# BestGC++: Optimizing Garbage Collection Selection Through Benchmarking

GUILHERME SOARES, Instituto Superior Técnico, Portugal

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.

Additional Key Words and Phrases: Microservices, Garbage Collection, Benchmarks, Java Runtimes

## 1 Introduction

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.[1]

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" [2] principle of the Java Virtual Machine (JVM). There are various JVM implementations used, with the Hotspot JVM and GraalVM being among the most popular.

### 1.1 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.

### 1.2 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.

## 2 Background

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" [1]. One of the earliest definitions of microservices architecture was made by Lewis and Fowler [3] in the year 2014 and since then its popularity has been increasing rapidly. They characterize this architecture as a way to modularize a single application into multiple services, all of them easily deployable with lightweight communication mechanisms between them.

Associated with the popularity increase many companies and organizations started to migrate their infrastructure giving rise to

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

[2] Sun Microsystems slogan for the Java platform

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

Author's Contact Information: Guilherme Soares, guilherme.luis.s@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Lisbon, Portugal.

many articles providing case studies and surveys about microservices migration.

## 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 [2].

Garbage Collector algorithms can be divided into two main categories [3]:

- Tracing Algorithms
- Reference Counting Algorithms

## 2.3 BestGC

BestGC [4] 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. 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.

**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: $1 = throughput\_weight + pause\_time\_weight$.

**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* 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* and *top* command, and if it reaches an average above 90% the application is classified as **CPU intensive** which can be leveraged so as 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**. With the *max heap* calculated we can already select the corresponding matrix i.e., 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.

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

# 3 Related Work

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 [5–7], 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 [8].

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 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 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.

**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 $\#eden\ regions + \#survivor\ regions >= newSizeRatio$ [6], 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 ZGC** The Z Garbage Collector (ZGC) [9] 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.[6]. To achieve concurrency, ZGC introduces two main novelties: colored pointers and load barriers.

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

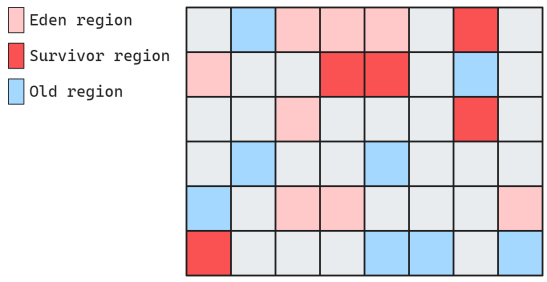


**Fig. 1.** G1 Heap Layout

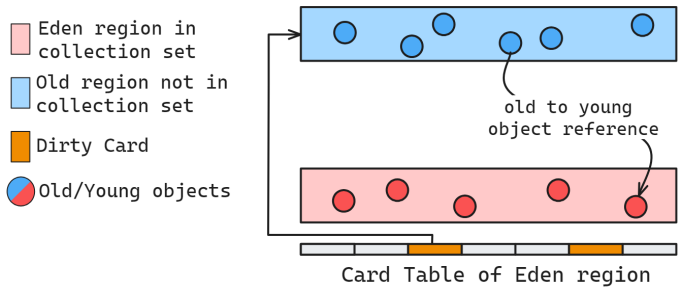


**Fig. 2.** Card Table Example

**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).

**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 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):

**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.[7]

**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.[8] Objects that are not updated in this phase, will be updated on the next marking phase or by previous load barriers.

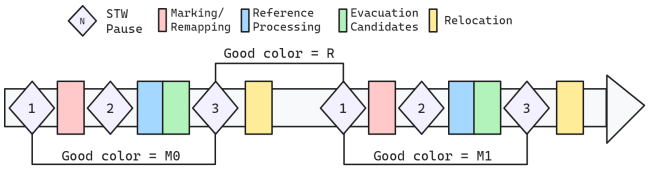


**Fig. 3.** ZGC Garbage Collection Cycle: The cycle keeps repeating alternating the first good color between M0 and M1

## 4 Solution

In this chapter, we will address the implementation architecture and components developed so as to improve BestGC. As mentioned previously in subsection 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 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 are explained more thoroughly later in subsection 4.2.

[7]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.

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

### 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 default benchmarks supported are DaCapo[9] and Renaissance[10], 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. 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.

The first building block of the application is the algorithm responsible for executing individual benchmarks and extracting relevant metrics. This algorithm, begins by fetching the appropriate command to run, taking into account the configuration in `benchmark_config.json`. For instance, an example of a command might be:

```
java -XX:+UseZGC -Xms4096m -Xms4096m
-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 subsection 3.1.2) while logging the garbage collection operations[11]. 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`[12] 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).

After an algorithm runs 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 2). After running all user-provided `benchmark_groups`, invalid benchmark reports are removed i.e., if a benchmark is not successful across all garbage collectors. 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

[9]https://www.dacapobench.org/

[10]https://renaissance.dev/

[11]https://docs.oracle.com/en/java/javase/11/jrockit-hotspot/logging.html#GUID-33074D03-B4F3-4D16-B9B6-8B0076661AAF

[12]https://man7.org/linux/man-pages/man1/top.1.html

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

$$\text{throughput}_{score} = \sum_{i=1}^{\#\text{benchmarks\_success}} \text{throughput}_i \quad (1)$$

$$\text{p90\_pause\_time}_{score} = \sum_{i=1}^{\#\text{benchmarks\_success}} \text{pause\_time}_i \quad (2)$$

As you can see in the Equation 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 1.

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

**Listing 1.** 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>,
            "benchmarks": ["..."] // List of benchmarks
        },
    // ...
}
```

**Listing 2.** Structure of GC Report

## 4.2 BestGC++

As mentioned before in subsection 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 subsection 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. 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**[12] 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**[13], 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.

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 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.

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

$$\text{throughput\_weight} = \begin{cases} 0 & \text{cpu\_avg} \in [0, 30] \\ \frac{\text{cpu\_avg}}{60} - 0.5 & \text{cpu\_avg} \in [30, 90] \\ 1 & \text{cpu\_avg} \in [90, 100] \end{cases} \tag{3}$$

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[14]) - 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*[15], 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:

*Core Services:*

**Matrix Service:** This service is responsible for loading the matrix generated by BenchmarkGC (see subsection 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 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**[12] 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.

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

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 predecessor, 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

## 5.1 Overview

This chapter provides a comprehensive evaluation of the two GC profiling applications developed, as detailed in subsection 4. We begin with an analysis of the results obtained with the BenchmarkGC application (see subsection 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 subsection 3.1.2, Generational ZGC was introduced in Oracle HotSpot, version 21, however, this feature is not available for GraalVM (with native image), which lacks support for ZGC entirely. Furthermore, Graal JIT only supports non-Generational ZGC[16], 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 subsection 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 subsection 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.

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

## 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 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.

**5.2.1 Runtime Analysis** Looking at the HotSpot and GraalVM runtimes results, 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.

**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.

## 5.3 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 subsection 5.2, wasn't included in the matrix generation phase, we opted to use it.

**5.3.1 Spring PetClinic - Benchmarks** For establishing a baseline of comparison, **PetClinic** was executed using the BenchmarkGC profiling application across both HotSpot and GraalVM runtimes. 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. 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.

**5.3.2 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 subsection 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 (subsection 2.3) is defined as follows:

$$Score = throughput_{weight} \times throughput_{score} + pause\_time_{weight} \times pause\_time_{score} \quad (4)$$

- 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 subsection 2.3 for more details).

Examining Equation 4 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 4. This leads to the piecewise Equation 5:

$$\text{Score} = \begin{cases} \text{throughput}_{\text{weight}} \times \text{throughput}_{\text{score}} & \text{if pause\_time}_{\text{weight}} = 0 \\ \text{pause\_time}_{\text{weight}} \times \text{pause\_time}_{\text{score}} & \text{if throughput}_{\text{weight}} = 0 \\ \text{throughput}_{\text{weight}} \times \text{throughput}_{\text{score}} \times \text{pause\_time}_{\text{weight}} \times \text{pause\_time}_{\text{score}} & \text{otherwise} \end{cases} \quad (5)$$

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 4 versus the modified Equation 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.

| 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 |

**Table 1.** Spring PetClinic BestGC++ GC Selection in GraalVM and HotSpot with a Heap Size of 512MB - Old Equation 4 vs New Equation 5

The results are summarized in Table 1. 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 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 previous observations. 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), BestGC++ made 5 out of 6 correct selections, compared to 3 out of 6 using the old Equation 4, revealing 33% improvement.

### 5.4 Summary

In this subsection 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 subsection, 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

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.

**6.0.1 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 [10]. 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++.

## References


[1] 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

[2] 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.

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

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

[5] 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

[6] 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

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

[8] 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

[9] 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

[10] 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.