Analysis of Garbage Collection Algorithms and Memory Management in Java

P. Pušek, H. Grgić and B. Mihaljević
Rochester Institute of Technology Croatia, Zagreb, Croatia
paula.puscek@mail.rit.edu, hrvoje.grgic@mail.rit.edu, branko.mihaljevic@croatia.rit.edu

Abstract – Significant elements of the Java Virtual Machine (JVM), as a part of the Java Platform, Standard Edition (Java SE), crucial for automatic memory management are various Garbage Collection (GC) algorithms. Since implementation of the Java Development Kit (JDK) is continuously being improved, it is difficult to ignore several new Java Enhancement Proposals (JEPs) correlated to memory management in JVM. Several different garbage collectors are implemented in addition to the existing aged Serial, Parallel, and Concurrent Mark & Sweep (CMS) GC algorithms, as well as newer Garbage-First (G1) GC. The major progress since JDK 10 is making Parallel Full GC for G1, as a default multi-threaded GC when performing collections on an entire heap. Redundant overheads could appear when GC algorithms perform garbage collection too frequently, or a too large amount of memory could be allocated when garbage is not collected regularly. The goal of new algorithms’ features is optimizing the overall process of releasing space so that pause times do not affect applications’ performances negatively. This paper explores several garbage collectors available in JDK 11 by using selected benchmarking applications of the DaCapo suite for comparison of the number of algorithms’ iterations and the duration of the collection time.

Keywords – Garbage Collector; Garbage Collection Algorithm; Memory Management; Java; G1; Garbage-First; DaCapo suite; Parallel Full GC for G1

I. INTRODUCTION

Memory management, the process which regulates the allocation and release of the memory for every application, should be optimized for a specific case to run efficiently. In lower-level programming languages, e.g., C or C++ languages, memory is managed explicitly by a developer, which might consequently lead to an application more prone to errors. In contrast, the main reason how some high-level object-oriented programming (OOP) languages, such as Java, differ from those, is the capability of performing the automatic reclamation and management of memory. The important component of an abstract computing machine with execution environment for other applications, such as Process Virtual Machine, is the process of automatic memory management, achieved through garbage collection algorithms.

The primary principles of automatic memory management system within Java Virtual Machine (JVM), the embedded part of Java Runtime Environment (JRE), are provisioning and recovery of the necessary memory from objects which are no longer being referenced in the run-time memory area called the heap.

In modern software development, automatic tuning of memory release is preferred in contrast to manual handling rearrangement of memory, as it helps to decide which parts of the heap space, where created objects reside, is available for use. Instead of relying on the developer’s expertise with the deep understanding of runtime data areas’ architectures, garbage collection algorithms and other internal optimization of JVM’s instructions [1], have been embedded as part of both runtime environment and JVM.

A. Heap Memory

The heap memory, a specific part of memory which is created on JVM’s start-up is used for allocating memory to all class instances and arrays. Also, all JVM threads are able to access the objects stored at the heap and perform allocations until there is enough space. Apart from the heap, as presented in Figure 1, another important element of JVM contributes to memory management, JIT compilation controls the running application by tracing information and allows making decisions based on it with the goal of optimization of the memory [2].

B. Garbage Collection

The component of JVM responsible for automatic process of managing the memory is oftentimes referred to as garbage collection. Garbage, the heap memory that is no longer in use by the application, is recovered and reused by performing the Mark and Sweep algorithm, which is the underlying principle of garbage collection algorithms. In its simplest form, this algorithm stops all running program threads, and examines all live objects, which have a reference in any stack’s frame of any application thread. Afterward, it searches through the reference tree of used objects and marks any object found as live. Every object left is considered garbage and is ready to be collected (swept) [3].

Figure 1. Crucial components of Java Virtual Machine related to memory management
A garbage collector is responsible for several tasks [4], including:

- Allocating from and giving back memory to the operating system
- Handing out that memory to the application as it requests it and determining which parts are still in use by the application
- Reclaiming the unused memory for reuse by the application

The essence of Java’s garbage collection is that the runtime keeps track of objects and automatically destroys those which are no longer needed [2]. However, garbage collectors do not solve all problems related to automatic memory management. For instance, an application may continue keeping unnecessary references or new objects could be continuously allocated, and consequently out of memory error might be encountered.

Automated reclamation comes with the performance cost, which can be significantly enlarged when running applications that might require more computational resources [5]. Therefore, there is a need for the improvement of the garbage collection algorithms which allow the applications’ runtimes to be optimized in a way that they are rarely interrupted by the garbage collection process [3]. Therefore, new enhancements are proposed in order to move closer to the accomplishment of this goal.

II. GARBAGE COLLECTORS (GCs)

There are many garbage collectors (GCs) that we can use in the JVM, ranging from the most commonly used older GCs, including Serial GC, Parallel and Parallel Old GC, Concurrent Mark and Sweep (CMS) GC, and Garbage-First (G1) GC, up to the latest experimental ones like Z Garbage Collector (ZGC) and Shenandoah GC.

The Parallel Old GC was the default garbage collector in JDK 7 and JDK 8 before it was substituted with G1 GC in JDK 9. The G1 GC has been improved since its official introduction and upgraded with new Parallel Full G1 GC, default in JDK 10 and onwards. Additionally, JDK 11 introduced an experimental GC named Epsilon GC, mostly for testing purposes. Moreover, the Java Zing VM’s default garbage collector algorithm is upgraded Azul’s C4 (Continuously Concurrent Compacting Collector) GC.

A. Current Java Garbage Collectors

Serial, Parallel, and Parallel Old GCs are all implemented as full Stop-The-World (STW) collectors, and take advantage of the Weak Generational Hypothesis, claiming that the vast majority of objects are short-lived, and others have longer life expectancy [2].

Following this hypothesis, the heap is divided into three generations: young, old, and permanent. Objects are first allocated in the young generation, and when more memory is required for the new objects, minor garbage collection is performed. Memory that is no longer used is discarded, and live objects are moved from the young generation to the old generation. Since objects are eventually moved, at some point of time it is necessary to collect garbage on the whole heap, and thus, perform full garbage collection, which might require more computational resources.

On the contrary, due to the scanning of unused objects without stopping the application running threads, the CMS and G1 GC are known as concurrent low-pause collectors. Moreover, both are able to perform full collection but occasionally tend to avoid it [6]. The application experiences fewer pauses because the duration of phases in which application threads are stopped is decreased compared to the STW GCs. G1 GC’s heap is based upon the set of regions, unlike the Parallel or CMS GCs with a generational heap. Since JDK 9, the CMS GC has been deprecated, and G1 GC has become the default.

1) Parallel Full Garbage-First (PF1) GC

In Garbage-First GC, both young and old generations are sets of regions, and most garbage collection operations can be performed per region at a time, rather than on the entire heap or an entire generation. Also, there is no need to decide which regions will be part of young or old generation in advance; thus, it is possible that, after a young generation collection, a whole region becomes a part of an old generation and vice versa [7]. Although some GCs have a generational design and finish collection relatively fast on a small portion of the heap, they inevitably have to enter a full collection at some point in time to collect the whole heap space and thus are subjected to considerable pause times.

Full garbage collection uses a compaction-based algorithm that copies all live objects into the beginning of the old space to vacate a vast continuous free memory area. Specifically, Parallel Scavenge GC (PSGC), which is compatible with Parallel Old GC, implements full GC algorithm that partitions the heap into regions which represent tasks assigned to multiple GC threads for further concurrent processing. To understand the primary principles of the full collection algorithm, PSGC’s implementation might be separated in three phases [8]:

- Mark phase – GC’s threads search for live objects, for instance, on-stack references or static variables from known roots which are marked as alive and their locations are recorded for later use
- Summary phase – PSGC calculates a heap summary for all live objects based on the pre-generated records, divides the heap into continuous regions of equal size (default 512KB) and summarizes objects within the same region
- Compacting phase – GC threads concurrently fetch destination regions and fill them with found live objects in corresponding source regions

Sometimes, G1 GC is unable to find a free region when trying to copy live objects from a young region or during evacuation from an old region. At such times, the collector will try to increase its usage of the heap, and if the expansion of the space is not successful, G1 will trigger its “fail-safe” mechanism in which a single thread marks, sweeps and compacts all regions constituting heap generations [7].

Since JDK 10, there have been some improvements regarding the worst-case latencies of G1 GC resulting in
development of Parallel Full GC for G1 GC\textsuperscript{1}. The major improvement is a multi-threaded parallel mark-sweep-compact algorithm which should use the same number of threads as the young and mixed collections while performing full garbage collection.

B. Experimental and Other Garbage Collectors

In the last few years, several experimental GCs were introduced, but most of them are still under development and used only for special purposes or within a specific environment.

Epsilon GC \textsuperscript{2} was presented with the latest JDK 11, as a garbage collector that handles memory allocations without performing any actual memory reclamation. The goal of this approach is not to introduce manual memory management features to the JVM or new heap management API, but to use it as a reference point for other GCs comparisons, when performing performance analysis for other GCs, memory or VM testing, or for last-drop latency and throughput improvements. It is a passive GC implementation with a bounded allocation limit and the low latency overhead, at the expense of memory footprint and throughput. In a way, Epsilon is not a "real" GC and is designed for testing purposes or special cases only.

As a part of their Zing JVM, Azul Systems has also created a high-performance concurrent and parallel GC named C4 (Continuously Concurrent Compacting Collector) \cite{9}, which provides consistent and contained application response times by eliminating GC pauses, thus enabling Java applications to scale up easier. Zing uses concurrent compaction with a single 64-bit word object header. However, since C4 is not a part of an open-source project like OpenJDK, and not easily comparable in our test environment, we will not consider it in this analysis.

In this research paper, we will also discuss ZGC and Shenandoah GC, although both are still considered experimental and under development.

1) ZGC Garbage Collector

The ZGC \textsuperscript{3} is an experimental scalable low-latency garbage collector available in JDK 11. At the moment of writing this paper, it is only available for 64-bit Linux, which is its main platform, but other platforms could be added later. It handles heaps ranging from relatively small to very large multi-terabyte sizes, and ZGC’s pause times do not increase with a heap or live-set size. Therefore, the heap can vary from GBs to TBs, and that will not have a significant impact on pause times because STW phases are limited to root scanning, and its pause times should not exceed 10ms.

Compared to G1, ZGC is also region-based, but it has a flexible sizing scheme, and has a better way to deal with object allocations; there is no generational separation of the heap. The heap is internally divided into many small regions, and we can choose to compact subset of those, typically one that contains the most garbage.

Load barriers and colored pointers are two core techniques chosen for achieving concurrency in ZGC. Pointer coloring is a core method which stores metadata in unused bits of 64-bit pointers. Because there is not enough unused space in 32-bit pointers for using pointer coloring, 32-bit platforms are currently not able to support ZGC. Metadata holds information about the object itself or about the object that it points to, and currently, it stores marking and relocation related information. By storing metadata in a pointer, we can use a load barrier when an object reference has to be loaded from the heap, which is conceptually similar to the decoding of Compressed Oops (Ordinary Object Pointers). Once a reference is loaded, the load barrier will check if the metadata bits (i.e., the colors) are "good" or "bad," and make a decision. "Bad" color means that particular action should be performed, whether that is marking, relocating, or remapping (depending on the phase of GC), and "good" color indicates that there is no need to again load the object reference from the heap \cite{10}.

Colored-pointers allow reuse of the memory before reclaimed regions are fixed during the relocation-compaction phase, which helps to keep the general heap overhead down, and therefore, an additional separate mark-compact algorithm that should handle full collections does not have to be implemented. Also, only few simple GC barriers keep the runtime overhead down and enable easier implementation, optimization, and maintenance of the GC barrier code in interpreter and JIT compilers \textsuperscript{3}.

2) Shenandoah Garbage Collector

The main reason behind Red Hat’s development of an experimental Shenandoah GC \textsuperscript{4} was to reduce the pause times by performing concurrent compaction. Shenandoah GC [11] is an open-source region-based low-pause parallel and concurrent collector, similar to ZGC, which attempts to reduce pause times by evacuating objects (compacting) concurrently with running application threads. Although GC performance depends on the heap size, with the Shenandoah the pause times are independent of the size of the heap.

The concurrent compaction is a complex task, as with moving live objects, all references to that object have to be updated to point to the new location, and to find those references the entire heap must be scanned. The object layout for Shenandoah GC adds its word on every object’s header, and it is allocated only when this collector is used. This enables moving the objects without updating all references of an object, but the thread which is copying the object performs an atomic compare and swap to have it point to the new address. All future readings or writings on this object will be done on a forwarded copy via the forwarding pointer. Similar to ZGC, Shenandoah’s heap is

\textsuperscript{1} JEP 307: Parallel Full GC for G1, http://openjdk.java.net/jeps/307
\textsuperscript{2} JEP 318: Epsilon: A No-Op Garbage Collector (Experimental) https://openjdk.java.net/jeps/318
\textsuperscript{3} JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental), https://openjdk.java.net/jeps/333
\textsuperscript{4} JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental), https://openjdk.java.net/jeps/189
partitioned in equal sized regions, and a region may keep newly allocated objects, long-lived objects, or a mix of both, and any subset of those regions might be collected during a GC cycle, similar to G1.

III. BENCHMARK AND PRELIMINARY RESULTS

A. Benchmark Suite

Benchmark suites are practical research tools, indispensable when features’ quality of systems is being tested, which report descriptive results to demonstrate the advantages within a new or improved system [12]. For the purpose of this research, we have chosen the DaCapo suite, which consists of a set of real-world Java applications for performance and memory management analysis, with non-trivial memory loads. In general case, open source DaCapo benchmarks are mostly larger, more complex, and richer than the commonly used SPEC Java benchmarks, thus ideal for scientific evaluations [13].

For exploring characteristics of new and experimental GCs available in JDK 11, we decided to use version 9.12-bach of DaCapo benchmark suite which has also been used when our previous preliminary research was conducted, and G1 GC was being compared to garbage collectors in JDK 9 [14]. The selected set of testing applications is similar to those which were used in prior testing and includes the following tests: fop (a single thread application which parses XSL-FO to generate PDF), h2 (a multi-threaded in-memory database which executes JDBC by performing transactions against a banking model), pmd (an application that analyzes Java source code in multiple threads), xalan (transforms XML documents into HTML by using multiple threads), sunflow (a multi-threaded rendering system for realistic images using ray tracing), and tradebeans (derived Daytrader benchmark with multiple numbers of threads).

B. Test Environment

Our testing environment relies on Java HotSpot™ Client VM of build 25.201-b09, mixed mode as well as Java HotSpot 64-Bit Server VM 18.9 of build 11.0.2+7-LTS, mixed mode. Moreover, OpenJDK 64-Bit Server VM of build 12-ea+29, mixed mode and sharing was included. The hosting environment which provided the testing was Windows 10 Pro OS. Regarding memory reclamation, the JVM’s choice for GC in JDK 8 was Parallel GC, whereas in JDK 11 the G1 GC was selected. Additionally, other garbage collection algorithms were tested, including CMS and Serial GC.

The results were gathered by using a graphical tool, Visual VM5, version 1.4.2, as well as its Visual GC plugin, which enables the in-depth exploration of measurements of both the number of algorithms collections and the duration of each. We repeated each test scenario within three different versions of Java. Moreover, each case consisted of five independent application runtime iterations, and both the average duration and collection count was calculated. In addition, the live set size was determined within each runtime by setting the heap size of 1GB.

C. Preliminary Results

Based on the generated traces, the metrics, i.e., the number of collections as well as the duration of executions, have been extracted for chosen test cases within JDK 8 and JDK 11, as well as the OpenJDK 12 Early-Access Build.

Even though G1 GC was expected to perform better in most benchmarks, that was not always the case, and G1’s advantages have not been obviously presented in the preliminary results within JDK 8. However, with an in-depth exploration of gathered data, interesting enhancements in performances of G1 and Parallel GCs might be noticeable in newer versions of Java.

In JDK 8, G1’s durations of collections are longer in contrast to Parallel GC for every application excluding tradebeans (Figure 2). When comparing these results to Serial GC, the number of collections (Figure 3) is more similar to Parallel GC’s, whereas in most cases, the execution time differs as it is increased in comparison with both G1 and Parallel GC. Besides, for other chosen applications, except for fop and h2, the CMS collector performed more collections than the other collectors in a relatively longer period than G1 and Parallel GCs.

![Figure 2. Comparison of durations (in milliseconds) of collections within JDK 8 for selected DaCapo benchmark applications](image-url)

![Figure 3. Comparison of number of collections within JDK 8 for selected DaCapo benchmark applications](image-url)

After conducting the identical benchmark within JDK 11, the behavior of the mentioned algorithms has changed. The most interesting difference is visible in the total garbage collection time of G1’s and Parallel GC’s collections when G1 was faster, in fact, for pmd application even almost two times faster than Parallel GC (Figure 4), but in most cases Parallel GC did more collections than G1 (Figure 5). However, for the test case h2, the major decrease of G1’s collection time within JDK 11 can be detected, i.e., new Parallel Full G1 performing similar number of collections as in JDK 8, but with recognizably reduced duration.

---

5 DaCapo Benchmarks, http://dacapobench.org/
Also, apart from G1 having outperformed Parallel GC in case of pmd, another interesting difference in G1’s behavior can be noticed in case of xalan application compared to Parallel GC. G1 had 13.34% less collections, but in doubly decreased amount of time, whereas within JDK 8, the Parallel GC was certainly a better choice for xalan.

The observation in case of fop benchmark is that G1 performed more operations than Parallel GC as in all other cases, Parallel GC performed a greater number of collections, e.g., for sunflow application 44.74% more and as well for tradebeans that number is significantly increased. Furthermore, for sunflow benchmark, Parallel GC surpasses the G1 as it did almost two times more collections with slightly reduced execution time within JDK 8, and in JDK 11 both collections and the durations are proportional as well as increased for both collectors.

With the quite interesting outcome in case of fop where G1 has shown to be superior over other collectors as a result of enlarged collection count, another benchmark luindex was included as part of the testing because both fop and luindex are run by a single thread. Luindex is a single-threaded full-text searching library and has a low nursery survival rate (a measure of how closely a program follows the generational hypothesis). Moreover, it has a high heap turnover ratio which might suggest substantial GC workload [13]. However, only G1 was able to perform collections in such environment with positive outcome as, for instance, Parallel GC was able to perform 10 collections in approximately 130 milliseconds when excluding argument –no-pre-iteration-gc, which reported worse performance of Parallel GC compared to G1 GC due to G1’s region-based heap architecture.

Although it might be concluded that G1 and Parallel GC are the main competitors for some test cases, there are also indicators that sometimes the best collector in terms of optimization is CMS GC, which is a rare case in our results. For example, in case of xalan, CMS is certainly the best choice, but with taking into consideration that it is deprecated, the Serial collector is the preferred choice for xalan, rather than G1 or Parallel GC. However, concerning an optimal amount of time spent, we consider G1 GC a better option in comparison with other collectors in this case. In contrast, with the exclusion of CMS, the best collector for sunflow is Parallel GC with the shortest execution time. It is important to mention that sunflow is one of the most allocation-intensive benchmarks, and might allocate up to 134 GB respectively.

The figures that demonstrate the behavior of garbage collectors within upcoming JDK 12 don’t differ much from JDK 11. However, when taking a closer look, Parallel GC performed less operations in case of pmd in the shorter period than in JDK 11. Also, G1’s total collection time was shorter in this case. An interesting difference might be seen as well as in CMS’s and Serial’s behavior when measuring the number of collections, which is similar to JDK 11, but the overall garbage collection time is almost doubly reduced for Serial, and remarkably increased when CMS was used to perform collection for pmd. As presented (Figure 6 and Figure 7), the overall time duration for G1 is significantly reduced since JDK 8. It is also clear to see that Parallel was a better option for pmd (Figure 8) until the release of Java 11 in which G1 has shown the better behavior than Parallel GC, although, within JDK 12, Parallel’s execution time noticeably decreased. Similarly, with utilizing the algorithms on another benchmark, i.e., tradebeans, within all versions of Java, G1 performed in the best manner and it reduced total collection time since its first release.
MIPRO 2019/SP

IV. DISCUSSION

Through Java’s history, Garbage Collection Algorithms have been tediously reanalyzed for achieving an environment where GCs operate in an optimal manner. According to P. Lengauer, V. Bitto, H. Mössenböck and M. Weninger [12], the minor pause time of the G1 is only 71% of the pause time of the ParallelOld GC, e.g. the h2 application’s pause time is 300 ms with ParallelOld GC, and only 81 ms with the G1 GC which is mostly due to the G1’s ability to choose the regions which will be collected and, as a consequence, control pause times as well as number of regions in which memory will be released.

The performance of G1 and Parallel GC in JDK 8 and 11 has been tested on OptaPlanner benchmarks [15]. As Parallel GC is focused on throughput, whereas G1’s main goal is low-latency, Parallel GC is still the better collector for OptaPlanner in terms of optimization according to the results. However, for almost every given data set, the average percentage of improvement of G1 in Java 11 is 16%, which can be supported with the newly introduced Parallel Full Garbage Collection with multiple threads.

Considering the fact that new experimental GC algorithms have been introduced within newer versions of Java, already existing collectors have also been compared to Shenandoah GC by running DaCapo benchmarks [11], and in 7 out of 11 test cases, G1’s overall time duration was decreased or equal to Shenandoah GC. However, when measuring the performance of Shenandoah and other collectors, i.e., G1, Parallel and CMS GC, with a benchmark that simulated a real customer application, Shenandoah had recognizably fewer GC pauses opposed to other algorithms.

V. CONCLUSION

Although there might be conditions, for instance, an application running with multiple or rather single thread, which might have an impact on algorithm performance, distinguishable changes might be seen in garbage collection algorithms’ implementations. G1 has operated better or in an equal manner as Parallel GC in cases of h2, pmd, xalan, and tradebeans considering the overall duration of iterations in the latest Java release. However, CMS and Serial, have not introduced any convincing evidence that would prevent us from questioning its strongly held position over the past years, but they have not shown to be assuredly superior over G1 and Parallel GC. Therefore, with improved G1, it certainly represents a direction in which future GCs are heading to. We believe that the future experimental GCs will as well contribute to overall systems’ optimizations. It will be worthy to explore the enhancements that will inevitably benefit those environments in which specified GC manages memory without any additional barriers which might decrement the productivity of an application.

REFERENCES


