A Study on the Causes of Garbage Collection in Java for Big Data Workloads

Aiswarya Sriram
Computer Science Dept.
PES University
Bengaluru, India
aiswarya.spaa@gmail.com

Advithi Nair
Computer Science Dept.
PES University
Bengaluru, India
advithi.nair@gmail.com

Alka Simon
Computer Science Dept.
PES University
Bengaluru, India
alkasimon23@gmail.com

Subramaniam Kalambur
Computer Science Dept.
PES University
Bengaluru, India
subramaniamkv@pes.edu

Dinkar Sitaram
Computer Science Dept.
PES University
Bengaluru, India
dinkar.sitaram@gmail.com

2020 IEEE International Conference on Big Data (Big Data) | 978-1-7281-6251-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/BigData50022.2020.9378113

***Abstract*—Garbage collection in Java is critical to the performance of Big Data applications. Our study brings forth the causes that invoke the 3 Java garbage collectors for Big Data and non Big Data workloads. We conclude that allocation failure occurs for about 31-99 per cent of the total garbage collector execution time and is the most common cause for garbage collection in Big Data workloads. In case of Dacapo workloads, we observe that the prominent cause for garbage collection is system gc calls which occur for about 40-99 per cent of the total garbage collector execution time.**

***Index Terms*—big data, concurrent mark sweep (CMS), dacapo, garbage collector (GC), garbage first (G1), parallel**

I. Introduction

Big Data applications handle massive amounts of data and are memory intensive. Most of these applications are written in memory managed object oriented programming languages like Java/Scala. Unlike C/C++, these languages use garbage collectors to reclaim memory of objects that are no longer in use.

The goal of our study is to analyze what triggers these garbage collections for Big Data workloads in a multi core setup. Insight into the different causes for garbage collection can be used to improve performance of the GC or workload. We have conducted our experiments with workloads sort, grep, wordcount and kmeans from BigDataBench [1] and sunflow, avrora from DaCapo benchmark [2] suites on OpenJDK 8 [3]. The three garbage collectors (GCs) considered are Parallel [4], ConcurrentMarkSweep [5] and G1 [6].

II. Related Work

There are multiple works on scalability [7] and performance impact of Garbage collectors on multicore environments [8]. However, these focus on non Big Data Java Benchmarks.

Lengauer et al. [9] described memory and GC characteristics of the DaCapo suite, the DaCapo Scala suite and the SPECjvm2008 benchmark suite. They further compared the default GC (ParallelOld GC) with G1 GC and concluded that G1 GC performed better with most workloads.

Iqbal et al. [10] performed a comparative analysis on garbage collectors in Sun HotSpot, Oracle JRockit and IBM J9 on different hardware systems.

While a lot of research has been done on smaller Java benchmarks like Dacapo, the focus has shifted to Big Data Benchmarks only in recent times.

Gidra et al. [11] considered a GC for Big Data on NUMA machines. This NumaGic was found to improve performance of throughput oriented applications. However, applications that required a concurrent collector were not analyzed. Bruno et al. [12] performed experiments on Big Data environments. They analyzed classic garbage collector algorithms and investigated scalability of these in Big Data environments. Xu et al. [13] performed a study on G1, parallel and CMS GCs using Spark applications. They analyzed patterns and inefficiencies of the GCs and proposed optimisation strategies for Big Data friendly collectors.

Our paper however, focuses on the behavior of the chosen Big Data Applications in order to determine the causes for a garbage collection. This investigation could be used to optimize the application and tune spark for better performance.

III. Theoretical Framework

A. Java Heap

The Java Heap space is used for dynamic memory allocation and deallocation of objects and JRE classes. When the heap becomes full, the automatic process of garbage collection clears objects that are no longer in use by the program and reclaims memory. The heap is divided into three generations to enhance overall JVM performance - *Young Generation, Old (Tenured) Generation* and *Permanent Generation*. *Young Generation* has three parts - *Eden, Survivor 0 and 1 spaces*. Newly created objects are allocated in the Young Generation. When it fills up, a Minor garbage collection is triggered which clears

TABLE I
TOP 2 GC CAUSES PER WORKLOAD PER GC

Workload	GC	GC Cause	Time%
Kmeans	CMS	Allocation failure	87.45
		Others	12.55
	G1	Others	84.06
		G1 Evacuation Pause	14.43
	Parallel	Allocation failure	99.32
		Metadata GC Threshold	0.41
Sort	CMS	Others	65.98
		Allocation failure	31.67
	G1	G1 Evacuation Pause	70.24
		Others	23.22
	Parallel	Allocation failure	81.7
		Ergonomics	12.42
Grep	CMS	Allocation failure	50.1
		Others	49.9
	G1	Others	65.25
		G1 Evacuation Pause	33.82
	Parallel	Allocation failure	93.63
		Metadata GC Threshold	6.37
Wordcount	CMS	Allocation failure	83.79
		Others	16.21
	G1	G1 Evacuation Pause	99.84
		Metadata GC Threshold	0.16
	Parallel	Allocation failure	89.3
		Metadata GC threshold	8.56
Sunflow	CMS	Allocation failure	59.79
		System.gc() calls	40.21
	G1	System.gc() calls	88.89
		G1 Evacuation Pause	6.49
	Parallel	System.gc() calls	68.25
		Allocation failure	31.75
Avrora	CMS	System.gc() calls	99.35
		Allocation failure	0.65
	G1	System.gc() calls	98.6
		G1 Evacuation pause	1.4
	Parallel	System.gc() calls	95.45
		Allocation failure	4.55

all the unreferenced objects. Objects that have survived many cycles of garbage collection and attained the age threshold are promoted to the *Old Generation*. Major garbage collection occurs when this generation is full. *Permanent Generation* contains the application metadata required by JVM to describe classes and methods used. A full garbage collection clears all the sections of the heap.

B. Garbage Collector Overview

The three Garbage Collectors considered for this study are:

Parallel GC - It is popularly termed as a Throughput Collector. This is the default Garbage Collector for OpenJDK 8.

ConcurrentMarkSweep GC (CMS GC) - It uses multiple threads for garbage collection. It marks the regions to be garbage collected and then sweeps those instances.

Garbage First (G1 GC) - This GC is mostly used for larger heap space. Garbage collection is prioritized in regions with the most garbage.

IV. METHODOLOGY

A. Hardware Specification

Experiments were run on an AMD EPYC 7301 machine with a 16-Core processor and x86_64 CPU architecture. It consisted of 32 logical cores, 4 NUMA nodes and Little Endian byte order. The max and min CPU speed of the machine was 3GHz and 1.2GHz respectively.

B. Software Specification

The software tools, packages and frameworks used are listed below:

- Linux operating system - Ubuntu(version 16.04)
- Apache Spark - version 2.3.2 [14]
- Apache Hadoop - version 2.7.3 [15]
- OpenJDK 8
- BigDataBench Spark - version 3.1.5 [16]
- DaCapo Benchmarks - version 9.12 [17]
- GCeasy [18]

C. Benchmark

We have used BigDataBench, an open-source Big Data benchmark suite. Three micro benchmarks - sort, grep, word-count and a component benchmark from the Social Networks domain - kmeans were chosen for the study. 30GB of input data was generated for these workloads.

Benchmarks - sunflow and avrora from the DaCapo benchmark suite were also chosen. These are open-source, real-world Java applications.

D. *Configuration and Procedure*

Apache Spark was configured in standalone mode with master and 4 worker instances each having a single executor. JVM options were added in the spark configuration file to switch garbage collectors, disable GC overhead limit, enable GC logging and redirect GC output to a file. Each Big Data workload was then run with 3 GCs and 30 GB of input data. Dacapo benchmarks were run with similar configuration. The garbage collection log files obtained were forwarded to the GCeasy tool to determine events that triggered the GC for these workloads.

V. Results

For Big Data workloads, we observe that the most common reason for a GC event is Allocation Failure. It is a regular event that occurs when the Young Generation does not have free space to create new objects. It triggers a Young garbage collection. The number of allocation failure related GC events could be reduced by increasing heap size of the Spark executor. However, this should be done with caution since increasing heap sizes indeterminately could cause higher pause times for garbage collection. Allocation failure is seen when the Big Data workloads are run with CMS and Parallel GC.

G1 GC is generally triggered due to a G1 Evacuation Pause. This occurs when live objects are being moved from one region to another. Depending on the region being copied, either a Young GC (Young region) or a Mixed GC is triggered (Young + Tenured regions).

The other GC causes are explained below:

Ergonomics: GC ergonomics tries to increase/decrease heap size based on application's requirement of throughput or latency.

Metadata GC Threshold: This occurs if the allocated metaspace size is too small or when there is a class loader leak.

Table 1 shows the top two GC causes for three GCs in Big Data and Dacapo workloads. The rightmost column gives the percentage of GC execution time used by that GC event.

The trend in GC causes seen per workload are:

- Kmeans - Allocation failure, G1 Evacuation Pause, Metadata GC Threshold
- Sort - Allocation failure, G1 Evacuation Pause, Ergonomics
- Grep - Allocation failure, G1 Evacuation Pause, Metadata GC Threshold
- Wordcount - Allocation failure, G1 Evacuation Pause, Metadata GC Threshold

In case of Dacapo workloads - sunflow and avrora, we see that GC causes are mainly of two types -

- System.gc() calls: Call to explicitly invoke the GC
- Allocation failure

However, it is not recommended to explicitly invoke System.gc() calls as they can degenerate GC performance.

VI. Conclusions

We see that the event which triggers the GC for Big Data workloads is mainly an allocation failure (31-99% of GC execution time). In the non Big Data workload, Dacapo, the prominent GC cause is System.gc() which takes 40-99% of GC Execution time. With these results, we wish to show why garbage collections occur. This can be used to improve application performance and also help with garbage collector tuning to optimize performance of the garbage collector.

Acknowledgment

The authors wish to thank Dr. Prakash Raghavendra from AMD India Pvt Ltd for his guidance.

References

[1] Gao, W., Zhan, J., Wang, L., Luo, C., Zheng, D., Wen, X., ... & Tang, H. (2018). Bigdatabench: A scalable and unified big data and ai benchmark suite. arXiv preprint arXiv:1802.08254.

[2] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., ... & Hirzel, M. (2006, October). The DaCapo benchmarks: Java benchmarking development and analysis. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (pp. 169-190).

[3] OpenJDK 8. https://openjdk.java.net/projects/jdk8/

[4] Parallel GC. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[5] CMS GC. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[6] G1 GC. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html

[7] Gidra, L., Thomas, G., Sopena, J., & Shapiro, M. (2013). A study of the scalability of stop-the-world garbage collectors on multicores. ACM SIGPLAN Notices, 48(4), 229-240.

[8] Carpen-Amarie, M., Marlier, P., Felber, P., & Thomas, G. (2015, February). 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).

[9] Lengauer, P., Bitto, V., Mssenbck, H., & Weninger, M. (2017, April). A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (pp. 3-14).

[10] IQBAL, S., KHAN, M., & MEMON, I. (2012). A Comparative Study of Garbage Collection Techniques in Java Virtual Machines. Sindh University Research Journal-SURJ (Science Series), 44(4).

[11] Gidra, L., Thomas, G., Sopena, J., Shapiro, M., & Nguyen, N. (2015). NumaGiC: A garbage collector for big data on big NUMA machines. ACM SIGARCH Computer Architecture News, 43(1), 661-673.

[12] Bruno, R., & Ferreira, P. (2018). A study on garbage collection algorithms for big data environments. ACM Computing Surveys (CSUR), 51(1), 1-35.

[13] Xu, L., Guo, T., Dou, W., Wang, W., & Wei, J. (2019, January). An experimental evaluation of garbage collectors on big data applications. In The 45th International Conference on Very Large Data Bases (VLDB'19).

[14] Apache Spark. https://spark.apache.org/docs/2.3.2/

[15] Apache Hadoop. https://hadoop.apache.org/docs/r2.7.3/

[16] BigDataBench User Manual. http://prof.ict.ac.cn/BigDataBench/wp-content/uploads/2014/12/BigDataBench-User-Manual.pdf

[17] DaCapo benchmarks. http://dacapobench.org

[18] Gceasy Tool. https://gceasy.io