UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Gabriel Alves Tavares

A NEW TAKE OF JAVA 11 GC PERFORMANCE: THE HEAPOTHESYS CASE

CAMPINA GRANDE - PB

2022

Gabriel Alves Tavares

A NEW TAKE OF JAVA 11 GC PERFORMANCE: THE HEAPOTHESYS CASE

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Thiago Emmanuel Pereira da Cunha Silva

CAMPINA GRANDE - PB

2022

Gabriel Alves Tavares

A NEW TAKE OF JAVA 11 GC PERFORMANCE: THE HEAPOTHESYS CASE

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em: 06 de abril de 2022.

CAMPINA GRANDE - PB

RESUMO (ABSTRACT)

Os *benchmarks* são essenciais à investigação científica, uma vez que proporcionam uma forma fiável de comparar abordagens inovadoras com o padrão académico. Especificamente, *benchmarks* são amplamente utilizados em Java para avaliar novas versões da JVM e dos Coletores de Lixo (CL). À medida que novas cargas de teste e CLs chegam à indústria, é fundamental expandir a nossa compreensão da gestão dinâmica de memória, estudando como funcionam essas novas estratégias. Este trabalho estuda o desempenho dos coletores de lixo modernos e estabelecidos na indústria utilizando HyperAlloc, uma carga de trabalho do Heapothesys Benchmark da Amazon que prevê com precisão o comportamento de alocação de memória e facilita as comparações entre algoritmos de CL. A análise fornecida neste documento serve como guia sobre a adequação da Heapothesys para avaliar os CLs modernos e fornece informações sobre os seus trade-offs de desempenho.

A new take of Java 11 GC performance: the Heapothesys case

Gabriel Alves Tavares*
gabriel.tavares@ccc.ufcg.edu.br
Universidade Federal de Campina Grande (UFCG)
Campina Grande, Paraíba, BR

Thiago Emmanuel Pereira
temmanuel@computacao.ufcg.edu.br
Universidade Federal de Campina Grande (UFCG)
Campina Grande, Paraíba, BR

Daniel Lacet de Faria Fireman
daniel.fireman@ifal.edu.br
Instituto Federal de Alagoas (IFAL)
Arapiraca, Alagoas, BR

ABSTRACT

Benchmarks are essential to scientific research as they provide a reliable way of comparing novel approaches with the academic standard. Specifically, benchmarks are widely used in Java to evaluate new JVM versions and Garbage Collectors (GC). As new benchmark suites and collectors arrive in the industry, it is fundamental to expand our comprehension of memory management by understanding how those novel strategies work. This work studies the performance of modern garbage collectors established in the industry by using HyperAlloc, a workload of Amazon's Heapothesys Benchmark suite that precisely predicts memory allocation behavior and facilitates comparisons between GC algorithms. The analysis provided in this paper serves as a guide on how suitable Heapothesys is to evaluate modern collectors and provides insights on their performance trade-offs.

KEYWORDS

Heapothesys, benchmarks, garbage collector, Java

ACM Reference Format:
Gabriel Alves Tavares, Thiago Emmanuel Pereira, and Daniel Lacet de Faria Fireman. 2018. A new take of Java 11 GC performance: the Heapothesys case. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Benchmarks are a reliable and efficient way to compare new systems implementations to the industry's established standards. Systems and optimization techniques can be evaluated using benchmarks, such as compiler optimizations, garbage collectors (GC) algorithms, search engines, and virtual machines. Then, it is standard to compare the benchmarks results between different options and choose the best for a specific service. Choosing the best version, framework, or garbage collector for a service can have huge implications on resource utilization, thus impacting the cost of such service.

There are significant efforts in evaluating the suitability of benchmarks. The ideal scenario for the industry is to map real-world applications into representative benchmarks, making it easy to choose a benchmark that suits one's needs. Consequently, we need to characterize the workloads in benchmarks suites available. For Java, established suites such as DaCapo (BLACKBURN et al, 2006, [1]), DaCapo Scala (SEWE et al, 2011, [2]), and SPECjvm2008 (SHIV et al, 2009, [3]) have been submitted to a comprehensive characterization in the literature (LENGAUER et al, 2017 [4]). Nonetheless, since the publication of these studies, there have been new benchmarks, Java versions, and GCs, which now lack documented comparisons.

In this work, we extend the literature on Java benchmarking by adding a comparative analysis with established GCs on the market while using new assets like Amazon Corretto, a workload from the Heapothesys benchmark suite, and Shenandoah GC. Amazon Corretto is an OpenJDK distribution of Java supported by Amazon. This JDK distribution is a promising alternative for Amazon Web Service users because of the long-term commitment of the Corretto team to make it a default for their clients. The same team announced in 2020 the Heapothesys benchmark suite, a set of workloads that aims to test fundamental application characteristics that affect garbage collectors. Finally, Shenandoah is a concurrent compacting garbage collector that promises consistently low pauses times and overall improved performance over the default GCs currently on Java.

Our goal is to provide insights into the behavior of the GCs analyzed in light of the metrics those GCs try to improve. We set the scope of this work not to comprehend all possible scenarios enabled by Heapothesys, as there are multiple workloads with various parameters to configure. In the end, we analyzed the HyperAlloc workload, a benchmark that dynamically tunes allocation behavior to simulate fundamental application characteristics.

The paper is structured as follows: Section 2 introduces key concepts on garbage collection in Java and the benchmarks used. Next, Section 3 details our methodology and experimental tooling. Section 4 discusses all results, while Section 5 presents previous work on java memory management. Finally, Section 6 exposes our conclusions.

2 CONTEXT

2.1 Garbage collection in Java

Garbage Collectors (GC) are responsible for dynamic memory management in modern programming languages. They are a fundamental tool of languages with managed environments such as Python, Java, and Rust. In general, GC algorithms execute three fundamental operations:

- `Mark`: determines the reachability of all objects in the JVM heap;
- `Sweep`: removes unreachable objects detected in the previous phase;

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

- `Compress`: optimizes the heap space usage by compressing the remaining objects.

Although all GCs can be summarized in these operations, each GC algorithm implements a different strategy for memory management. One of the significant decisions one needs to make when implementing a GC is how to use generations. Heap generations are partitions that separate old and newly created objects. The efficiency of this method is based on the observation that most objects are short-lived, called the generational theory. Heap generations need data structures and operations for moving objects between them, potentially creating an overhead on the pause times. For that reason, each GC algorithm chooses different numbers and sizes for their generations, and some ignore this model entirely. Shenandoah, for example, is a non-generational GC that aims to collect regions of the heap with the most garbage, whether they are old or young objects (FLOOD el at, 2016, [5]). To analyze the effects of the use of generations, we will compare both generational (G1 and Parallel) and non-generational (Shenandoah) GCs.

Another important aspect of garbage collections is choosing which events stop application threads and balancing the maximum and the total pause times. The maximum pause time is associated with latency because it can be translated as the maximum time an application will be unresponsive. While in this unresponsive state, the application can not process requests arrived, thus impacting latency. On the other hand, the total pause time accumulates all GC pauses' duration. The higher the total pause time, the worst is the application throughput because the application spends less time processing valuable work and more time blocked by GC threads.

Usually, GCs aim to optimize a specific metric such as throughput, latency, and responsiveness by controlling average, maximum, and total pause time. Depending on the chosen metric, there are different ways of executing the operations of marking, sweeping, and compressing the heap. One approach is to execute a Stop The World (STW) pause, block all application threads, and use the machine resources exclusively to speed up those operations. Another strategy is to execute some of those operations concurrently with the application. Furthermore, it is possible to mix STW pauses with the concurrent approach. Each design choice has trade-offs: prioritizing throughput may impact latency, or a particular generation layout may lead to caveats with big objects.

As an example, we can briefly analyze Parallel's strategy. ParallelGC parallelizes the collection on the Young region, where newly created objects are stored, and stops all application threads when doing it so. This decision leads to more frequent but shorter STW pauses. In an interactive microservice running Parallel, requests that hit the application when it was executing one of these frequent STW pauses will be greatly impacted in latency. However, the shorter pause times cause the total pause time to be lower than most GCs. In summary, the application spends a significant percentage of the time doing valuable work (i.e., greater throughput), but this strategy increases the variance of the requests' latency.

2.2 Amazon's Heapothesys Benchmark

Amazon's Heapothesys is a benchmark suite for the JVM. The Amazon Correto team designed it to test GCs, valuate new garbage collection strategies, and detect peculiar behavior in latency and other performance metrics. The Heapothesys suite has two workloads that serve different purposes: Extremem and HyperAlloc. The first mimics production-like behavior mixed with processes of allocation and deallocation of heap space, while the second focuses on the intricacies of allocation rate and heap occupancy. We focused on the latter because it can accurately predict the allocation behavior, which enables comparisons of the predicted and observed performances of the GCs.

Understanding the fundamental concepts behind the HyperAlloc workload is essential to evaluating how the GCs performed on this benchmark. Corretto's team modeled HyperAlloc as a synthetic workload focused on simulating rigid constraints of heap occupancy and allocation rate. Those two constraints, or factors, are directly responsible for collector stress. HyperAlloc tries to accurately predict allocation behavior based on parameters that can be tuned by the user, if necessary. That leads to a benchmark where we can interpret the allocation rate as a factor, i.e., the predicted allocation rate, and as a metric, i.e., the observed allocation rate. GCs handle a predicted allocation rate differently, some performing better than others, as we will discuss later in the Results section.

3 METHODOLOGY

3.1 Experimental Design

This study aims to answer the following question: how do modern Java GCs perform under the Heapothesys' HyperAlloc workload? To answer this question, we conducted benchmark experiments to evaluate a set of modern GCs. Another goal of this study is to create insights into the HyperAlloc behavior to assist future users in tuning the available benchmarks and which metrics to evaluate.

Our experimental design consists of running the selected benchmark for three different GCs. We ran three repetitions for each GC to increase confidence in our results. Our key metrics are related to object allocation behavior and GC pauses, both of which we dive deeper into in the Results section.

The chosen GCs were Parallel, Shenandoah, and G1. This set of collectors provides a variety of memory management strategies to evaluate. Parallel is known as the throughput collector, i.e., it aims to maximize the percentage of time running application threads instead of garbage collection threads. Shenandoah has the unique feature of running concurrent compacting of the heap to achieve the lowest maximum pause time. Finally, Garbage First (G1) is the default GC for Java 11, making it a great base model to compare the other two.

3.2 Experimental Setup

In all experiments, we used Amazon EC2 virtual machines running on Amazon Linux 2. We chose the t2.large flavor (2vCPUs, 8 GB of RAM, general-purpose SSD). We initialized a new virtual machine for each repetition to guarantee that previous runs do not impact the following ones. It is well-known in the literature that JVM warm-up effects can harm experiment validity, so, before each repetition, we executed a warm-up run and discarded its results (BLACKBURN et al, 2008 [6]). Each repetition lasts 600 seconds, as configured by a HyperAlloc parameter (-d 600). As for the JVM flags, we used a 512MB heap size (-Xms512m -Xmx512m) and additional flags for GC logging.

We had difficulty finding the definitive Java version for this study because different versions can drastically change how to analyze logs or configure the benchmarks. For instance, Shenandoah's logs in Java 8 do not follow the unified GC logging system, but in Java 11 or greater, it does. Using Java 8 would allow us to directly compare previous works like (LENGAUER et al, 2017 [4]). However, we prioritized the unified GC logging system to guarantee a fair comparison between GCs (especially Shenandoah) and a more straightforward analysis pipeline; thus, we used Java 11. Specifically, we used Amazon Correto 11, a distribution of the OpenJDK 11 maintained by Amazon, which guarantees compatibility with the Heapothesys suite and solves our logging problems.

3.3 Tooling

We built a benchmark coordinator to run the experiments. This coordinator is composed of Terraform scripts that automate the infrastructure and Ansible scripts that set up the machines and execute the benchmarks. When the repetition ends, the coordinator transfers the benchmarks' output and GC logs from the VMs to the machine that triggered the experiment.

Regarding the infrastructure automation, the user configures the number of machines and the EC2 machine configuration (i.e., subnet, tags, flavor), then runs the "terraform apply" command. These options allowed us to run benchmarks simultaneously in isolated VMs, saving time when executing repetitions. Destroying the whole infrastructure is as simple as running the "terraform destroy" command.

The machine coordination can be divided into four phases: update dependencies, build benchmarks, execute benchmarks, and backup results. We used Ansible to execute the whole pipeline of commands because it allows us to write a playbook of pre-defined actions to configure all virtual machines simultaneously. The user can simply execute an "ansible-playbook" command to run the entire experiment with multiple repetitions.

First, the virtual machines need to have all Heapothesys and GC dependencies. The most important dependencies are JDK tools (i.e., jar, java, javac), which are not simple to configure. We needed to handle conflicts between java versions installed by other dependencies (e.g., maven) and the specific versions we wanted to use. There is plenty of room for improvement in usability, but our tooling solves the dependencies problems for multiple benchmarks executions.

The last three phases of machine coordination are straightforward. The build phase compiles the benchmarks set to run. Next, the execute phase runs the benchmark in the order set in the Ansible configuration file. Finally, the backup phase saves all results data in a local directory of the machine running the coordinator. These features help test new configurations and organize data of multiple experiments.

4 RESULTS

The following sections describe the behavior of the garbage collection and heap usage for the experiment using three GCs (Shenandoah, G1GC, and ParallelGC) and Heapothesys' HyperAlloc workload. We describe our insights on the GC behavior in the light of the benchmark's predefined goals (e.g., Parallel aims to improve throughput, and Shenandoah tries to minimize GC pause time).

4.1 Allocation Behavior

Heapothesys' HyperAlloc benchmark, as the name suggests, focuses on achieving a specific object allocation rate. Because of that, we can trust that the benchmark will pressure the JVM in a very similar fashion across all GCs tested. Then we can compare results in order to get insights into GC behavior.

We used the average creation rate of objects and the total MBs allocated to analyze how the benchmark impacted heap usage for each GC. As discussed in Section 2, we differentiate the observed and the expected allocation rates. The expected is set and maintained by the benchmark, and the value for all experiments is the workload's default: 1024 MBs/sec. The observed allocation rate is presented as the average creation rate, shown in 1. The goal of HyperAlloc is to dynamically change its parameters to sustain a high allocation rate based on the number of CPU cores and heap size. For that reason, the higher the observed allocation rate and the total allocated memory, the better.

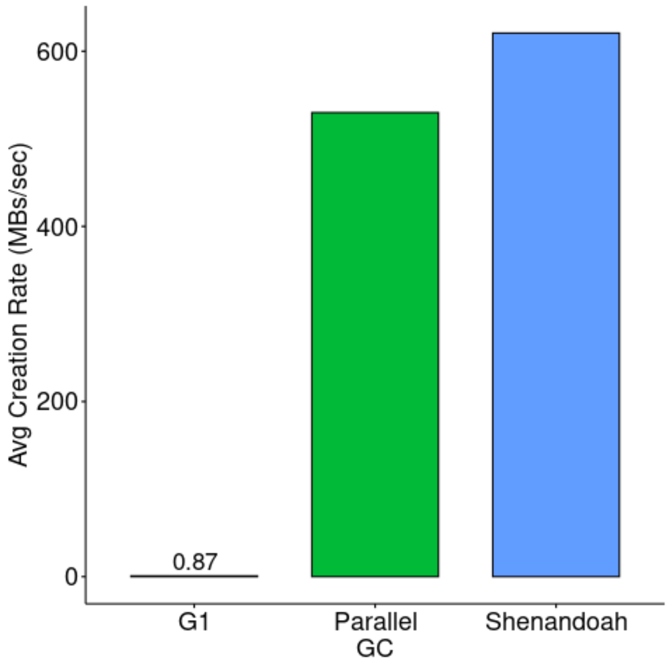

Figure 1: The average creation rate of objects on the JVM heap in MBs per second. Smaller values are explicitly shown to facilitate comparison.

Knowing how to interpret those metrics, our experiments show that G1 has problems sustaining a high creation rate. 1 shows that G1 allocates less than 1Mb/sec. 2 also shows that G1 allocated 300x less in total than the other alternatives. Parallel and Shenandoah allocated over half a GB per second and more the 300 GB in total. Shenandoah is known for working as intended regardless of heap size, but Parallel also allocated more than 500 MBs/sec even with the small heap size.

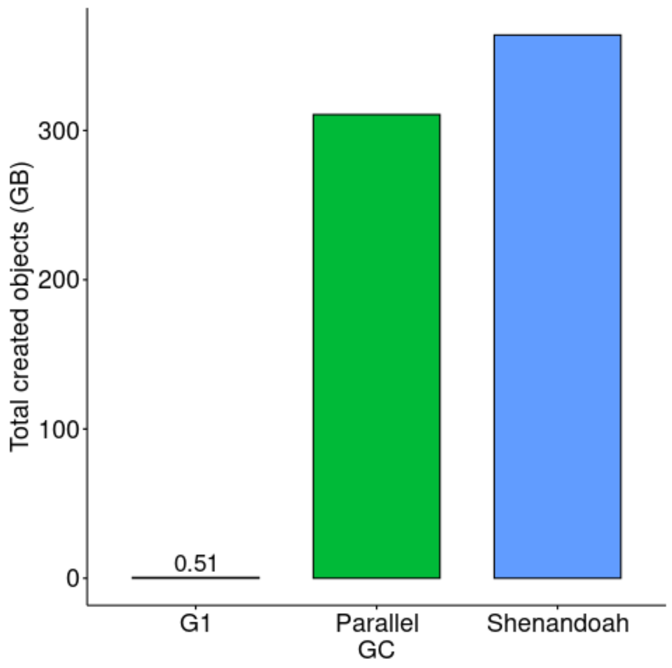

Figure 2: The total amount of allocated heap in GB. G1 results are explicitly shown to facilitate comparison.

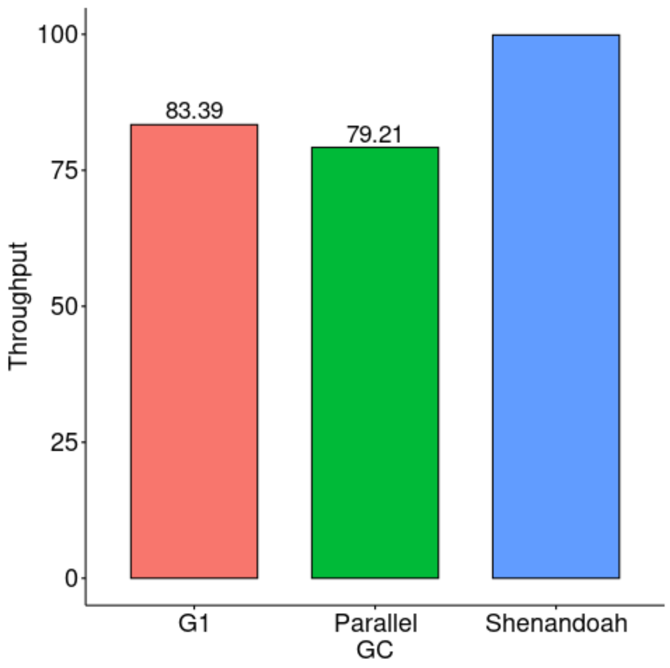

Figure 3: Application throughput relative to GC pauses, i.e., percentage of time the application spent processing actual transactions versus time spent in GC activity.

One possibility for the G1 behavior is that the HyperAlloc benchmark, with the default configuration, caused a common G1 caveat known as allocating Humongous Objects. This problem occurs when the application creates numerous objects with half (or more) the GC region size. These Humongous Objects are allocated on the Old Generation and, due to their size, leave a free and unused space in the region they are allocated. This will increase the need for costly defragmentations that, while executing, stop application threads. HyperAlloc default object size has a random size between 128 bytes and 1 KB. The default G1 region size for a heap of 512MB is 1MB. We should not have problems with Humongous objects by these numbers, so we are left with two options: a new caveat for G1 or Hyperalloc dynamic object creation rate is unpredictable. One way to investigate this in the future is to use an internal agent on the JVM, like AnTracks, that monitors every object allocated and enables an in-depth analysis.

4.2 Garbage Collection

The most analyzed metrics in JVM benchmarks are related to GC pauses. These pauses can be responsible for significantly degrading response time because they force the application into an unresponsive state. We measure this unresponsiveness with the percentage of time the application threads were executing freely instead of running a collection, i.e., the application throughput. On the other hand, the average pause time and the total number of pauses (GC count) help measure a GC's impact on response time. High pause times and frequent collections increase the chances of impacting the application processing.

HyperAlloc throughput differs for each GC. Surprisingly, Parallel, promoted as the throughput collector, presented the worst results. G1 is 4% better than Parallel, and Shenandoah reaches nearly 100% throughput. Parallel's algorithm uses stop-the-world pauses and concurrent threads to accelerate the GC collection and increase throughput. However, that strategy only works if pause times or the number of pauses is low, which was not the case for the Parallel in our experiments, as shown in 4 and 5. The throughput of G1, although slightly higher than Parallel's, is still tiny considering that G1 allocated orders of magnitude fewer objects than its competitors.

Parallel also presented the worst pause times, reaching 2x the average for G1 and 10x for Shenandoah (4). This result alone is not alarming because pause time is not a priority for Parallel; instead, the throughput is. However, to compensate for a high average pause time, it is necessary to lower the number of collections, which was not low enough for Parallel to perform as well as the others (5).

G1 kept the average pause time close to 10ms thanks to its intelligent choosing of heap regions to evacuate. However, the GC count for this collector was the worst, which explains why this G1 is only slightly better than Parallel in terms of throughput. The results on allocation and GC pauses for G1 indicate that this GC needs application-specific tunning or a bigger heap size to perform appropriately.

Meanwhile, Shenandoah had the best performance in all metrics. Its strategy of minimizing STW pauses and doing most of the work concurrently with the application proved to be the best for our scenario. The results show that Shenandoah can withstand small heap sizes and great allocations rates while maintaining high throughput

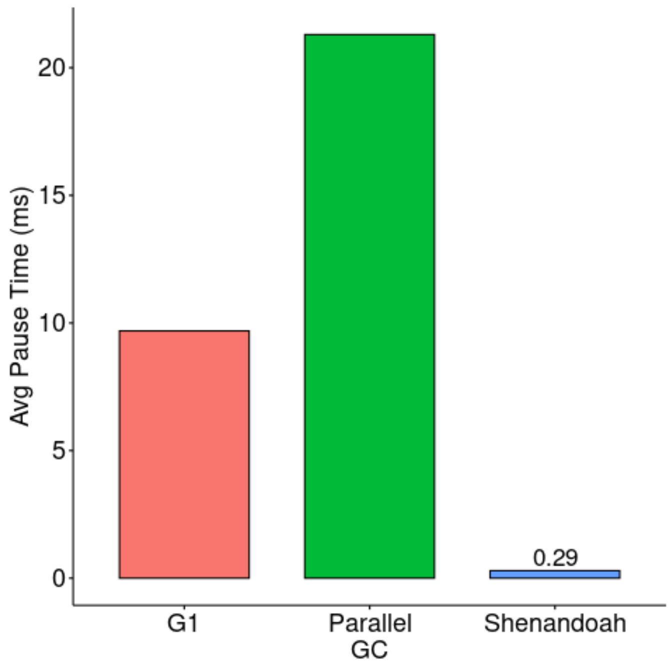

Figure 4: Average GC pause time for each benchmark and GC.

and low pause times. We can safely say that this GC is an excellent out-of-the-box, i.e., with little to no configuration, solution for applications with strict constraints on the metrics analyzed.

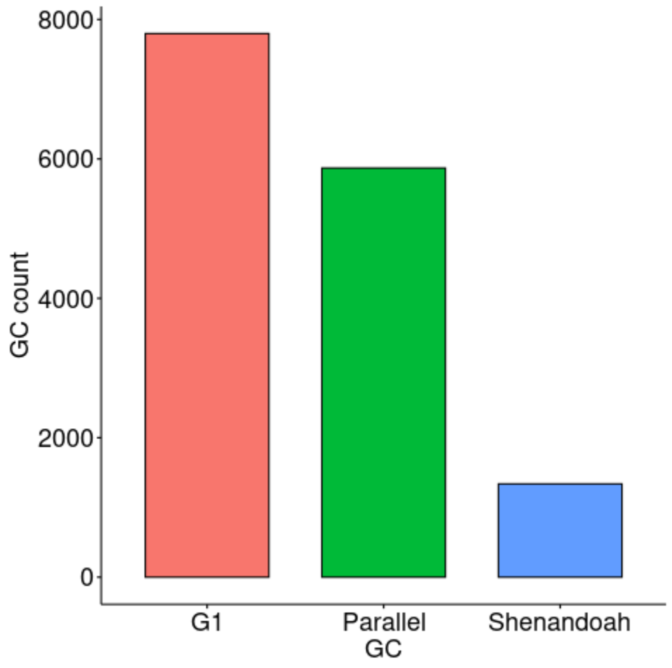

Figure 5: The number of GC collections.

5 RELATED WORK

In 2017, LENGAUER et al [4], published "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008". This work provides a detailed view on how to configure benchmarks (e.g., choosing warm-up duration and heap size) and documents the effects of those workloads on heap memory usage. Furthermore, the analysis explains the performance differences between G1 and Parallel GCs. Lengauer et al. was a major inspiration for our work, as we executed similar experiments and analysis. However, we made new and promising additions: Shenandoah GC and Heapothesys benchmarks. Another critical difference between our works is the use of Anttracks for memory and GC monitoring. This tool, or one with similar capabilities, is highly recommended for future works because it enables collecting precise data on every allocated object, thus helping diagnose GC problems.

Another work that evaluates GC performance is "Selecting a GC for Java Applications" (TAVAKOLISOMEH et al, 2021, [7]). In this study, the authors classified applications in two types, CPU-intensive or I/O-intensive, and for each type, they discovered the best GC for metrics such as heap usage, throughput, and pause time. The comparisons in Tavakolisomeh's work helped us understand trade-offs between GC algorithms and possible causes for underperforming. Although the authors included Shenandoah in their experiments, they did not used the Heapothesys benchmark suite.

6 CONCLUSION

In this paper, we analyzed the behavior of G1, Parallel, and Shenandoah GCs when submitted to HyperAlloc, a workload from the Heapothesys benchmark suite. Shenandoah was the leading GC in all metrics explored. Its out-of-the-box configuration can handle HyperAlloc's allocation-heavy default configuration while maintaining the expected pauses times and throughput. Both Parallel and G1 underperformed on our tests, as they did not manage to accomplish what their algorithms proposed. Parallel presented the worst throughput of all tested GCs, and G1 had difficulty reaching the expected allocation rate. Our results serve as a foundation for future studies that may dive deeper into the configurations of Heapothesys benchmarks and optimization of modern GCs.

ACKNOWLEDGMENTS

We thank our institution, the Universidade Federal de Campina Grande (UFCG), and its Computer Science undergraduate course for its support. We also thank Daniel Fireman and Thiago Emmanuel for their invaluable guidance. Finally, we thank the industry-research partnership between VTEX and UFCG, supported through the Embrapii and VTEX Lab program, for providing opportunities to students to grow as engineers and scientists.

REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, oct 2006.

[2] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. *SIGPLAN Not.*, 46(10):657–676, oct 2011.

[3] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. Specjvm2008 performance characterization. In David Kaeli and Kai Sachs, editors, *Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. page 3–14, 2017.

[5] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. 2016.

[6] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, aug 2008.

[7] Sanaz Tavakolisomeh, Rodrigo Bruno, and Paulo Ferreira. Selecting a gc for java applications. In *Norsk IKT-konferanse for forskning og utdanning*, number 1, pages 2–15, 2021.