Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology

Programming Language and Systems
CSE 6305

Report on

A Comparison of Modern JVM Based Garbage Collectors: In Big Data Benchmarks

Prepared By:

Zahin Wahab, 0421052013
Bishal Basak Papan, 0421052033

Supervised By:

Dr. Rifat Shahriyar

November 20, 2022

1 Introduction

In the Java programming language the automated memory management process known as garbage collection is a critical component of the Java Runtime Environment (JRE) and Java Virtual Machine (JVM). Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory. The Garbage Collector (GC) mechanism has considerable influence over the overall performance and efficiency of a running application [1] [2]. Today there exists a great need for continuous and uninterrupted services in modern software applications. It is therefore valuable to analyze how different garbage collectors are performing when managing memory and how they impact the application execution.

The biggest benefit of Java garbage collection is that it automatically handles the deletion of unused objects or objects that are out of reach to free up vital memory resources. Programmers working in languages without garbage collection (like C and C++) must implement manual memory management in their code. Despite the extra work required, some programmers argue in favor of manual memory management over garbage collection, primarily for reasons of control and performance. While the debate over memory management approaches continues to rage on, garbage collection is now a standard component of many popular programming languages. For scenarios in which the garbage collector is negatively impacting performance, Java offers many options for tuning the garbage collector to improve its efficiency.

With the release of Java 13, multiple garbage collectors are offered by the JVM. Serial GC, Parallel GC, Concurrent Mark Sweep GC, Epsilon GC, Garbage First GC (G1GC), Shenandoah GC [3], and Z Garbage Collector (ZGC) [4]. G1GC is the default collector of Java 13 while ZGC and Shenandoah GC are 2 out of the 3 latest released GCs for Java (along with Epsilon GC) and have been highly praised for their performance benefits.

Typically when looking at the performance of a garbage collector the key measurement values are latency and throughput. Latency is the responsiveness of an application, the delay experienced by the application due to garbage collection activity. For certain collection phases, the garbage collector needs to suspend all application threads to conduct necessary memory management. This suspension duration, known as a pause time or Stop-The-World (STW) event, results in latency as the application is unable to progress. The goal when optimizing for low latency is to reduce the duration of pause times and/or the number of times they occur. Throughput is instead a measurement of the workload managed by an application within a specific unit of time. It is a measurement of application production efficiency. For example, throughput can be measured by the number of completed transactions in an hour or percentage of the total execution time of an application not spent on garbage collection activities, measured over long periods of time. When opting to maximize throughput the workload managed per time unit, in the long run, is prioritized and higher pause times can often be overlooked. However, in large scale real world systems a delicate balance of throughput and latency is required.

Most "Big Data" systems are written in managed languages, such as Java, C#, or Scala. These systems suffer from severe memory problems due to the massive volume of objects created to process input data. Allocating and deallocating a sea of data objects puts a severe strain on existing garbage collectors (GC), leading to high memory management overheads and reduced performance. This memory management in Big Data systems is often prohibitively expensive. For example, garbage collection (GC) can account for close to 50% of the execution time of these systems severely damaging system performance. The problem becomes increasingly painful in latency-sensitive distributed cloud applications where long GC pause times on one node can make many/all other nodes wait, potentially delaying the processing of user requests for an unacceptably long time. Hence memory management in big data systems is a challenge which we like to explore in our work.

Garbage collectors traditionally try to make assumptions and/or observations of application behavior to form the memory management after memory behavior. For example, the Garbage First GC concentrate space-reclamation efforts on young objects based on the assumption that objects more recently allocated are more likely to have turned into garbage. Another example is the Adaptive heuristic of Shenandoah GC which initiates a garbage collection cycle based on observations of time, heap occupancy, and allocation pressure from previous garbage collection cycles. Thereby relying on the assumption of repetition of application memory behavior. The more the GC can work in symbiosis with the application the lesser the overhead caused by GC activity will be. However not every GC is equipped to work in every environment. Our project takes that fact into account and try to evaluate Garbage First GC (G1GC), Shenandoah GC and ZGC against each other in different benchmarks, taking different heap sizes into account. While choosing benchmarks we tried to work with those which were written on big data library like Spark[5].

2 Background

All of the garbage collectors compared in this study are reference tracing algorithms. The reason for this is primarily due to the problems associated with reference counting, i.e., reference counting needing additional techniques to be a complete algorithm which is computationally costly, and reference counting requiring a write barrier in all reference modification instructions, which incur an extra overhead on performance. Therefore, reference counters algorithms are not utilised in most production JVMs.

2.1 Garbage First (G1)

The Garbage-First [6] is the current default collector for OpenJDK Hotspot JVM and is one of the most widely used garbage collectors. G1 is a two-generational garbage collector. Instead of having the young and old generation be a contiguous chunk of memory where garbage collection is monolithic, in G1 both the young and old generations are a set of regions where most GC operations can be applied individually to each region. Also, regions that belong to the same set and therefore same generation do not need to be contiguous in memory.

For the young generation, similar to the previous collectors, G1 employs a parallel "stop the world" copying collector which collects all regions belonging to the young generation region set (monolithic). For the old generation, on the other hand, G1 does not require the whole generation to be collected. Instead, just a subset of the old generation region set is collected at any one time during a mixed collection, using a mostly concurrent mark and sweep collector. A mixed collection is a young generation collection, where the subset of the old generation region set chosen is also collected together. Using a full heap trace allows G1 to track the amount of garbage in each region accurately and therefore, preferentially target regions that will yield the most garbage. In addition, incremental compaction is employed by G1, which means that on every old generation collection, all objects in the subset of regions being collected are relocated to unused regions. Regions that have all their reachable objects relocated (due to incremental compaction) become unused regions, which later can be used by either generation. When objects can no longer be promoted to the old generation, it falls back to a monolithic "stop the world" compaction of the old generation. However, with the incremental compaction and with sufficient tuning, G1 is designed in such as way that a full garbage collection being required should not occur. Nonetheless, this requires some application profiling to best tune the garbage collector to the application needs.

In addition to a memory barrier to update remembered sets, G1 also employs a snapshot at the beginning (SATB) barrier. During a concurrent marking phase, an object initially believed to be garbage may become reachable due to a new allocation; therefore, a mechanism (SATB) that avoids the collection of these objects is employed. This mechanism is an additional hook that gets called whenever mutator threads write references field in objects during concurrent marking so that these objects may be marked as live after the marking ends.

2.2 Shenandoah

The Shenandoah garbage collector, like the ZGC, has the goal of reducing pause times on large heaps. It is also a region-based non generational collector that uses similar principles to ZGC but follows a different implementation strategy. Instead of coloured pointers, Shenandoah makes use of Brooks pointers, for allowing concurrent compaction of memory. The main idea behind a Brooks pointer is that each object has an additional reference field that always points to the current location of the object. The referenced location can either be the object itself or, as soon as the object gets copied to a new location, to that new location.

During compaction, an object that is set to be relocated must be copied from the "from-space" to the "to-space". The from-space, as the name implies, is the original location of the object, and the to-space is the destination of the object after the copying. A classic "stop the world" compaction would then stop application threads so it could update all references to the old "from space" object, to the current "to space" reference. However, with Brooks pointers, we no longer need to stop the application to update all references leading to the "from space" object. Every object now has an additional reference field (forward pointer) that points to the object itself, or, as soon as the object gets copied to a new location, to that new location. To assure that all writes are done on the "to space" object, a write barrier is triggered for all writes. This write barrier is responsible for dereferencing the forward pointer on the old object, to reach the current location of the object. Additionally, if during a copying phase a write barrier is triggered on a "from-space" object that is set to be copied but has yet to be, the procedure is as follows: i) creates the "to space" object; ii) updates the "from space" forward pointer; iii) writes the value in the "to space" copy; iv) updates the reference that triggered the barrier to point to the new location. Eventually, when the copying phase terminates, references that still point to "from space" objects that were copied are updated during concurrent marking. When all references are updated, the "from space" object is collected.

This technique introduces some overheads, e.g., memory overhead caused by the forward pointer (usually one word per object), more instructions to verify that writes and reads are always done in the "to space" object, and the high possibility of cache misses due to pointer-chasing indirection. Furthermore, contrary to the ZGC collector, Shenandoah claims to work exceptionally well even in small heaps.

2.3 ZGC

The Z Garbage Collector, also known as ZGC, is an experimental scalable low-latency collector that is built to handle heaps varying from relatively small to potentially multi-terabytes sizes. Also, ZGC's pause time is supposed not to exceed 10ms even when increasing the heap or live-set size.

Compared to the Garbage-First collector, ZGC is also a region-based collector; however, it is not generational. It improves upon G1 by achieving concurrent compaction with the introduction of two core techniques, read barriers and coloured pointers. The coloured pointer is a technique that uses 4 of the 22 unused bits of a 64-bit reference to store some important metadata. AThe first 42-bits of an object reference are reserved for the actual address of the object, which gives a theoretical heap limit of 4TB address space. From the remaining unused bits, four of those are used as flags named finalizable, remapped, marked1 and marked0:

- finalizable - The bit is set if the object is only reachable through a finalise method.

- remapped - The bit is set if the reference points to the current address of the object. When the collector is performing a concurrent relocation, and an object is loaded by the application, a read barrier is triggered when loading the reference from the heap. The application will first check if the remapped bit is set. In case it is, it means the reference points to the current address of the object and the reference to the object is returned. Otherwise, it checks if the object is in the relocation set. If the object is not in the relocation set it means the reference points to the current address, but the remapped bit is yet to be set, so the application sets the bit and returns the address. In case the object is indeed in the relocation set, the application checks if the object has already been relocated or not. If it has then the reference is updated to the new current address of the object and returned. In case it has yet to be relocated, the application threads relocate the object and return the updated reference.

- marked0 and marked1 - These are used to flag reachable objects. When the relocation phase concludes there may still be references that need to be remapped and hence have still one of the marked bits set from the last marking cycle set. If the subsequent marking phase used the same marking bit, the read-barrier would see this reference as already marked, incorrectly. Therefore, the marking phase alternates between using the marked0 and marked1 bit in each cycle.

When a read barrier is invoked by loading a reference, there are a few assembly instructions that need to be executed. Due to the high frequency of reads and writes in an application, both write and read barriers need to be extremely efficient as not to cause large performance overheads. Even though ZGC does not make use of write barriers, it uses read barriers called load-value barriers (LVB) to do concurrent compaction. In addition to the overhead caused by the read barrier, there is a cost associated with using coloured pointers, more specifically due to the need of dereferencing the object address from the pointer. To avoid this cost, when allocating a page, ZGC maps the same page to 3 different addresses, corresponding to the original address plus an offset caused by precisely one of the flags being set. This approach works because when reading from memory precisely one bit of marked0, marked1 and remapped is set.

3 Experiment

3.1 Chosen Benchmarks

For experimenting the performance of these three modern day garbage collectors, we chose to run these GCs on a number of benchmarks from two well known benchmark suite: the DaCapo Benchmark Suite [7], [8] and the Renaissance Benchmark Suite [9]. We downloaded the DaCapo benchmark suite from DaCapo Benchmark Suite and the Renaissance benchmark suite from Renaissance Benchmark Suite in *.jar* formats.

From the DaCapo benchmark suite, we chose the following four benchmarks to examine:

- h2: Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark
- lusearch: Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
- lucene: Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
- pmd: analyzes a set of Java classes for a range of source code problems

From the Renaissance benchmark suite, we chose the following eight benchmarks to examine:

- chi-square: Runs the chi-square test from Spark ML library
- dec-tree: Runs the Random Forest algorithm from the Spark ML library
- movie-lens: Recommends movies using the ALS algorithm in Apache-Spark
- page-rank: Runs a number of PageRank iterations, using RDDs
- dotty: Runs the Dotty compiler on a set of source code files
- philosophers: Solves a variant of the dining philosophers problem using ScalaSTM.

- scala-kmeans: Runs the K-Means algorithm using Scala collections
- finagle-http: Sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response

Among these eight benchmarks, we chose to vary the maximum allocated heap size for chi-square and movie lens from 500 MB to 1500 MB in one experiment. For page-rank benchmark, we chose to vary the maximum allocated heap size from 1000 MB to 2000 MB in one experiment.

3.2 Experimental Setup

3.2.1 Environment

The DaCapo benchmarks and the Renaissance benchmarks both are run on OpenJDK Java version 11.0.15 with HotSpot JVM. The DaCapo benchmarks are executed on Ubuntu 20.04 Virtual OS with 4 GB RAM and the Renaissance benchmarks are executed on Ubuntu 20.04 Virtual OS with 8 GB RAM.

3.3 GC and Benchmark Options

In this section we denote the GC options used within the various configurations formed for the performance tests.

To see experimental options in the logs, we added the *-XX:+UnlockExperimentalVMOptions* option in the command for running Shenandoah and ZGC garbage collectors in case of both DaCapo and Renaissance benchmarks. To get a detailed log for each execution, we added the *-Xlog:gc*:file* option in the command for each of the three garbage collectors in case of both benchmarks.

The Renaissance benchmarks run for different numbers of iterations by default. So, we explicitly define the iteration count as 10 for each of the Renaissance benchmarks, not bounding the maximum allocated heap size, by using *-r 10* option in the command. While bounding the maximum allocated heap size for three Renaissance benchmarks, we define the iteration count as 2 by using *-r 2* option in the command as these three benchmarks generate a large amount of objects and take a lot of time to complete execution in our environment.

For three of the Renaissance big data benchmarks, we vary the maximum allocated heap size by using *-Xmx:* option in the command for all three garbage collectors and keep the iteration count as 2 as described earlier.

3.4 Execution

In Ubuntu Virtual OS, we wrote three shell scripts, one for running three garbage collectors (G1, Shenandoah and ZGC) on DaCapo benchmarks not specifying or varying the maxium allocated heap size, one for running three garbage collectors on Renaissance benchmarks not specifying or varying the maximum allocated heap size and one for running three garbage collectors on three Renaissance benchmarks varying the maximum allocated heap size.

We executed the following three commands for executing three GCs on DaCapo benchmarks:

- **sh_default**: java -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xlog:gc*:file=shgc_$benchmark.log -jar dacapo-9.12-MR1-bach.jar $benchmark
- **z_default**: java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xlog:gc*: file= zgc_$benchmark.log -jar dacapo-9.12-MR1-bach.jar $benchmark
- **g1_default**: java -XX:+UseG1GC -Xlog:gc*:file=g1gc_$benchmark.log -jar dacapo-9.12-MR1-bach.jar $benchmark

Here *dacapo-9.12-MR1-bach.jar* is the name of the downloaded benchmark suite file, *$benchmark* is a command line argument to specify the name of the benchmark of the suite and shgc(or ZGC or G1GC)_$benchmark.log is the output log file name for Shenandoah(or ZGC or G1GC) GC information. In case of Renaissance benchmark suite (not varying maximum allocated heap size), the *dacapo-9.12-MR1-bach.jar* part in each of the three commands needs to be replaced by the name of the downloaded jar file for Renaissance benchmark suite, and a *-r 10* option to be added at the end.

- **sh_default**: java -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -Xlog:gc*:file=shgc_$benchmark.log -jar renaissance-mit-0.14.0.jar $benchmark -r 10

- **z_default**: java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xlog:gc*:file= zgc_$benchmark.log -jar renaissance-mit-0.14.0.jar $benchmark -r 10

- **g1_default**: java -XX:+UseG1GC -Xlog:gc*:file=g1gc_$benchmark.log -jar renaissance-mit-0.14.0.jar $benchmark -r 10

If we want to specify the maximum allocated heap size equal to a command line argument *size* in MB, we just need to add *-Xmx${size}M* to the commands. For example, in case of running ZGC on a Renaissance benchmark, with maximum allocated heap size as 500 MB the command will be:

java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -Xmx${size}M -Xlog:gc*:file= zgc_$benchmark_${size}.log -jar renaissance-mit-0.14.0.jar $benchmark -r 2

We executed the shell script for DaCapo benchmark on Ubuntu 20.04 having 4 GB RAM, and the shell scripts for Renaissance benchmarks on Ubuntu 20.04 having 8 GB RAM. The output log files for each of the benchmarks are stored separately.

3.5 Examining Output Logs

Running three garbage collectors on four DaCapo benchmarks, we get 12 log files. Subsequently, from eight Renaissance benchmarks, we get 24 log files not varying the maximum allocated heap size and 45 log files varying the maximum allocated heap size on three Renaissance benchmarks. We export these 81 log files separately to GCEasy, which is an online universal GC log analyzer guided by Machine Learning algorithms. This online tool automatically examines the logs and calculate important performance metrics like throughput, maximum allocated heap size, maximum used heap size, average pause GC time, maximum pause GC time, total concurrent GC time, total created bytes, average allocation rate, young and old generation GC time etc.

Among all the metrics, we focused on seven metrics:

- Throughput percentage: Percentage of time spent in application workload compared to time spent in GC activity of the total time. A higher percentage is a good indication that GC overhead is low. Concurrent GC work is not regarded as GC time but included in application time.

- JVM heap size allocation: The allocated size of heap memory.
- JVM heap size peak: The memory peak utilization size.
- Average allocation rate: The average memory allocation rate representing the object creation rate by the application.
- Average pause time: The average amount of time taken by one STW (Stop the World) GC pause.
- Maximum pause time: The maximum amount of time taken by one STW GC pause.
- Total concurrent time: Total accumulated time of GC activity concurrent with application threads.

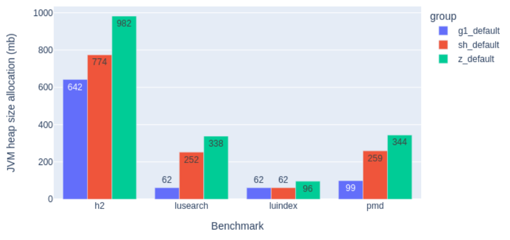
We store the values of these seven metrics for each of the 81 execution instances in a csv file. Then we developed a python script and used that csv file to generate graphs showing the performance variation of the three garbage collectors in different benchmarks. The next section will focus on these graphs and output analysis.

4 Result

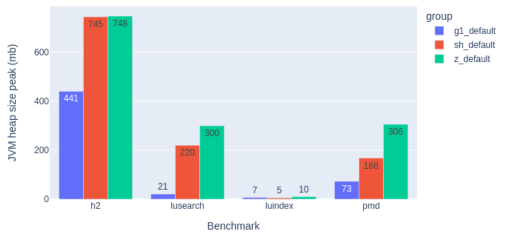
4.1 DaCapo Benchmark Suite

This benchmark suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory loads.

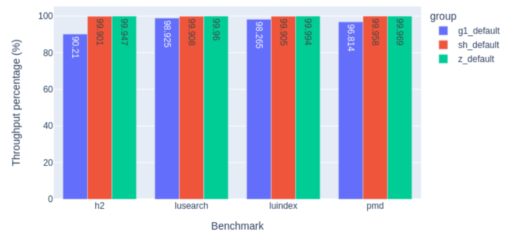
We evaluated G1, Shenandoah and ZGC using their default settings (commands are given in section 3.4) in DaCapo benchmark suite with respect to the metrics discussed in section 3.4. As per our expectation, ZGC gave the highest concurrent time and throughput in maximum cases whereas G1GC had maximum pause time in all benchmarks. G1GC gave maximum average allocation rate in every case. The detailed results are shown in fig. 1.



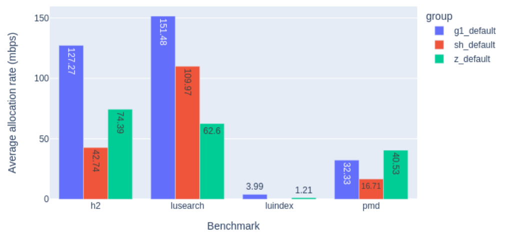
((a)) Heap size allocation (mb)



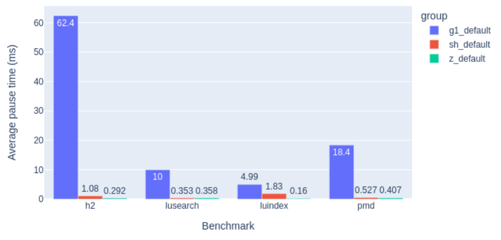
((b)) Heap size peak (mb)



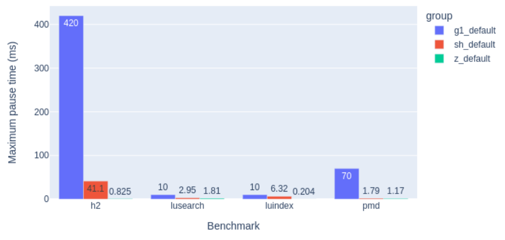
((c)) Throughput percentage



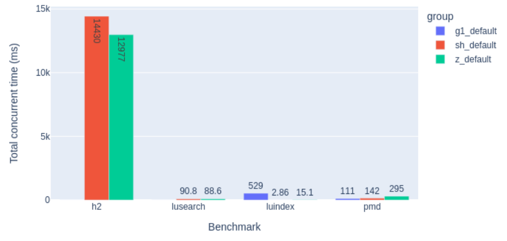
((d)) Average allocation rate (mbps)



((e)) Average pause time (ms)



((f)) Maximum pause time (ms)



((g)) Total concurrent time (ms)

Figure 1: Evaluation of G1GC, Shenandoah and ZGC in DaCapo benchmark using different metrics

4.2 Renaissance Benchmark Suite

We evaluated G1, Shenandoah and ZGC using their default settings (commands are given in section 3.4) in Renaissance benchmark suite with respect to the metrics discussed in section 3.4. As per our expectation, ZGC gave the highest concurrent time and throughput in maximum cases whereas G1GC had maximum pause time in all benchmarks. The detailed results are shown in fig. 2.

4.3 Renaissance Benchmark Suite with varying heap size

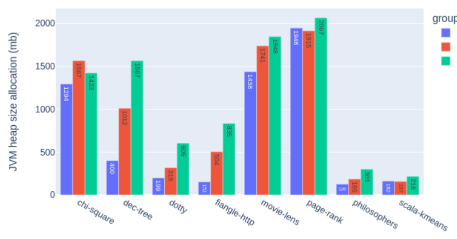
We worked with chi-square, movie-lens and pagerank benchmark from Renaissance benchmark suite. We varied heap size from 500 to 1500 mb for both chi-square and movie-lens benchmarks. For movie-lens benchmarks, throughput increases with the increase of heap size in case of all garbage collectors whereas for chi-square, throughput does not increase (rather is static) with varying heap size. We could not observe such pattern for pagerank benchmark. In all these benchmarks, throughput of ZGC and Shenandoah is greater than G1GC for any heap size.

Total concurrent time for both ZGC and Shenandoah in above mentioned benchmarks show a downward pattern with the increase of heap size.

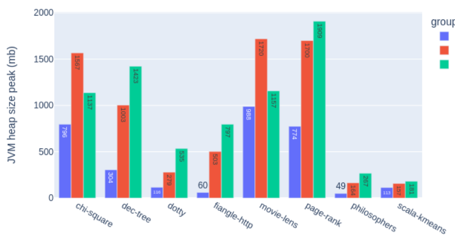
Results for movie-lens, chi-square and pagerank benchmarks are shown in fig. 3, fig. 4 and fig. 5 respectively.

5 Conclusion

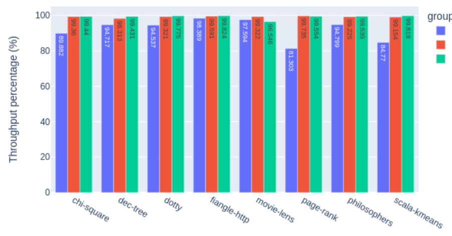
In this project, we mainly focused on the performance of three modern day JVM based garbage collectors: G1GC, Shenandoah and ZGC in big data benchmarks. We have found that, in almost all of the benchmarks, Shenandoah and ZGC perform reasonably better than G1GC. Though we have compared seven performance metrics of these three GCs, we could not compare some other metrics due to lack of time and effort. We also varied only one parameter of Garbage Collector, that is the maximum allocated heap size, but to better understand the performance under different constraints, we need to vary other parameters and examine the logs. Moreover, we did not examine the logs manually rather depended on the online GCEasy tool for analyzing



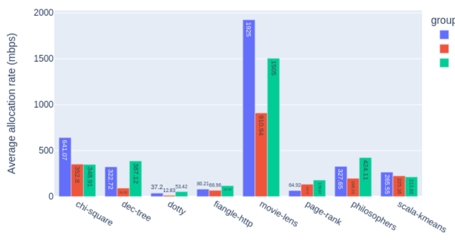
((a)) Heap size allocation (mb)



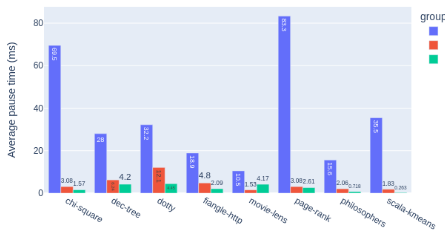
((b)) heap size peak (mb)



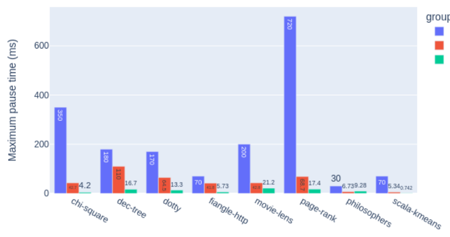
((c)) Throughput percentage



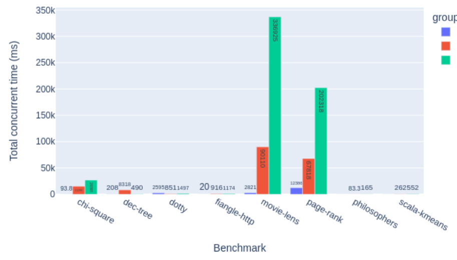
((d)) Average allocation rate (mbps)



((e)) Average pause time (ms)

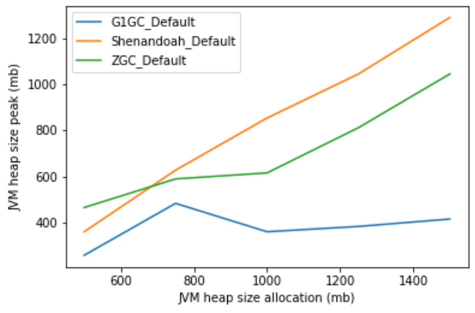


((f)) Maximum pause time (ms)

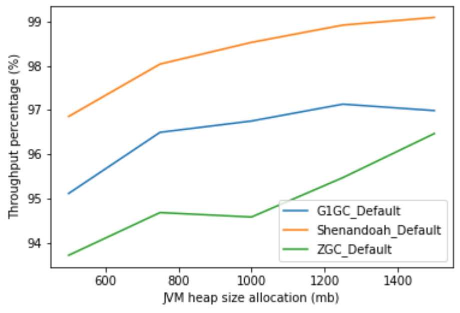


((g)) Total concurrent time (ms)

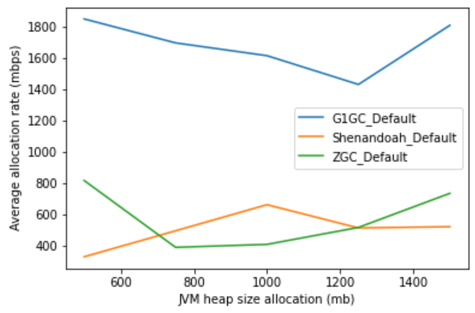
Figure 2: Evaluation of G1GC, Shenandoah and ZGC in Renaissance benchmark using different metrics



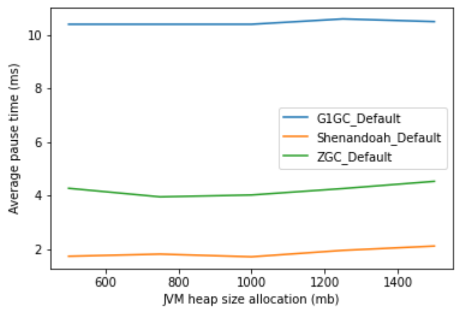
((a)) Heap size peak (mb)



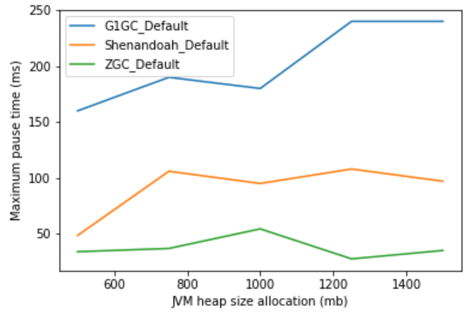
((b)) Throughput percentage



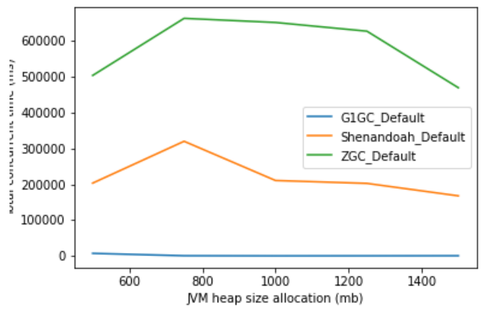
((c)) Average allocation rate (mbps)



((d)) Average pause time (ms)

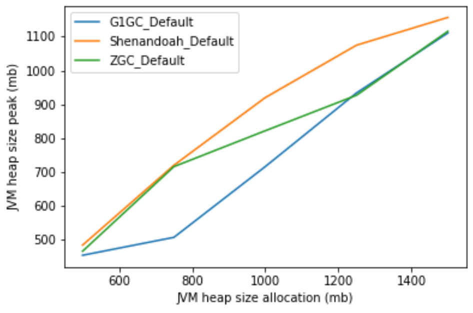


((e)) Maximum pause time (ms)

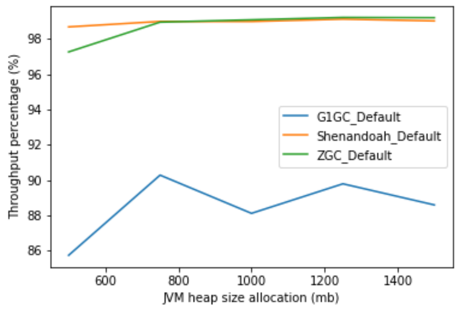


((f)) Total concurrent time (ms)

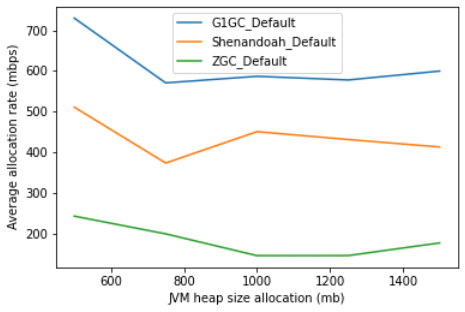
Figure 3: Evaluation of G1GC, Shenandoah and ZGC in movie-lens benchmark using different metrics by varying heap size (500MB to 1500MB)



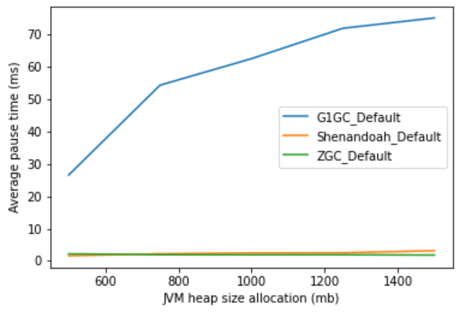
((a)) Heap size peak (mb)



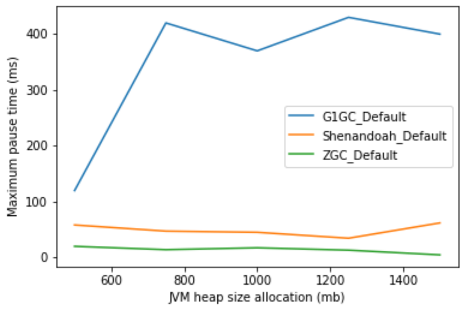
((b)) Throughput percentage



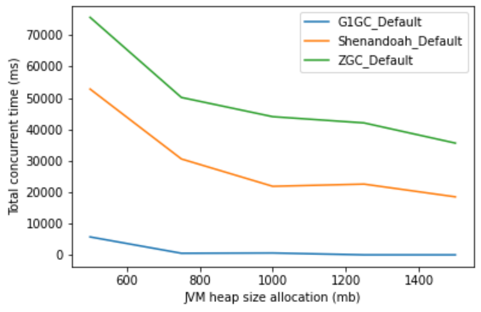
((c)) Average allocation rate (mbps)



((d)) Average pause time (ms)

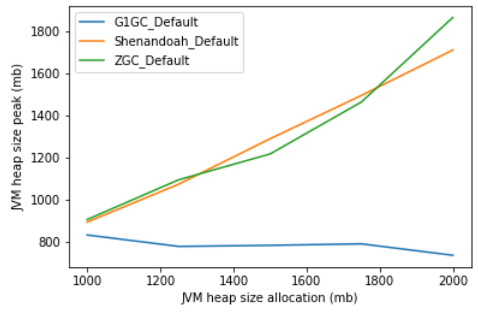


((e)) Maximum pause time (ms)

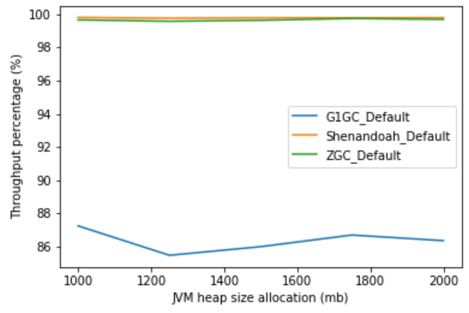


((f)) Total concurrent time (ms)

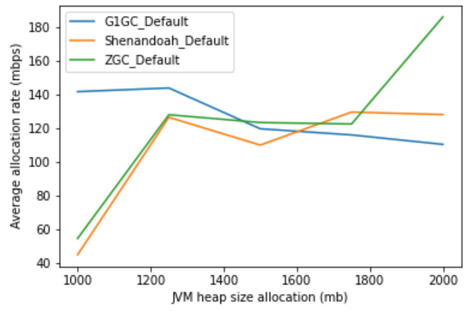
Figure 4: Evaluation of G1GC, Shenandoah and ZGC in chi-square benchmark using different metrics by varying heap size (500MB to 1500MB)



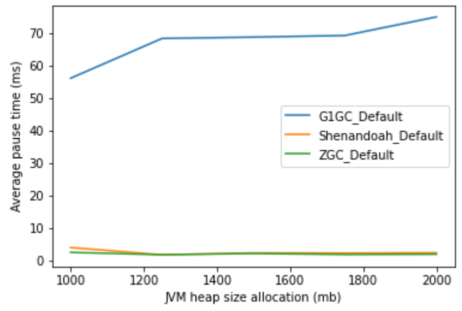
((a)) Heap size peak (mb)



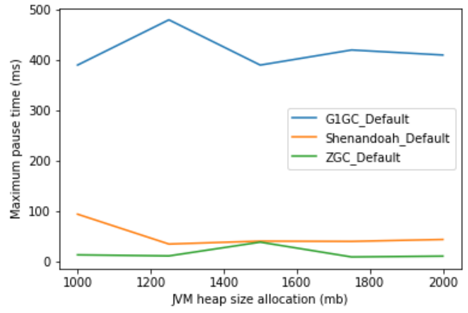
((b)) Throughput percentage



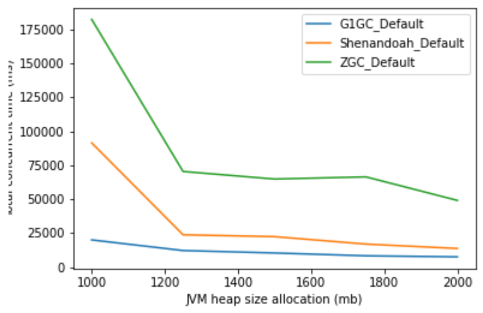
((c)) Average allocation rate (mbps)



((d)) Average pause time (ms)



((e)) Maximum pause time (ms)



((f)) Total concurrent time (ms)

Figure 5: Evaluation of G1GC, Shenandoah and ZGC in pagerank benchmark using different metrics by varying heap size (1000MB to 2000MB)

the tool automatically. One problem with this tool is that it analyzes the logs using Machine Learning algorithms, so the performance metrics output may not fully accurate. This experiment may be helpful to the greater community interested in system and compiler analysis by learning how to experiment and analyze JVM based garbage collector performance on different benchmarks.

References

[1] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, "A performance study of java garbage collectors on multicore architectures," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 20–29, 2015.

[2] P. Lengauer and H. Mössenböck, "The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors," in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pp. 111–122, 2014.

[3] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for openjdk," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pp. 1–9, 2016.

[4] P. Lidén and S. Karlsson, "The z garbage collector," 2018.

[5] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th international symposium on Memory management*, pp. 37–48, 2004.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The

DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java benchmarking development and analysis (extended version)," Tech. Rep. TR-CS-06-01, 2006. http://www.dacapobench.org.

[9] 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: A modern benchmark suite for parallel applications on the jvm," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2019, (New York, NY, USA), p. 11–12, Association for Computing Machinery, 2019.