Selecting a GC for Java Applications [⋆]

Sanaz Tavakolisomeh[1], Rodrigo Bruno[2], and Paulo Ferreira[3]

[1] University of Oslo, Oslo, Norway sanazt@ifi.uio.no
[2] INESC-ID / Técnico, ULisboa Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt
[3] University of Oslo, Oslo, Norway paulofe@ifi.uio.no

Abstract. Nowadays, there are several Garbage Collector (GC) solutions that can be used in an application. Such GCs behave differently regarding several performance metrics, in particular throughput, pause time, and memory usage. Thus, choosing the correct GC is far from trivial due to the impact that different GCs have on several performance metrics. This problem is particularly evident in applications that process high volumes of data/transactions especially, potentially leading to missed Service Level Agreements (SLAs) or high cloud hosting costs.
In this paper, we present: i) thorough evaluation of several of the most widely known and available GCs for Java in OpenJDK HotSpot using different applications, and ii) a method to easily pick the best one. Choosing the best GC is done while taking into account the kind of application that is being considered (CPU or I/O intensive) and the performance metrics that one may want to consider: throughput, pause time, or memory usage.

Keywords: Garbage collector · Automatic memory management.

1 Introduction

In object-oriented programming languages, e. g. Java, automatic memory management regulates all the objects' allocations and deallocations in memory using Garbage Collection algorithms. A Garbage Collector (GC) in Java, which is responsible to free objects that are no longer referenced by any part of running applications and processes, is even more important when applications are dealing with high volumes of data (i.e., Big Data and/or Cloud Services [5]).

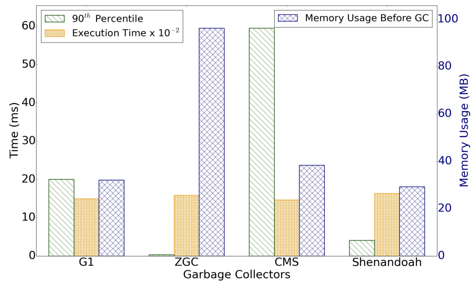

Fig. 1. Throughput, 90th percentile of pause times, and memory usage before GC, in Luindex benchmark.

Existing garbage collection solutions apply a trade-off between different performance metrics based on how the GC algorithm is designed. Taking Luindex

⋆ Rodrigo Bruno's research was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020.

benchmark (included in DaCapo benchmark suites [2]) as an example, 90th percentile of pause times along with execution time and memory usage for four different GCs is depicted in Figure 1. We can conclude that ZGC [28] has almost 12x and 57x less pause time compared to Shenandoah [8] and G1 [7] GCs, respectively. regarding pause time, CMS [12] would be the last choice when compared to the other GCs.

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

The existence of several GCs with diverse functionality and purposes makes it hard for a developer to choose the most suitable GC solution for a given application. This happens especially when the application deals with a large amount of data and demands a significant amount of resources like CPU or I/O. To select the most suitable GC, a user must know whether a given application is CPU-intensive or I/O-intensive; then, the user only requires to decide what GC performance metric (throughput, pause time, and memory usage) he wants to prioritize. In this paper, we evaluate how several widely known GCs behave regarding the above mentioned performance metrics, then we propose a methodology to choose the best GC for a given Java application running on the most widely used Java Virtual Machine (JVM) implementation, OpenJDK HotSpot JVM.

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

In this work, we use both Renaissance [21] and DaCapo [2] benchmark suites that represent a large set of applications. We also developed an application, called BufferBench (B2), that does read/write operations in memory to better study the GCs' behavior and the cost associated with the read and write barriers that some GCs use. Furthermore, we use Spring Boot based PetClinic application [16] to investigate our methodology using a real-world application. Using these applications, while applications are classified as being CPU-intensive or I/O-intensive, we investigate four well-known GCs available in the JVM. G1, which is the default GC since JDK 9, ZGC, Shenandoah, and CMS to evaluate their performance metrics. According to the results, we recommend the best GC solution for a specific application to fulfill its requirements: maximize throughput, minimize pause-time, or minimize memory usage.

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

2 Related Work

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

2.1 Background

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

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

There are several designs and implementations for GC solutions [11]. Some GCs partition the heap into multiple partitions or sub-heaps [26]. This is the case of generational GCs in which the partitioned heap holds objects that are segregated based on their lifetime. In this case, a newly created object is placed in the young generation partition; should it survive several GC cycles, it will be transferred to the old generation partition. In serial GCs [23] garbage collection is done serially using just one single thread in both generations, or multiple threads can be used by GC in the young generation (parallel GC) or in both young and old generations (Parallel Old GC) [23]. In the GCs to organize the free spaces in the heap, compaction and copying algorithms may be used. Through the compaction, GC moves all the live objects to the beginning of the memory segment, whilst through copying, a GC groups the live objects by moving them from multiple memory segments into a single one [3].

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

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

There are multiple GC algorithms available for the JVM. **Concurrent Mark/ Sweep collector (CMS)** [6] is a tracing collector (i. e. attempts to identify all the reachable objects in the heap by tracing the root objects) and generational GC including young and old generations. CMS employs an STW mark and copy technique in the young generation while there is a concurrent mark and sweep collector in the old generation to collect the marked unreachable objects. Also, It uses a write barrier that is run every time a reference in an object is updated. Since CMS does not have the compacting step; this may lead to heap fragmentation [27].

Garbage first (G1) [7] is the default GC since JDK9. G1 is a generational GC with fixed-size regions in the heap that uses a compacting algorithm. In STW young generation GC phase, G1 starts to traverse the object graph to find the live objects and copies them to the old regions. The old generation GC employs write barriers to mark live objects concurrently with the running application [29]. In fact, the main strategy in G1 is to reclaim regions with the least live objects, i.e., most garbage first (as its name suggests).

The main goal in **Shenandoah** [8] GC is having short pause times regardless of the heap size. It maintains the heap as a collection of regions and uses both concurrent copy and compaction techniques. To achieve object relocation concurrently with the application threads, Shenandoah uses a data structure that needs an additional field per object which points back to the object itself, when initializing the object and, with using the write barriers, to the object's new location once it is moved. It also uses read barriers to read objects from the exact current location in the heap memory.

ZGC [28] is a non-generational GC that divides the heap into regions of different sizes (small, medium, and huge) to hold the objects based on their size. To relocate the live objects concurrently, it uses read barriers and *colored pointers*, and stores some important metadata to hold information about an object itself and marking and relocation-related information [22]. ZGC applies STW pauses to mark live objects and mark the regions for compaction.

2.2 Existing Work

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

Ossia et el. [15] designed an incremental, and concurrent collector, with threads running in parallel, to ensure low pause time; Also, Pizlo et al. [17] propose a GC, STOPLESS, that uses a mark and sweep collector and a compactor to avoid fragmentation, for multi-processors running multithreaded applications. Pizlo et al. [18], also propose two other solutions for concurrent real-time GCs, CLOVER and CHICKEN, to improve the complexity of STOPLESS. Detlefs et el. [7] propose a GC that attempts to achieve high throughput for the applications running on multi-processor systems while they have a high allocation rate and need large heaps. Printezis et al. [19], propose a GC algorithm that works mostly concurrent with the application and attempts to reduce the worst-case pause time in generational memory systems.

Zhao et el. [29] decompose G1 into several key components and reimplement it. Also, they produce six collectors to evaluate different algorithmic elements of G1 using benchmarks including the DaCapo benchmark suite (also used by us as shown in Section 5). They developed a concurrent, region-based moving collector to find the shared elements in G1, ZGC, C4 [25], and Shenandoah. Pufek et al. [22] used selected benchmarks of the DaCapo suite and compared the results of G1, Parallel, Serial, and CMS GCs regarding the time spent in the garbage collection process as well as the number of collections in JDK versions 8, 11, and 12. Grgic et al. [10] analyzed the same GCs as Pufek's also using DaCapo benchmarks. They concluded that G1 performed better than CMS and Parallel GC regarding the total number of garbage collections and CPU utilization in multi-threaded applications. Prokopec et al. [20] describe Renaissance benchmark suite and explain the focus of each benchmarks. They also provide information about different metrics, including average CPU utilization, during a single steady-state execution of Renaissance and DaCapo benchmark suites.

Lengauer et al. [13] provides a description of commonly used benchmarks, including DaCapo, DaCapo Scala [24], and SPECjvm2008,[4] in terms of memory and garbage collection behavior. They compared G1 and Parallel Old (parallel old generation) regarding GC counts, the GC time relative to total run time, and the total pause time. They concluded that G1 performs better than the Parallel Old regarding pause time as it can select which regions to collect.

There are several works that studied GCs in Big Data systems. Bruno et al. [3] describes existing GC solutions for Big Data environments and the scalability of some memory management algorithms in terms of throughput and pause time. Nguyen et al. [14] propose a GC for Big Data systems with low pause time and high throughput To reduce the objects managed by the GC, the GC divides the heap into data space and control space and uses generation-based and region-based algorithms in the spaces respectively. However, the developer has to mark

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

the beginning and end points of the data path in the program. Broom [9] and NG2C [4] propose two GCs for Big Data systems. In Broom the regions in the heap are explicitly created by the programmer; in NG2C, the programmer identifies the generation where a new object should be allocated

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

3 Architecture

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

3.1 User Level Overview

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

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

Then, based on the GC metrics that the user wants to make better, the best GC regarding throughput is CMS in both CPU and I/O-intensive applications. Also, ZGC minimizes the pause time in almost all the CPU and I/O-intensive applications; and CMS has acceptable results regarding using memory before garbage collection, especially in CPU-intensive applications, while CMS can be replaced with Shenandoah in I/O-intensive applications; G1 is a good choice in reducing memory usage after GC in CPU-intensive applications, CMS works as well as G1 regarding this metric in I/O-intensive applications (see Section 5).

3.2 Application Classification

In this section, we focus on classifying applications based on their CPU and I/O usage. As described in Section 4, all the process mentioned above is done with G1 activated as a default GC in the OpenJDK and by employing representative applications available in well-known benchmark suites: Renaissance [21] and DaCapo [2]. These two Java-based benchmark suites allow us to disable/enable multiple features (e.g., disabling garbage collection before each iteration in benchmarks, or change input parameters like input size), and are representative of most existing applications.

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

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

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

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

a) Renaissance benchmark suite

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

b) DaCapo benchmark suite

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

c) B2

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

d) PetClinic Application

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

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

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

3.3 Main Aspects

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

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

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

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

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

4 Implementation

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

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

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

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

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

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

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

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

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

5 Evaluation

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

5.1 Methodology

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

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

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

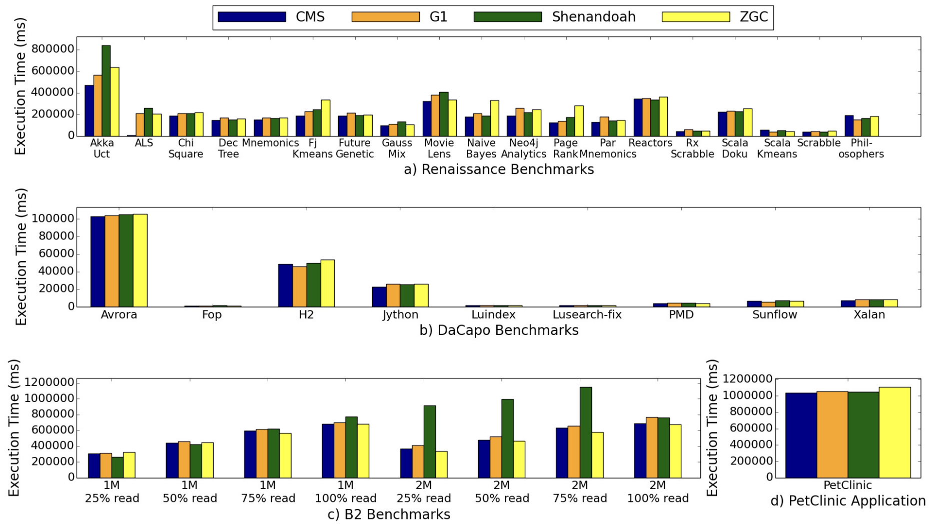

Fig. 2. Average execution time in applications using different GCs.

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

Moreover, to evaluate the GC performance metrics with a real-world application we examine the GCs using the PetClinic Spring Boot application. First, we investigate if the application is CPU or I/O intensive, then we evaluate for each GC, the throughput, pause time, and memory usage. To test the PetClinic application we use Apache JMeter [1] (version 5.4). Using JMeter, the APIs in the Petclinic application were called using 100 threads and data files containing records as input values for different entities in the application. We also set the number of iterations to 10 for each thread group. Choosing these numbers after several evaluations, revealed a typical usage of the application, reduced the errors resulting from using simultaneous threads, triggered several GC cycles and used resources significantly in the server.

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

5.2 Results

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

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

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

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

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

Table 2. 90th Percentile of pause times (ms).

a) Renaissance benchmark suite

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

b) DaCapo benchmark suite

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

c) B2

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

d) PetClinic Application

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

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

Pause Time Table 2 shows the 90th percentile of all the pause times obtained from all the executions for each application (because of the very small values obtained using ZGC and a huge difference between the CMS and ZGC results, we preferred to show the results in a table). ZGC has a pause time of less than 1 millisecond in most of the applications, and has the minimum pause time in comparison with other GCs. Although Shenandoah has on average a pause time about 6x more than ZGC, it performs significantly better than CMS and G1. Based on the values in Tables 2.a, 2.b , and 2.c, CMS is not able to manage the pause time as well as other GCs. In addition, in both DaCapo benchmarks and B2, CMS has a pause time of about 400x more than ZGC; in Renaissance benchmarks, CMS has a pause time around 100x more than ZGC on average. Also, Table 2.d shows that G1 performed slightly better than other three GCs regarding pause time in PetClinic.

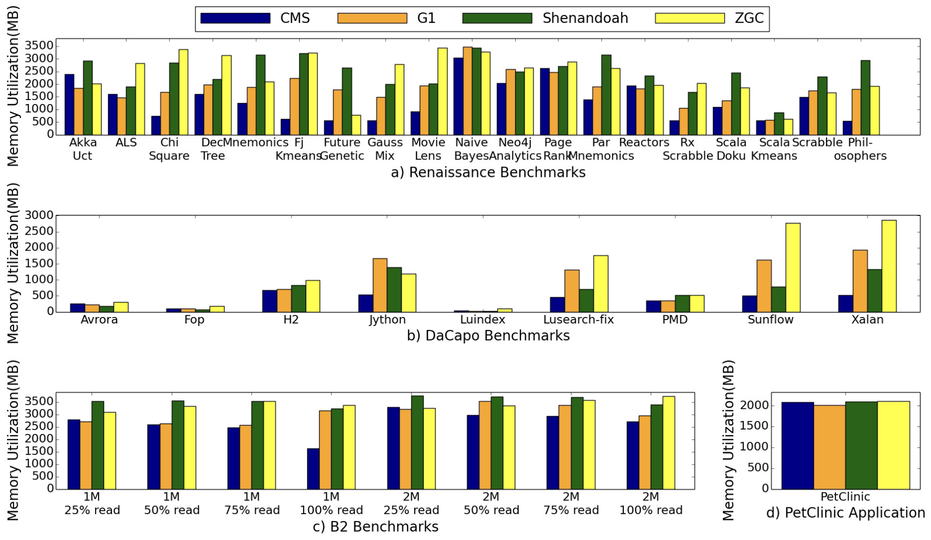

Fig. 3. Average memory usage before garbage collection.

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

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

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

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

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

In B2, memory reduced not more than 65% after garbage collection by the GCs (since most of the objects, held in a hashmap data structure, are still alive). However, ZGC is the preferred solution in almost 90% of experiments in

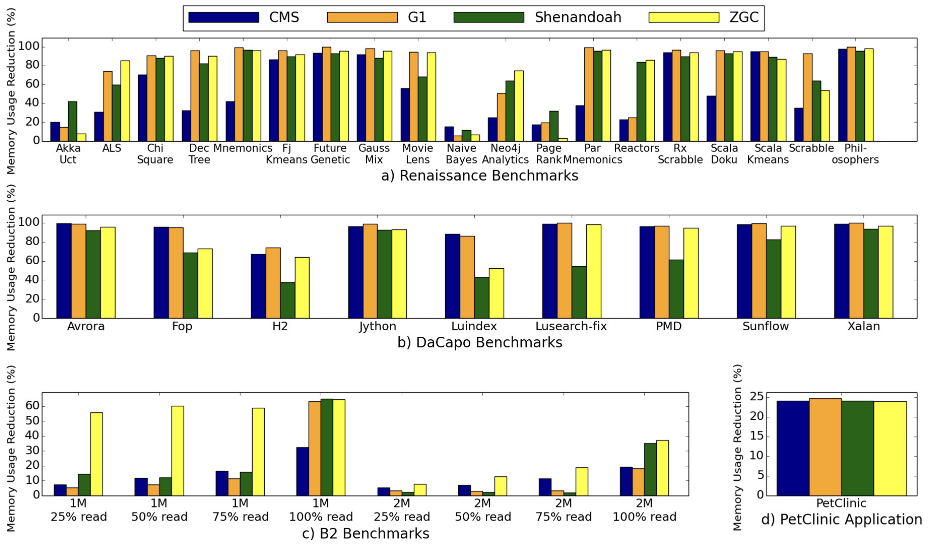

Fig. 4. Memory usage reduction (%) after GC.

B2. Finally, in the PetClinic application (Figure 4.d), all the GCs reduced the memory usage by around 24%. However, G1 is the GC that performed better.

5.3 Discussion

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

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

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

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

Regarding the heap usage reduction after garbage collection, in the CPU-intensive category, G1 is the leading GC in both DaCapo and Renaissance benchmark suites. However, ZGC is the GC that reduces most the amount of heap usage in B2. Also, as shown in Figure 4, for the PetClinic application, G1 reduced the heap utilization the best. According to the results, G1 is the best solution regarding average memory reduction for about 57% of applications in CPU-intensive category. In the I/O-intensive category, G1 and CMS could reduce the heap usage in around 43% of benchmarks. The results match one of the main design principles in G1. G1 attempts to reclaim the heap spaces with the most garbage [7].

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

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

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

6 Conclusion

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

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

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

References

1. Apache jmeter, https://jmeter.apache.org/
2. Blackburn, S.M., et al.: The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 169–190 (2006)
3. Bruno, R., Ferreira, P.: A study on garbage collection algorithms for big data environments. ACM Computing Surveys (CSUR) **51**(1), 1–35 (2018)
4. Bruno, R., Oliveira, L.P., Ferreira, P.: Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management. pp. 2–13 (2017)
5. Chen, M., Mao, S., Liu, Y.: Big data: A survey. Mobile networks and applications **19**(2), 171–209 (2014)
6. Concurrent mark sweep (cms) collector, https://docs.oracle.com/en/java/javase/11/gctuning/concurrent-mark-sweep-cms-collector.html
7. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of the 4th international symposium on Memory management. pp. 37–48 (2004)
8. Flood, C.H., Kennke, R., Dinn, A., Haley, A., Westrelin, R.: Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In: Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. pp. 1–9. ACM, New York, NY, USA (2016)
9. Gog, I., Giceva, J., et al.: Broom: Sweeping out garbage collection from big data systems. In: 15th Workshop on Hot Topics in Operating Systems (HotOS {XV}) (2015)
10. Grgic, H., Mihaljević, B., Radovan, A.: Comparison of garbage collectors in java programming language. In: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). pp. 1539–1544 (2018)
11. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 1st edn. (2011)

12. Jones, R.E.: Garbage collection: algorithms for automatic dynamic memory management. John Wiley and Sons (1996)
13. Lengauer, P., Bitto, V., Mössenböck, H., Weninger, M.: A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. pp. 3–14 (2017)
14. Nguyen, K., Fang, L., Xu, G., Demsky, B., Lu, S., Alamian, S., Mutlu, O.: Yak: A high-performance big-data-friendly garbage collector. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation. pp. 349–365 (2016)
15. Ossia, Y., Ben-Yitzhak, O., Goft, I., Kolodner, E.K., Leikehman, V., Owshanko, A.: A parallel, incremental and concurrent gc for servers. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 129–140 (2002)
16. Petclinik application (2007), https://projects.spring.io/spring-petclinic/
17. Pizlo, F., Frampton, D., Petrank, E., Steensgaard, B.: Stopless: A real-time garbage collector for multiprocessors. In: Proceedings of the 6th International Symposium on Memory Management. p. 159–172 (2007)
18. Pizlo, F., Petrank, E., Steensgaard, B.: A study of concurrent real-time garbage collectors. ACM SIGPLAN Notices **43**(6), 33–44 (2008)
19. Printezis, T., Detlefs, D.: A generational mostly-concurrent garbage collector. In: Proceedings of the 2nd international symposium on Memory management. pp. 143–154 (2000)
20. Prokopec, A., et al.: On evaluating the renaissance benchmarking suite: Variety, performance, and complexity. arXiv preprint arXiv:1903.10267 (2019)
21. Prokopec, A., et al.: Renaissance: benchmarking suite for parallel applications on the jvm. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 31–47. ACM (2019)
22. Pufek, P., Grgić, H., Mihaljević, B.: Analysis of garbage collection algorithms and memory management in java. In: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). pp. 1677–1682 (2019)
23. Java garbage collection basics, https://www.oracle.com/webfolder/technetwork /tutorials/obe/java/gc01/index.html
24. Sewe, A., Mezini, M., Sarimbekov, A., Binder, W.: Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In: Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 657–676. OOPSLA '11, ACM, New York, NY, USA (2011)
25. Tene, G., Iyengar, B., Wolf, M.: C4: The continuously concurrent compacting collector. In: Proceedings of the international symposium on Memory management. pp. 79–88 (2011)
26. Ungar, D., Jackson, F.: Tenuring policies for generation-based storage reclamation. ACM SIGPLAN Notices **23**(11), 1–17 (1988)
27. Xu, L., Guo, T., Dou, W., Wang, W., Wei, J.: An experimental evaluation of garbage collectors on big data applications. Proc. VLDB Endow. **12**(5), 570–583 (2019)
28. Zgc: A scalable low-latency garbage collector (2018), https://openjdk.java.net/jeps/333
29. Zhao, W., Blackburn, S.M.: Deconstructing the garbage-first collector. In: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 15–29 (2020)