer from the cost of the GC, because of many remote memory accesses during the GC reference graph scan, that saturate the interconnect between memory nodes. These remote memory accesses are due to the creation of *inter-node references* (a reference to an object allocated on a different node).

NumaGiC [23] was created to resolve these problems. It has a *mostly distributed* design and aims to maximize memory access locality and avoid the drawbacks of a purely distributed design which tends to decrease parallelism. The main optimization was to delegate the processing of objects to the GC thread running on the remote node, by sending a message. This design can decrease parallelism because a GC thread remains idle when there are no local objects to be collected. Enter the *mostly distributed* design, where a GC thread has two modes: **Local** and **Work Stealing** mode. NumaGiC also introduces several policies that aim to reduce the number of *inter-node references*, and to balance the amount of objects across nodes.

Note: NumaGiC targets long-running computations that use large data sets, for which a throughput-oriented stop-the-world GC algorithm is suitable. It is based on an improved version of **Parallel Scavange** (see 3.2.1) called **PSB** (a stop-the-world, parallel, generational GC which is the default throughput-oriented GC for Hotspot)

Numa friendly policies

The next memory placement optimization policies were created taking into account that the placement decision should take less time than the expected benefit. The object graphs of 5 applications were analysed with a customised version of PSB (Parallel Scavenge Baseline), which ensures that the objects allocated by a mutator thread running on node *i*, always stay on that node (pure distributed design).

Applications:

Spark: A multi-threaded map-reduce engine

Neo4j: An embedded, disk-based, fully transactional Java persistence engine that manages graph data

SPECjbb2013: Business logic service-side application benchmark that defines groups of threads with different behaviours

SPECjbb2005: Business logic service-side application benchmark where all the threads have the same behaviour

H2: An in-memory database from the DaCapo 9.12 benchmark

Based on the analysis it was possible to observe that the proportion of clustered references is always high, specifically between young objects (most of applications with $>= 90\%$), and old-to-young objects (3 applications with $>= 70\%$). It was also possible to conclude that the memory allocation imbalances can vary greatly depending on the application, with a range from $7\% \; to \; 40\%$.

Derived from these results, four policies were created:

1. *Node-Local Allocation*: An Object is placed on the same node as the mutator thread. Improves both GC locality and application locality.

2. *Node-Local Root*: The roots of a GC thread are chosen to be located mostly on its running node.

3. *Node-Local Copy*: During young collection, a live object is copied to the node where the GC thread is running. This in conjunction with the GC thread *stealing mode* re-balances the load.

4. *Node-Local Compact*: During the compacting phase of a full collection, an object being compacted remains on the same node.

To map addresses to nodes, NAPS (see 3.2.1) *fragmented space* is used (each virtual address range is allocated on a different node). However, unlike NAPS, it is extended to the old generation too. The card table (Section 3.1.1) is divided into old-generation segments. This means that each segment has the old-to-young roots of a given node enforcing *Node-Local Root* policy.

GC threads Mode

Local Mode: In this mode, a GC thread only collects its local memory. While processing references, if an object from another node is encountered the GC thread sends the reference to the ”home node” of the object. When a GC thread idles in local mode, it can steal work from its local nodes's pending queues. [5]

Communication Infrastructure: NumaGiC uses a communication channel per each pair of nodes (to avoid contention), implemented with a *array-based lock-free bounded queue*. To mitigate the *atomic compare and swap* sync cost of multiple threads sending on the same queue, the references are sent in bulk.

Work-stealing Mode: In *work-stealing* mode, a GC thread may steal work from any node and access remote memory. It steals from 3 groups:

- Its own transmit buffers (cancelling the messages that were not delivered).
- The receive side of other nodes.
- Pending queues from other GC threads.

When stealing from one group, the GC thread will keep stealing for as long as possible from the same group to avoid failed attempts.

When a GC thread does not find references to steal, it waits for termination. To solve async communication termination problems, before entering the termination protocol, the GC thread checks the remote memory to see if all its messages were delivered.

Switching between modes: A GC thread enters work-stealing mode when it does not find local work: its local pending queue is empty; can't steal from local GC threads; its receive channels are empty.

However, it regularly re-enters *local mode* (every 1024 stolen objects) because local work can become available again, and this will ensure that all groups from the *work-stealing mode* will be checked again when the GC thread leaves *local mode*.

[5] Parallel Scavenge keeps tasks on pending queues

Results

Testing NumaGiC against previous NAPS (see 3.2.1) showed improvements between 12% to 45% in throughput. Those improvements were due to the high memory access locality without decreasing parallelism, which translated into a reduction of interconnect traffic.

3.2.3 iGC

Latency Critical services experience heavy-tailed latency due to performance interference of the concurrent garbage collection (GC) as well as multi-tenancy (several different cloud customers are accessing the same computing resources). The root cause is a semantic gap in resource allocation between JVM and the underlying Linux OS in multi-tenant systems. This leads too:

- GC threads competing for CPU
- Co-located batch jobs interfering with Latency Critical (LC) services due to Simultaneous Multi-Threading (SMT)

iGC [24] is a middleware that separates GC threads from worker threads of LC services by assigning different CPUs for the GC threads and worker threads, avoiding interference. To mitigate SMT interference, iGC tries to assign threads of batch jobs and worker threads of LC services to different cores. Furthermore, iGC limits the resource usage of batch threads when GC threads are co-located with batch threads and GC is triggered. As a result, it effectively mitigates the contention of memory hierarchy in a multi-tenant system. Now we will describe the iGC architecture and all its components (Figure 3.4).

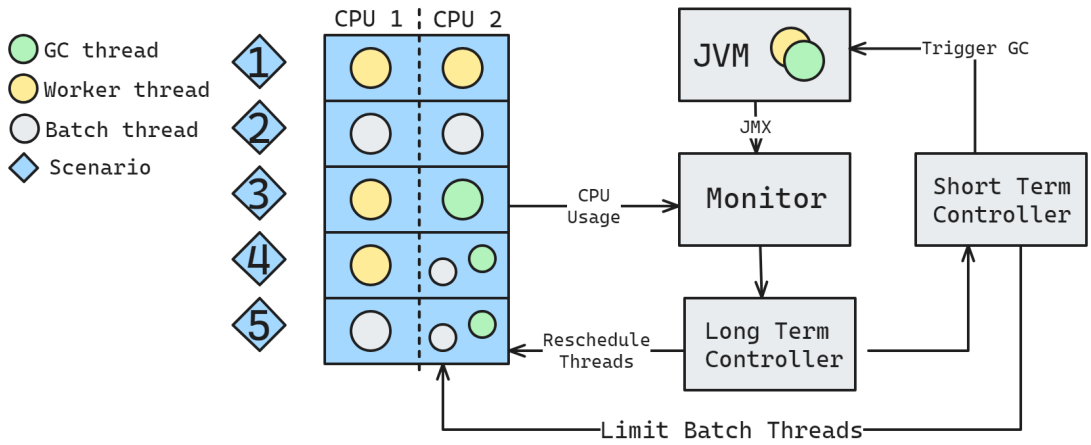

Figure 3.4: iGC Architecture

Monitor

The monitor periodically collects the resource utilization information including CPU usage and heap usage of a target JVM.

Short Term Controller

The controller manages the adaptive GC triggering procedure. It triggers GC based on CPU usage, heap usage, and interval set by Monitor and the Long Term controller. It limits the CPU usage of batch threads on the allocated CPU where GC threads are running (scenarios 4 and 5 in Figure above) by setting a low priority in CPU usage for batch threads so they can be resumed when GC is finished.

If the JVM is started with root privilege, GC is triggered directly as the OS scheduler will treat it with high priority. If not, GC is triggered if the cpu_usage where GC threads run is smaller than cpu_threshold (70% in this case). If the CPU resource is not enough, iGC limits the CPU usage of batch threads sharing the same CPU with GC threads by cgroup.

GC trigger situations:

1. There is enough CPU resource so it proactively triggers GC
2. The heap usage increases beyond a threshold such that GC must be triggered to avoid the allocation stall.

Long Term Controller

This unit has two key functions, dynamic CPU allocation and thread placement and adaptive monitoring interval. The former addresses non-trivial CPU contentions between GC threads and worker threads and between batch threads and worker threads. It also leverages Hyper-Threading (HT) for cache performance improvement. The latter controls parameters used in GC, including the heap usage threshold, the CPU usage threshold, and the number of GC threads.

JVM Problems:

- **JVM running in user space**: GC threads and worker threads are treated equally by the Linux OS scheduler and the use of round-robin time-slicing based scheduling leading to higher latency during GC.
- **JVM running as root**: the priority levels of the threads are converted to the OS-level priority levels by the JVM and the OS schedules the threads based on their priority leading to higher latency if GC threads have higher priority, or expensive Stop the World problems if GC threads have lower priority.

- **Solution:** To solve this, iGC pins the Worker and GC threads to different CPUs.

Thread Placement Scenarios:

1. It assigns both CPUs of one core to worker threads, which avoids the interference from batch threads and makes use of the memory hierarchy of the core.
2. It assigns both CPUs of one core to batch threads, which avoids the impact on worker threads.
3. When there are enough CPUs, worker threads and GC threads are assigned to the two CPUs of one core, respectively (this allows to use HT to our favor).
4. When CPUs are not enough, GC threads share the CPU with batch threads on one CPU, while worker threads use the other CPU.
5. In the worst scenario, GC threads share one CPU with batch threads, while other batch threads use the other CPU of the same core.

Results

To test iGC, three NoSQL Latency Critical services were selected (Cassandra, HBase, Solr), and multiple workloads with varying percentages of read/update/insert and scan operations. The comparisons were made against two GCs: Garbage-First and ZGC. The results allowed them to conclude that the iGC middleware improved throughput in multi-tenant systems when used with both previously mentioned Garbage Collectors.

3.2.4 J-NVM

Many modern data stores and big data analytics platforms are written in Java. Because they manipulate large amounts of persistent data, these systems can greatly benefit from the recent technological advances in Non-Volatile Main Memory (NVMM).

To access NVMM two designs currently exist:

1. External design: NVMM remains outside Java Heap. The JVM accesses it through the file system (FS) or Java native interface (JNI)
2. Integrated design: JVM stores Java objects in NVMM, and the application accesses them directly (read/write instr.)

The external design is inefficient due to the cost of converting data back and forth between the NVMM and Java representations. The internal design has a fundamental flaw: because NVMM now has Java

objects it has to run GC (very expensive when data is in the order of a hundred GBs as is the case with NVMM).

J-NVM [25] avoids the problem mentioned before by noticing that the deletion of persistent objects is often related to a specific event e.g., deleting data from a database, which is rare and well-defined - so very few deletion sites. So it makes sense to avoid GC altogether.

Decoupling principle

Because Java is a GC language the authors introduce a decoupling principle between the data structure of a persistent object and its representation in the Java world. Based on this principle, a persistent object now consists of two parts: a data structure stored off-heap in NVMM, and a proxy that remains in volatile memory. The data structure holds the fields of the persistent object (avoids collection), while the volatile proxy provides the methods that manipulate them.

The J-NVM uses a **class-centric** approach which means that a persistent object has to be explicitly annotated as such. Despite not having the elasticity of the **instance-centric** approach of creating persistent and volatile objects alike, it reduces problems in reliability (e.g., bugs can appear from hiding durability from the compiler and programmer); and avoids cross-heap references (e.g., a reference stored in the NVMM could refer to a persistent or volatile object).

J-NVM framework

The J-NVM framework offers two paradigms:

1. Low-level interface that focuses on performance
2. High-level interface that trades performance for usability

The Low-level interface defines the methods for the proxy to access the persistent data structure. The High-level interface additionally provides failure-atomic blocks (block of code that is executed entirely or not).

Properties:

- Association: The proxy maintains the persistent data on an array that holds the addresses of all its blocks.
- Liveness by reachability: Implemented in J-NVM using a recovery-time GC, that transverses the persistent object graph when application resumes after a crash.

- Fragmentation: J-NVM relies instead on a memory layout inspired by the work of Pizlo et al.. This layout splits the heap into blocks of fixed size. If a large object does not fit into a single block, J-NVM creates a linked list of blocks to store its content. Using blocks of fixed size eliminates the fragmentation problem by design since we can always allocate large objects. However, this memory layout also increases the complexity of accessing large objects (fixed by the array of addresses).

- Low-level interface **pwb, pfence, psync** are used to: add the cache line to pending write queue; ensure previous **pwbs** and stores to volatile/persistent memory are executed; (same as previous) and additionally the writes in the pending queue are propagated to the NVMM.

Results

Both the low-level and the high-level interfaces systematically outperform the external design. In YCSB benchmark the low-level interface is between 3.6 and 10.5x faster than other existing solutions.

3.2.5 Elastic Memory Management

Stream processing engines are very prominent in the modern technology landscape, mostly due to Big Data and the need for real-time analytics in fields such as finance, telecommunications, and e-commerce. The applications in these sectors require short End-to-End (E2E) latency, to achieve better user experience, respect service level agreements, and generate profit.

The data stream can run through multiple tasks until the final output is computed, with each task having different latency values. This means that if a task in the pipeline has a very high latency, it greatly affects the E2E latency of the application. One of the causes of high latency in these streaming tasks is the mismatch between required and allocated resources. Using static schedulers or having the user predict the memory requirements of the task can incur in some having high latency due to under-provisioning, or the opposite, compromising E2E latency in the end.

Other approaches like scaling entities or moving workload between entities still lead to costly operations like state migration and synchronization operations.

Elastic memory management would not have those problems, because there isn't the need to manipulate state. It is possible already to elastic reconfigure Linux containers using "cgroup" (Linux tool), however, because on SPEs the Linux container normally contains all the executing tasks, it is not possible to use this method because it doesn't provide enough granularity. The proposed solution is a task-level elastic memory manager - EMM [26] - built on top of Apache Flink, a state-of-the-art SPE.

EMM

EMM has the ability to dynamically balance the memory between tasks. It does that using a provisioning plan, built by a quantitative model that uses task-level latency ($queuing\ delay + processing\ time$) and memory size. It monitors the real-time performance of the application and analyses the metrics, determining the optimal memory for each task to minimize latency.

Motivation Examples

Exactly-Once Guarantees: SPEs have these guarantees in order to ensure the correctness and consistency of stateful operations in case of failures. Apache Flink uses a Pipelined Consistent Snapshot mechanism in which the system periodically inserts barriers into the stream. When there are multiple parallel tasks, and one of them gets delayed, it will affect a downstream task, that has to wait for all the upstream barriers. So the upstream task latency will greatly influence the E2E latency.

Event-time Processing: Some streaming operations occur over data between a window of time. Similar to the previous example, the downstream task has to wait for the slowest task to output the result. Again an example of an upstream task influencing the E2E latency.

Structure

EMM consists of two parts: the EMM runtime and a module embedded in the SPE (Figure 3.5).

EMM runtime:

- Datastation: Pre-processes the data collected from Apache Flink into mature data to estimate α (time taken to read an item from memory) and β (additional time taken to read an item from disk compared to memory) parameters of the online learning method (method in which the model is updated as new data points arrive).
- Solver: Gets the solution of a non-linear programming model.
- Resource Negotiator: Parses the provision plan from the solver and sends the resizing requests. First sends and waits for the memory shrinkage requests. After that, it sends expansion requests.

Modules in SPE:

- MemUpdateCoordinator: Located in the job master of the streaming application. Parses the resizing request and maps the request to the correct task.
- Elastic Memory: Memory area used to cache state and handled by each task.

Results

Varying the execution interval of the metric pooling affected the performance of the system, With short intervals leading to overreaction to metrics and long ones to a large delay in memory management. The value on Flink was set to 30 seconds to achieve good performance. Compared with Flink Slot Sharing capabilities, EMM could reduce significantly the E2E latency in multiple tests, with 46% to 62% (P99), and 40% to 51% (mean) reduction.

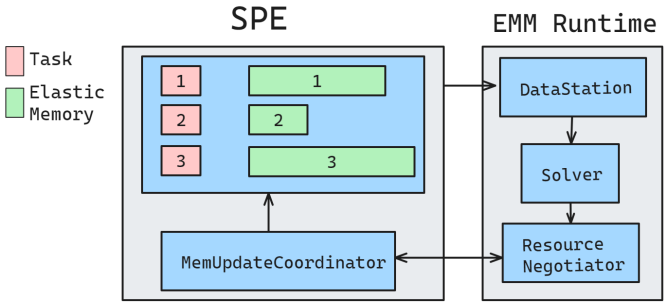

Figure 3.5: EMM architecture

GC Comparison

The GC algorithms mentioned earlier can be compiled and classified as illustrated in Table 3.1.

Table 3.1: Garbage Collectors

Algorithm	Concurrent concurrency	Generational	Parallel	Copy/Evacuating	Mark	Objective
G1	Partial	Yes	Yes	No/Yes	Partial(Old Gen)	Latency
Shenandoah	Yes	No1	Yes	No/Yes	Yes	Latency
ZGC	Yes	No2	Yes	No/Yes	Yes	Latency
Parallel/NAPS/NumaGiC	No	Yes	Yes	Young Gen/Old Gen	Partial (Old Gen)	Throughput

An observed correlation is that the Garbage Collector objective is directly tied to algorithm concurrency. It becomes evident that, for improved application latency, a preference is given to more concurrent algorithms due to fewer Stop-the-World pauses, whereas the opposite holds true for throughput. We can also observe that all algorithms are fully parallel, prefer an Evacuating strategy for object compaction (lower memory requirements), and the ongoing push for generational support in newer GCs like Shenandoah and ZGC, due to its potential benefits [19]. Looking at Table 3.1 we can denote that having generational support implies a different approach for young and old generations, to avoid frequent expensive operations like marking, however on newer algorithms with lower pause times that is not the case.[6]

[5]We consider here that a fully concurrent algorithm implements all expensive operations e.g., global marking, concurrently to the application

[6]See 1 and 2

3.3 Runtime Optimizations

3.3.1 GraalVM Native Image

Nowadays, server applications are being deployed and executed using cloud computing services such as Function as a Service (**FaaS**). When the workload rises on these systems, the underlying platform spawns new instances to handle the workload, however, the new instance will have a “cold start“. The cold start happens due to the language runtime having to be initialized. These initializations can be very slow due to resource-intensive processes like code verification, class loading, bytecode interpretation, profiling, and dynamic compilation (e.g., in the Java VM). This can result in slow startups and a high memory footprint, potentially breaking service-level agreements (SLA) and raising costs.

GraalVM Native Image [27] aims to solve these problems by points-to analysis and heap snapshotting, followed by ahead-of-time (AOT) compilation.

Overview

The designed solution is based on a closed-world assumption, which means all Java classes must be known at build time (Figure 3.6). All the Java bytecode of the application, libraries, VM, and JDK are processed the same way by points-to analysis, heap snapshotting, and initialization (callback execution). This process is iterative, and it only stops when a fixed point is reached.

Order of steps:

1. Points-to analysis: Used to discover all classes, methods, and fields reachable at run time. It starts with all entry points and builds iteratively a type-flow graph that covers the application. It terminates when there are no more possible type additions to the graph.

2. After building the type-graph, the initialization code can be run. The source of this code is Java class initializers, which compute the initial values of static fields, and custom callbacks that can be run at build time, before, during or after the analysis step.

3. Heap Snapshooting: Builds the object graph with all reachable objects and writes them into the heap image (the initial heap on application start).

This process is repeated until there are no changes in all previous steps. After that, all reachable methods are compiled into machine code using Ahead of Time Compilation (AOT), and some optimizations are made according to points-to analysis information e.g., marking fields as final.

The built Image Heap will be considered root-pointers by the Garbage Collector and can be used by multiple VMs at the same time. This is called ”isolates”, and they provide the ability to host multiple VMs

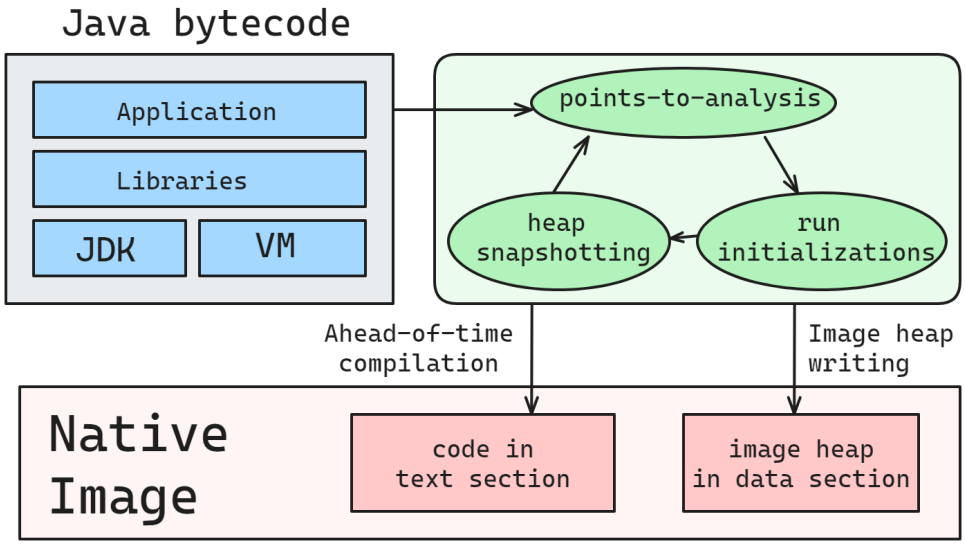

Figure 3.6: GraalVM Native Image

in the same process, each one sharing the same AOT compiled code. Furthermore, each isolate has a separate heap, so it isn't possible to have direct object references between two isolates, and Garbage Collection can happen separately in each isolate. When the isolate is destroyed all the isolate heap can be freed without the need of running Garbage Collection.

Results

Testing with three recent frameworks (Quarkus (version 0.18.0), Micronaut (version 1.1.3), and Helidon (version 1.0.1)) showed nearly two orders of magnitude improvement in startup time and instruction count, with also a good improvement in memory size. All the framework start-up times were less than 30 milliseconds while using the Java Hotspot VM runtime took a second or more.

3.3.2 GraalVM Isolate Proxy

As mentioned before, while serverless platforms are known for their cost efficiency and relative ease of use, they still have some downfalls. These include high latency experienced due to cold starts and memory duplication of runtimes, libraries, and program state.

There are already solutions to reduce memory duplication for the JVM runtime e.g., Photons a framework designed to share the language runtime reducing memory footprint and cold starts [28]. However, Photos still has a cold-start problem due to the JVM startup time, so developing a similar solution using GraalVM (a language runtime with lower start-up times) is an option worth pursuing.

As previously indicated in Section 3.3.1, GraalVM has the concept of *isolates*, a feature that enables

multiple VMs to be allocated on the same process, and share the same AOT compiled code, although with different heaps and no state sharing. When not needed, isolates are destroyed, so there isn't an object caching functionality. This was designed with the thought that isolates should be short-lived with low memory allocated, which is not true for all applications e.g., interactions with databases. Isolate Proxy [29] aims to fix these limitations.

Solution Design

Isolate Proxy solves the previously mentioned limitations by implementing:

Object Pooling: Create a pool of warm isolates. When a thread detaches, the isolate is still alive on the object pool, allowing for further invocations to utilize the previously allocated objects. When an isolate is idle for a long time it can be destroyed, releasing all its allocated resources.

State Sharing: To enable state sharing, a specific isolate is created for that effect - a global shared Isolate. Worker Isolates, when they want to access the shared state, send requests to the global shared Isolate and get the result via argument passing.

Results

Testing the Isolate Proxy against Photons in a simulated cluster revealed that Isolate Proxy reduces the overall memory usage for the cluster by 30%. When both solutions have the same cluster size, there are fewer occurrences of cold starts on Isolate Proxy, which directly translates into better performance due to Photons' high startup time. Isolate Proxy also has lower memory requirements, allowing for more simultaneous warm containers and collocated isolates, culminating in reduced cold starts as well.

4

Solution

Contents

In this chapter, we will address the implementation architecture and components developed so as to improve BestGC. As mentioned previously in Section 2.3 to run BestGC there is a matrix generation phase, where a matrix that scores all available garbage collectors (GCs) in the user's Java environment is computed. So that a user can easily reproduce this matrix, a tool called BenchmarkGC[1] (see Section 4.1) was developed, which we will describe in more detail. Furthermore, BestGC was enhanced to support web invocations, which are run in a nearly fully automated fashion; and provide an overview of the users' running applications. Those improvements culminated in the tool BestGC++[2], which is explained more thoroughly later in Section 4.2.

[1] https://github.com/guilhas07/benchmark-gc
[2] https://github.com/guilhas07/bestgc-plus-plus

4.1 BenchmarkGC

This tool allows a user to run multiple benchmarks so it classifies Java garbage collectors with respect to application $throughput$ and garbage collector pause time, or for short $pause\_time$. The inputs supported by the tool are described in the table 4.1.

Table 4.1: Command-line options for BenchmarkGC

Option	Description
-h, –help	Show this help message and exit
-d, –debug	Enable benchmark debugging
-c, –clean	Clean the benchmark stats
-s, –skip-benchmarks	Skip the benchmarks and compute the matrix with previously obtained garbage collector results. You must specify the java jdk used to obtain previous results. The jdk version is present in the name of each benchmark stat file. See `–jdk`.
-i, –interactive	Run the benchmark interactively
-j, –jdk JDK	Specify the Java JDK version when skipping benchmarks and calculating the matrix
-n, –number-iterations ITERATIONS	Number of iterations to run benchmarks. Increase this number for more reliable metrics
-t, –timeout TIMEOUT	Timeout for each benchmark
-b, –benchmarks {{Renaissance,DaCapo} ...}	Specify the group of benchmarks to run

As you can see by the option *-b* or *–benchmarks* description, the default benchmarks supported are DaCapo[3] and Renaissance[4], tools whose main purpose is to profile the Java Virtual Machine with non-trivial applications and workloads. Due to the existence of multiple applications in each benchmark suite, the need for custom options can rise dramatically e.g., multiple benchmarks in DaCapo don't support Java with $version >= 17$. To ensure users can execute the benchmark suites without extraneous problems, a configuration file, `benchmarks_config.json`, is provided to override or extend the benchmark settings. Its structure is shown below in Listing 4.1. Users can specify extra parameters for the JVM by writing a list of *string* options, and override the number of iterations or timeout (in seconds) for benchmarks that may take longer to run in less performant environments.

We will now explain the flow of the BenchmarkGC application. The first building block of the application is the algorithm responsible for executing individual benchmarks and extracting relevant metrics. This algorithm, detailed in Listing 4.2, begins by fetching the appropriate command to run, taking into account the configuration in `benchmark_config.json` (see 4.1). For instance, an example of a command might be:

[3]https://www.dacapobench.org/
[4]https://renaissance.dev/

```
{
    "DaCapo": {
        "kafka": { "java": ["-XX:+ExitOnOutOfMemoryError"] },
        "h2o": {
            "iterations": 5,
            "timeout": 900,
            "java": ["-Dsys.ai.h2o.debug.allowJavaVersions=21"]
        },
    }
}
```

Listing 4.1: Benchmarks Config Example

```
java -XX:+UseZGC -Xms4096m -Xmx4096m -Xlog:gc*,safepoint:file=example.log -r 10
--no-forced-gc
```

This command runs a Renaissance benchmark `scala-kmeans` using the Z Garbage Collector (see Section 3.1.3) while logging the garbage collection operations[5]. The option `Xlog:safepoint` in particular, logs the duration of each garbage collection pause, allowing us to compute the garbage collectors' average pause time and the 90^{th} percentile pause time, used in the garbage collection scoring, as mentioned earlier. During the benchmark execution, system statistics are collected every $0.10\ seconds$ using the `top`[6] command. The specific command used is `top -bn 1 -p process_id`. Here, the **-b** flag enables batch mode, while the **-n 1** option ensures the command runs only one iteration. By providing the $process\_id$ we can retrieve key performance metrics such as user-space $cpu\_time(us)$, representing the time spent in application code, $io\_time(wa)$, indicating the time spent waiting for I/O operations, and the percentage of cpu_usage of the benchmark process since previous **top** invocation. During the benchmark execution, if the process crashes or if the execution time surpasses the user's timeout value, a benchmark report is created with the error code and message (if present). An example of a benchmark report is shown in Listing 4.2.

The main algorithm in Listing 4.1 is responsible for running the selected benchmarks (or loading previous reports from the user's filesystem), as well as compiling every successful benchmark report into a Garbage Collector report (whose format is described in Listing 4.4). See Listing A.2 in Appendix A for a full example. After running all user-provided `benchmark_groups`, also known as benchmark suites, invalid benchmark reports are removed i.e., if a benchmark is not successful across all garbage collectors we can't use it to compute the score matrix. Having purged all invalid results from `benchmark_reports` the algorithm ends by creating an error report, so the user can see what failed in each benchmark, and the previously mentioned Garbage Collector report for each GC. The valid `benchmark_reports` are returned to compute the scoring matrix.

Finally, we compute the scoring matrix for the garbage collectors given the `benchmark_reports`. For

[5]https://docs.oracle.com/en/java/javase/11/jrockit-hotspot/logging.html#GUID-33074D03-B4F3-4D16-B9B6-8B0076661AAF
[6]https://man7.org/linux/man-pages/man1/top.1.html [8]

```
{
    "garbage_collector": "G1",
    "jdk": "HotSpot_21.0.4",
    "benchmark_group": "DaCapo",
    "benchmark_name": "avrora",
    "heap_size": "256",
    "error": null,
    "avg_cpu_usage": 17.5,
    "avg_cpu_time": 17.2,
    "avg_io_time": 0.1,
    "p90_io": 0.0,
    "number_of_pauses": 18,
    "total_pause_time": 15879736,
    "avg_pause_time": 882207.56,
    "pauses_per_category": {
        "G1CollectForAllocation": 7,
        "ICBufferFull": 1,
        "Cleanup": 10
    },
    "total_pause_time_per_category": {
        "G1CollectForAllocation": 15231688,
        "ICBufferFull": 57392,
        "Cleanup": 590656
    },
    "avg_pause_time_per_category": {
        "G1CollectForAllocation": 2175955.43,
        "ICBufferFull": 57392.0,
        "Cleanup": 59065.6
    },
    "p90_pause_time": 2329170.8,
    "throughput": 73643900878
}
```

Listing 4.2: Example of Benchmark Report

each combination of heap size and GC, the scores are given by the following equation:

$$\text{throughput}_{score} = \sum_{i=1}^{\#\text{benchmarks\_success}} \text{throughput}_i \tag{4.1}$$

$$\text{p90\_pause\_time}_{score} = \sum_{i=1}^{\#\text{benchmarks\_success}} \text{pause\_time}_i \tag{4.2}$$

As you can see in the Equation 4.2, to compute the *throughput* and *pause_time* scores we compute the sum between all successful benchmarks because we want each successful benchmark to have an equal contribution to the score. In the end, all scores will be normalized against the values of G1 garbage collector 3.1.1. The resulting matrix will follow the format shown in Listing 4.3. For a fully detailed matrix consult the Listing A.1 in Appendix A.

```
{
  "matrix": {
      "heap_size": {
          "garbage_collector" : {
              "throughput": <value>,
              "pause_time": <value>
          }
      }
  }
}
```

Listing 4.3: Structure of GC Scoring Matrix.

```
{
    "garbage_collector": <gc_name>,
    "jdk": <jdk>,
    "stats": [
        {
            "heap_size": <heap_size>,
            "number_of_pauses": <value>,
            "total_pause_time": <value>,
            "avg_pause_time": <value>,
            "p90_avg_pause_time": <value>,
            "avg_throughput": <value>,
            // List of benchmarks
            "benchmarks": ["..."]
        }
    ]
}
```

Listing 4.4: Structure of GC Report.

Algorithm 4.1: Run Benchmarks and build reports

```
Initialize benchmark_reports as a nested dictionary with default list values;
Initialize heap_sizes by calling get_heap_sizes();
Initialize failed_benchmarks as a nested dictionary with default list values;
foreach gc in garbage_collectors do
    foreach heap_size in heap_sizes do
        if skip_benchmarks is true then
            Load benchmark_reports[gc][heap_size] with reports for gc, heap_size, and jdk;
        else
            Append results of running benchmark_groups to
             benchmark_reports[gc][heap_size];
            Pass gc, heap_size, iterations, jdk, timeout, and benchmark_groups as
             arguments;
        foreach result in benchmark_reports[gc][heap_size] do
            if result is not successful then
                Append error details to
                 failed_benchmarks[result.heap_size][result.benchmark_name];
                Store result.garbage_collector and result.error;
Build error report using jdk and failed_benchmarks;
Update benchmark_reports by removing failed reports based on failed_benchmarks;
Build garbage collectors report using jdk and the updated benchmark_reports;
return benchmark_reports;
```

Algorithm 4.2: Run Benchmark Algorithm

```
Input: benchmark group, benchmark, gc, heap size, iterations, jdk, timeout
Output: Benchmark Report or Error
Function kill_process(process, cmd):
    process.kill();
benchmark_command ← get_benchmark_command(benchmark_group, benchmark, gc,
  heap_size, iterations);
file ← DummyTimerAndFile();
file_path ← get_benchmark_debug_path(gc, benchmark_group.value, benchmark, heap_size);
if debug then
    file ← open(file_path, "w");
    process ← subprocess.Popen(benchmark_command, stdout=file,
      stderr=subprocess.STDOUT);
else
    process ← subprocess.Popen(benchmark_command, stdout=subprocess.PIPE,
      stderr=subprocess.PIPE);
timer ← if timeout == None then
    DummyTimerAndFile();
else
    Timer(timeout, kill_process, (process, benchmark_command));
time_start ← time.time_ns();
pid ← process.pid;
cpu_usage_stats, cpu_time_stats, io_time_stats ← [], [], [];
timer.start();
while process.poll() == None do
    p ← subprocess.run(["top", "-bn", "1", "-p", pid], capture_output=True, text=True);
    lines ← p.stdout.splitlines();
    us, wa ← extract_cpu_io_times(lines);
    io_time_stats.append(wa);
    cpu_time_stats.append(us);
    cpu_usage ← calculate_cpu_usage(lines);
    cpu_usage_stats.append(cpu_usage);
    time.sleep(0.1);
timer.cancel();
throughput ← time.time_ns() - time_start;
cpu_usage_avg ← mean(cpu_usage_stats);
cpu_time_avg ← mean(cpu_time_stats);
io_time_avg ← mean(io_time_stats);
p90_io ← percentile(io_time_stats, 90);
if process.returncode == 0 then
    result ← build_benchmark_report(gc, benchmark_group.value, benchmark, heap_size,
      cpu_usage_avg, cpu_time_avg, io_time_avg, p90_io, throughput, jdk);
else
    if debug then
        error ← "Check " + file_path + " for error logs.";
    else if process.stderr then
        error ← process.stderr.read().decode();
    result ← build_benchmark_error(gc, benchmark_group.value, benchmark, heap_size,
      cpu_usage_avg, cpu_time_avg, io_time_avg, p90_io, jdk, process.returncode, error);
file.close();
result.save_to_json();
return result;
```

4.2 BestGC++

As mentioned before in Section 2.3, BestGC is a tool that aims to select, as the name implies, the best Garbage Collector for a user-given Java application, and it does so by previously profiling multiple Java applications to score GCs based on metrics. Afterwards, when profiling a given application, it can use the previously collected data and select the best GC based on some parameters. The benchmark profiling was already covered in the previous Section 4.1, so we will now dedicate to explaining what modifications and improvements were made to create BestGC++.

4.2.1 Metrics and Parameters Rational

One of the possibilities to improve the BestGC Garbage Collector selection is to generate different matrices based on a value of some metric gathered while executing the application, which could enhance GC selection accuracy due to the increased matrix specificity. Previously, the authors of BestGC tried to classify applications into CPU and non-CPU intensive, looking at the CPU usage during benchmark execution. However, this raises some problems:

- What should be the chosen CPU usage percentage threshold for classifying an application as CPU-intensive or non-CPU-intensive?
- Once that value is chosen, we will categorize our available benchmarks into two groups, which may result in an imbalance of data between them or a lack of data for a specific matrix category.
- An application with CPU usage close to the selected threshold may experience significantly worse performance if it ”lands” on the incorrect side.

Another approach explored was classifying applications as either I/O or CPU intensive. To do this, we initially used the **top**[8] tool, focusing on the ”wa: time waiting for I/O completion” metric (in percentage). To understand how this metric behaves, we created various test cases with differing levels of file manipulation. After running these examples, it became clear that to trigger a high rise in **wa%**, the application needs to spend significantly more time waiting for I/O operations than performing CPU work. This could be indicative of an issue within the application. Moreover, even if we set a low **wa%** as a threshold to classify applications as I/O or CPU intensive, there remains the possibility that an application with high CPU usage could still be classified as I/O intensive. Switching to a different classification method, such as monitoring disk reads using **iostat**[7], would still suffer from the same limitations.

The key takeaway is that, given the complex interplay between multiple factors like I/O and CPU utilization, a simplistic binary classification would create a false dichotomy and fail to capture the full nuance of the data.

[7] https://linux.die.net/man/1/iostat

After exploring these possibilities, we shifted our focus to alternative methods for improving GC selection while also enhancing user experience. Although we ultimately decided against these ideas, they provided valuable insights into the metrics that can affect application performance, and, as a result, we are now tracking them with the BenchmarkGC profiling application (see 4.1).

As mentioned earlier, running BestGC currently requires users to manually assign weights based on their desired emphasis on *throughput* and latency, the latter defined as GC *pause_time*. A way to enhance user experience would be to automatically determine the optimal weights, allowing users with minimal technical expertise to run their applications with the utmost performance. To solve that, we return to the topic of CPU usage, we made the following observation: if an application shows high CPU utilization during profiling, it suggests that a GC that competes with the application for CPU resources would be detrimental, impacting the already stressed application. This means the GC should prioritize application throughput and minimize interference. Based on this insight, we decided to compute the throughput weight using the application's average CPU usage, *cpu_avg* for short. In the piecewise equation 4.3, we clamp average CPU values below 30% to 0 and those above 90% to 1. For average CPU values within this range, we apply a linear function to ensure a smooth transition.

$$\text{throughput\_weight} = \begin{cases} 0 & \text{cpu\_avg} \in [0, 30] \\ \frac{\text{cpu\_avg}}{60} - 0.5 & \text{cpu\_avg} \in [30, 90] \\ 1 & \text{cpu\_avg} \in [90, 100] \end{cases}, \quad 0 \leq \text{cpu\_avg} \leq 100 \tag{4.3}$$

Important to remember that given the computed *throughput* weight value the *pause_time* weight will be $\text{pause\_time}_{weight} = 1 - \text{throughput}_{weight}$.

4.2.2 Application Architecture and Overview

To further enhance user experience, we enabled BestGC to function as a web application. This approach simplifies the process for users, allowing them to submit their compiled Java application (in Jar format[8]) - from now on referred to only as java application - and specify parameters to achieve optimal performance with minimal effort.

We developed the web application using *Spring*[9], a widely adopted open-source framework for building web applications in Java. The architecture is designed to support flexibility and performance profiling for Java applications, with the following key services:

[8]https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html
[9]https://spring.io/

Core Services:

Matrix Service: This service is responsible for loading the matrix generated by BenchmarkGC (see Section 4.1). Once the matrix is loaded, it handles the scoring of all Garbage Collectors (GCs). If no weights are provided, it will automatically calculate them using the equation in 4.3.

Profile Service: As the name suggests, this service profiles the user's Java application. During profiling, it captures metrics such as heap size, CPU usage, I/O wait time, and CPU time percentages (which correspond to the **us** and **wa** values from the **top**[8] tool). These metrics provide insight into the application's performance characteristics.

Run Service: This service manages the execution of the user's Java application. It tracks running applications and stores relevant information such as process IDs, application names, and the commands used to execute them. This service is integral for managing multiple runs and gathering runtime data.

Main Endpoints for User Functionality

POST /profile_app: This endpoint allows users to submit their Java application along with any necessary arguments. Optionally, users can specify *throughput* and *pause_time* weights. The application is profiled, and then executed with the best-performing GC based on the profiling results. The final execution can be toggled on or off by the user.

POST /run_app: Similar to `/profile_app`, but in this case, the user manually selects the heap size and GC without a profiling stage. The application is run directly with the specified parameters.

GET /poll_apps?ids=application_ids: This endpoint accepts a list of comma-separated application IDs and returns the current performance metrics for the specified applications. The metrics are gathered from the **Profile Service**, giving users a real-time view of heap usage, CPU consumption, and more.

Now that we have covered the architecture, we can focus on the user workflow. With an emphasis on simplicity and minimizing friction in the user experience, the workflow for getting an application up and running follows these steps:

1. **Accessing the Application**: The user navigates to the application's root endpoint e.g., http://your-domain.com.
2. **Input Submission**: The user fills in his application arguments (if applicable) and uploads their Java application.
3. **Profiling and Execution**: The application is then profiled, weights are computed, optimal heap size is determined, and it runs automatically.

4. **User Notification and Dashboard Access**: The user receives a notification indicating that the application is being profiled and is redirected to a dashboard when done. Here, they can view all running applications and monitor performance metrics.

Expanding on item 3, the BestGC++ application employs a new approach to profiling. Since we use CPU usage percentage as a key metric for selecting the most suitable Garbage Collector, it is crucial to manually specify the heap size during the profiling process. We begin with a minimum heap size of 256MB and double it until we reach a size that allows the application to run successfully. This method ensures that the captured CPU usage metrics closely resemble those observed during the application's final execution, allowing us to accurately determine the appropriate *throughput* weight.

An important implementation detail is the inclusion of a dashboard on the BestGC++ application, which is crucial for its users, transforming it into a full-fledged service. This dashboard provides valuable insights into application performance, including metrics related to I/O, CPU usage, and heap size. These metrics are displayed only when the user expands the application details, reducing the number of unnecessary web requests.

Lastly, while BestGC++ can now function as a service, it is important to note it is still fully capable of being used as a console application while using the new automatic weight calculation functionality.

4.3 Summary

In this chapter, we presented the solution implemented to enhance BestGC by introducing two major components: BenchmarkGC and BestGC++, each designed to optimize the selection of the best-performing garbage collector (GC) for a given Java application.

The BenchmarkGC tool was developed to automate the generation of a scoring matrix, which classifies available GCs based on their *throughput* and *pause_time* performance across multiple Java benchmarks. The tool supports a flexible configuration, allowing users to run benchmarks with custom JVM options, benchmark iterations, and timeouts. It also generates detailed reports for each GC and benchmark, which are then compiled into a scoring matrix. This matrix provides normalized scores for each GC, making it easy to compare their performance in a user's environment.

The second enhancement, BestGC++, improves upon the original BestGC by introducing new features and improving its user experience. BestGC++ now functions as a web service, allowing users to upload their compiled Java applications (in JAR format), profile them, and automatically select the best GC for optimal performance using the scoring matrix generated by BenchmarkGC. Furthermore, BestGC++ introduces automatic weight determination based on application profiling, especially focusing on CPU usage, allowing it to dynamically balance *throughput* and *pause_time* weights without requiring manual input from the user. Despite these enhancements, BestGC++ retains the functionality of its pre-

decessor, allowing it to operate as a console application while incorporating the new weight calculation feature.

The chapter concludes with the architecture and workflow of BestGC++, detailing how it profiles applications, computes GC scores, and facilitates user interaction through a web-based dashboard. This comprehensive solution simplifies the process of selecting the best GC, reducing the complexity for users while ensuring their Java applications run with optimal performance. In the next Chapter 5 we will test these developed tools to assess their improvements and effectiveness.

5

Evaluation

Contents

5.1 Overview

This chapter provides a comprehensive evaluation of the two GC profiling applications developed, as detailed in Section 4. We begin with an analysis of the results obtained with the BenchmarkGC application (see Section 4.1), an application designed to facilitate the benchmark of various Garbage Collectors (GCs). By utilizing this application in combination with different Java Development Kits (JDKs), we can derive meaningful insights into the performance of these GCs and Java runtimes. To carry out this evaluation, we selected two widely used Java runtimes, Oracle HotSpot and Oracle GraalVM with Just-in-Time (JIT) compiler. These runtimes are frequently deployed in real-world applications, making them

ideal for performance comparison. Additionally, we focused on benchmarking three prominent garbage collectors: G1, Parallel, and ZGC. However, a key limitation must be noted. As previously stated in Section 3.1.3, Generational ZGC was introduced in Oracle HotSpot, version 21, however, this feature is not available for GraalVM (with native image) (see Section 3.3.1), which lacks support for ZGC entirely. Furthermore, Graal JIT only supports non-Generational ZGC[1], so as a result, any comparisons of ZGC between these two runtimes (HotSpot and GraalVM JIT) will use the default non-Generational version.

The primary metrics we will focus on during our evaluation are *throughput*-the time taken to complete each workload (see workload definition in Section 5.2)-and the p90 GC *pause_time*, which represents the 90^{th} percentile of garbage collection pause durations. These metrics will allow us to assess the overall performance and efficiency of the garbage collectors across different configurations and runtimes.

To evaluate BestGC++, we must select an application that didn't contribute to the results obtained during the matrix generation phase (mentioned in Section 4.1). This ensures an unbiased evaluation of BestGC's ability to identify the appropriate Garbage Collector for a given random application. Only after analyzing the application with the BenchmarkGC tool, can we determine if BestGC was capable of selecting the correct GC for the application, based solely on the previously obtained data.

5.2 Testbed and Hardware Specifications

The evaluation was conducted on a machine running Arch, a Linux-based Operating System, equipped with an i7 6700k CPU (4 cores, 4.00 GHz) and 16GB of RAM. For runtime performance profiling, we used Oracle HotSpot and Oracle GraalVM with JIT both with version *21.0.4*. In this evaluation, we define **Benchmarks** as a group consisting of multiple **Workloads**, each of which, is a distinct computational task designed to stress test the Java runtime and Garbage Collector. The purpose of these benchmarks is to provide a diverse set of workloads to evaluate how different GCs perform under various conditions. The Benchmarks and Workloads used with the tool BenchmarkGC are listed in Table 5.1. The DaCapo and Renaissance benchmarks were selected due to the wide range of real-world workloads they encompass, from small to big applications, that are frequently selected to profile garbage collectors (GCs) and Java runtimes.

To evaluate BestGC, we chose **Spring PetClinic**[2], a very popular application in the Java ecosystem due to being an entry point to the Spring framework[3]. This application was chosen because it is a typical Java web application. However, if we examine the DaCapo workloads we can see that it has a **spring** workload, which indeed uses the **Spring PetClinic**. We will remove that workload from the BenchmarkGC, meaning it will not be used in the matrix generation phase, which as highlighted in

[1]Generational ZGC already exists in the development branch of GraalVM JDK, being planned for 24.1 release https://github.com/oracle/graal/issues/8117

[2]https://github.com/spring-projects/spring-petclinic

[3]Spring is an open source framework for the Java language

Table 5.1: List of Benchmarks/Workloads used by BenchmarkGC - spring is removed, so it can be used to test BestGC++.

Benchmarks	Workloads
DaCapo	avrora batik biojava cassandra eclipse fop graphchi h2 h2o jme jython kafka luindex lusearch pmd ~~spring~~ sunflow tomcat tradebeans tradesoap xalan zxing
Renaissance	scrabble page-rank future-genetic akka-uct movie-lens scala-doku chi-square fj-kmeans rx-scrabble neo4j-analytics finagle-http reactors dec-tree scala-stm-bench7 naive-bayes als par-mnemonics scala-kmeans philosophers log-regression gauss-mix mnemonics dotty finagle-chirper

the previous Section 5.1, is fundamental for an unbiased assessment of BestGC++'s decision-making capabilities.

5.3 BenchmarkGC Evaluation

5.3.1 Performance and Benchmark Analysis

We will start by analyzing both benchmarks (DaCapo and Renaissance) independently on the two chosen Java runtimes (HotSpot and GraalVM) with respect to *throughput* and p90 *pause_time*. Each benchmark was evaluated with varying heap sizes, allowing us to extract how different GCs react when facing varying degrees of stress in terms of memory availability. Due to the existence of a multitude of workloads with different requirements, some couldn't run while using a specific combination of Garbage Collector and heap size. In spite of this being a very good pointer while choosing a Garbage Collector for environments with low available memory, we couldn't correctly attribute a *throughtput* and *pause_time* value for GCs that couldn't complete said workload. Due to this fact, we will see that for smaller heap sizes there are less considered workload results, due to the necessity of pruning them from the results.

Very important to reiterate, that *throughput* and *pause_time* are both metrics with time units, meaning that a low score is **always** better. For *throughput* meaning the workload took less time to complete, and for *pause_time* meaning that the GCs' duration of pauses is smaller.

HotSpot Results

Beginning the analysis with an overview of the number of workloads used, and how many were successful i.e., run until completion with no errors for each Garbage Collector. This information is in Table 5.2. In total 45 workloads were executed, and all GCs were successful when using heap sizes greater than 2048MB. However, we can see that for smaller heap sizes, G1 has more workloads run until completion e.g., at 256MB heap size it completes successfully three more workloads than Parallel GC and two more than ZGC. This tells us that G1 can better handle systems with less memory available, not indicative,

however, that the performance will be in any way acceptable.

Table 5.2: HotSpot - Workloads overview.

Heap Sizes (MB)	GC	Success	Error
256	G1	34	11
	Parallel	31	14
	Z	32	13
512	G1	41	4
	Parallel	40	5
	Z	40	5
1024	G1	44	1
	Parallel	44	1
	Z	43	2
2048	G1	45	0
	Parallel	45	0
	Z	45	0
4096	G1	45	0
	Parallel	45	0
	Z	45	0
8192	G1	45	0
	Parallel	45	0
	Z	45	0

DaCapo Results Let's begin by examining the DaCapo results in terms of the p90 *pause_time*, as shown in Figure 5.3. The most notable takeaway is the consistently low pause times achieved by the nearly fully concurrent ZGC. Across all workloads in the DaCapo benchmark, regardless of the heap size, ZGC manages to maintain p90 pause times under 0.01 seconds.

In contrast, the other garbage collectors, G1 and Parallel, show similar results across most workloads, although the G1 Garbage Collector shows slightly higher pause times across most workloads. The notable exception is when using a heap size of 256MB (see Figure 5.3a). In that case, the Parallel GC exhibited higher pause times in 18 out of 19 benchmarks compared to G1. Parallel GC consistently shows a higher maximum pause time across most heap sizes (see Figure 5.3), with the exception of Figure 5.3f, where G1 has the higher p90 *pause_time*, although the difference is minimal.

The data raises some questions regarding the inconsistency of *pause_time* values when the heap size is modified. One might expect that increasing the heap size would lead to lower pause times across all workloads, as more memory should alleviate the system's memory pressure, reducing the likelihood of longer Full GC pauses. However, this is not always the case. For instance, in the **batik** workload,

the p90 *pause_time* increases from 0.19 seconds with a 256MB heap (Figure 5.3a) to 0.27 seconds with a 512MB heap (Figure 5.3b) when using the Parallel GC. A plausible hypothesis is that with a larger heap, the total amount of garbage that accumulates and needs to be collected also increases, potentially causing longer pauses during GC phases. However, this does not necessarily indicate a negative trend. If we examine additional data (see Listing 5.1) provided by the BenchmarkGC tool (see Section 4.1), we can observe that the **average pause_time** decreased when the heap size increased to 512MB. This suggests that while the p90 pause time might occasionally increase with a larger heap, the overall pause behaviour of the system may still improve, with shorter average pauses benefiting the system's performance.

```
{
"avg_pause_time": 102161303.76,
"avg_pause_time_per_category": {
    "Cleanup": 19367.2,
    "ParallelGCFailedAllocation":
    107023688.71,
    "ICBufferFull": 130851.5
}}
```

(a)

```
{
"avg_pause_time":  76053790.0,
"avg_pause_time_per_category": {
    "Cleanup": 33892.33,
    "ParallelGCFailedAllocation":
    89975202.93,
    "ICBufferFull": 220712.0
}}
```

(b)

Listing 5.1: DaCapo Hotspot batik - a) 256 MB heap size and b) 512 MB heap size - Parallel GC Pause times (in ns)

```
{
    "throughput": 575998378675,
    "number_of_pauses": 1520,
    "total_pause_time": 653447466
}
```

(a)

```
{
    "throughput": 347383293671,
    "number_of_pauses": 659,
    "total_pause_time": 366088051,
}
```

(b)

Listing 5.2: DaCapo Hotspot h2 - a) 2048 MB heap size and b) 4096 MB heap size - ZGC Pause Time and Throughput (in ns)

Let's now shift our focus to the *throughput* results. After analyzing the *pause_time* data, it is reasonable to expect that the Z Garbage Collector (ZGC) would underperform in terms of throughput compared to more throughput-optimized GCs like G1 and Parallel. As shown in Figure 5.4, this assumption holds true. The results show a clear pattern where, in terms of throughput (remember, a smaller throughput value is better), the Garbage Collectors rank as follows: $Z > G1 > Parallel$.

Interestingly, the results of the workload **batik** show that in spite of having a very high p90 *pause_time*, exhibits remarkably low *throughput*. This suggests that this workload is highly memory-intensive, placing

significant stress on the Garbage Collector while being less computationally demanding intensive than other workloads.

Most workloads, tend to display similar values independent of the heap size used. However, an obvious exception is the **h2** workload when using ZGC. To understand this anomaly, we need to delve deeper into the statistics (refer to Listing 5.2). One potential explanation is the substantial reduction in the number of pauses (nearly 2.3 times fewer), which results in a 1.8 fold reduction in total pause time. Nevertheless, this alone cannot fully account for the dramatic 228-second reduction in execution time when increasing the heap size from 2048MB to 4096MB. It suggests that other underlying factors, possibly related to memory management efficiencies or reduced GC pressure, may also be at play, further improving the workload's performance at larger heap sizes.

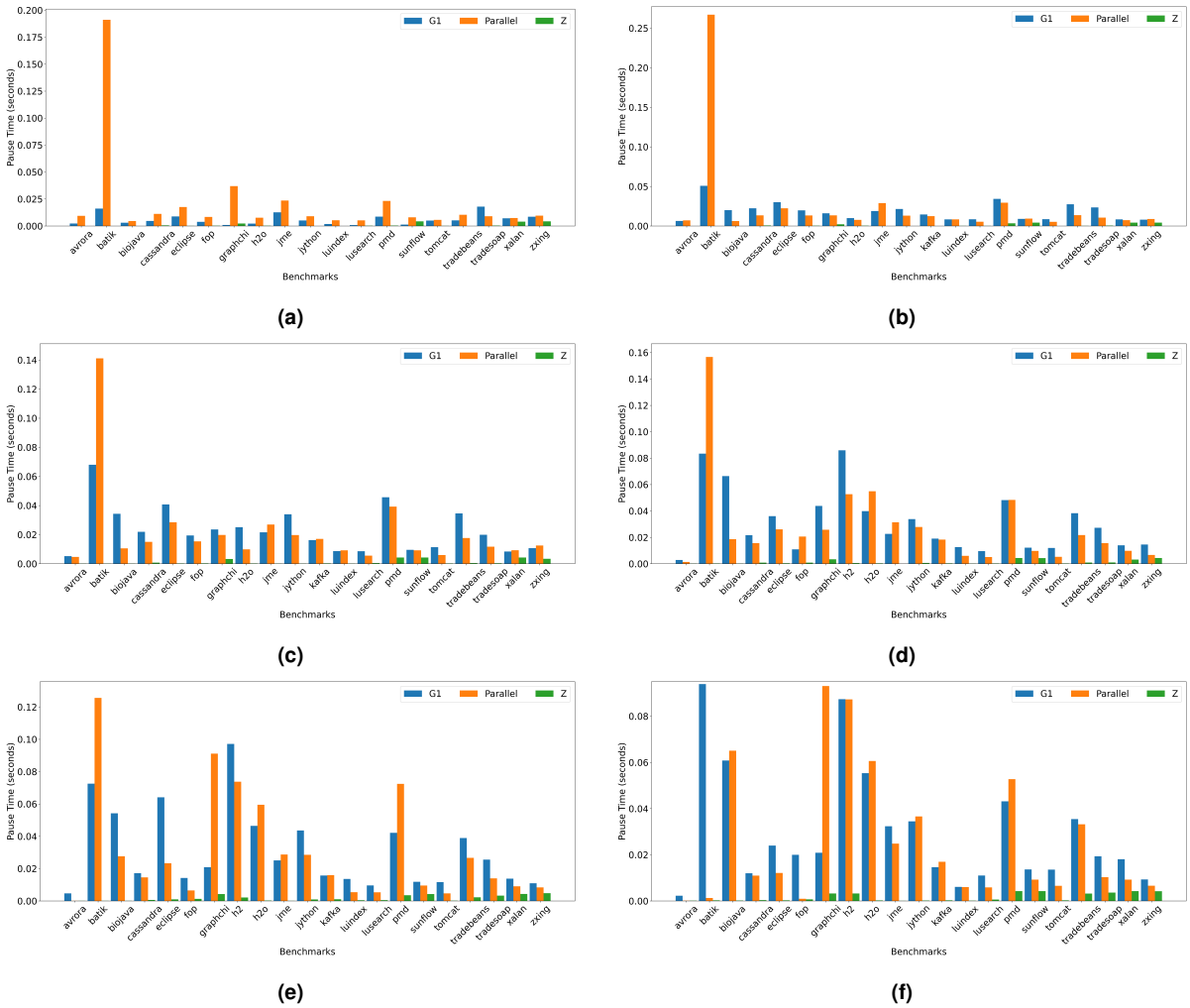

Figure 5.3: HotSpot DaCapo Results (P90 Pause Time - Lower is better) - The following graphs show P90 pause time with different heap sizes across all DaCapo Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

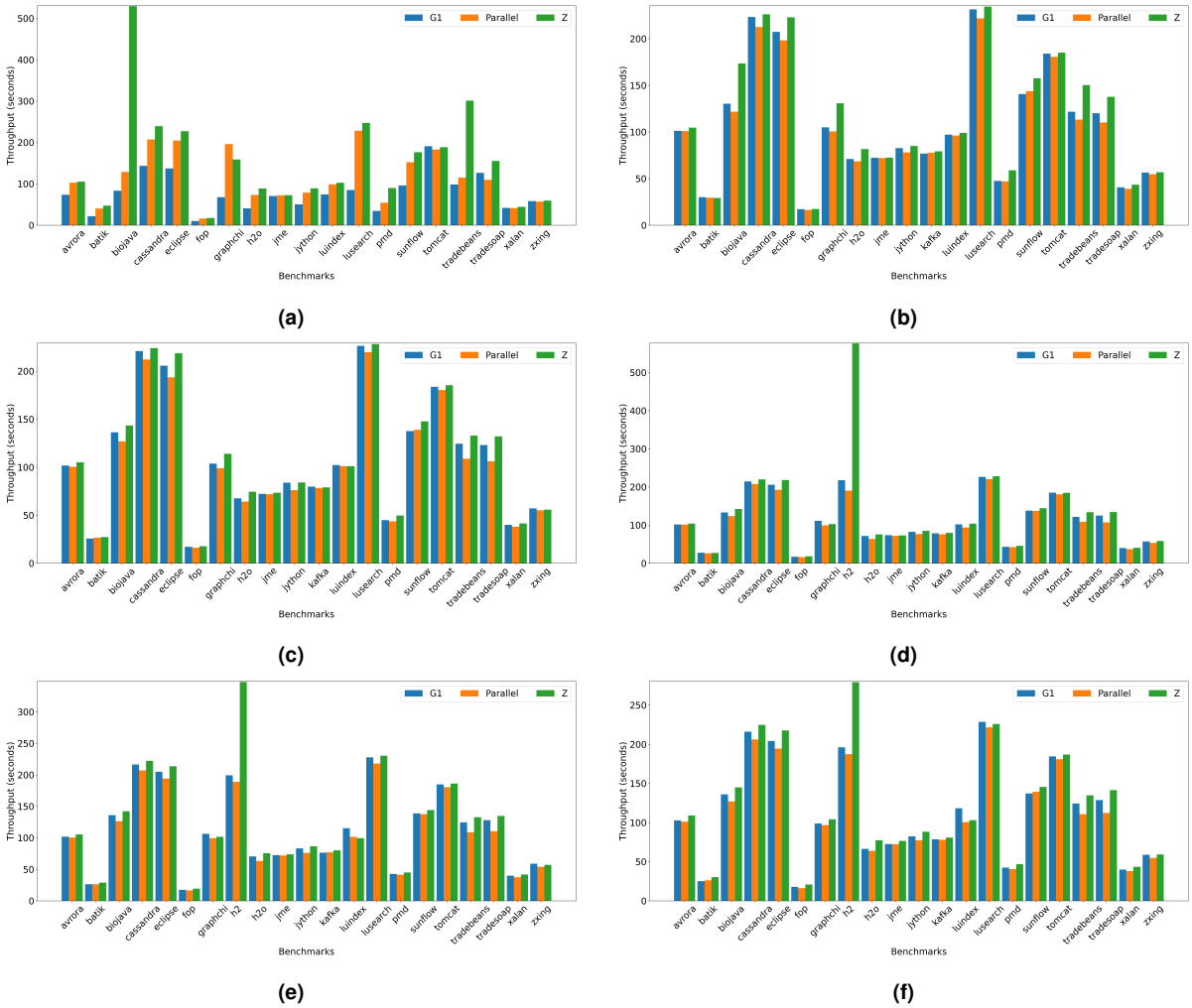

Figure 5.4: HotSpot DaCapo Results (Throughput - Lower is better) - The following graphs show throughput time with different heap sizes across all DaCapo Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

Renaissance Results Let's now analyze the Renaissance Benchmark results, focusing on *pause_time*. Figure 5.6 presents the findings, which, similar to the previous results obtained of DaCapo in Figure 5.3, show that Z Garbage Collector continues to present impressive low p90 *pause_time* values. However, in contrast to what happened in the DaCapo 256MB (see Figure 5.3a), the Renaissance workloads reveal a different trend for the G1 and Parallel GCs. Specifically, in 9 out of 12 workloads with a 256MB heap size, Parallel GC shows lower p90 pause times than G1 GC (Figure 5.6a). The same pattern can be observed for the remaining heap sizes: 13/20 (512MB); 16/23 (1024MB); 17/24(2048MB). However, for a heap size of 4096MB, there is a tie i.e., there are equal amounts of workloads where G1 and Parallel outperform each other. With an 8192MB heap size (see Figure 5.6f), the trend reverses, with Parallel GC showing worse p90 pause times compared to G1 in 15 out of 24 workloads.

An interesting exception occurs with the workload **scala-kmeans**, where ZGC records the worst

p90 *pause_time* among all GCs. This is due to the low memory stress in this workload at larger heap sizes, which favours throughput-oriented GCs. In these scenarios, ZGC, designed to pause more often to avoid full garbage collection cycles (see 3.1.3), maintains consistent *pause_time* values across all heap sizes. While, on the other hand, G1 and Parallel when not under heavy heap applications, avoid long pauses altogether as you can see on the Listing 5.3. At 4096MB heap size, despite the number of pauses remaining very similar, the *pause_time* increases again due to the following pauses: **ParallelGCFailedAllocation** in Parallel GC and **G1CollectForAllocation** in G1 GC.

```
{
"Parallel": {
    "total_pause_time_per_category": {
        "ICBufferFull": 125121,
        "Cleanup": 95571,
        "ParallelGCFailedAllocation":
        19036923
    }
},
"G1":{
    "total_pause_time_per_category": {
        "ICBufferFull": 65346,
        "G1CollectForAllocation": 17317822
    }
}}
```

(a)

```
{
"Parallel": {
    "total_pause_time_per_category": {
        "ICBufferFull": 62346,
        "Cleanup": 104511
    }
},
"G1":{
    "total_pause_time_per_category": {
        "ICBufferFull": 57755,
        "Cleanup": 103218
    }
}}
```

(b)

Listing 5.3: Renaissance Hotspot scala-kmeans - a) 4096 MB heap size and b) 8192 MB heap size - Pause Time per category for G1 and Parallel GCs(in ns)

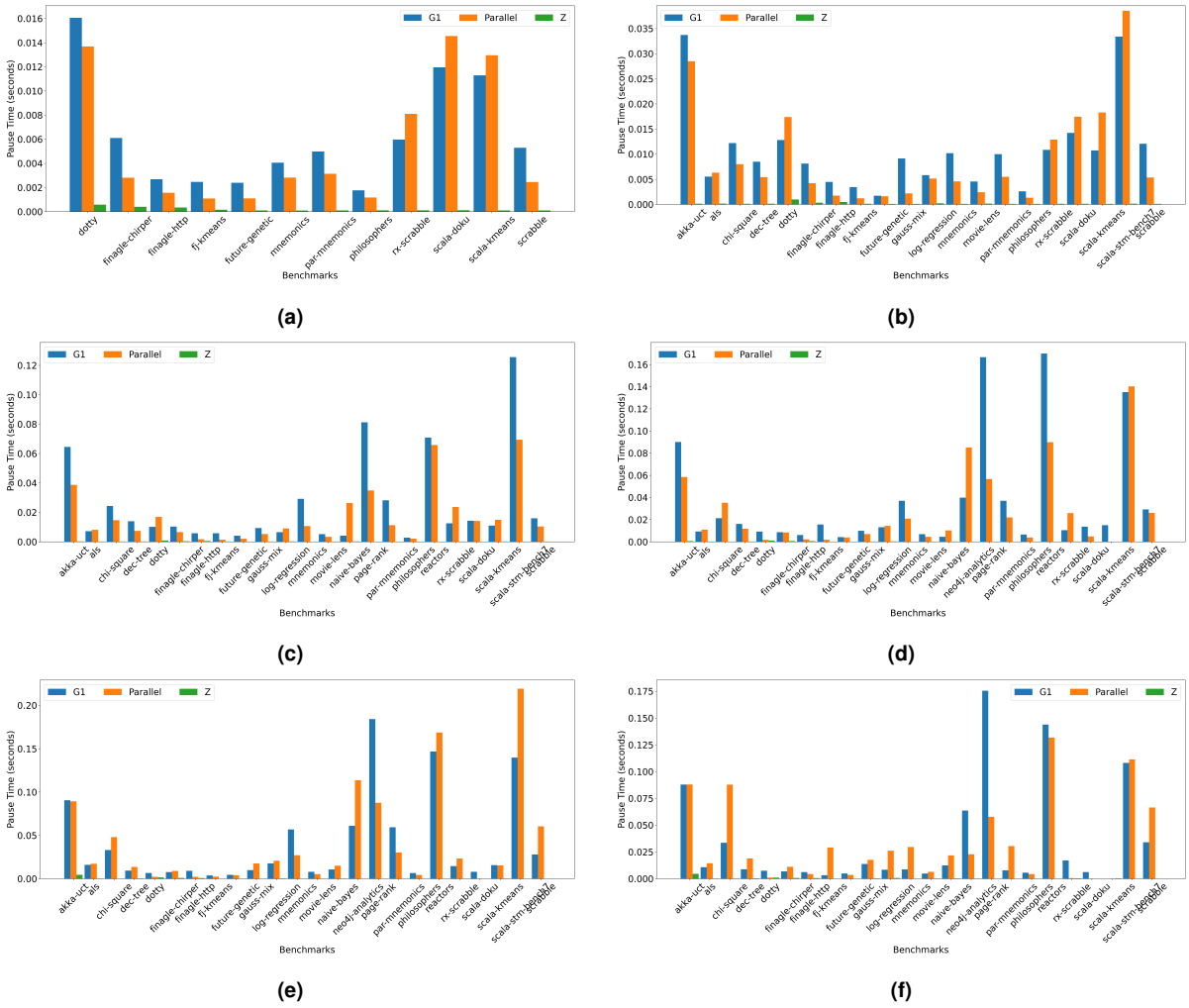

Figure 5.6: HotSpot Renaissance Results (P90 Pause Time - Lower is better) - The following graphs show P90 pause time with different heap sizes across all Renaissance Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

Starting the *throughput* analysis of the Renaissance workloads, we first observe that, similar to the results from the DaCapo Benchmark (refer to Figure 5.4), ZGC consistently shows higher throughput values across all heap sizes (see Figure 5.8). Moreover, the Parallel GC outperforms both G1 and ZGC in terms of *throughput*, achieving consistently lower values across all tested heap sizes. Notably, the workload **akka-uct** in Figure 5.8b stands out due to a difference exceeding 300 seconds in throughput between ZGC and the other GCs. This gap narrows to less than 50 seconds only when the heap size reaches 4096MB. Examining the data in Listing 5.4, we can confirm that ZGC's total pause time for a 512MB heap is significantly lower-100 times less than Parallel and 87 times less than G1. When the heap size increases to 4096MB, ZGC's pause time decreases by a factor of 1.24, while the pause times for Parallel and G1 decrease more sharply (by 3.69 and 6.24 times, respectively). This reduces the gap, with Parallel's pause time being 33 times higher than ZGC's and G1's being 17 times higher,

raising further questions about ZGC's throughput decline. One might intuitively attempt to explain this conundrum with an equation like the following:

$$throughput_{\text{512MB}} - throughput_{\text{4096MB}} \approx total\_pause\_time_{\text{512MB}} - total\_pause\_time_{\text{4096MB}} \tag{5.1}$$

Based on the data in Listing 5.4, we apply and evaluate the previous Equation 5.1:

Parallel:

$$\frac{103483032311 - 76203010562}{10^9} = 27.28$$
$$\frac{47119299019 - 12759848804}{10^9} = 34.36$$
$$27.28 \approx 34.36$$

G1:

$$\frac{109273124016 - 74241081333}{10^9} = 35.03$$
$$\frac{40811559264 - 6543404594}{10^9} = 34.27$$
$$35.03 \approx 34.27$$

Z:

$$\frac{425930406666 - 105217020071}{10^9} = 320.71$$
$$\frac{466408601 - 376147881}{10^9} = 0.09026072$$
$$320.71 \not\approx 0.09026072$$

After calculating Equation 5.1 for each GC, we conclude that the relationship between throughput and pause time holds for both G1 and Parallel GCs. However, it is not a perfect one-to-one mapping. For instance, in the case of Parallel GC, there is a 7-second discrepancy between both sides of the equation. This difference could be attributed to factors such as concurrent GC overhead or non-GC-related operations like I/O, thread contention, or general system overhead. A key factor likely explains why the equation does not hold for ZGC: **allocation stalls**. Allocation stalls occur when the memory allocation rate exceeds the speed at which the GC can reclaim memory. When this happens, ZGC by not having the ability to do an STW pause to retrieve large memory chunks, will essentially behave like a Stop-the-World GC until it retrieves enough memory, greatly impacting throughput.

```
{
"Parallel": {
    "total_pause_time": 47119299019,
    "throughput": 103483032311
},
"G1":{
    "total_pause_time": 40811559264,
    "throughput": 109273124016
},
"Z":{
    "total_pause_time":   466408601,
    "throughput": 425930406666
}}
```

(a)

```
{
"Parallel": {
    "total_pause_time": 12759848804,
    "throughput": 76203010562
},
"G1":{
    "total_pause_time": 6543404594,
    "throughput": 74241081333
},
"Z":{
    "total_pause_time": 376147881,
    "throughput": 105217020071
}}
```

(b)

Listing 5.4: Renaissance Hotspot akka-uct - a) 512 MB heap size and b) 4096 MB heap size - Throughput and Total Pause Time(in ns) for all GCs

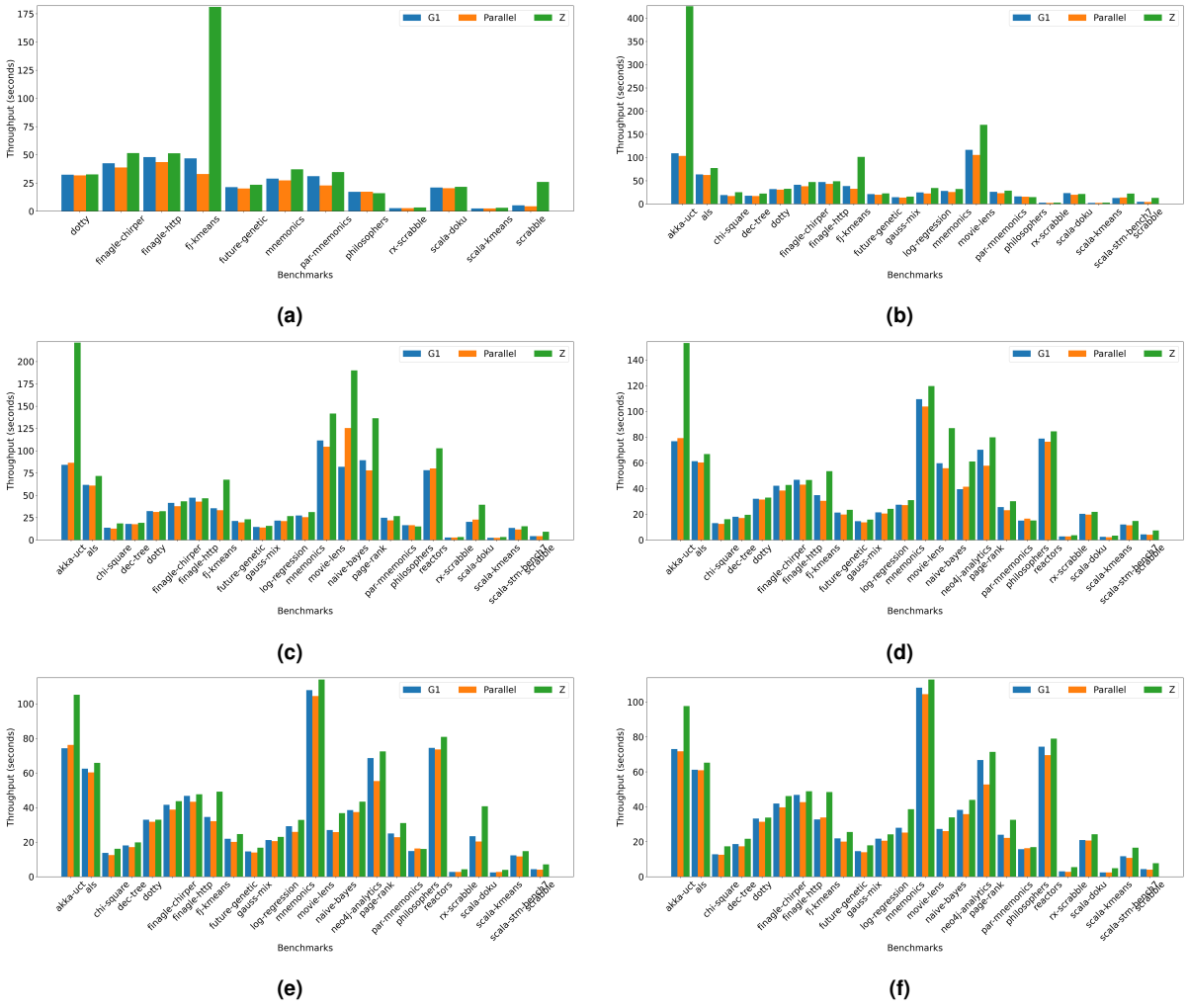

Figure 5.8: HotSpot Renaissance Results (Throughput - Lower is better) - The following graphs show throughput time with different heap sizes across all Renaissance Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

GraalVM Results

With the transition to the GraalVM runtime, we first present the number of workloads profiled, as shown in Table 5.3. Similar to the results obtained with the HotSpot runtime, all workloads were successfully executed with heap sizes of 2048 MB or larger. A comparison between Table 5.3 and Table 5.2 reveals no significant differences. This indicates that, under the GraalVM runtime, G1 remains the superior choice when dealing with memory-constrained environments.

Table 5.3: Graal - Workloads overview.

Heap Sizes (MB)	GC	Success	Error
256	G1	34	11
	Parallel	31	14
	Z	32	13
512	G1	41	4
	Parallel	40	5
	Z	40	5
1024	G1	44	1
	Parallel	44	1
	Z	43	2
2048	G1	45	0
	Parallel	45	0
	Z	45	0
4096	G1	45	0
	Parallel	45	0
	Z	45	0
8192	G1	45	0
	Parallel	45	0
	Z	45	0

DaCapo Results Beginning with the DaCapo Benchmark, we analyze the results in terms of the p90 *pause_time*, representing the 90^{th} percentile of pause times. As shown in Figure 5.10, the Z Garbage Collector consistently exhibits lower *pause_time* values across all heap sizes. Focusing on Figure 5.10a, which presents workload results for a heap size of 256 MB, we observe that the Parallel GC demonstrates lower *pause_time* values compared to G1 in 13 out of 19 workloads. This trend persists across different heap sizes; for instance, in Figure 5.10d, Parallel GC outperforms G1 in 17 out of 21 workloads at a heap size of 2048 MB.

Interestingly, despite being a throughput-oriented garbage collector, Parallel GC achieves lower p90 *pause_time* values than G1. To illustrate this, let's analyze the **h2** workload at a heap size of 2048MB (see Figure 5.10d), which clearly demonstrates that G1 exhibits a worse p90 *pause_time* compared to Parallel GC. As shown in Listing 5.5, G1 has a lower total pause time of less than 1.35 seconds when compared to Parallel, primarly due to having 100 fewer pauses while maintaining a similar average *pause_time* (only 0.00617 seconds higher). Notably, the difference in the p90 percentile is more signifi-

cant, at 0.033 seconds. So for more restricted *pause_time* requirements, Parallel GC would still be the safer option.


```
{
"p90_pause_time": 83211244.4,
"number_of_pauses": 332,
"total_pause_time": 13331332086,
"avg_pause_time": 40154614.72
}
```


(a)


```
{
"p90_pause_time": 50437108.6,
"number_of_pauses": 432,
"total_pause_time": 14681780210,
"avg_pause_time": 33985602.34
}
```


(b)

Listing 5.5: DaCapo Graal h2 - a) G1 and b) Parallel - Pause time details for 2048MB Heap Size (in ns)

(a) **(b)** **(c)** **(d)** **(e)** **(f)**

Figure 5.10: Graal DaCapo Results (P90 Pause Time - Lower is better) - The following graphs show P90 pause time with different heap sizes across all DaCapo Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

Turning to the *throughput* results in DaCapo depicted in Figure 5.12, we start by referring that the Z Garbage Collector, as expected due to its latency-oriented architecture, shows higher *throughput* values across all heap sizes, being the worse GC with respect to throughput. Analogously to what happened in the *pause_time* results, Parallel GC exhibits better performance compared to G1. However, it is worth mentioning that the difference in *throughput* between all GCs is pretty low across most workloads, with only ZGC exhibiting larger differences due to allocation stalls (see workload h2 in Figure 5.12). However, one of the workloads at 256MB of heap size exhibits a different pattern, meaning that the Parallel GC got worse *throughput* among all GCs, the workload **graphchi**. To diagnose the motives for why this happened we need to consult the data in Listing 5.6, and it becomes pretty evident when we observe that the difference in *throughput* between G1 and Parallel is approximate to the difference in total pause time ($|-92.47| \approx |-90.76|$).

Turning to the throughput results in DaCapo depicted in Figure 5.12, we note that the Z Garbage Collector, consistent with its latency-oriented architecture, exhibits the highest *throughput* values across all heap sizes, making it the least effective garbage collector in terms of overall *throughput*. Similar to the observations in the *pause_time* results, Parallel GC demonstrates better performance compared to G1. However, it is important to highlight that the differences in *throughput* among the garbage collectors are relatively minor across most workloads, with only ZGC showing significant variations due to allocation stalls, as seen in the h2 workload in Figure 5.12. Notably, one workload at a heap size of 256 MB exhibits a different pattern: the Parallel GC underperforms compared to the other garbage collectors in terms of *throughput*, specifically for the workload **graphchi**. To investigate the reasons behind this anomaly, we can refer to the data in Listing 5.6. It becomes evident that the difference in throughput between G1 and Parallel GC closely aligns with the difference in total pause time ($|-92.47| \approx |-90.76|$). The larger total *pause_time* is largely attributed to "ParallelGCFailedAllocation" pauses, that contribute to 752 more pauses than G1. This reinforces our earlier analysis in Table 5.3, where G1 demonstrates superior performance compared to other garbage collectors in low-memory environments. This is likely due to its efficient generational-based memory management, which optimizes resource usage under constrained conditions.

```
{
"throughput": 101929334256,
"total_pause_time": 10000028850,
"avg_pause_time":  3640345.41,
"number_of_pauses": 2747,
"pauses_per_category": {
    "Cleanup": 5,
    "G1CollectForAllocation": 1812,
    "G1TryInitiateConcMark": 49,
    "G1PauseRemark": 441,
    "G1PauseCleanup": 440
}
}
```


(a)


```
{
"throughput": 194402623047,
"total_pause_time": 100759511173,
"avg_pause_time": 28796659.38,
"number_of_pauses": 3499,
"pauses_per_category": {
    "Cleanup": 6,
    "ParallelGCFailedAllocation": 3493
}
}
```


(b)

Listing 5.6: DaCapo Graal graphchi - a) G1 and b) Parallel - Throughput details (in ns) at 256MB heap size

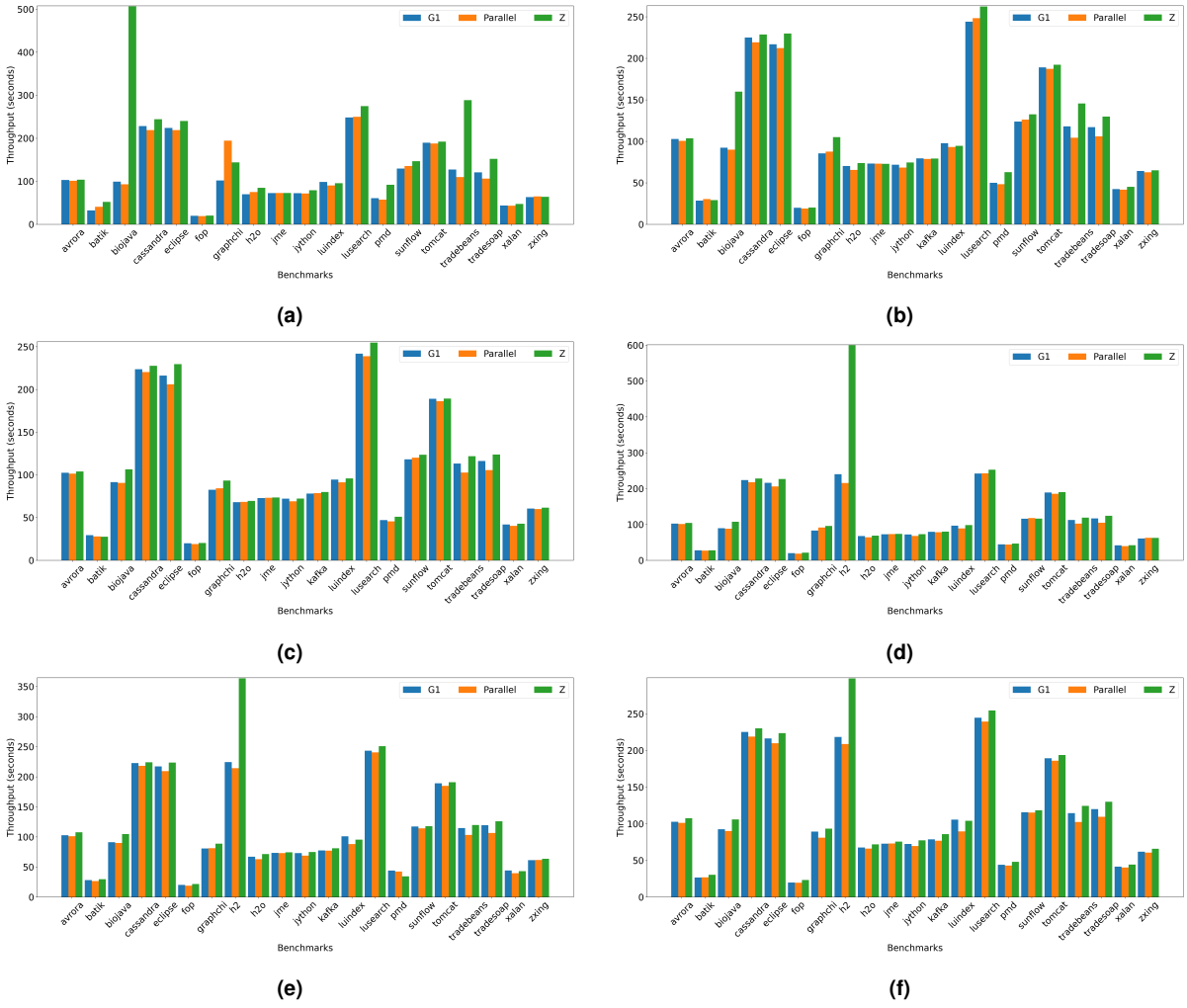

Figure 5.12: Graal DaCapo Results (Throughput - Lower is better) - The following graphs show throughput time with different heap sizes across all DaCapo Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

Renassaince Results Switching to the Renassaince benchmark, we start by analyzing the p90 *pause_time* metrics in Figure 5.13 following the same approach as before. Not surprisingly, ZGC shows again across all workloads and heap sizes that is capable of outperforming the other GCs when it comes to achieving the lowest p90 *pause_time*. Using the workload **finagle-http** at 256MB heap size (see Figure 5.13a) and referring to the additional statistics in Listing 5.7, we observe a remarkable result by solving Equation 5.2: ZGC's p90 *pause_time* is less than 30% of G1's average pause time and less than 40% of Parallel's GC pause time.

$$\frac{\text{Z p90\_pause\_time}}{\text{average\_pause\_time } \{\text{G1, Parallel}\}}$$
$$\text{G1: } \frac{480413.6}{1614107.67} = 0.298 \tag{5.2}$$
$$\text{Parallel: } \frac{480413.6}{1098527.98} = 0.437$$

Switching to the Renaissance benchmark, we begin by analyzing the p90 *pause_time* metrics in Figure 5.13, following the same approach as before. As expected, ZGC consistently achieves the lowest p90 *pause_time* across all workloads and heap sizes, outperforming the other garbage collectors. Focusing on the **finagle-http** workload at a heap size of 256 MB (see Figure 5.13a) and referring to the additional statistics in Listing 5.7, we observe a remarkable result by solving Equation 5.2: ZGC's p90 *pause_time* is less than 30% of G1's average pause time and less than 40% of Parallel GC's pause time, reforcing even more it's superiority when it comes to latency-sensitive.

```
{
"Z":{"avg_pause_time": 202858.01, "p90_pause_time": 480413.6},
"G1":{"avg_pause_time": 1614107.67, "p90_pause_time": 2930847.0},
"Parallel":{"avg_pause_time": 1098527.98,"p90_pause_time": 1464950.6}
}
```

Listing 5.7: Renaissance Graal finagle-http - Z,G1 and Parallel Pause Time details (in ns) at 256MB heap size

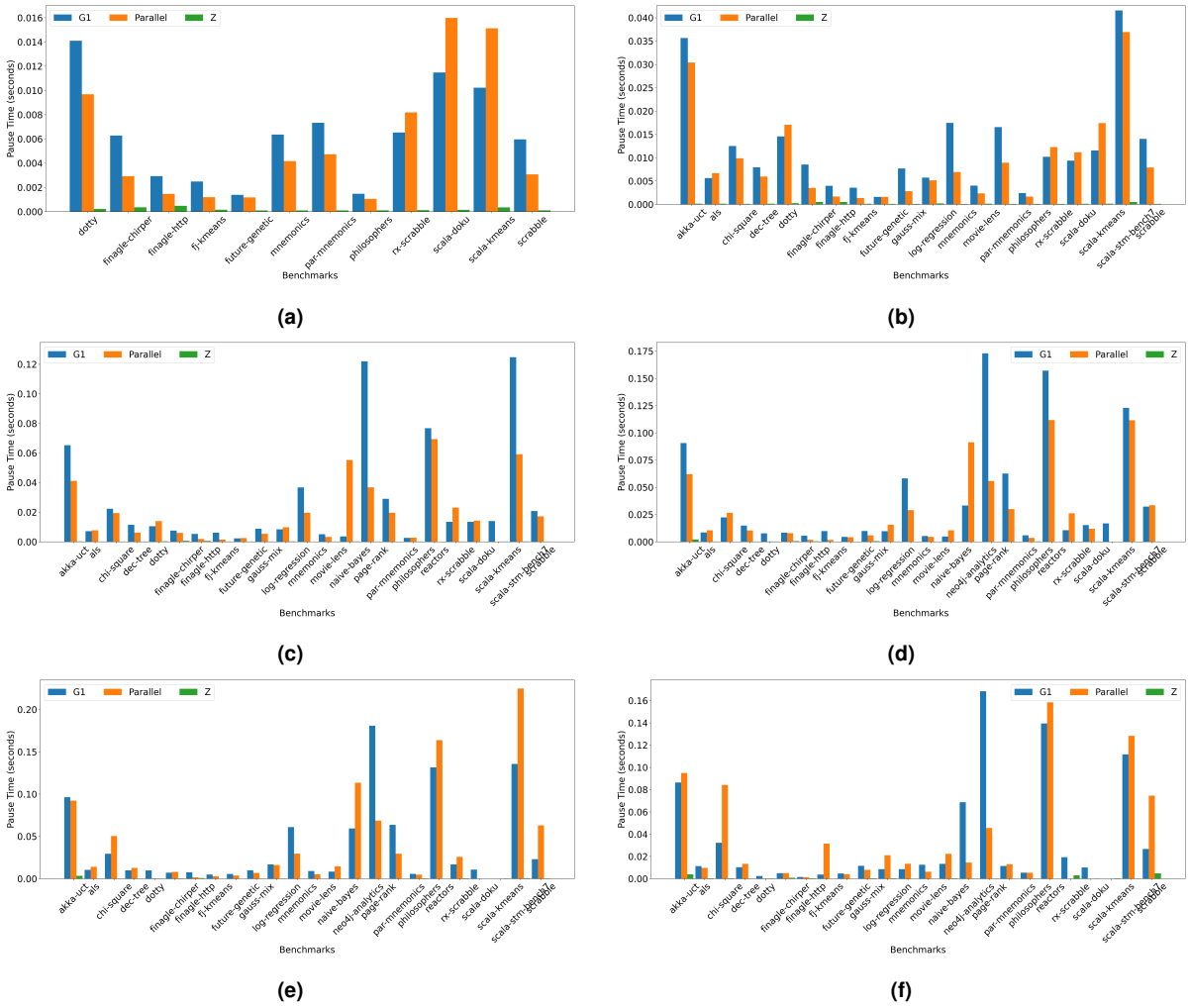

Figure 5.13: Graal Renaissance Results (P90 Pause Time - Lower is better) - The following graphs show P90 pause time with different heap sizes across all Renaissance Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

Examining the *throughput* results in Figure 5.14, comparable to what happened to the DaCapo workloads, Parallel GC shows on average better *throughput* times on every analyzed heap size. On the other hand, Z Garbage Collector is the least performant, showing higher *throughput* times. However, upon closer inspection, we notice that for multiple workloads the difference in *throughput* among all GCs is less than five seconds compared to Parallel GC. For example, at a heap size of 1024MB, workloads **future-genetic**, **gauss-mix**, **dec-tree** and **dotty** display minimal differences in *throughput*. This suggests that for certain workloads, GCs like G1 and ZGC, which are not primarly designed for *throughput*, can still achieve comparable performance, making them viable options. One workload that stands out is the **naive-bayes** workload with a 1024MB heap size (see Figure 5.14c), where G1 unexpectedly shows the best *throughput*, while Parallel GC performs similarly to ZGC. Consulting the Listing 5.8 we can analyze the difference between throughput and total pause time for Z and Parallel compared to G1. The

results are the following:

$$\frac{\text{throughput}_{Z} - \text{throughput}_{G1}}{10^9} \equiv \frac{167979862433 - 4807492792}{10^9} = 167.52\text{sec}$$
$$\frac{\text{total\_pause\_time}_{Z} - \text{total\_pause\_time}_{G1}}{10^9} \equiv \frac{985669542 - 4807492792}{10^9} = -3.82\text{sec}$$
$$\frac{\text{throughput}_{Parallel} - \text{throughput}_{G1}}{10^9} \equiv \frac{172323381000 - 4807492792}{10^9} = 163.17\text{sec}$$
$$\frac{\text{total\_pause\_time}_{Parallel} - \text{total\_pause\_time}_{G1}}{10^9} \equiv \frac{106203311236 - 4807492792}{10^9} = 101.4\text{sec}$$

As shown, although both ZGC and Parallel GC exhibit worse *throughput* by 167.52 and 163.17 seconds respectively, this disparity cannot be attributed solely to differences in total GC pause time. For one, Z's total pause time is actually 3.82 seconds less than G1, and the difference for Parallel GC is only 101.4 seconds. This leaves nearly 160 seconds in the ZGC case and 60 seconds in Parallel's case unaccounted for. A potential explanation for ZGC could be allocation stalls, which are not counted as GC pause times but instead occur when threads wait for available memory. In the case of Parallel GC, allocation stalls can still happen if a mutator thread has to wait for the start of an STW Garbage Collection Cycle, which involves the stoppage and synchronization of all mutator threads. If the mutator threads are busy, the synchronization time can add up. Furthermore, additional factors related to the workload itself could be influencing these results, and further analysis using more profiling tools and deeper inspection of the source code would be useful to fully understand the cause.

```
{
"Parallel":{
    "total_pause_time": 106203311236,
    "throughput": 167979862433
},
"Z":{
    "total_pause_time": 985669542,
    "throughput": 172323381000
},
"G1":{
    "total_pause_time": 4807492792,
    "throughput": 77778448188
},
}
```

Listing 5.8: Renaissance Graal naive-bayes - Z,G1 and Parallel Throughput and Pause Time details (in ns) at 1024MB heap size

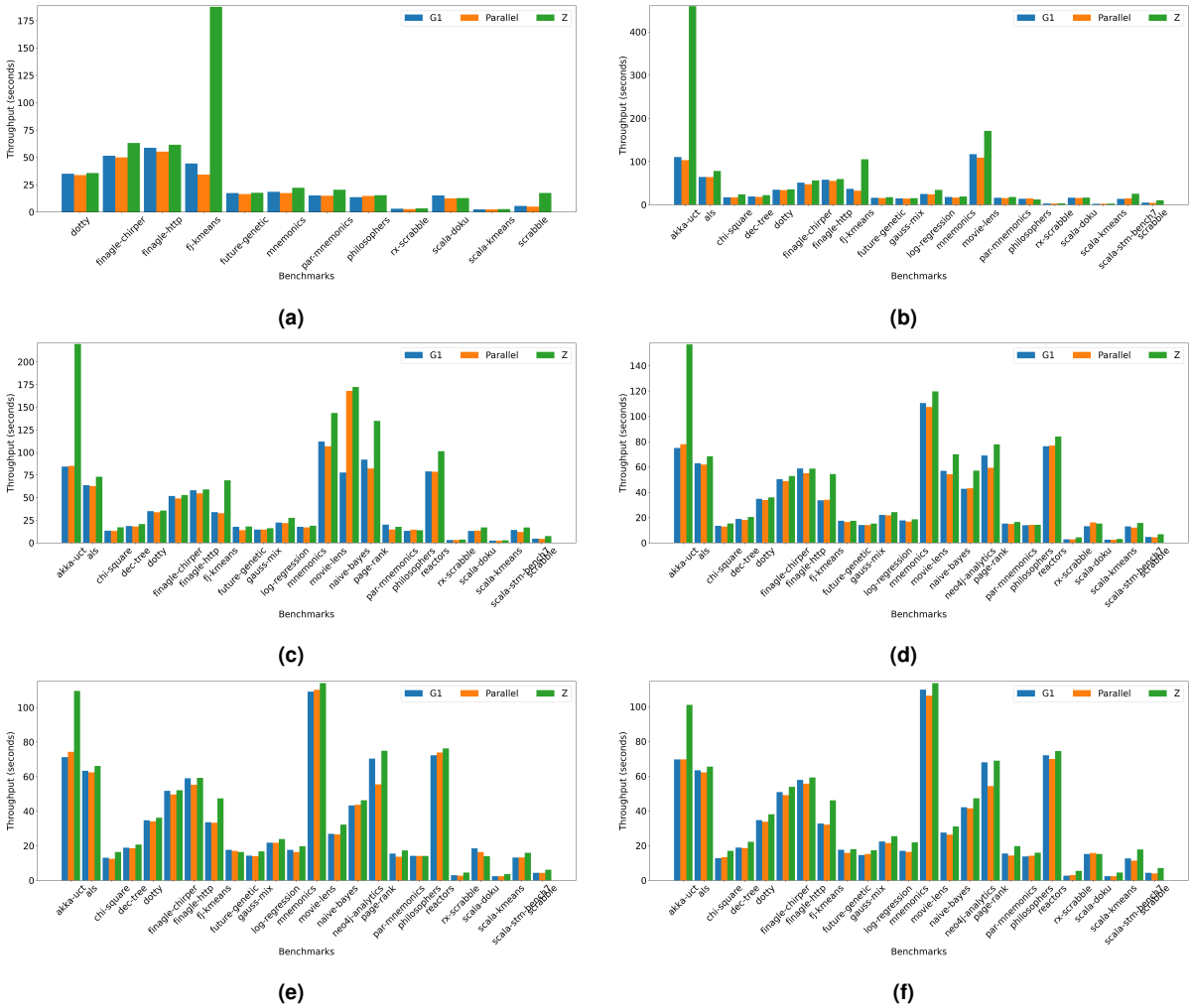

Figure 5.14: Graal Renaissance Results (Throughput - Lower is better) - The following graphs show throughput time with different heap sizes across all Renaissance Workloads and GCs (G1, Parallel, Z). The heap sizes used are: **a)** 256; **b)** 512; **c)** 1024; **d)** 2048; **e)** 4096; **f)** 8192.

5.3.2 HotSpot vs GraalVM

Now that all Benchmarks and Workloads have been thoroughly analyzed, we can finally compare each runtime directly, focusing on the matrices computed from the Benchmark and Workload results (refer to Section 4.1). As explained earlier, the scoring matrix ranks each Garbage Collector based on *throughput* and p90 *pause_time* for each heap size. Similar to the workload metrics, a lower score indicates better performance. An example of the scoring matrix is shown in Listing A.1, and a visual representation of the matrices we are going to analyze can be found in Figure 5.15.

As discussed in Section 4.1, the GC scores are computed by summing the successful workload metrics and normalizing them against the G1 Garbage Collector. This means a GC can have a worse (i.e., higher) score while still showing better performance across more workloads. This happens because

each workload contributes equally, and poor performance in one workload can skew the overall score. Using this graph and the previous analysis from Section 5.3.1, we can easily identify these situations.

Starting with GraalVM's p90 *pause_time* scores in Figure 5.15a, we clearly see that ZGC has the best pause time scores. Additionally, Parallel GC ties with G1, outperforming G1 in 3 of the heap sizes. However, to determine overall performance, we can apply the following equation:

$$\begin{aligned}&\sum_{i\in\{256,512,1024,2048,4096,8192\}} \text{g1}_{\text{score}} - \text{parallel}_{\text{score}} = \\ &= (1-1.38)+(1-1.19)+(1-0.77)+(1-0.8)+(1-1.07)+(1-0.94) = \\ &= -0.15\end{aligned} \tag{5.3}$$

This indicates that, overall, Parallel GC had worse performance, meaning G1 had a lower overall score. An example where a GC might have a worse score despite better performance in more workloads is Parallel GC at a heap size of 256MB. In Section 5.3.1, we observed that for a 256MB heap, Parallel GC outperformed G1 in 13 out of 19 workloads. However, in Figure 5.15a, Parallel GC has the worst performance, mainly due to the **batik** workload.

Moving on to GraalVM's *throughput* scores if Figure 5.15b, we observe that, as expected after our previous analysis, Parallel GC consistently performs better than its counterparts. It ties with G1 at the 1024MB heap size and loses with G1 by just 0.01 at the 256MB heap size. ZGC, on the other hand, shows the worst *throughput* performance, though it reveals an interesting observation: ZGC's worst *throughput* score is for the 256MB heap, where it is 41% higher than G1's score. However, for the same heap size, ZGC's p90 *pause_time* is 94% lower than G1's. This wide range is even more pronounced in larger heaps (*pause_time* $= 98\% lower$, *throughput* $= 9\%$ higher for a 4096MB heap), providing the basis for a new GC scoring method in Section 5.4.

Turning to the HotSpot p90 *pause_time* results in Figure 5.15c, we again see a tie between Parallel GC and G1, but with a greater performance gap at the 256MB heap size. In GraalVM, Parallel GC performed 1.38 times worse than G1; in HotSpot, this difference increases to 2.45 times. While it may be tempting to conclude that Parallel GC is less optimized for smaller heaps on HotSpot compared to GraalVM, it's important to remember that these results are normalized to G1. By examining the Garbage Collector Reports data (provided by BenchmarkGC) and the calculated statistics in Table 5.4, we find that Parallel GC improved its p90 *pause_time* by 4% when switching to GraalVM. Conversely, G1's performance dropped by 71% in GraalVM, indicating that the performance difference is largely due to G1's optimization in HotSpot rather than poor Parallel GC performance in HotSpot. Furthermore, if we focus on heap sizes $\geq$ 1024MB, Parallel GC consistently outperforms G1 in p90 *pause_time*, with only a minor surplus of 0.03 at 4096MB, indicating better optimization for larger heaps. As in GraalVM, ZGC remains the clear winner in p90 *pause_time*.

Focusing now on Figure 5.15d, which shows the *throughput* scores for HotSpot GCs, we observe a

greater disparity than in the GraalVM results, particularly for the smaller heap sizes (256MB and 512MB). To identify the source of this difference, we refer again to Table 5.4. At 256MB, the disparity is primarily due to G1's superior performance in HotSpot (GraalVM being 32% less performant). At 512MB, however, Parallel GC performs better in GraalVM (11%), which explains why Parallel GC has higher *throughput* than G1 in HotSpot for that heap size.

One notable observation in the table is the 29% improvement in ZGC's p90 *pause_time* at the 4096MB heap size when switching to GraalVM. This improvement is not evident by looking at the Figures 5.15a and 5.15c, as ZGC's *pause_time* values are orders of magnitude lower than G1's.

Table 5.4: Improvement in Throughput and P90 Pause Time when changing from HotSpot to GraalVM (**(+)** value: decreased performance; **(-)** value: increased performance).

Heap Sizes (MB)	GC	(GraalVM - HotSpot)/HotSpot (%)		More Performant	
		Throughput	P90 Pause Time	Throughput	P90 Pause Time
256	G1	32	71	HotSpot	HotSpot
	Parallel	-1	-4	GraalVM	GraalVM
	Z	-2	2	GraalVM	HotSpot
512	G1	-3	1	GraalVM	HotSpot
	Parallel	-11	2	GraalVM	HotSpot
	Z	-4	-6	GraalVM	GraalVM
1024	G1	-3	6	GraalVM	HotSpot
	Parallel	0	2	Equal	HotSpot
	Z	-4	-7	GraalVM	GraalVM
2048	G1	-3	1	GraalVM	HotSpot
	Parallel	-6	1	GraalVM	HotSpot
	Z	-2	-6	GraalVM	GraalVM
4096	G1	-7	-1	GraalVM	GraalVM
	Parallel	0	2	Equal	HotSpot
	Z	-3	-29	GraalVM	GraalVM
8192	G1	-2	-1	GraalVM	GraalVM
	Parallel	-2	-2	GraalVM	GraalVM
	Z	-2	10	GraalVM	HotSpot

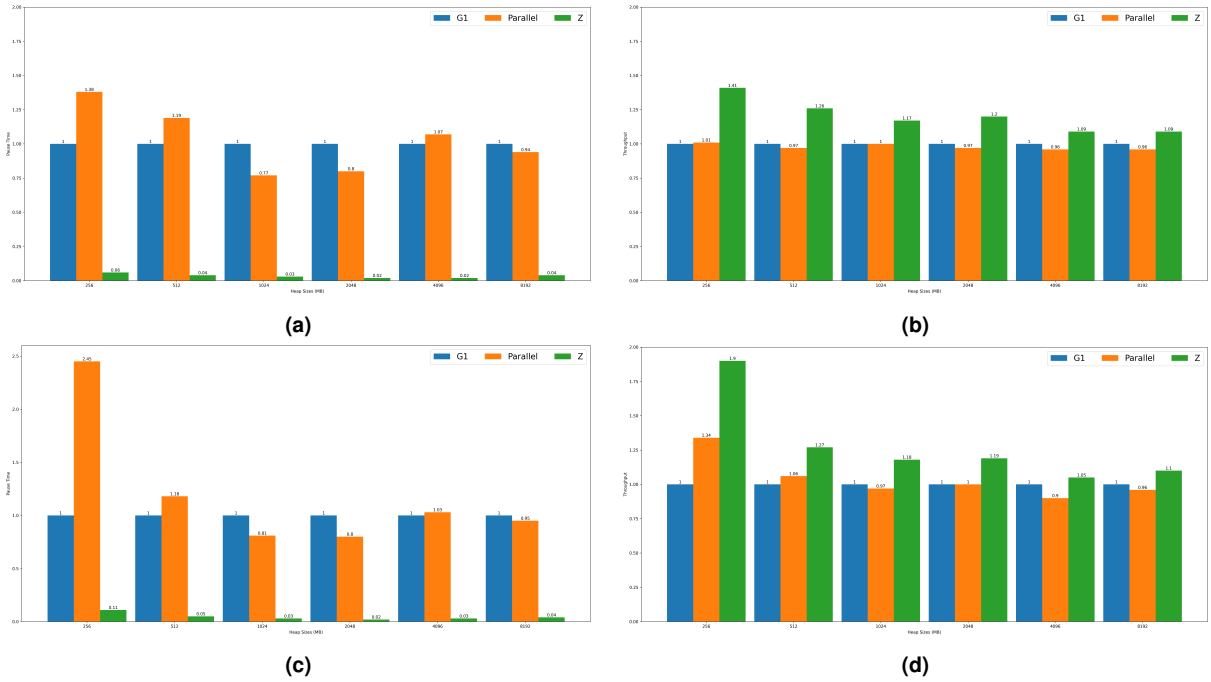

Figure 5.15: Runtime Scoring Matrices (Lower score is better) - The following graphs show Garbage Collector Throughput and P90 Pause Time scores across multiple heap sizes for GraalVM (**a,b**), and HotSpot (**c,d**). All scores are normalized to G1 Throughput and Pause time values. **a)** GraalVM P90 Pause Time Scores; **b)** GraalVM Throughput Scores; **c)** HotSpot P90 Pause Time Scores; **d)** HotSpot Throughput Scores.

5.3.3 Final Thoughts

After analyzing both the HotSpot and GraalVM runtimes, each evaluated with two benchmarks-DaCapo and Renaissance-several we can draw the following key conclusions:

Z Garbage Collector is the best choice in terms of p90 *pause_time*: Throughout all Benchmarks and Workloads, independent of the Java runtime used, ZGC achieved significantly smaller pauses compared with the other garbage collectors. However, this advantage may also present a downside, which we will explain when discussing *throughput*.

Parallel GC offers better *throughput* and p90 *pause_time* than G1: As noted earlier, the throughput values were quite close among all GCs, but Parallel GC consistently achieved lower *throughput* values (remember, a smaller *throughput* value is **better**). It's easy to think, that if we extrapolate this to a long-running application, the benefits would accumulate. However, regarding p90 *pause_time*, this claim may be somewhat misleading; while Parallel GC has the best p90 *pause_time* values compared to G1, the results are fairly divided. For each heap size and benchmark, Parallel GC maintained an advantage in more than half of the analyzed workloads. Conversely, in cases where Parallel GC shows worse throughput compared to G1, we found that the **average** *pause_time* is the principal culprit, suggesting that G1 can perform better in terms of *throughput* and average *pause_time* in

certain scenarios.

ZGC exhibits the worst throughput (sometimes by a large margin): As expected, an almost fully concurrent garbage collector lacks the characteristics to present the best throughput among all available GCs. Nonetheless, for many workloads, the differences were not as significant as previously anticipated, indicating that a latency-oriented collector like ZGC can still achieve acceptable throughput values. However, we identified a potential pitfall: when ZGC handles applications with high memory allocation rates while having low available memory, it can incur **allocation stalls**, dramatically increasing workload execution times and negatively impacting *throughput*.

HotSpot and GraalVM exhibit similar characteristics: Overall, the previous statements apply to both runtimes, suggesting that each garbage collector demonstrates comparable behaviour, regardless of the runtime used.

GraalVM is more performant (in throughput): In 15 out of 18 configurations (combinations of GC and heap size), GraalVM demonstrates significant improvements in *throughput*. Of the remaining 3 configurations, two are draws, showing no noticeable difference between GraalVM and HotSpot. However, one stands out as an exception: GraalVM exhibits 32% lower performance when using G1 with 256MB heap size, suggesting that G1 in HotSpot may be better optimized for low memory environments (see Table 5.4).

HotSpot is more performant (in p90 pause time): HotSpot delivers superior *p90 pause time* performance in 10 out of 18 configurations, with the most notable improvement when using G1 with a 256MB heap, where GraalVM has 71% performance reduction. Nevertheless, GraalVM shows a notable result with ZGC at a 4096MB heap size, achieving a 29% improvement in pause time. This is particularly impressive given ZGC's already low p90 *pause_time* baseline.

5.4 BestGC++ Evaluation

To evaluate BestGC++, as previously mentioned, we use the **Spring PetClinic** application. Given that the DaCapo benchmark suite already includes a **Spring PetClinic** workload that as mentioned in Section 5.2, wasn't included in the matrix generation phase, we opted to use it.

5.4.1 Spring PetClinic - Benchmarks

For establishing a baseline of comparison, **PetClinic** was executed using the BenchmarkGC profiling application across both HotSpot and GraalVM runtimes. The results are presented in Figure 5.16. We observe that Parallel GC exhibits consistent performance across all heap sizes in both runtimes. Notably, G1 shows better optimization in HotSpot at the 256MB heap size compared to GraalVM's G1 at the same heap size, which is consistent with previous findings in Section 5.3.2. Additionally, G1 in

HotSpot achieves its best performance in terms of *throughput* and p90 *pause_time* at 256MB. However, in both runtimes, G1 experiences worse p90 *pause_times* as heap sizes increase, which is expected as larger memory allows GC cycles to be delayed, leading to larger collections and longer pauses. Regarding *throughput*, G1 remains consistent across all heap sizes, except for HotSpot at 256MB, where its performance stands out.

ZGC maintains consistent p90 *pause_times* in both runtimes, achieving the lowest pause times among all GCs, followed by Parallel GC most of the time. Regarding *throughput*, Z Garbage Collector shows improvement up to the 1024MB heap size (in both runtimes), after which performance plateaus. Overall, ZGC is the best-performing GC in terms of p90 *pause_time* across both runtimes, while Parallel GC delivers the highest *throughput* in most heap sizes.

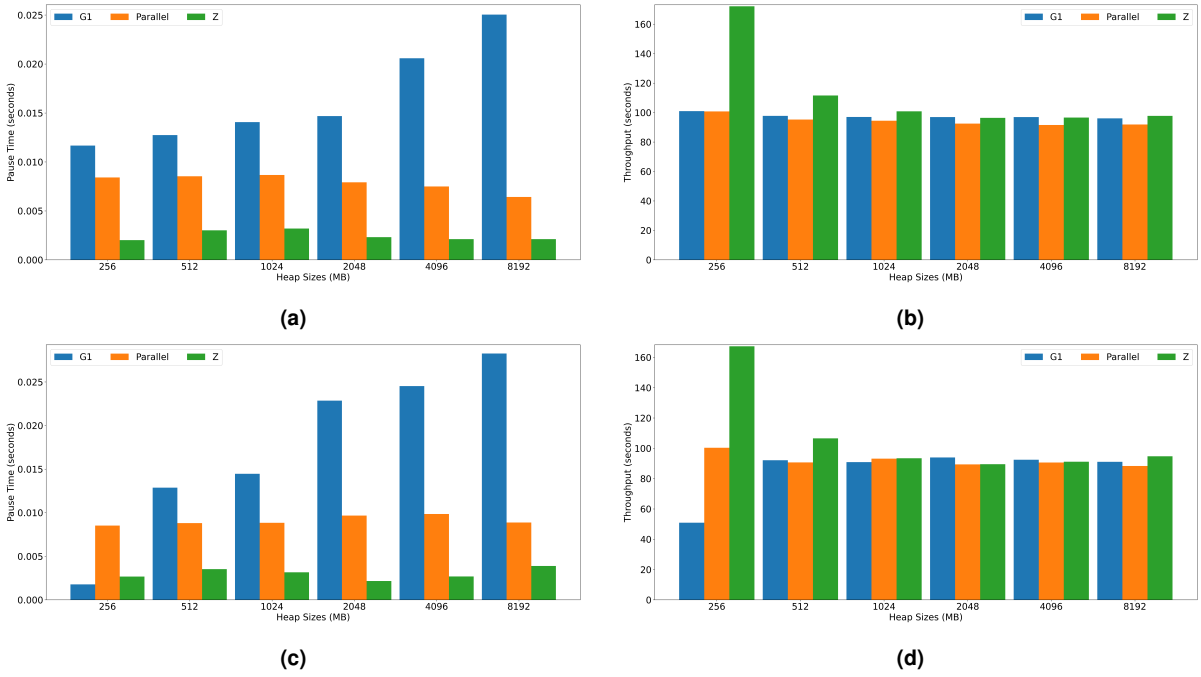

Figure 5.16: Spring PetClinic Benchmark Throughput and P90 Pause Time Results (Lower is better) in seconds - The following graphs show PetClinic's Throughput and P90 Pause Time for GraalVM and HotSpot runtimes. **GraalVM**: **a)** p90 pause_time and **b)** throughput; **HotSpot**: **c)** p90 pause_time and **d)** throughput.

5.4.2 BestGC++ Testing Methodology and Results

Evaluating BestGC++ focuses on testing its ability to select the most performant Garbage Collector, i.e., its accuracy, using a pre-existing GC scoring matrix (see Section 4.1) and optionally user-provided weights for *throughput* and p90 *pause_time*.

Key considerations include:

- The sum of the weights is always equal to 1, e.g., if the *throughput* weight is 0.6, the p90 *pause_time* weight will be 0.4.
- The equation used in the original BestGC implementation (Section 2.3) is defined as follows:

$$Score = throughput_{weight} \times throughput_{score} + pause\_time_{weight} \times pause\_time_{score} \tag{5.4}$$

- Like its predecessor, BestGC++ calculates the applications' maximum heap size by increasing the observed maximum heap size by 20% and rounding it up to the nearest heap size in the scoring matrix (see Section 2.3 for more details).

Examining Equation 5.4 and considering the range discrepancy issue mentioned in Section 5.3.2, we identify that if the *throughput_score* and *pause_time_score* have different ranges, the equation may become skewed. For instance, consider the following scenario: a GC has a *pause_time* score of 0.5 and a *throughput* score of 2. Calculating the score with equal *throughput* and *pause_time* weights, each 0.5, yields:

$$\text{throughput\_score\_contribution} = \text{throughput\_score} \times 0.5 = 2 \times 0.5 = 1$$
$$\text{pause\_time\_score\_contribution} = \text{pause\_time\_score} \times 0.5 = 0.5 \times 0.5 = 0.25$$

As we can see, although both scores are proportionally related to 1, their impact on the overall score differs significantly. To address this imbalance, we multiply the terms, ensuring that if any weight is zero, the corresponding term is effectively ignored, effectively becoming equivalent to the old Equation 5.4. This leads to the piecewise Equation 5.5:

$$\text{Score} = \begin{cases} \text{throughput}_{\text{weight}} \times \text{throughput}_{\text{score}} & \text{if pause\_time}_{\text{weight}} = 0 \\ \text{pause\_time}_{\text{weight}} \times \text{pause\_time}_{\text{score}} & \text{if throughput}_{\text{weight}} = 0 \\ \text{throughput}_{\text{weight}} \times \text{throughput}_{\text{score}} \times \text{pause\_time}_{\text{weight}} \times \text{pause\_time}_{\text{score}} & \text{otherwise} \end{cases} \tag{5.5}$$

To evaluate **PetClinic**, we profiled the application under BestGC++ in both automatic mode and manual mode with *throughput* weights set to 1 and 0 for both GraalVM and HotSpot runtimes. The command used for the automatic mode is: `java -jar bestgc.jar dacapo-23.11-chopin.jar --args="spring -n 10 --no-pre-iteration-gc" --automatic --monitoringTime=50`. For manual mode, we remove the `--automatic` flag and specify the weight using `--wt=<value>`.

The results compare BestGC++'s accuracy using the original Equation 5.4 versus the modified Equation 5.5. In each case, BestGC++ profiles the **Spring PetClinic** application, selects the optimal heap

size and determines the best GC according to the selected equation. The accuracy is then measured by comparing the selected GC against the most performant GC for the given **Spring PetClinic** workload, calculated using the same weights and equation across all GCs for the selected heap size.

Table 5.5: Spring PetClinic BestGC++ GC Selection in GraalVM and HotSpot with a Heap Size of 512MB - Old Equation 5.4 vs New Equation 5.5.

Runtime	Throughput Weight	Old Equation			New Equation			Selected GC		Correct Option	
		G1	Parallel	Z	G1	Parallel	Z	Old Equation	New Equation	Old Equation	New Equation
GraalVM - Manual Mode	1	1	0.97	1.26	1	0.97	1.26	Parallel		Parallel	
GraalVM - Manual Mode	0	1	1.19	0.04	1	1.19	0.04	Z		Z	
GraalVM - Automatic Mode	0.72	1	1.03	0.92	1	0.23	0.01	Z	Z	Parallel	Z
HotSpot - Manual Mode	1	1	1.06	1.27	1	1.06	1.27	G1		Parallel	
HotSpot - Manual Mode	0	1	1.18	0.05	1	1.18	0.05	Z		Z	
HotSpot - Automatic Mode	0.71	1	1.09	0.91	1	0.26	0.01	Z	Z	Parallel	Z

The results are summarized in Table 5.5. BestGC++ identified a heap size of 512MB as the most optimal. In automatic mode, both runtimes showed similar average CPU usage when executing *Spring PetClinic*. This can be derived by remembering that *throughput* weight is calculated using CPU average and the Equation 4.3. The only significant difference occurred with a *throughput_score* of 1, where G1 was selected in HotSpot, while Parallel was selected in GraalVM, consistent with the observations in Section 5.3.2. In other cases, Z Garbage Collector was chosen in both **Old** and **New** equations.

Finally, and most important to test our hypothesis, the computed **Correct Options** show that with the new scoring method (Equation 5.5), BestGC++ made 5 out of 6 correct selections, compared to 3 out of 6 using the old Equation 5.4, revealing 33% improvement.

5.5 Summary

In this Section we evaluated the two GC profiling tools developed, BenchmarkGC, a tool that allows us to benchmark Java workloads, collect performance metrics, and ultimately classify Garbage Collectors in a scoring matrix, and BestGC++, a profiling tool meant to select the best Garbage Collector for a given application, using the BenchmarkGC's scoring matrix. They were executed with multiple Java runtimes and workloads, explaining the methodology and reasoning behind the testing. The Java runtimes HotSpot and GraalVM (JIT), were chosen, due to having similar capabilities in terms of available Garbage Collectors, so a fair comparison could be made in the future. Workloads were selected by choosing two of the most popular Java benchmarking suites (DaCapo and Renaissance). However, special attention was paid so as to not repeat workloads used in the BenchmarkGC and BestGC++ testing, because the latter is dependent on the former. Specifically, the **Spring PetClinic** workload that was present in DaCapo, so it ended up being removed so as to be used by BestGC++.

Analyzing BenchmarkGC's result we made several findings when it comes to Garbage Collection performance, in different runtimes. One of which, is the fact that GraalVM is more performant than HotSpot when it comes to *throughput* in 15 out of 18 configurations. However, the opposite also happened for a

heap size of 256MB, where G1 showed a performance reduction of 32% when switching from HotSpot to GraalVM. The better GCs in terms of performance related to *throughput* and p90 *pause_time* were also identified, with ZGC being the better GC for *pause_time* and ParallelGC for *throughput*. Although opposite to Z, Parallel had closer GCs to its values e.g., G1.

Reasons for application performance decay were also identified, like allocation stalls with low memory heap sizes e.g., ZGC; higher *pause_time*s due to a heap size increase, which increases the duration of GC cycles; GC overhead, and other system phenomenons, proving that the metrics collected by the BenchmarkGC profiling application are useful when it comes to diagnosing applications.

Furthermore, BestGC++ was tested with **Spring PetClinic** and a new scoring method was developed so as to improve on the old scoring equation. Analyzing the new scoring method, revealed that it had improved 33%, increasing from 50% to 83%. This indicated that the BestGC++ tool with the new scoring method as a higher accuracy, further increasing its usability value.

In this section, we evaluated two profiling tools developed for garbage collection analysis: BenchmarkGC and BestGC++. BenchmarkGC enables benchmarking of Java workloads, collecting performance metrics to classify garbage collectors within a scoring matrix. BestGC++ is designed to identify the most suitable garbage collector for a specific application based on the scores derived from BenchmarkGC. We conducted tests across multiple Java runtimes and workloads, detailing the methodology and rationale behind our testing approach.

The Java runtimes selected for this evaluation were HotSpot and GraalVM (JIT), as they offer similar capabilities regarding available garbage collectors, facilitating a fair comparison. Workloads were chosen from two of the most popular Java benchmarking suites, DaCapo and Renaissance. To ensure the validity of our results, care was taken to avoid repeating workloads used in BenchmarkGC and BestGC++ testing, particularly excluding the Spring PetClinic workload from DaCapo, which was utilized in BestGC++.

Our analysis of BenchmarkGC's results yielded several insights into garbage collection performance across different runtimes. Notably, GraalVM outperformed HotSpot in terms of throughput in 15 out of 18 configurations. However, a performance reduction of 32% was observed in G1 when switching from HotSpot to GraalVM at a heap size of 256MB. The best-performing garbage collectors regarding throughput and p90 pause time were also identified, with ZGC excelling in pause time and ParallelGC in throughput, despite ParallelGC having closer values to G1 in comparison.

We also identified key factors contributing to application performance degradation, such as allocation stalls at low memory heap sizes (e.g., with ZGC), lower throughput due to pause time increase, higher GC cycle durations in larger heap sizes resulting in increased pause times, and overall GC overhead. These findings showcase the value of the metrics collected by the BenchmarkGC profiling tool in diagnosing application performance issues.

Furthermore, we tested BestGC++ using the Spring PetClinic workload, developing a new scoring method to enhance the existing scoring equation. The analysis revealed a significant improvement of 33%, increasing from 50% to 83% in accuracy. This enhancement indicates that BestGC++ with the new scoring method offers greater accuracy, further increasing its usability and effectiveness as a profiling tool.

6

Conclusion

Contents

This thesis began by examining the current technology landscape, which is now deeply integrated with cloud computing. We highlighted the rise of various cloud service models, such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS), and serverless solutions like Function-as-a-Service (FaaS). In terms of application architecture, large enterprises are increasingly adopting microservices, where applications are composed of independent services. We identified Java as a predominant language in microservices development, and thus, set our goals toward optimizing its surrounding environment—including the language runtime, Garbage Collector (GC), and inter-microservice communication.

We then reviewed the current state of research on microservices architecture, contrasting it with monolithic and serverless designs. This analysis showed that while microservices offer advantages, they also introduce challenges such as increased communication complexity and distributed system issues like failures and timeouts. Despite these drawbacks, the widespread adoption of microservices justified our focus on developing solutions tailored to this architecture. In terms of garbage collection, we identified two main types of GC algorithms: Tracing Algorithms and Reference Counting Algorithms. We

began by exploring Mark-and-Sweep, the first tracing algorithm, which marks live objects by traversing root objects and then sweeps unmarked objects. Next, we examined a reference-counting algorithm, which keeps track of the number of references to each object. Additionally, we explored a profiling tool called BestGC, designed to identify the most suitable Java GC by using a precomputed matrix score. BestGC operates in four phases: matrix generation, execution, monitoring, and GC scoring, ultimately selecting the best-performing GC. This tool served as the foundation for our developed solution.

Next, we reviewed several studies, particularly on the performance of modern Java GCs such as G1, Shenandoah, and ZGC. The latter two are noteworthy for being almost fully concurrent algorithms, with a focus on minimizing pause times—a critical factor in cloud environments where Service Level Agreements (SLAs) often prioritize tail latency. We also explored various approaches to improving GC performance, such as middleware for better GC thread placement on CPU cores, reducing interference with latency-sensitive application threads, and optimizations for NUMA architectures. Additionally, we touched on runtime optimizations, such as GraalVM Native Image, which emphasizes fast startup times, mitigating cold starts. Graal's ahead-of-time (AOT) compilation allows it to share runtime resources efficiently across multiple processors (Isolate Proxy), reducing memory footprint in server environments.

After reviewing these studies, we introduced two GC profiling tools we developed. BenchmarkGC is designed to simplify benchmarking Java applications, collecting comprehensive metrics on GC performance and automatically computing a scoring matrix for easier evaluation of which GC is most suitable for a given heap size. BestGC++, as the name suggests, builds on the original BestGC tool. It was refactored to function as a web service, allowing users to profile their applications and identify the optimal GC with minimal input required. This improvement makes BestGC++ more accessible to non-expert users by automating the calculation of weights, that previously required user input, now calculated based on application CPU usage. Another enhancement is its ability to monitor applications already in production to analyze their performance in real time.

We then tested both profiling tools to validate the improvements. BenchmarkGC was executed with two Java runtimes, GraalVM and HotSpot. By comparing GCs and runtimes, we found that the collected metrics enabled the identification of GC-related issues, such as allocation stalls and high pause times, which increased overall execution time. Our results showed that ZGC performed best in both runtimes in terms of p90 pause time, while ParallelGC excelled in throughput, with G1 performing similarly. GraalVM demonstrated better performance in most configurations for throughput, while HotSpot had an edge in p90 pause time. For BestGC++, we evaluated its accuracy by running the Spring PetClinic application in both HotSpot and Graal, allowing the tool to determine the optimal GC based on the weights (calculated from average CPU usage). During this evaluation, we identified and addressed an issue in the original BestGC scoring equation, resulting in a new scoring equation for BestGC++. The new equation showed an accuracy improvement of 33%, raising the accuracy from 50% to 83%. These results confirm the value

of the profiling tools we developed. BenchmarkGC offers deep insights into application performance, while BestGC++ provides a practical tool for improving Java application performance by selecting the best GC accurately.

6.1 Future Work

Future work could focus on further enhancing BestGC++'s ability to select the most suitable GC. This was discussed earlier in Section 4.2.1, where we suggested classifying applications as I/O or CPU-intensive to refine the scoring process. However, simply using a binary classification fails to capture the full complexity of the data, as applications are often exceptions to such rigid categories, and this approach could reduce the available workload pool. A promising avenue would be to employ artificial intelligence (AI), which can handle a wider range of parameters and capture more intricate, non-linear relationships, as was done previously but just for one runtime and just one algorithm [30]. A potential implementation would be a Multilayer Perceptron (MLP), a type of feed-forward neural network capable of learning complex patterns through backpropagation. This approach is well-suited to our problem, given the diverse parameters that influence GC performance. In addition, BenchmarkGC has to be enhanced to collect more data points, such as disk activity and other JVM events beyond GC pauses. With a richer dataset and sufficient workloads, AI could offer a more accurate and dynamic classification method for BestGC++.

Bibliography

[1] P. M. Mell and T. Grance, “Sp 800-145. the nist definition of cloud computing,” Gaithersburg, MD, USA, Tech. Rep., 2011.

[2] O. Zimmermann, “Microservices tenets,” *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, Jul 2017. [Online]. Available: https://doi.org/10.1007/s00450-016-0337-0

[3] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[4] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[5] R. Shrestha and B. Nisha, “Microservices vs serverless deployment in aws: A case study with an image processing application,” in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022, pp. 183–184.

[6] M. Chadha, V. Pacyna, A. Jindal, J. Gu, and M. Gerndt, “Migrating from microservices to serverless: An iot platform case study,” in *Proceedings of the Eighth International Workshop on Serverless Computing*, ser. WoSC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 19–24. [Online]. Available: https://doi.org/10.1145/3565382.3565881

[7] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, “Survey on serverless computing,” *Journal of Cloud Computing*, vol. 10, no. 1, p. 39, Jul 2021. [Online]. Available: https://doi.org/10.1186/s13677-021-00253-7

[8] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, “Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures,” in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.

[9] X. Ma, J. Yan, W. Wang, J. Yan, J. Zhang, and Z. Qiu, “Detecting memory-related bugs by tracking heap memory management of c++ smart pointers,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 880–891.

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

[11] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part i,” *Commun. ACM*, vol. 3, no. 4, p. 184–195, apr 1960. [Online]. Available: https://doi.org/10.1145/367177.367199

[12] D. Patrício, R. Bruno, J. Simão, P. Ferreira, and L. Veiga, “Locality-aware gc optimisations for big data workloads,” in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, and R. Meersman, Eds. Cham: Springer International Publishing, 2017, pp. 50–67.

[13] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira, “Runtime object lifetime profiler for latency sensitive big data applications,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys ’19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303988

[14] G. E. Collins, “A method for overlapping and erasure of lists,” *Commun. ACM*, vol. 3, no. 12, p. 655–657, dec 1960. [Online]. Available: https://doi.org/10.1145/367487.367501

[15] S. Tavakolisomeh, R. Bruno, and P. Ferreira, “Bestgc: An automatic gc selector,” *IEEE Access*, vol. 11, pp. 72 357–72 373, 2023.

[16] D. Detlefs, C. Flood, S. Heller, and T. Printezis, “Garbage-first garbage collection,” in *Proceedings of the 4th International Symposium on Memory Management*, ser. ISMM ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 37–48. [Online]. Available: https://doi.org/10.1145/1029873.1029879

[17] W. Zhao and S. M. Blackburn, “Deconstructing the garbage-first collector,” in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 15–29. [Online]. Available: https://doi.org/10.1145/3381052.3381320

[18] Oracle, *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*, 2023.

[19] D. Ungar, “Generation scavenging: A non-disruptive high performance storage reclamation algorithm,” in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering*

Symposium on Practical Software Development Environments, ser. SDE 1. New York, NY, USA: Association for Computing Machinery, 1984, p. 157–167. [Online]. Available: https://doi.org/10.1145/800020.808261

[20] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "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*, ser. PPPJ '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2972206.2972210

[21] A. M. Yang and T. Wrigstad, "Deep dive into zgc: A modern garbage collector in openjdk," *ACM Trans. Program. Lang. Syst.*, vol. 44, no. 4, sep 2022. [Online]. Available: https://doi.org/10.1145/3538532

[22] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 229–240, mar 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451142

[23] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "Numagic: A garbage collector for big data on big numa machines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 661–673, mar 2015. [Online]. Available: https://doi.org/10.1145/2786763.2694361

[24] J. Zhao, A. Pi, X. Zhou, S.-Y. Chang, and C. Xu, "Improving concurrent gc for latency critical services in multi-tenant systems," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, ser. Middleware '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 43–55. [Online]. Available: https://doi.org/10.1145/3528535.3531515

[25] A. Lefort, Y. Pipereau, K. Amponsem, P. Sutra, and G. Thomas, "J-nvm: Off-heap persistent objects in java," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 408–423. [Online]. Available: https://doi.org/10.1145/3477132.3483579

[26] S. Ravindra, M. Dayarathna, and S. Jayasena, "Latency aware elastic switching-based stream processing over compressed data streams," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 91–102. [Online]. Available: https://doi.org/10.1145/3030207.3030227

[27] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: Application initialization at build time," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360610

[28] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 45–59. [Online]. Available: https://doi.org/10.1145/3419111.3421297

[29] S. Wang, “Thin serverless functions with graalvm native image,” Master Thesis, ETH Zurich, Zurich, 2021-04-22.

[30] J. Simão, S. Esteves, A. Pires, and L. Veiga, “Gc-wise: A self-adaptive approach for memory-performance efficiency in java vms,” *Future Generation Computer Systems*, vol. 100, pp. 674–688, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18304898

A

Code of Project

```
{
    "matrix": {
        "256": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 1.36,
                "pause_time": 2.47
            },
            "Z": {
                "throughput": 1.93,
                "pause_time": 0.12
            }
        },
        "512": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 0.96,
                "pause_time": 1.18
            },
            "Z": {
                "throughput": 1.24,
                "pause_time": 0.05
            }
        },
        "1024": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 0.97,
                "pause_time": 0.8
            },
            "Z": {
                "throughput": 1.17,
                "pause_time": 0.03
            }
        },
        "2048": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 0.94,
                "pause_time": 0.79
            },
            "Z": {
                "throughput": 1.19,
                "pause_time": 0.02
            }
        },
        "4096": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 0.95,
                "pause_time": 1.02
            },
            "Z": {
                "throughput": 1.1,
                "pause_time": 0.03
            }
        },
        "8192": {
            "G1": {
                "throughput": 1.0,
                "pause_time": 1.0
            },
            "Parallel": {
                "throughput": 0.95,
                "pause_time": 0.94
            },
            "Z": {
                "throughput": 1.1,
                "pause_time": 0.03
            }
        }
    },
    "garbage_collectors": [
        "Parallel",
        "G1",
        "Z"
    ],
    "benchmarks": { // List of benchmarks used
        "256": ["..."],
        "512": ["..."],
        "1024": ["..."],
        "2048": ["..."],
        "4096": ["..."],
        "8192": ["..."]
    }
}
```

Listing A.1: GC Scoring Matrix

```
{
    "garbage_collector": "G1",
    "jdk": "HotSpot_21.0.4",
    "stats": [
        {
            "heap_size": "256",
            "number_of_pauses": 17836,
            "total_pause_time": 43527753550,
            "avg_pause_time": 2440443.68,
            "p90_avg_pause_time": 6024536.22,
            "avg_throughput": 57974666909.47,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        },
        {
            "heap_size": "512",
            "number_of_pauses": 13583,
            "total_pause_time": 109548964900,
            "avg_pause_time": 8065152.39,
            "p90_avg_pause_time": 14456188.89,
            "avg_throughput": 71026543908.37,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        },
        {
            "heap_size": "1024",
            "number_of_pauses": 10028,
            "total_pause_time": 91535440538,
            "avg_pause_time": 9127985.69,
            "p90_avg_pause_time": 23743435.2,
            "avg_throughput": 70715007539.57,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        },
        {
            "heap_size": "2048",
            "number_of_pauses": 6257,
            "total_pause_time": 84939964591,
            "avg_pause_time": 13575190.12,
            "p90_avg_pause_time": 33744149.2,
            "avg_throughput": 72125366917.17,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        },
        {
            "heap_size": "4096",
            "number_of_pauses": 3928,
            "total_pause_time": 56826444177,
            "avg_pause_time": 14467017.36,
            "p90_avg_pause_time": 35331718.0,
            "avg_throughput": 71255828500.39,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        },
        {
            "heap_size": "8192",
            "number_of_pauses": 3546,
            "total_pause_time": 41580511333,
            "avg_pause_time": 11726032.52,
            "p90_avg_pause_time": 31194526.16,
            "avg_throughput": 70757142346.61,
            "benchmarks": [
                "..." // List of successful benchmarks
            ]
        }
    ]
}
```

Listing A.2: Garbage Collector Report