Received 24 June 2023, accepted 3 July 2023, date of publication 11 July 2023, date of current version 19 July 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3294398*

RESEARCH ARTICLE

# BestGC: An Automatic GC Selector

**SANAZ TAVAKOLISOMEH[1], RODRIGO BRUNO[2], AND PAULO FERREIRA[1], (Senior Member, IEEE)**
[1]Department of Informatics, University of Oslo, 0373 Oslo, Norway
[2]INESC-ID/Técnico, ULisboa, 1000-029 Lisbon, Portugal

Corresponding author: Sanaz Tavakolisomeh (sanazt@ifi.uio.no)

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

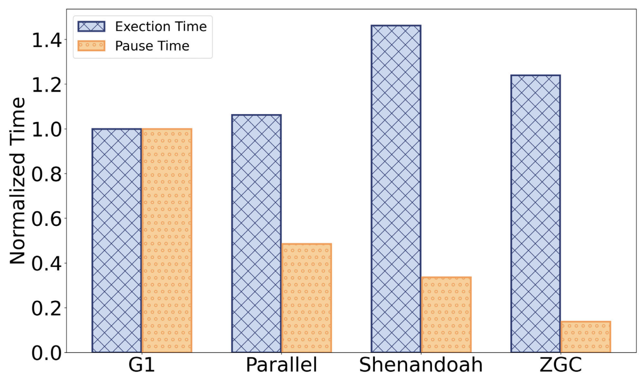
**INDEX TERMS** Garbage collection, java virtual machine GCs, pause time, throughput, memory management, generational GC, concurrent GC.

## I. INTRODUCTION

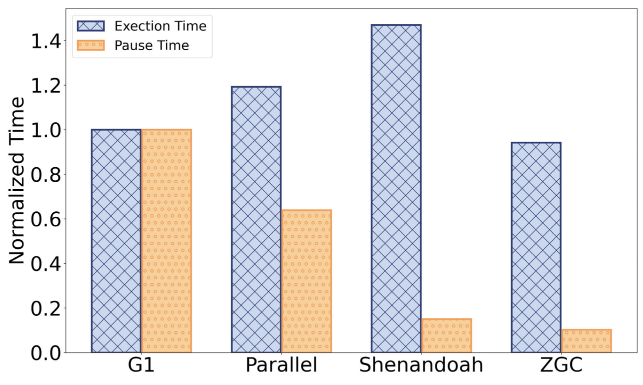
A significant portion of today's applications use managed runtime languages like Java, and therefore, take advantage of automatic memory management, also commonly known as Garbage Collection. It is a crucial component in managed runtimes as it eliminates developers' effort to manually allocate and deallocate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

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

The associate editor coordinating the review of this manuscript and approving it for publication was P. Venkata Krishna.



(a) Heap Size = 256 MB



(b) Heap Size = 4096 MB

**FIGURE 1. Normalized application execution time and 90th percentile of GC pause time for Philosopher workload (from Renaissance benchmark suite). Lower is better.**

an operation is widely distinct for the workloads we use. In addition, for GC pause time, the usual 90th percentile is used.

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

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

We extensively evaluate four widely used GCs (G1, Parallel, Shenandoah, and ZGC) in OpenJDK version 15. This evaluation considers application throughput (in fact, total application execution time), GC pause time, and diverse heap sizes. We do so while aiming at the following sub-goals:

- evaluating four well-known GCs (G1, Parallel, Shenandoah, and ZGC) in a production JVM implementation (OpenJDK) considering application execution time and GC pause time, with different amounts of memory available (256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, or 8192 MB);
- conducting the above mentioned experiments based on a set of workloads from standard benchmark suites (DaCapo [2] and Renaissance [1]) that represent existing real-world applications to reveal the costs of the GCs; and
- proposing an approach to suggest a suitable GC for a user's application while considering how the user cares about the GC pause time and application throughput.

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

We evaluate G1 [3], Parallel [4], Shenandoah [5], and ZGC [6], the most relevant GC implementations available in OpenJDK 15. We use workloads from two widely-used benchmark suites, DaCapo [2] and Renaissance [1]. These two benchmarks include diverse workloads that mostly need CPU (CPU-intensive), which is also a critical resource for GC. Using a forthcoming release of DaCapo, called DaCapo Chopin,[1] we have several latency-sensitive workloads.

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

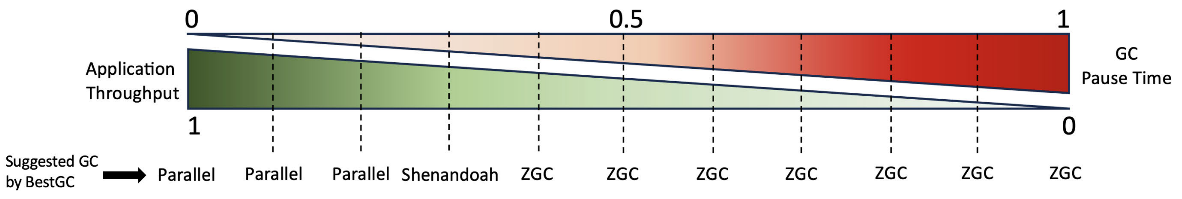


**FIGURE 2.** Suggested GCs by BestGC for varying user preferences in terms of application throughput and GC pause time, for the Compress workload (included in SPECjvm2008 benchmark suite) and with a heap size of 512 MB. The suggested GC algorithm is Parallel when prioritizing application throughput,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

These workloads enqueue requests if they can not be processed immediately; this causes requests to be served after some time. This latency can be affected by a GC and results in longer application execution times (i.e., less throughput). Therefore, using these workloads, we can consider the concurrency overhead of the GCs on our studied performance metrics [7].

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

- extensive measurement of four GCs (G1, Parallel, Shenandoah, and ZGC) using two widely accepted benchmarks suites (DaCapo and Renaissance);
- developing a system, BestGC (by using a set of matrices we created from the results obtained from our extensive measurements done in the previous step) that frees the user from the complicated process of selecting a GC that fits a user's application performance goals; and
- validation of BestGC with the workloads from another widely used benchmark (which is SPECjvm2008 [8]).

Due to the importance of the GCs, extensive research analyzed, characterized, and proposed new GCs to either improve GCs' capabilities or fix current GC issues [9], [10], [11], [12]. Several studies also compared GC solutions regarding performance metrics like application throughput, GC pause time, and memory usage [13], [14], [15], [16], [17], [18], [19]. These performance metrics are the most common, and we also consider them the most critical metrics that highly affect users' applications. However, we believe that there is a lack of a system that suggests the best GC, among those existing powerful and in-production GCs, that matches a user's application performance goals. We believe BestGC is the first system that allows the user/developer to overcome the challenges of selecting a suitable GC that meets the application's performance requirements. BestGC is extendable to work with new JDK versions and new GCs. Besides, the methodology used in BestGC could be used in other runtimes in which there are GCs available to manage the heap.

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

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

## II. BACKGROUND

This section briefly presents GCs' key terminology and algorithms (for more detailed background, see Jones and Lins's book [13]). Then, we give an overview of the GCs evaluated in this paper.

### A. GARBAGE COLLECTION ALGORITHMS

The Java HotSpot VM [20] provides different GCs to satisfy various performance requirements of Java applications. These GCs provide dynamic memory management that facilitates allocating memory to objects, managing the heap, and reclaiming unused memory for future use. There are two types of garbage collection algorithms: reference counting [21], and reference tracing [22].

A reference counting (RC) algorithm counts the number of references (e.g., pointers) to an object, keeps that number updated, and removes the objects when their counter falls to zero (providing immediate memory reuse). However, it comes with the cost of counters' space and updating overhead, in addition to skipping cyclic structures with destroyed references. Tracing collectors identify objects in use by the running applications (also called live objects) by tracing the object graph, starting from a set of roots (registers, stacks, and global variables), and moving live objects to another space or sweeping them [23]. Although it eliminates the additional field in RC, it requires tracing the whole object graph and needs mechanisms to deal with application threads' (mutator) activities. GCs we evaluate in this work follow the reference tracing approach.

To optimize the collection, GCs divide the heap into generations or regions. Generational GCs divide the heap into two age groups. Newly created objects are placed in the young generation, and those that survive multiple collection cycles are moved into the old generation. As the generational hypothesis [24] states, most objects die (i.e., become unreferenced) after a short period of time. Therefore, the young generation often fills up with dead objects (objects that are not being referenced by the mutator threads); consequently, minor collections occur more frequently and move live objects to the old generation (and implicitly remove unused objects). A major collection runs in the old generation if it fills up. Conversely, some GCs maintain the heap as a single generation. However, GCs may use mechanisms to divide the heap into regions to make the collection and improve their performance goals (for example, to evacuate the regions with the most unused objects first [3]).

To monitor the object graph, keep track of references in generations, and provide concurrency in the collectors, GCs employ read and write barriers [15]. A read barrier (also known as a load barrier) is executed when loading an object reference from the heap [19]. In contrast, a write barrier's code snippet is invoked before any write operations in the heap [19]. Stop-the-world (STW) collectors pause mutator threads as long as the garbage collection runs, then use write barriers to update pointers to the evacuated objects. However, concurrent GCs do not pause the mutator as STW GCs. Concurrent GCs use write barriers to mark live objects and also may employ read barriers to do evacuation and reclamation simultaneously with the running mutator threads. These barriers impose performance costs on the GCs [25].

### B. EVALUATED GCs

In this work, as already mentioned, we evaluate four production GCs in OpenJDK 15:[2] G1 [3], Parallel [26], Shenandoah [5], and ZGC [6]. In this section, we highlight the GCs' key features that provide the background to understand the results obtained in the evaluation (see Section VI) that are used in BestGC (see Sections IV and V).

Parallel (also known as throughput collector [26]) is a STW generational collector. It performs garbage collection in its generations using multiple threads in parallel. So, it is well suited for applications that can tolerate long application pauses (as it is not a concurrent GC).

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

Garbage First (G1) has become the default collector of OpenJDK since version 9. It tries to keep a balance between pause time and throughput [26]. G1 divides the heap into fixed-size regions, and, as its name implies, it targets the regions containing the most existing garbage (dead objects) to start the collection from them. It is a generational GC that uses a concurrent tracing mechanism to mark live objects, yet it performs a STW collection to evacuate objects. It also uses write barriers to make sure that all the live objects remain alive during the concurrent tracing phase.

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

Z Garbage Collector, ZGC (available from JDK version 15), tries to keep the GC pause times below 10 ms. It is a single-generation collector (a generational version is under development) that divides the heap into regions of different sizes. It uses part of an object's reference (called colored bits) to keep marking and relocation-related information [19]. A read barrier checks on the colored bits once a reference is loaded to take action for the object.

## III. RELATED WORK

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

### A. NOVEL GC STRATEGIES

Many studies introduce new GCs created over existing collection algorithms. Ossia et al. [9] designed a parallel, incremental (which performs the collection in steps), and mostly concurrent GC to meet the low GC pause time goal. The collector is suitable for shared-memory and multiprocessor servers and supports highly multi-threaded applications. Pizlo et al. [10] propose STOPLESS, which uses a tracing collector and a compactor to control fragmentation to support multi-threaded applications. Pizlo et al. [11] also propose two other solutions for concurrent real-time GCs, CLOVER, and CHICKEN, to reduce the complexity of STOPLESS. Frampton et al. [12] designed an incremental, young-generation STW GC for real-time systems. The collector uses both read and write barriers to track remembered set changes.

Tene et al. [27] propose C4, the Continuously Concurrent Compacting Collector. It supports concurrent compaction, and incremental update tracing through the use of a read barrier. C4 enables simultaneous-generational concurrency that allows different generations to be collected concurrently. It continuously performs concurrent young generation collections, even during long periods of concurrent full heap collection, maintaining high allocation rates and efficiency without sacrificing response times. Wu et al. [28] introduces Platinum, a novel concurrent GC designed to reduce latency in interactive services. Platinum creates an isolated execution environment for concurrent mutators, improving application latency without restricting GC thread execution. Additionally, Platinum utilizes a new hardware feature to make access control to different regions of memory more efficient and therefore minimize software overhead present in previous concurrent collectors.

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

Bruno et al. [29] described how objects created by Big Data applications are kept in memory and surveyed the existing GCs for big data environments. They also addressed scalability issues in classic garbage collection algorithms and analyzed several relevant systems that try to solve these scalability issues. Xu et al. [30] used the Apache Spark [31] application to analyze GCs like G1 and Parallel. They proposed strategies for designing GCs specified for Big Data. Nguyen et al. [32] also propose a GC with low pause time and application high throughput based on the logical distinction between the data path and the control path. The data path consists of data manipulation functions, while the control path is responsible for cluster management, setting communication channels between nodes, and interacting with users. They believe these paths differ in heap usage and object creation patterns. So, based on the previously mentioned paths, they divide the heap into data and control spaces to reduce the objects managed by the GC. Not many objects are created in the control spaces, and they are subject to generation-based collection, while the objects in the data space, which creates the most objects, are subject to region-based collections. However, the developer is responsible for marking the beginning and end points of the data path in the program. Broom [33] and NG2C [34] also propose two GCs for Big Data systems. However, in Broom, the programmer has to create the regions in the heap explicitly; in NG2C, the programmer identifies the generation in which a new object should be allocated. These GCs require developer efforts and are prone to errors. Then, the authors of NG2C proposed POLM2 [35], an offline profiler that automatically infers generations for object allocation, and finally, ROLP [36], an online profiler to select an object generation with no user intervention.

### B. GC COMPARATIVE ANALYSIS

Zhao et al. [15] introduce LXR to deliver low GC pause times and high application throughput. Their approach includes STW collections to increase responsiveness and an RC mechanism to deliver scalability and promptness. They evaluated and compared barrier overhead and pause time for GCs like G1, Shenandoah, and ZGC that we also used in this paper; yet, note that RC-based GCs are not widely used in production and, therefore, they are out of the scope of this document. Zhao et al. [14] decomposed G1 into several key components to evaluate the impact of different algorithmic elements of G1 on performance. Pufek et al. [19] analyzed several garbage collectors like G1, Parallel, Serial (a GC that uses the same thread as the mutator [37]), and CMS [38] (a generational GC with a concurrent tracing in the old generation).[3] They used the DaCapo benchmark suite [2] to evaluate the GCs. They compare the number of algorithm iterations and the total GC pause time for applications running on top of JDK 8 and JDK 11. Also, they compared ZGC and Shenandoah, which were experimental GCs by then, with G1. They concluded that G1 and Parallel operated better than Serial and CMS regarding the overall duration of collections. They also showed that those experimental GCs would contribute to overall system optimizations. Their results also emphasize the importance of evaluating ZGC and Shenandoah, as we do in this paper.

Grgic et al. [17] analyzed Serial, G1, Parallel, and CMS using the DaCapo benchmark suite in JDK version 9. They concluded that G1 is a better choice regarding the total number of garbage collections and CPU utilization for multi-threaded environments (but not for single-threaded environments).

Lengauer et al. [39] describes commonly used benchmarks, including DaCapo, DaCapo Scala [40], and SPECjvm2008 [8] in terms of memory and garbage collection behavior. They compared G1 and Parallel Old (parallel collection in the old generation) regarding the number of full collections (GC counts), the GC time relative to the total execution time of the application, and GC pause time. They concluded that G1 performs better than the Parallel Old concerning GC pause time by selecting different regions to collect.

Beronić et al. [18] investigated and compared memory issues, heap allocation, CPU usage, and duration of collection in three GCs: G1, ZGC, and Shenandoah. They found that GCs are sensitive to the heap size and that G1 uses less heap than the two other GCs. However, G1 is more CPU intensive since it occupies more OS threads necessary for scanning the live objects in a heap.

Cai et al. [7] introduced a new methodology that defines a practical lower bound on the costs of GCs for any given cost metric. They showed that GCs are sensitive to heap size and indicated that low GC pauses achieved by concurrent GCs do not translate into low application latency. To achieve the conclusions, the authors used latency-sensitive workloads from the DaCapo Chopin benchmark suite. These latency-sensitive workloads serve requests arriving at a determined rate, and if they are not able to process a request instantly, they enqueue the request. Therefore, they used a metric, called metered latency, to show the delay both for the executing and

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

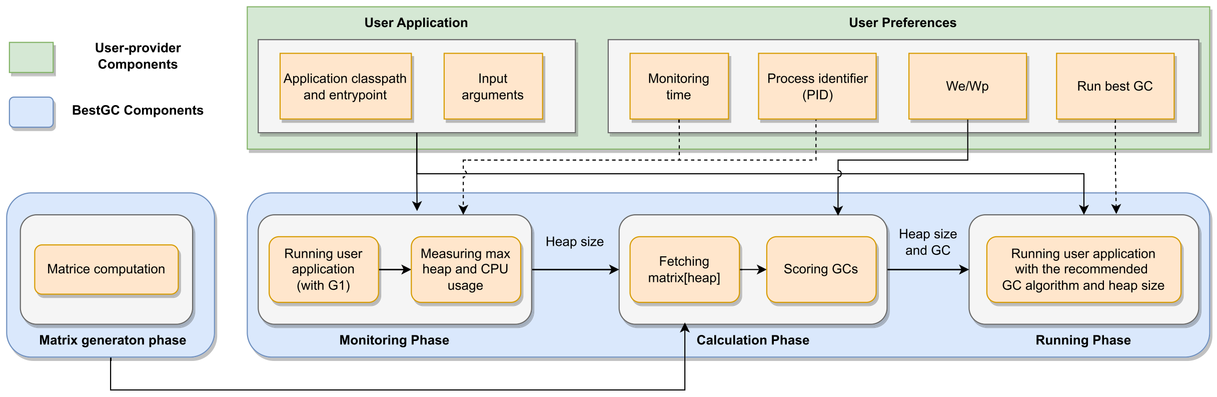


**FIGURE 3.** **Overall architecture of BestGC. Solid arrows for mandatory inputs and dotted arrows for optional inputs.**

the enqueued requests. Using this metric (metered latency), the authors show that the GCs, especially concurrent ones, cause delays for both executing and enqueued requests and therefore affect the latency. In this document, we do not measure such metric, but we include latency-sensitive workloads in our evaluations; in fact, we do so because it impacts the application's total execution time, the metric we use to report the throughput.

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

## IV. BestGC

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

### A. MATRICES GENERATION PHASE

A set of matrices for different heap sizes is included in BestGC. They hold the manually evaluated application execution time and pause time for four GCs (G1, Parallel, Shenandoah, and ZGC). Such evaluated results are obtained by measuring the average application execution time and GC pause time (with different heap sizes) of the workloads in DaCapo [2] and Renaissance [1] in OpenJDK version 15 (more details in Section VI). These matrices are generated only once and inserted as constants in BestGC. BestGC fetches the matrix corresponding to the measured heap size to score GCs, suggests the best of the GCs (see Section IV-D), and runs the user/developer application with it.

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

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

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

**FIGURE 4. Example of a matrix for the heap size of 8192 MB.**

**TABLE 1. Available switches for BestGC.**

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

Figure 4 shows such a matrix (as an example). As can be seen, the matrix shows the application execution time and GC pause time when the heap size is 8192 MB normalized to G1. For each evaluated GC and heap size, the first column contains the average application execution time of all the workloads; the second column contains the average 90th percentile of the GC pause times (values are normalized to G1) also for all workloads. This is detailed in Section IV-D.

## B. INPUT OPTIONS FOR BestGC

BestGC requires a few simple inputs to run.[4] The user (who is defined as she/he who wants to run a Java application with the best performance possible) needs to pass two inputs (see Table 1): first, the mandatory absolute path to the application's jar file, plus all the application-specific input arguments (if the user's application is already running, the user must pass its process id to BestGC using the *pid* command option); second, a mandatory weight (between 0 and 1) for application execution time ($w_e$) and/or GC pause time ($w_p$) to show how the user cares for these two performance metrics (1 for the highest importance) in such a way that $w_p + w_e = 1$. Users have to specify at least one of these weights to run BestGC.

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

**Algorithm 1** Calculation of Maximum Heap Usage

```
1:  $t \leftarrow 0$
2:  $pid \leftarrow user\_app\_pid$
3:  **while** $t \leq monitoringTime$ **do**
4:      $heap \leftarrow 0$
5:      $jstat \leftarrow jstat(pid)$
6:      $heap \leftarrow jstat.s1u + jstat.eu + jstat.ou + jstat.ccsu$
7:      $heap\_usage\_list.add(heap)$
8:      $t++$
9:  **end while**
10: $max\_heap \leftarrow heap\_usage\_list.max()$
```

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

## C. MONITORING PHASE

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

Based on the recorded maximum heap usage in this phase, BestGC offers a reasonable heap configuration for future executions of the user's application. As already mentioned, such maximum heap values can be 256 MB, 512 MB, 1024 MB, 2048 MB, 4096 MB, or 8192 MB. We now describe how a Java application's heap usage is obtained by BestGC. The pseudocode in Algorithm 1 shows how BestGC measures the heap usage of a Java application. To capture the maximum heap usage, every second, BestGC invokes the *jstat*[5] command with the *gc* option for the user's application using its Process ID (line 5). This option displays statistics about the heap behavior. BestGC considers all the metrics that report the capacity of the different parts of the whole heap (line 6). Considering G1, which BestGC uses to run a user's

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

**Algorithm 2** Calculating Number of Total Engaged Cores

```
1: $t \leftarrow 0$
2: $pid \leftarrow user\_app\_pid$
3: $engaged\_cores \leftarrow 0$
4: while $t \leq monitoringTime$ do
5:     $cpu\_by\_core\_list.add(read(/proc/stat))$
6:     for all $core\_usage$ in $cpu\_by\_core\_list$ do
7:         $new\_core\_cpu\_usage \leftarrow core\_usage.all\_usage\_values$
8:         $new\_core\_cpu\_idle \leftarrow core\_usage.idle\_value$
9:         $delta\_core\_cpu\_usage \leftarrow new\_core\_cpu\_usage - last\_core\_cpu\_usage$
10:        $delta\_core\_cpu\_idle \leftarrow last\_core\_cpu\_idle - last\_core\_cpu\_idle$
11:        $core\_cpu\_usage \leftarrow 100 \times (delta\_core\_cpu\_usage - delta\_core\_cpu\_idle)/delta\_core\_cpu\_usage$
12:        if $core\_cpu\_usage \geq 50$ then
13:            $engaged\_cores\_per\_second + +$
14:        end if
15:    end for
16:    $engaged\_core\_list.add(engaged\_cores)$
17:    $t + +$
18: end while
19: $engaged\_cores \leftarrow engaged\_core\_list.average()$
```

**Algorithm 3** Calculating Average CPU Usage per Engaged Core

```
1: $t \leftarrow 0$
2: $pid \leftarrow user\_app\_pid$
3: while $t \leq monitoringTime$ do
4:     $cpu\_usage \leftarrow top(pid)$
5:     $cpu\_usage\_list.add(cpu\_usage)$
6:     $t + +$
7: end while
8: $average\_cpu \leftarrow cpu\_usage\_list.average()$
9: $average\_cpu\_per\_core \leftarrow$
       $average\_cpu/engaged\_cores$
```

application in this phase, these metrics are: *S1U* (survivor space 1 utilization), *EU* (Eden space utilization), *OU* (old space utilization), and *CCSC* (compressed class space used). Finally, the maximum heap usage is detected by BestGC at the end of the *monitoring-time* (line 10).

BestGC also reports if a user's application is CPU-intensive or not. BestGC obtains a Java application's CPU usage through two steps; first, it calculates the number of engaged cores while running the Java application (Algorithm 2), and then the average CPU usage per engaged core (Algorithm 3). When running a CPU-intensive application, the choice of GC is even more important (than running a non-CPU-intensive application) given that the CPU is a shared resource between the application and the GC. This feature is also included in BestGC for adding future GC selection factors in the system. For best results, BestGC should run alone on the user's machine to avoid interference from other applications that may impact CPU measurements during the monitoring phase. BestGC captures the amount of CPU the user's application uses every second during the *monitoring-time* interval (Algorithm 2, line 4). Moreover, since not all the applications utilize all the CPU cores in the machine, there is a function (pseudocode presented in Algorithm 2) to calculate the number of engaged CPU cores for the user's application. The *proc/stat*[6] command reports different metrics for each CPU core in Linux. BestGC sums all consumed values (line 6) and the CPU idle time (line 7) for each CPU core. Then, to calculate the core CPU usage at the current second relative to the last second, BestGC calculates the difference between the new and last CPU usage (also the CPU idle time) for the selected core. Finally, it computes the CPU usage for the core at the current second (lines 8-10). Should the value be above 50% for a core, BestGC increases the number of engaged CPU cores per second by one. Then, the average number of engaged cores is calculated at the end of the *monitoring-time*. Also, the average total CPU utilization (Algorithm 3) is computed by averaging the recorded total CPU usage every second using the *top*[7] command and dividing it by the average number of engaged CPU cores. Should the results be over 90%,[8] the application is CPU-intensive; otherwise, it is considered non-CPU-intensive (Algorithm 3, lines 4-8).

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

## *D. CALCULATION PHASE*

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

BestGC selects the suitable matrix among the matrices available for all six heap sizes depending on the maximum heap size used by the user's application (see Section IV-C).

[6]https://manpages.ubuntu.com/manpages/xenial/man5/proc.5.html
[7]https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html
[8]We used the considerations in the Linux command *atop*, which assumes the CPU usage to be critical when the total CPU usage percentage is 90% and above; https://manpages.ubuntu.com/manpages/bionic/man1/atop.1.html

BestGC considers 20% headroom for the application (i.e., *max_heap* × 1.2) and picks the closest bigger heap configuration's matrix available. Due to our evaluation, 20% additional heap space is big enough to let GCs manage the heap. For example, during a 30-second *monitoring-time*, BestGC reports that the maximum heap memory used by *Tomcat* is 308.1 MB. It simply computes the required heap, 308.1 × 1.2 = 369.72, and looks for the closest next heap size in its heap sizes set, which is 512 MB. Then, it uses matrix_512 for further calculations.

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

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

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

### E. RUNNING PHASE

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

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

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

## V. IMPLEMENTATION

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

### A. BestGC

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

JDK provides tools like *jcmd*[9] and *jstat* to measure the heap usage. Using *jcmd* to monitor heap usage requires enabling the Native Memory Tracking feature (`-XX:NativeMemoryTracking=[summary | detail]`) when starting the JVM. However, Native Memory Tracking will cause a 5-10% performance overhead to JVM [41] which is not the case with *jstat*. Therefore, we decided to use *jstat* in BestGC. *Jstat* provides different output options to display various statistics. We use the *gc* option, which represents the behavior of the garbage-collected heap. Computing the utilized capacities reported for separated parts of the heap reveals the heap consumption at the moment in which *jstat* is invoked. Since BestGC sets no JVM options while executing the user's application, it employs the JVM default GC (G1 since JDK version 9). Therefore, *jstat* reports heap consumption, including both young and old generations statistics. Recording the total heap usage every second lets BestGC find the adequate heap size to decide on the most suitable GC for the application.

### B. HEAP SIZE VARIETY

GCs are sensitive to the heap size [2]. The way GCs manage the heap is one of their critical performance features. Large heap memory size shows GCs' behavior when there exists much memory to assign the new objects and GCs have to manage a large memory area. However, small heap memory causes frequent garbage collections to free more memory; this has obvious impacts on the GCs' pause time and may lead to GCs' failure in the worst case. This research also involved heap size in GC evaluations to monitor its impact on the other performance metrics, such as application execution time and GC pause time. As already mentioned, we set the heap size in our evaluations to 256, 512, 1024, 2048, 4096, and 8192 MB. First, heap sizes in powers of two are commonly used by the users [42], also, these include the required heap sizes to run all workloads in our evaluations. If the maximum heap usage by the application is higher than 8192 MB, BestGC runs the application with the actual maximum heap usage (plus extra headroom for running); however, it chooses the GC based on the GC scores for 8192 MB heap size (conversely, the same process applies for heap usage bellow 256 MB).

## VI. EVALUATION

First, we describe some details regarding the benchmark suites we use: DaCapo and Renaissance. Then, we present the application execution time and GC pause time results (per each heap size) for the selected workloads. As already mentioned, these benchmark workloads are representative of real-world applications. Finally, we validate BestGC using workloads from a third benchmark suite, SPECjvm2008 [8]; BestGC runs the workloads from SPECjvm2008 as any other

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

**TABLE 2.** **Workloads from DaCapo and Renaissance benchmark suites used in the experiments.**

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

application a user may run. All the results were obtained on a server running GNU/Linux, Ubuntu 16.04.4 LTS, with eight *Intel(R) Xeon(R) E5506 @ 2.13GHz* CPUs and *16 GB* of RAM.

## A. BENCHMARKS

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

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

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

Table 2 shows the workloads in DaCapo and Renaissance benchmark suites we used in this work. Some of the workloads in DaCapo and Renaissance benchmark suites fail due to incompatibility with JDK version 15 (e.g., Cassandra in DaCapo or Db Shootout in Renaissance) or other errors like database connection failures (in Tradesoap and Tradebeans workloads). Also, the strategy we employ to evaluate GCs provides GCs with the same fixed heap sizes and eliminates the dependence of the GCs on the available heap memory on the machine; also, it prevents GCs from freely choosing heap sizes by their different policies. Since they have to manage the same heap size, it makes their abilities comparable. This heap selection strategy in BestGC results in facing failures while running some workloads due to an *Out of Memory* error. Thus, we omit all the non-running workloads from the evaluations with all the heap sizes. This narrows our set of workloads down but allows us to make the GC comparison fair since they are working on the same workloads with the same heap availability.

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

## B. APPLICATION EXECUTION TIME AND GC PAUSE TIME

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

### 1) APPLICATION EXECUTION TIME

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

In the Renaissance, Parallel outperforms G1 (value 1), Shenandoah, and ZGC, respectively, by about 8%, 40%, and 60% on average. Parallel can keep the workloads' execution time almost the same for all the heap sizes, while application execution time in Shenandoah and ZGC worsens when decreasing the heap size. For example, with a 256 MB heap size, the application execution time of the workloads

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

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

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

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

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

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

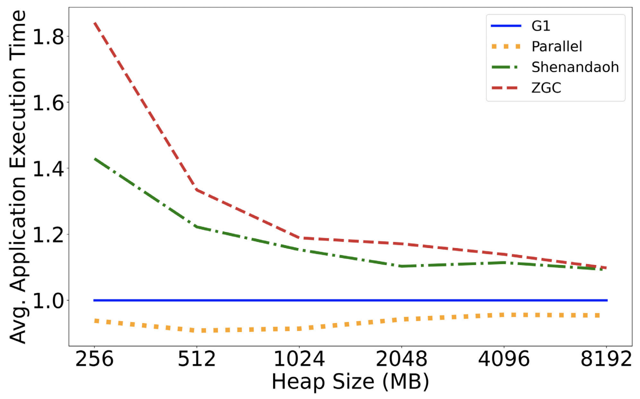
| GC | a) DaCapo | | | | | | b) Renaissance | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Heap Size (MB) | | | | | | Heap Size (MB) | | | | | |
| | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 256 | 512 | 1024 | 2048 | 4096 | 8192 |
| **Parallel** | 2.298 | 2.208 | 2.485 | 2.188 |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

is 2.5× and 1.8× better with Parallel compared to ZGC and Shenandoah, respectively. This shows that the workloads in Renaissance are more aggressive regarding memory usage than DaCapo; in fact, using concurrent GCs, which work concurrently with the application threads, significantly affects the application execution time of these workloads. This difference is not happening in G1 (value 1), even with a 256 MB heap size. Unlike ZGC and Shenandoah, G1 is a generational collector that keeps a balance between throughput (application execution time in our case) and GC pause time. Also, since the minor collection happens more frequently, G1 is able to manage the heap and, consequently, the overall application execution time better.

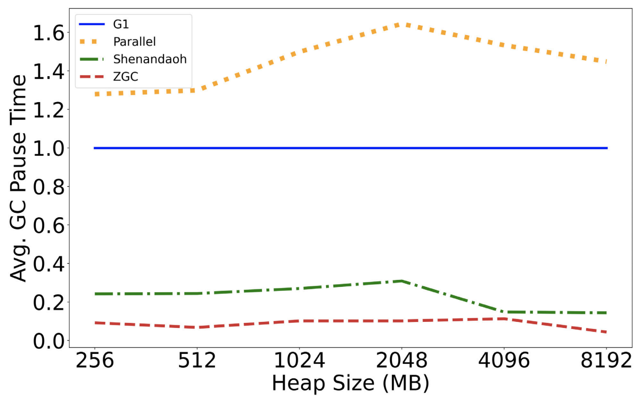
### 2) GC PAUSE TIME

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

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



(a) application execution time



(b) GC pause time

**FIGURE 5.** **Normalized (to G1) average application execution time and the normalized average of 90th percentile of GC pause time for all the DaCapo and Renaissance workloads.**

using its concurrent evacuation phase, performs better than Parallel. After ZGC, Shenandoah manages the GC pause time better than G1 (value 1) and Parallel; it obtained results closer to ZGC than G1 (value 1) and Parallel. However, because of the overheads of the different barriers [5] it uses to provide concurrent collection, it manages the GC pause time worse than ZGC.

### 3) DISCUSSION

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

As Figure 5 shows, Parallel outperforms other GCs for all heap sizes. It keeps application execution times almost the same while changing the heap size. Parallel, which is a throughput-oriented GC, adjusts the generations' sizes to reach the best throughput (application execution time in our case). G1 comes after Parallel regarding the time it takes to run a workload. Shenandoah and ZGC have worse results and show different behavior with different heap sizes. Even with an 8192 MB heap size, they both have higher application execution times than G1 and Parallel. These two GCs' application execution times (Shenandoah and ZGC) are almost the same for 8192 MB; their execution times worsen when the heap size gets smaller. In fact, reducing the heap size from 1024 MB to 512 MB results in a significant application execution time degradation in Shenandoah and especially in ZGC. With a 256 MB heap, ZGC shows the worst application execution time among the other GCs. Compared to Shenandoah, which is also a concurrent GC, ZGC sacrifices more of the workloads' application execution time to provide concurrency since it requires more room in the heap to allow object allocations while the GC is running [43]. Parallel and G1 are generational collectors. As already mentioned, since newly created objects tend to die in a short time (generational hypothesis), garbage collection happens most frequently in the young generation. So, in each collection, a GC detects a noticeable amount of dead objects by tracing a portion of the heap (not the entire heap). Therefore, as Figure 5 shows, these two GCs (Parallel and G1) deliver better application execution time, in our study, than ZGC and Shenandoah.

As Figure 5b shows, unlike the application execution time results, Parallel performs poorly regarding the GC pause time. Parallel tries to change one generation size at a time (the generation with the more significant GC pause time [26]). With a 2048 MB heap capacity, it experiences the highest GC pause time to meet its throughput (application execution time) goal. While more heap capacity is available for Parallel, it may increase the young generation size; also, the objects created by the workloads fill this capacity, resulting in longer GC pause times for the frequent collections in the young generation with 2048 MB heap. For heap sizes above 2048 MB, generation sizes created by Parallel are large enough not to invoke a collection repeatedly. GC pause times for concurrent GCs, especially ZGC, are shorter than generational G1 and Parallel, as Figure 5b illustrates. In other words, utilizing concurrent tracing and copying mechanisms, ZGC and Shenandoah results are significantly better than G1 and Parallel regarding the GC pause time. Although Shenandoah keeps the GC pause time very small, it shows no predictable pattern for different heap sizes. However, ZGC maintains almost a steady average GC pause time for all the heap sizes because its main goal is keeping GC pause times small, regardless of the heap size [6].

## C. CPU USAGE AND GC

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

These benchmark suites mostly contain workloads with high CPU consumption, yet, we selected some workloads with low and very high CPU usage per engaged cores (as discussed in Section IV-C).

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

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

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

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

### D. VALIDATION OF BestGC RESULTS

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

- We use the `Lagom` switch (available in SPECjvm2008) to run each workload. Lagom provides a fixed-size workload for the benchmarks, i.e., it does a fixed number of operations in each benchmark (just like DaCapo and Renaissance benchmark suites). Then, we use the corresponding application execution time as a metric for throughput.
- We select eleven workloads (see Table 7) from SPECjvm2008. We excluded the Startup workload since it has a very short execution time (less than one second), as well as the Compiler workload since it is not compatible with JDK version 15.
- Just as with the workloads from DaCapo and Renaissance benchmark suites shown in the previous sections, we invoke all the workloads of the SPECjvm2088 several times to the point the execution times remain stable, employing one of four GCs (G1, Parallel, Shenandoah, ZGC) at a time. Then, we use the application execution time and GC pause times of the last iteration.

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

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

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

The heap sizes obtained from the empirical maximum heap size measurement and the one obtained with BestGC are shown in Table 8. In the second column, the real max

**TABLE 6.** Average performance benefit of the selected GC for each workload with different heap sizes.

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

**TABLE 7.** Workloads from SPECjvm2008 benchmark suite used in the validation.

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

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

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

heap size utilized by each SPECjvm2008's workload is multiplied by 1.2. We consider 20% extra headroom for GC (see Section V) as we did with BestGC. In the third and fourth columns, it can be seen that the measurement of the heap in BestGC is consistent with the measurements we took empirically.

Table 9 shows the most appropriate heap size (empirically obtained), the suggested GC by BestGC (for different $w_e$ and $w_p$ values), and the most proper GC, all for the Compress workload. Note that the last row is the GC with the minimum score obtained empirically (for the Compress workload) corresponding to the heap used. As the table shows, for all the $w_p$ larger than 0.3, the suggested GC by BestGC and the GC obtained empirically are exactly the same. With $w_p = 0.3$ and $w_e = 0.7$, although the GC suggested by BestGC and the GC empirically obtained are not exactly the same, they are both in the fully-concurrent GC category. For other values of $w_p$ and $w_e$, the suggested GC by BestGC and the GC empirically obtained are neither the same nor of the same category. In these cases, we think that such a mismatch may happen because BestGC did not use even more workloads. In the future, as mentioned in Section VIII, we plan to include further evaluations of other classes of workloads that rely on resources other than CPU, such as I/O. This will allow BestGC to have even better results.

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

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

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

The table indicates, for instance, that BestGC's exact GC suggestion percentage is 63.64% for the Compress workload. At the same time, it offers a GC with the exact category in 81.82% of the cases. BestGC failed to suggest the exact GC category in 18.18% of the cases (worst-cases) in Compress; however, the suggested GC by BestGC still causes a 4.18% improvement (worst-case performance benefit) in GC score compared to running the user application with the default GC (G1). In addition, for this workload, it is shown in the table that using the suggested GC (with respect to

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

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

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

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

all failures and successes of BestGC for all the $w_e$ and $w_p$) makes, on average, a 37.53% improvement compared to default GC (G1) when the user does not use BestGC. The worst-case performance benefit for MPEGaudio and XML workloads are 0. It shows that for these two workloads, BestGC suggested G1, where it failed to detect both the exact GC and the exact category; thus, there is 0% improvement between the suggested GC and the default GC (G1). Also, according to the table, BestGC suggests the exact GC for all the application execution time and GC pause time weights in the Serial workload. For Sunflow and Crypto.aes workloads, although the *exact GC* metric for BestGC is not 100%, it could suggest a GC with the correct GC category (Figures 5a and 5b demonstrates that most of the time, GCs with the same category have close scores). In Scimark.large and Derby workloads, the worst-case performance benefit is 0.55% and 4.97%, respectively. For these workloads, using BestGC still results in significant overall performance benefits compared to G1. The average of all the worst-case performance benefits for the workloads in SPECjvm2008 is 1.75%. It shows that using the suggested GC by BestGC has a 1.75% performance benefit compared to the default GC (G1). BestGC suggests the exact GC for the workloads in 51.24%, while it suggests the best GC category in 85.95%, on average. BestGC's average overall performance benefit is reported in the last column in Table 10. It indicates that using BestGC's suggested GC results in an average improvement of 36.75% (including all the failures and successes of BestGC) compared to the situation in which applications are run with the default GC (G1).

## VII. CONCLUSION

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

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

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

We also evaluated BestGC using SPECjvm2008 workloads in which each workload is used as any user application. On average, BestGC suggests the most proper GC for about 51.24% of the time; also, it suggests a GC with the best category (concurrent/non-concurrent) on average about 85.95% of the time. When BestGC fails, still using the suggested GC by BestGC results in about a 1.75% improvement in GC score compared to using the default OpenJDK GC (G1). This improvement for a highly optimized environment like OpenJDK would greatly affect the performance of the users' applications, especially for big data and cloud applications. Using BestGC results in having a GC with an average of 36.75% overall performance benefit, considering both BestGC's failures and successes, compared to running the user's application with the default GC (when not using the BestGC).

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

## VIII. FUTURE WORK

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

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

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

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

Furthermore, in our future work, we also plan on exploring Machine Learning techniques [44] to achieve a more accurate model and replace the formula we used in BestGC. It has the potential to enhance the accuracy of the GC offered by BestGC.

## REFERENCES

[1] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the JVM," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2019, pp. 31–47.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, and A. Diwan, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Appl.*, 2006, pp. 169–190.

[3] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proc. 4th Int. Symp. Memory Manage.*, Oct. 2004, pp. 37–48.

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

[5] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK," in *Proc. 13th Int. Conf. Princ. Practices Program. Java Platform, Virtual Mach., Lang., Tools*, Aug. 2016, pp. 1–9.

[6] Liden and Karlsson. (2018). *ZGC: A Scalable Low-Latency Garbage Collector*. [Online]. Available: https://openjdk.java.net/jeps/333

[7] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, May 2022, pp. 46–57.

[8] (2008). *SPECjvm2008*. [Online]. Available: http://www.spec.org/jvm2008/index.html

[9] Y. Ossia, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, and A. Owshanko, "A parallel, incremental and concurrent GC for servers," in *Proc. ACM SIGPLAN Conf. Program. Lang. design Implement.*, May 2002, pp. 129–140.

[10] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard, "Stopless: A real-time garbage collector for multiprocessors," in *Proc. 6th Int. Symp. Memory Manage.*, Oct. 2007, pp. 159–172.

[11] F. Pizlo, E. Petrank, and B. Steensgaard, "A study of concurrent real-time garbage collectors," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 33–44, May 2008.

[12] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove, "Generational real-time garbage collection," in *Proc. Eur. Conf. Object-Oriented Program.* Cham, Switzerland: Springer, 2007, pp. 101–125.

[13] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Boca Raton, FL, USA: CRC Press, 2016.

[14] W. Zhao and S. M. Blackburn, "Deconstructing the garbage-first collector," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, Mar. 2020, pp. 15–29.

[15] W. Zhao, S. M. Blackburn, and K. S. McKinley, "Low-latency, high-throughput garbage collection," in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement.*, Jun. 2022, pp. 76–91.

[16] S. Tavakolisomeh, "Selecting a JVM garbage collector for big data and cloud services," in *Proc. 21st Int. Middleware Conf. Doctoral Symp.*, Dec. 2020, pp. 22–25.

[17] H. Grgic, B. Mihaljevic, and A. Radovan, "Comparison of garbage collectors in Java programming language," in *Proc. 41st Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2018, pp. 1539–1544.

[18] D. Beronic, N. Novosel, B. Mihaljevic, and A. Radovan, "Assessing contemporary automated memory management in Java—Garbage first, shenandoah, and Z garbage collectors comparison," in *Proc. 45th Jubilee Int. Conv. Inf., Commun. Electron. Technol. (MIPRO)*, May 2022, pp. 1495–1500.

[19] P. Pufek, H. Grgic, and B. Mihaljevic, "Analysis of garbage collection algorithms and memory management in Java," in *Proc. 42nd Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2019, pp. 1677–1682.

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

[21] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, Dec. 1960.

[22] (2019). *Concurrent Mark Sweep (CMS) Collector*. [Online]. Available: https://docs.oracle.com/en/java/javase/13/gctuning/concurrent-mark-sweep-cms-collector.html

[23] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 25–36, Jun. 2004.

[24] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, no. 6, pp. 419–429, Jun. 1983.

[25] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, "Barriers reconsidered, friendlier still!" *ACM SIGPLAN Notices*, vol. 47, no. 11, pp. 37–48, Jan. 2013.

[26] Oracle. (2020). *Hotspot Virtual Machine Garbage Collection Tuning Guide, JDK15*. [Online]. Available: https://docs.oracle.com/en/java/javase/15/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf

[27] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proc. Int. Symp. Memory Manage.*, Jun. 2011, pp. 1–11.

[28] M. Wu, Z. Zhao, Y. Yang, H. Li, H. Chen, B. Zang, H. Guan, S. Li, C. Lu, and T. Zhang, "Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 1–10.

[29] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–35, Jan. 2019.

[30] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 570–583, Jan. 2019.

[31] The Apache Software Foundation. (2018). *Apache SparkT—Unified Engine for Large-Scale Data Analytics*. [Online]. Available: https://spark.apache.org/

[32] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, "YAK: A high-performance big-data-friendly garbage collector," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement.*, 2016, pp. 349–365.

[33] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *Proc. 15th Workshop Hot Topics Operating Syst.*, 2015, pp. 1–5.

[34] R. Bruno, L. P. Oliveira, and P. Ferreira, "NG2C: Pretenuring garbage collection with dynamic generations for HotSpot big data applications," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage.*, Jun. 2017, pp. 2–13.

[35] R. Bruno and P. Ferreira, "POLM2: Automatic profiling for object lifetime-aware memory management for hotspot big data applications," in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, Dec. 2017, pp. 147–160.

[36] R. Bruno, D. Patricio, J. Simão, L. Veiga, and P. Ferreira, "Runtime object lifetime profiler for latency sensitive big data applications," in *Proc. 14th EuroSys Conf. 2019*, 2019, pp. 1–16.

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

[38] T. Printezis and D. Detlefs, "A generational mostly-concurrent garbage collector," in *Proc. 2nd Int. Symp. Memory Manage.*, Oct. 2000, pp. 143–154.

[39] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, "A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo scala, and SPECjvm2008," in *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2017, pp. 3–14.

[40] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, "Da capo con scala: Design and analysis of a scala benchmark suite for the Java virtual machine," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2011, pp. 657–676.

[41] Oracle. (2001). *Native Memory Tracking*. [Online]. Available: https://docs.oracle.com/en/java/javase/15/vm/native-memory-tracking.html

[42] B. Evans. (Mar. 2020). *What Tens of Millions of VMS Reveal About the State of Java*. [Online]. Available: https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/

[43] P. Liden. (2022). *The Z Garbage Collector*. [Online]. Available: https://wiki.openjdk.org/display/zgc/Main

[44] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proc. 6th Int. Symp. Memory Manag.*, 2007, pp. 91–102.

**SANAZ TAVAKOLISOMEH** received the B.S. and M.S. degrees in information technology in Iran. She is currently pursuing the Ph.D. degree with the University of Oslo.

During the master's studies, her research focused on network function virtualization (NFV), which involves virtualizing network services. Her work in this area has contributed to the development of new techniques for optimizing NFV. Following the master's studies, she continued to build her expertise through several years of experience in java application development. She is also continuing her research on the JVM. Specifically, she is investigating java virtual machine garbage collectors, which are essential components of the automatic manage memory.

**RODRIGO BRUNO** received the Ph.D. degree in CS from Técnico, University of Lisbon. He is currently an Assistant Professor with Técnico, University of Lisbon, and a Senior Researcher with INESC-ID, Lisbon. Before, he was a Senior Researcher with the Oracle Laboratories, Zurich (working on GraalVM project). He joined the Oracle Laboratories after spending two years as a Postdoctoral Researcher with the Systems Group, ETH Zurich. His current research interests include the intersection between systems and programming languages and optimizing language runtimes for cloud environments, such as microservices and serverless. Besides language runtimes, he works on operating systems and parallel and distributed systems.

**PAULO FERREIRA** (Senior Member, IEEE) received the B.Sc. and M.Sc. degrees in electrotechnical engineering from Instituto Superior Tcnico, University of Lisbon, in 1988 and 1992, respectively, and the Ph.D. degree in computer science from the University of Pierre et Marie Curie, in 1996. He is currently pursuing the Agrega degree with the University of Lisbon.

He collaborates with INESC ID, where he did research for several years with the Distributed Systems Group. He has published, as the author or coauthor, more than 120 articles in international journals and conferences, one scientific book, 11 book chapters, and a pedagogical book (with three editions, being one in Brazil), serves on various program committees (e.g., Middleware, ICDCS, and DAIS), served as an expert to the European union for assessment of projects proposals under the Seventh Framework Program, led more than 15 research projects, and was one of the founders and board officer of EuroSys (ACM—European Chapter of the Special Interest Group on Operating Systems) being currently a member of the Steering Committee.

Dr. Ferreira is also a member of the Steering Committee of the ACM/IFIP/Usenix Middleware and ACM/IFIP/Usenix Adaptive and Reflective Middleware (ARM). He is also a member of the editorial board of the *Journal of Internet Services and Applications* (Springer). He is a Senior Member of ACM, was awarded two best paper awards at international events (ACM/IFIP/Usenix Middleware Conference, level A in both CORE and RADIST rankings), One Recognition of Service Award from ACM, and his Ph.D. thesis was top ranked from Université Pierre et Marie Curie.