International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

STRATEGIES FOR TUNING JAVA HEAP MEMORY TO ACHIEVE OPTIMAL PERFORMANCE IN HIGH THROUGHPUT ENVIRONMENTS

Rajesh Kotha
Senior Software engineer at Kroger

Abstract

Java heap space tuning is essential for optimal application performance where concurrency impacts resource consumption. This paper outlines various techniques for evaluating and managing Java heap memory, emphasizing the basic methods of addressing the issue, such as detecting and preventing memory leaks, garbage collectors, and JVM settings. This is particularly so for long-lived applications since memory leaks result in decreased throughput and increased latency. The problems described above can be easily identified and resolved with the help of tools like Java VisualVM and JProfiler. Also, the paper presents various kinds of garbage collection, such as serial, parallel, CMS, G1, and ZGC, along with the merits and demerits of each type. Garbage collection pause time and throughput depending on the chosen garbage collector and high throughput GC should be chosen according to application needs. Another element determining performance is the allocation patterns and settings in JVM that depict the distribution of memory, space usage, and space release. The paper also includes some examples of the discussed strategies and live examples of how these strategies help improve Java application performance with desired low latency and high throughput rates. The study impacts HPC by comparing diverse memory management strategies and providing suggestions to enhance Java application efficiency. Future research on enhancing these strategies and managing new concerns related to Java heap memory is provided

Keywords: Java heap memory, memory leak, garbage collection, JVM, Throughput.

I. INTRODUCTION

Java is widely used in high-performance computing applications where heap memory management is an acceptable skill. Memory management and garbage collection are controlled by the Java Virtual Machine (JVM) and are factors that directly impact application response time and throughput. High workloads that include using many applications and processing large amounts of data and user requests may cause critical issues connected with memory management. The paper presents different approaches for fine-tuning the Java heap memory, which will help prevent memory leakages and irregular garbage collections, among other problems.

II. LITERATURE REVIEW

Java heap memory management has grown in importance in the last few years, particularly for applications handling high traffic, when it comes to performance optimization. The many aspects of Java memory management have been extensively studied, with a focus on garbage collection strategies, memory leak detection, and JVM tuning. In their discussion of the effects of memory leaks in large-scale data applications, Sahin et al. (2016) stress the significance of appropriate JVM configuration management.

The research emphasizes how crucial it is to select the appropriate garbage collection approach to strike a compromise between delay and throughput. A variety of collectors, such as Serial, Parallel, CMS, G1, and ZGC, offer pros and cons of their own. Particularly, CMS and ZGC are commended for their low latency, which makes them appropriate for applications where pause periods must be kept to a minimum. In contrast, Parallel GC is preferred in throughput-oriented scenarios, despite the possibility of lengthier pauses. A thorough analysis of the various garbage collector kinds and how they affect application performance is given by Gupta (2018).
Additionally, programs like Java VisualVM, JProfiler, and YourKit have been quite helpful in locating memory leaks and assessing heap memory utilization, assisting programmers in optimizing memory allocation and averting ongoing performance problems. Research has also demonstrated how important JVM settings are for improving application performance, including memory size and garbage collection thread configuration. It has been demonstrated that properly adjusting these settings can significantly reduce GC pauses and improve system stability, especially in scenarios with a high user traffic and data volume. All things considered, the research shows that efficient management of Java heap memory is essential to preserving the dependability and responsiveness of Java programs, particularly in high-performance computing settings.

III. PROBLEM STATEMENT

The Java applications experience high volumes of data and user requests in high-throughput environments and may be bound by strict performance constraints. However, the default JVM settings may only partially be suitable for these conditions that exert much pressure on heap memory management. Suboptimal memory allocation and garbage collection can lead to frequent pauses and thus negatively impact the application's performance. These interruptions appear as delays where the application has a slower time responding to requests and lower rates of transactions or operations per time. Our extensive measurement investigation has produced several interesting results [1]. Application performance is not necessarily enhanced by larger heap sizes, and heap space problems do not always signify a full heap. Furthermore, modifying the heap structure parameters frequently suffices to fix these issues without expanding the heap. Furthermore, without altering the JVM, VM, or OS kernel, a JVM with a small heap can achieve comparable performance by adjusting the JVM heap topology and garbage collection parameters.
Moreover, failing to free up unused objects, called memory leaks, can gradually take up available memory until the application stops working or encounters an out-of-memory exception. This scenario is particularly undesirable in high-throughput zones where system stability and reliability are critical. To overcome these challenges, a developer must thoroughly learn about the Java heap memory and all the related garbage collection mechanisms, how the memory is allocated, and which specific JVM flags affect the application's performance. These elements must be optimized to ensure no resources are wasted, and the latency is minimal when the application is deployed in an environment with high traffic.

IV. SOLUTIONS

1. *Memory Leak Detection and Prevention*

Memory leaks in Java applications can be defined as a situation where no longer valuable objects are not deallocated. Therefore, only the application utilizes memory resources and could potentially exit. Memory leaks are severe problems and should be identified and addressed,

International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

mainly if memory management is critical in a heavily loaded system [5]. Java Visual VM, JProfiler, and YourKit tools assist in identifying memory leaks, visualizing heap data, identifying referent objects, and observing a difference in memory over time. These tools analyze the objects that are unnecessarily kept and help devs and elopers understand the reasons behind the memory leaks. Memory leaks can be prevented by following the object reference nullification standard to release the space in memory used by unreferenced objects. However, weak references can also be required to avoid memory leaks because objects referred to through weak references can be garbage collected.

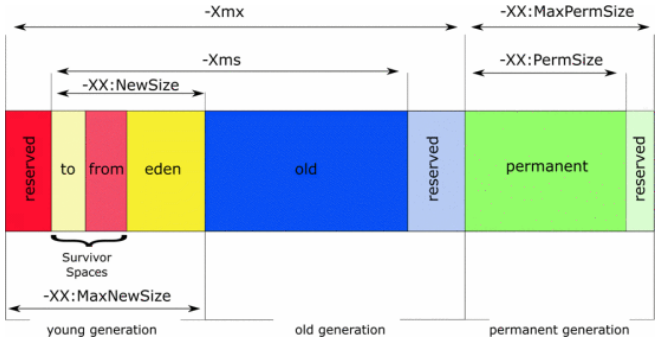

Figure 1: Java memory sections [1]

2. *Garbage Collection Mechanisms*

Several GC mechanisms are implemented in the JVM, and all have strengths and weaknesses. These mechanisms should be straightforward so that the proper adjustments for heap memory can be made.

a) Serial Garbage Collector

The Serial Garbage Collector is a single-threaded collector, which is ideal when a tiny heap is used. It is most suitable for small applications with unnecessary, complicated garbage collection schemes. In the Serial GC, all the garbage collection processes occur in a single thread, which may cause memory clearing to take much time. This usually is acceptable in those low-flow situations where the response time is not critical. However, in high-throughput applications, serial GC can be a problem because while garbage collection is under process, the execution of the application leads to higher latency and lower throughput.

b) Parallel Garbage Collector

The Parallel Garbage Collector (Parallel GC), or the throughput collector, works in throughput systems and is characterized by multiple threads for garbage collection. This approach differs from the Serial GC, where a single thread is used, as the Parallel GC takes advantage of multicore processors to increase the garbage collection rate and process more objects. Dividing the load among several CPU cores, the Parallel GC also significantly shortens garbage collection pauses and increases application throughput. This makes it most appropriate in systems with much computational power, where throughput is all important. However, while the Parallel GC performs very well in high-throughput systems, it is not as efficient in the systems that require low-latency operations because the primary goal of the Parallel GC is to achieve throughput, not to minimize pauses, which in turn may cause long pauses to harm the latency-sensitive applications.

International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

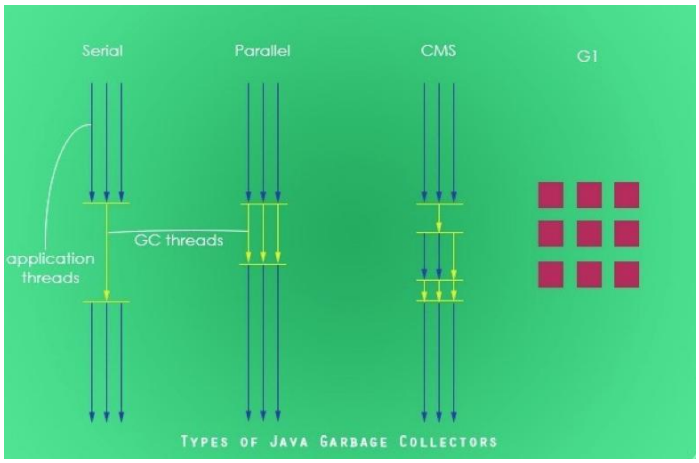

Figure 2: Types of GC[6]

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Concurrent Mark-Sweep (CMS) Garbage Collector mainly works concurrently with the program and is characterized by a minimum of long generational pauses for garbage collection. The CMS GC uses several threads to collect the unreachable objects, so the application continues to run as much as possible without interruption. It lessens the pause times, which can be faster than single-threaded collectors like the Serial GC [6]. However, the CMS GC still results in fragmentation eventually as the heap is not compacted during the GC process. Otherwise, it may lead to out-of-memory errors in the long run since it brings fragmentation. Moreover, the CMS GC requires additional heap space for efficient operation and is, therefore, unsuitable for configurations with a small amount of memory.

4. G1 Garbage Collector

The G1 Garbage Collector (G1 GC) is an advanced garbage collection mechanism for large heaps that balances throughput and pause times. While orthodox garbage collectors partition the heap into young and old generations, G1 GC divides the heap into multiple regions that can be collected separately [6]. This design enables G1 GC to do incremental garbage collection and collection to focus on regions with the first garbage, ensuring that garbage collection pauses do not significantly affect application performance. G1 GC is beneficial when pause times are expected to be short and predictable because the collector offers pause times as a goal to be met. However, to get the most out of G1 GC, some tuning is required to fine-tune the regions' size, heap occupancy, and the desired pause time.

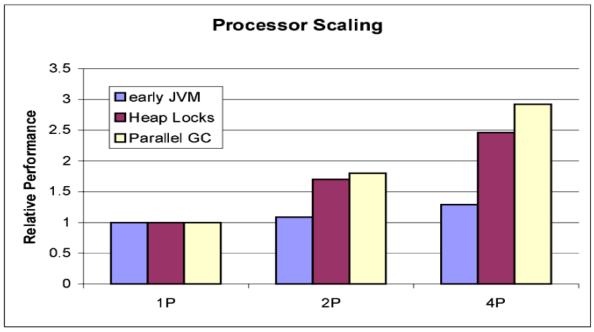

Figure 3: Impact of Parallel Garbage Collection on Processor Scaling [7]

International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

5. Z Garbage Collector (ZGC)

The Z Garbage Collector (ZGC) is a low-latency garbage collector designed to have few ms garbage collection pauses even when the heap is enormous. Like the CMS GC, ZGC performs most of its work concurrently with running applications, with the help of such mechanisms as colored pointers and load barriers. These mechanisms enable ZGC to monitor and manipulate object references effectively without interfering with application processing, avoiding heap fragmentation and sustaining low pause time. This makes ZGC particularly suitable for high-throughput and low-latency applications, which include real-time systems and large-scale data processing. However, as was already mentioned, ZGC is relatively young compared to other garbage collectors, and to achieve the best result, it might be necessary to fine-tune the utilization settings and thoroughly test it in different use cases, especially when it comes to enterprise apps where stability and predictable performance are crucial.

V. ALLOCATION PATTERNS

Java memory allocation is divided into three parts called stack, heap and non-heap. It is allocated as illustrated below [6]

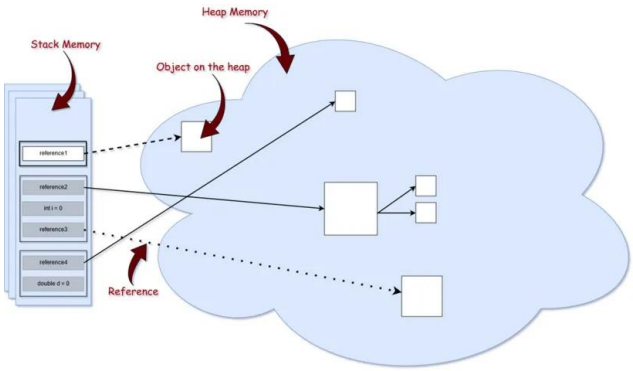

Figure 4: Java memory Allocation[6]

It is necessary to understand allocation patterns to improve the utilization of heap memory in Java applications and under high load conditions. Usually, objects are created in the young generation, where frequent garbage collection occurs. If these objects are not reclaimed for several garbage collection (GC) cycles, they are moved to the old generation, where collections are less frequent but more costly [3]. By studying these patterns, developers can control the values of young and old generation sizes to provide frequent but short Full Garbage Collections in the young generation and less frequent but longer Full Garbage Collections in the old generation. In this way, they can minimize the frequency and length of GC pauses, improving overall application performance [3]. There are techniques like object pooling wherein, instead of repeatedly creating and destroying the objects, the objects are stored in memory for reuse. Such practices help reduce the burden of garbage collectors. In addition, reducing the use of objects by utilizing them where possible and using stack rather than heap where possible can also reduce memory overhead. These strategies not only enhance the use of memory but also enhance the reaction time of the application by putting pressure on JVM's garbage collection, r, resulting in the enhancement of memory usage and, hence t, the overall performance of the application. Another frequently used allocation methodology is lazy allocation [2], a memory management method to maximize resource

International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

utilization. Lazy allocation postpones allocation memory or resources until after a request has been made, rater than doing it right away. This method can assist prevent over-allocation, enhance performance and cut down on needless memory use.

VI. JVM SETTINGS

1. *Heap Size (-Xms and -Xmx)*

Heap size is one of the most crucial JVM parameters directly involving the application's performance. The -Xms and -Xmx parameters determine the amount of memory the JVM allocates to the heap, consisting of the young, old, and meta space generations [4]. It is crucial to properly set these values to get the correct memory consumption and performance. If the heap is too small, there will be too many garbage collection processes, whereas if the heap is too large, the GC pauses will be too long. Therefore, when adjusting the heap size to be proportional to the memory needed by the application and the workload, application latency can be reduced, and out-of-memory errors can be avoided.

2. *Garbage Collection Threads (-XX)*

The -XX: Parallel GC Threads setting controls the number of threads used during garbage collection when multi-threaded GC algorithms such as Parallel GC are used. As for multicore systems, increasing the number of GC threads can improve garbage collection effectiveness due to better load distribution across cores [4]. However, this setting must be carefully tuned to avoid saturating the operation, which may lead to reduced application performance or excessive CPU utilization. In high throughput conditions, increasing or decreasing the number of GC threads in response to hardware and parallelism of the application increases the throughput and minimizes the latency caused by GC during the high load periods.

3. *GC Logging (-Xlog)*

GC logging is activated via the -Xlog: GC option, which is, in fact, very important for controlling JVM in terms of performance. This log of garbage collections includes JVM's pause time, frequency, and memory usage characteristics, which can help analyze JVM's memory management attributes. Information derived from these logs can indicate several performance problems, such as long GC pauses or memory leaks, that may impact application interactivity and system throughput. In GC logs, changes to JVM can be made with the right heap size e choice, algorithm for garbage collection, and other crucial tuning parameters. Additionally, GC logging can help identify patterns in memory use over time and make necessary corrections to enhance the application's efficiency. GC logging should be used as a step within the performance tuning process to ensure that Java applications function efficiently when managed with high throughputs that require memory.

4. *Survivor Ratio (-XX)*

The -XX: Survivor Ratio is another important JVM parameter used to control the amount of memory allocated to the young generation by determining the relationship between the Eden and survivor spaces. This ratio influences the promotion of objects from the young to the old, directly affecting garbage collection efficiency. The desired survivor ratio is appropriate for maintaining objects in the survivor space before they are collected or moved to the next level to cut short complete garbage collections and improve heap usage. If the survivor spaces are too small, objects

International Journal of Core Engineering & Management
Volume-5, Issue-06, September-2018, ISSN No: 2348-9510

may be moved to the old generation before they should be, increasing garbage collection overheads and reducing system performance. On the other hand, large survivor spaces mean that more heap space is allocated than required for the live generation. Therefore, depending on the allocations of an application and the object's lifetime, one can fine-tune the value of the survivor ratio and thus optimize memory usage, enhancing the performance of applications.

VII. IMPACT

The Optimization of Java heap memory significantly impacts the application's performance, especially in environments where the throughput is high, and the consumption of resources is high. Improper heap tuning leads to long GC pause times, which are undesirable in latency and make the application inefficient at handling transaction or data processing loads. One well-known example of how in-memory computing might improve computing efficiency is Spark. Nevertheless, Spark still faces difficulties in making effective use of memory resources. In order to minimize resource consumption and speed up data processing, we provide in this work an adaptive memory tuning strategy for Spark that allows dynamic selection of data compression and serialization techniques. One well-known example of how in-memory computing might improve computing efficiency is Spark[8] Nevertheless, Spark still faces difficulties in making effective use of memory resources. In order to minimize resource consumption and speed up data processing, we provide in this work an adaptive memory tuning strategy for Spark that allows dynamic selection of data compression and serialization techniques. This is because if GC pauses are reduced, the application can achieve higher throughput, thereby enhancing the overall experience of users and the effectiveness of the processes under execution. Session management also plays a critical role in performance optimization in large multi-user environments[9]. It's important to avoid over creating sessions as each session consumes memory. Also keeping session small minimize memory usage and ensures reduced response times. In addition, since memory leaks can be avoided, it is possible to guarantee that memory is released and reused as needed rather than slowly deteriorating from constant use. This stability is critical over long-lived application instances because memory errors and application crashes are disruptive and expensive in production. Thus, reflecting on the allocation patterns, the developers can guarantee that memory consumption is optimized, improving the application's performance and capacity. Specific considerations of garbage collection mechanisms of high generality enable better memory management that is proportional to the application load. Optimizing the values of the JVM settings, such as the heap size, survivor ratios, and GC threads, ensures that Java resources are utilized to their maximum benefit, which enables the application to run at optimal capacity. All these optimizations, taken together, improve application reliability and performance necessary in high throughput applications.

VIII. SCOPE

The techniques applied in the tuning of Java heap memory are general. They can be incorporated into almost any Java application and are focused on high-performance and high-efficiency applications. Heap memory management is critical in high throughput web servers and other business applications like banking and large-scale data processing applications since it profoundly affects how an application can perform many transactions processing, data analysis, and handling multiple requests simultaneously without lagging. This suggests that improving memory profile

management, identifying the most suitable garbage collection methods, and tuning the JVM will improve the capabilities of these systems to meet specific performance requirements. This is particularly so during the high loads where the memory leaks; poor garbage collection may cause the system to hang or slow. Heap size fine-tuning is still the only option for cost optimization in the clouds where resources can be met. This results in the effective use of resources, thus reducing operation costs and achieving the best results for applications since it is a component of memory management by specialists. They also enhance the responsiveness of applications based on workload, which is necessary in areas with scalability and cost of consumption. In addition, these techniques assist in making Java applications sounder and more reliable regardless of future changes in Java or improvements. This makes them capable of real-time performance optimization and long-term software applications used in various and complicated computational environments.

IX. CONCLUSION

- The management of Java heap is a fundamental aspect that can be managed in high-throughput systems to optimize performance.
- Understanding garbage collector types, memory allocation patterns, memory leaks, and JVM tuning is essential for optimizing performance in high-throughput Java systems.
- These strategies are effective in ensuring that Java applications are robust and performant on dynamic computing platforms.
- High-performance computing (HPC) systems can achieve higher throughput and lower latency by reducing GC pauses through proper heap optimization and GC design.
- New features and optimization techniques for Java programs will result from the ongoing evolution of the research of Java heap management in response to technological breakthroughs.
- A wide range of Java applications can use these heap management strategies, but they are especially useful in mission-critical setups like banking and massive data processing.
- As JVM technology develops, new tactics and optimizations to further improve Java application performance will be presented by future studies and updates.

REFERENCES

1. S. Sahin, W. Cao, Q. Zhang and L. Liu, "JVM Configuration Management and Its Performance Impact for Big Data Applications," 2016 IEEE International Congress on Big Data (BigData Congress), San Francisco, CA, USA, Oct. 06, 2016, pp. 410-417, https://ieeexplore.ieee.org/document/7584970.
2. J. Shi, W. Ji, L. Zhang, Y. Gao, H. Zhang and D. Qing, "Profiling and analysis of object lazy allocation in Java programs," 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai, China, July. 21, 2016, pp. 591-596, https://ieeexplore.ieee.org/document/7515964
3. Retheesh Soman "Java Memory Model & Garbage collection | by Retheesh Soman - Medium." 18 Aug. 2018, https://medium.com/@retheesh.soman/java-memory-model-garbage-collection-fae350fe5c56.
4. Thilina Ashen Gamage "Understanding Java Memory Model - Medium." 22 Aug. 2018, https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973.

5. J. Qian, X. Zhou, W. Dang and Z. Wang, "A Specification-Based Approach to the Testing of Java Memory Bloat," 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), Vienna, Austria, 2016, pp. 347-352, https://ieeexplore.ieee.org/document/7589814
6. Alankar Gupta "Java Memory Management - Medium." 24 Aug. 2018, https://medium.com/mindorks/java-memory-management-6e7ccafafcc1
7. K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren and O. Lindholm, "Impact of JIT/JVM optimizations on JAVA application performance," INTERACT-7 2003. Proceedings., Anaheim, CA, USA, 2003, pp. 5-13, doi: 10.1109/INTERA.2003.1192351. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1192351&isnumber=26739
8. Di Chen; Haopeng Chen; Zhipeng Jiang; Yao Zhao An adaptive memory tuning strategy with high performance for Spark October 6, 2017pp 276-286 https://www.inderscience.com/offers.php?id=86970
9. V. K. Myalapalli and S. Geloth, "High performance JAVA programming," April 16, 2015 https://ieeexplore.ieee.org/document/7087004