Java Garbage Collectors, their working and comparisons

Anmol Sehgal · Follow

Published in Geek Culture

25 min read · Jul 3, 2023

Java garbage collectors (GCs) are automatic memory management components responsible for reclaiming memory occupied by objects that are no longer in use. They play a crucial role in managing Java's dynamic memory allocation, allowing developers to focus on application logic without manual memory deallocation. GCs are needed to prevent memory leaks, optimize memory usage, and ensure the overall performance and stability of Java applications.

Before the advent of GCs, manual memory management was the norm in programming languages. Developers had to explicitly allocate and deallocate memory for objects, which often led to errors like memory leaks or dangling references. Such manual memory management was error-prone and time-consuming, requiring careful tracking and release of objects.

Java introduced automatic memory management through its GC mechanism, relieving developers from manual memory deallocation. GCs periodically identify and release memory occupied by objects that are no longer reachable or in use. This

process involves several stages, including marking reachable objects, identifying unreachable objects, and reclaiming their memory.

In this document, we will cover the 10 most widely used GCs:

1. Serial GC: A simple, single-threaded GC algorithm suitable for small applications or systems with low memory requirements.
2. Parallel GC: Uses multiple threads to speed up garbage collection, making it more efficient for applications that generate a large amount of garbage.
3. CMS (Concurrent Mark-Sweep) GC: Provides low-latency garbage collection by performing most of the collection work concurrently with the application's execution.
4. G1 (Garbage-First) GC: A server-style GC that divides the heap into regions and performs garbage collection on smaller subsets of the heap to minimize pauses and improve throughput.

5. ZGC: Designed for applications requiring ultra-low pause times, ZGC performs concurrent garbage collection without significant interruption to the application.

6. Shenandoah GC: Another concurrent GC algorithm, Shenandoah aims to reduce pause times by performing garbage collection concurrently with the application.

7. Epsilon GC: A no-op GC algorithm useful for testing and scenarios where the application handles memory management independently.

8. Azul C4 GC: Designed for high-performance and low-latency requirements, C4 GC utilizes pauseless, concurrent collection techniques.

9. IBM Metronome GC: An experimental real-time GC algorithm that focuses on predictable and consistent pause times.

10. SAP Garbage Collector: A concurrent GC algorithm optimized for large heap sizes and low-latency requirements.

. . .

Serial GC

Serial GC is a garbage collector algorithm used in Java for automatic memory management. It is a simple and straightforward algorithm that works efficiently for small-scale applications or systems with limited memory requirements. In this document, we will explore how the Serial GC algorithm works, its benefits, and considerations for its usage.

How Serial GC Works

The Serial GC algorithm operates in a stop-the-world manner, meaning it pauses the application's execution during garbage collection. When the Serial GC is triggered, it performs the following steps:

1. Initial Mark: The algorithm identifies and marks all objects directly referenced by the application's active threads. This marking process is performed while the application is paused briefly.

2. Marking: The algorithm traverses the object graph starting from the marked objects and identifies all reachable objects. It marks these reachable objects as live.

3. Remark: After the initial marking, the algorithm allows the application to resume briefly and continues marking any objects that became reachable during the pause.

4. Sweep and Compact: Once the marking is complete, the algorithm sweeps the memory regions, deallocating memory occupied by unreferenced objects. It then compacts the memory by moving live objects together, creating a contiguous free memory space.

5. Free Memory: After compaction, the algorithm updates the free memory space pointer to indicate the new location for object allocation.

Pros of Using Serial GC:

1. Simplicity: The Serial GC is straightforward to implement and understand, making it an ideal choice for beginners or smaller applications.

2. Low Overhead: The algorithm has lower CPU and memory overhead compared to more complex GC algorithms, making it suitable for resource-constrained environments.

3. Predictable Pauses: As a stop-the-world collector, the Serial GC provides predictable pauses during garbage collection, allowing for better control over the application behavior.

4. Single-threaded Execution: The Serial GC performs garbage collection using a single thread, ensuring determinism and avoiding potential multi-threading issues.

Cons of Using Serial GC:

1. Longer Pause Times: The stop-the-world nature of the Serial GC can result in longer pause times, causing interruptions in application responsiveness for larger heaps or memory-intensive applications.

2. Limited Scalability: The single-threaded nature of the Serial GC limits its scalability on modern multi-core processors, where parallelism can significantly improve GC performance.

3. Not Suitable for Large Applications: The Serial GC may not be the optimal choice for large-scale applications or systems with high memory requirements, as its performance may degrade due to increased garbage collection times.

Conclusion

Serial GC is a simple and efficient garbage collector algorithm suitable for smaller applications or environments with limited memory resources. Its straightforward implementation and low overhead make it an attractive choice for resource-constrained systems. However, it is important to consider its limitations, such as longer pause times and limited scalability, when evaluating its suitability for larger or memory-intensive applications. Understanding the characteristics and trade-offs of the Serial GC algorithm can help developers make informed decisions about GC selection based on their specific application requirements.

. . .

Parallel GC

Parallel GC is a garbage collector algorithm used in Java for automatic memory management. It is designed to take advantage of multi-core processors and provides improved garbage collection performance by parallelizing certain tasks. In this document, we will delve into the Parallel GC algorithm, and its workings in simple terms, and discuss the benefits and considerations of its usage.

How Parallel GC Works

The Parallel GC algorithm shares similarities with the Serial GC algorithm but introduces parallelism to expedite garbage collection. It operates in a stop-the-world manner, temporarily pausing the application's execution. The steps involved in the Parallel GC algorithm are as follows:

1. Initial Mark: Similar to the Serial GC, the algorithm identifies and marks all objects directly referenced by the application's active threads. This marking

process occurs while the application is paused momentarily.

2. Concurrent Marking: While the application is running, the algorithm concurrently marks reachable objects in the memory, utilizing multiple threads. This concurrent marking phase reduces the overall pause time required for garbage collection.

3. Remark: After the concurrent marking phase, the algorithm allows the application to continue running while it performs a final marking process to account for objects that may have become reachable during the concurrent phase.

4. Concurrent Sweep and Compaction: The Parallel GC concurrently sweeps and deallocates memory regions occupied by unreferenced objects. Simultaneously, it compacts the memory, moving live objects together to create contiguous free memory space. This concurrent sweep and compaction phase helps reduce the pause time further.

5. Free Memory: After compaction, the algorithm updates the free memory space pointer to indicate the new location for object allocation.

Pros of Using Parallel GC:

1. Improved Throughput: Parallel GC leverages multiple threads to execute garbage collection tasks concurrently, leading to improved throughput by utilizing the processing power of multi-core processors.

2. Reduced Pause Times: By parallelizing the marking, sweeping, and compaction phases, the Parallel GC minimizes the pause times required for garbage collection, resulting in improved application responsiveness.

3. Scalability: The algorithm's ability to utilize multiple threads makes it highly scalable, allowing it to handle larger heaps and memory-intensive applications more efficiently.

4. Enhanced Performance: Parallel GC performs well in scenarios where the application generates a significant amount of garbage, as it can leverage

parallelism to expedite garbage collection and keep up with memory demands.

Cons of Using Parallel GC:

1. Increased CPU Utilization: The Parallel GC algorithm utilizes multiple threads, leading to higher CPU utilization compared to single-threaded garbage collection algorithms. This increased utilization may affect the overall system performance for applications with limited CPU resources.

2. Longer Individual Pause Times: While the overall pause time may be reduced, the individual pause times for each garbage collection cycle might be longer compared to other algorithms. This aspect may impact the responsiveness of time-sensitive applications.

3. Not Suitable for Small Systems: The Parallel GC algorithm's multi-threaded nature and increased resource utilization make it less suitable for small-scale systems with limited resources or single-core processors.

Conclusion

Parallel GC is a garbage collector algorithm in Java that leverages parallelism to enhance garbage collection performance. By utilizing multiple threads, it achieves improved throughput, reduced pause times, and better scalability for memory-intensive applications. However, it is essential to consider the increased CPU utilization and potentially longer individual pause times when evaluating the suitability of Parallel GC for specific application scenarios. Understanding the characteristics and trade-offs of the Parallel GC algorithm is crucial for making informed decisions regarding garbage collector selection based on the specific requirements of the application.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate

. . .

CMS (Concurrent Mark-Sweep) GC

The Concurrent Mark-Sweep (CMS) Garbage Collector is a popular garbage collection algorithm used in Java to manage memory and reclaim unused objects. Unlike the stop-the-world approach of some other garbage collectors, CMS aims to minimize application pauses by running certain phases concurrently with the application threads. This document provides an in-depth overview of CMS, explaining its algorithm, benefits, and considerations for usage.

Algorithm Overview:

The CMS algorithm consists of several concurrent and stop-the-world phases:

1. Initial Mark: Pauses the application briefly to identify objects directly reachable from the root set.
2. Concurrent Mark: Concurrently traverses object graphs, marking objects that are still in use.
3. Concurrent Preclean: Continues marking objects concurrently while accounting for changes made during concurrent marking.

4. Final Remark: Pauses the application again to identify objects modified during concurrent marking and completes marking.

5. Concurrent Sweep: Concurrently reclaims memory by sweeping through and freeing unused objects.

6. Concurrent Reset: Concurrently resets internal data structures and prepares for the next garbage collection cycle.

How CMS Works

In simpler terms, CMS aims to perform garbage collection with minimal impact on application responsiveness:

1. The initial mark and final remark phases cause short pauses as they require analyzing root objects and marking reachable objects.

2. Concurrent marking occurs alongside application threads, identifying additional reachable objects.

3. The concurrent preclean phase handles any object modifications during concurrent marking.

4. Concurrent sweeping frees up memory by reclaiming unused objects while the application continues running.

5. Concurrent reset prepares the garbage collector for the next cycle, ensuring it is ready for future garbage collection.

Pros of Using CMS:

- Reduced Pause Times: CMS aims to minimize pauses by performing garbage collection concurrently with the application, resulting in better application responsiveness.

- Improved Scalability: CMS is well-suited for large applications with high thread concurrency, as it strives to run concurrently with application threads.

- Effective for Mixed Workloads: CMS performs well in scenarios where the application generates a significant amount of short-lived objects.

Cons and Considerations:

- Increased CPU Utilization: Concurrent execution can lead to higher CPU usage due to the additional threads involved in garbage collection.
- Fragmentation Concerns: CMS may suffer from memory fragmentation issues, leading to less efficient memory utilization.
- Limited Pause Reduction: While CMS reduces pauses compared to other algorithms, it may not eliminate pauses entirely, and long-running concurrent phases can still impact application performance.
- Deprecated in Java 9+: CMS is deprecated in Java 9 and later versions, with the intention to be removed in future releases. The introduction of the Garbage-First (G1) GC aims to provide a more efficient and scalable alternative.

Conclusion

The Concurrent Mark-Sweep (CMS) Garbage Collector in Java offers a balance between pause reduction and application responsiveness by performing concurrent garbage collection alongside the application threads. It reduces pause times,

improves scalability, and works well for mixed workloads. However, it comes with considerations such as increased CPU utilization, potential fragmentation issues, and the fact that it has been deprecated in recent Java versions. Understanding the characteristics and trade-offs of CMS is crucial for making informed decisions when selecting a garbage collector for your Java application.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate

. . .

G1 (Garbage-First) GC

The Garbage-First (G1) Garbage Collector is a generational garbage collection algorithm introduced in Java 7. It is designed to provide better performance and lower pause times for large heap sizes, making it suitable for modern Java

applications. In this document, we will explore the workings of the G1 GC, its benefits, and considerations for its usage.

Overview of G1 GC

The G1 GC is a parallel and concurrent garbage collector that divides the heap into multiple regions. It operates based on the concept of regions, where each region is a fixed-size block of memory. The heap is dynamically divided into Eden regions, survivor regions, and old regions. The G1 GC collects garbage in a region-based manner, allowing it to prioritize the most heavily garbage-filled regions first.

How G1 GC Works

1. Initial Marking: G1 GC starts by identifying the root objects directly reachable from application threads, such as static variables and reference objects.

2. Concurrent Marking: G1 GC concurrently marks live objects in the heap while the application threads continue to run. It uses a series of marking cycles to traverse the object graph, identifying live objects and updating the mark bits accordingly.

3. Remark: Once the concurrent marking phase is complete, G1 GC performs a stop-the-world remark phase to process any remaining objects that were modified during the concurrent marking.

4. Cleanup: G1 GC performs a cleanup phase to reclaim memory from the garbage-collected regions. It selects regions with the most garbage and compacts live objects into fewer regions to reduce fragmentation.

5. Evacuation: G1 GC uses the evacuation process to transfer live objects from one region to another, ensuring that regions are filled with mostly live objects and reducing the need for full garbage collections.

Pros of Using G1 GC

- Improved Pause Times: G1 GC aims to provide more predictable and shorter pause times, especially for applications with large heaps, by minimizing the duration of stop-the-world garbage collection pauses.

- Adaptive Behavior: G1 GC adjusts its heap partitioning and collection strategies based on the application's workload, allowing it to dynamically adapt to changing memory usage patterns.
- Better Utilization of Hardware Resources: G1 GC utilizes multiple threads for parallel garbage collection, which can lead to better utilization of available CPU resources, resulting in improved application throughput.
- Reduced Fragmentation: G1 GC's compacting algorithm reduces memory fragmentation by reclaiming memory from regions with a high garbage-to-live object ratio.

Cons of Using G1 GC

- Increased Overhead: Compared to other garbage collectors, G1 GC may introduce a slightly higher CPU overhead due to its more complex heap management and concurrent marking algorithms.

- Longer Application Warm-Up: G1 GC may exhibit longer warm-up times as it gathers statistics and adapts its behavior to optimize garbage collection performance.

Conclusion

The G1 Garbage Collector is a powerful garbage collection algorithm introduced in Java 7, specifically designed to address the challenges of managing large heap sizes with shorter pause times. With its concurrent and region-based approach, G1 GC offers improved garbage collection performance for modern Java applications. It is recommended to evaluate the specific requirements and characteristics of your application before choosing the G1 GC, ensuring it aligns with your desired performance goals and resource utilization.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High

. . .

ZGC

ZGC (Z Garbage Collector) stands out as a modern, low-latency garbage collector designed to handle large heaps with minimal pauses. Introduced in JDK 11, ZGC aims to provide excellent responsiveness and scalability for applications with large memory requirements. This document provides an overview of ZGC, explaining how it works, and explores its advantages and limitations.

What is ZGC and How is it Used?

ZGC is a garbage collector specifically designed for large heaps, typically ranging from a few gigabytes to several terabytes. It focuses on minimizing the impact of garbage collection pauses, making it suitable for latency-sensitive applications. ZGC is intended to be used in scenarios where applications require consistent response times and have stringent latency requirements.

How Does ZGC Work?

At a high level, ZGC employs a concurrent garbage collection approach, meaning it performs garbage collection concurrently with the application's execution. This concurrent process avoids long pauses that can disrupt application responsiveness. ZGC divides the heap into fixed-size regions and concurrently identifies and relocates live objects while the application continues to run. The key steps involved in the ZGC algorithm are as follows:

1. Initial Marking: ZGC starts by identifying the root objects and marking them as live. This phase is performed quickly and involves minimal impact on application execution.

2. Concurrent Marking: After the initial marking, ZGC performs a concurrent marking phase to identify additional live objects. This marking process is carried out alongside the application's execution, ensuring minimal impact on latency.

3. Remark: Once the concurrent marking phase is completed, ZGC performs a short stop-the-world remark phase to capture any objects that may have been

modified during the concurrent marking process.

4. Concurrent Relocation: In this phase, ZGC relocates live objects to new regions of memory to free up fragmented memory. The relocation process is done concurrently with the application, ensuring minimal impact on responsiveness.

5. Final Marking: ZGC performs a final marking phase to identify any objects that were missed during the concurrent marking and relocation phases. This step helps ensure the consistency of the garbage collection process.

6. Reference Processing: ZGC handles reference processing concurrently to keep track of live objects and facilitate their relocation.

Pros of Using ZGC:

- Low Latency: ZGC is specifically designed to minimize garbage collection pauses, resulting in low latency and consistent application responsiveness.

- Scalability: ZGC is capable of efficiently handling large heaps, making it suitable for applications with substantial memory requirements.

- Concurrent Execution: The concurrent garbage collection approach allows ZGC to perform collection work simultaneously with application execution, reducing the impact on throughput.

Cons of Using ZGC:

- Overhead: Like any garbage collector, ZGC incurs some overhead due to the concurrent marking and relocation processes. This overhead can be noticeable in certain scenarios.

- Limited Platforms: Currently, ZGC is only supported on certain platforms, such as Linux x64 and Linux ARM64.

Conclusion

ZGC is a cutting-edge garbage collector for Java that addresses the challenges of managing large heaps with minimal latency impact. Its concurrent approach to garbage collection allows it to provide low-pause times, making it suitable for latency-sensitive applications. While it has some limitations and platform

restrictions, ZGC's performance benefits and scalability make it a compelling choice for modern Java applications with stringent latency requirements.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate

. . .

Shenandoah GC

Shenandoah GC is a garbage collector designed for a low-pause time in Java applications. It is an open-source garbage collector that aims to reduce the impact of garbage collection on application responsiveness. Shenandoah GC is available as an experimental feature in OpenJDK 8 and 11 and as a production feature starting from OpenJDK 12.

Usage and Benefits

Shenandoah GC is primarily used in applications that require low latency and consistent pause times. It is suitable for applications with large heaps and stringent response time requirements. By minimizing pause times, Shenandoah GC enables applications to maintain high throughput and responsiveness even under high load.

How Shenandoah GC Works

Shenandoah GC employs a unique approach called concurrent evacuation. Unlike traditional garbage collectors that halt the application to perform garbage collection, Shenandoah GC works concurrently with the application threads. It allows the application to continue running while the garbage collector performs its tasks, resulting in significantly shorter pause times.

Here's a simplified explanation of how Shenandoah GC works:

1. Initial Mark: Shenandoah GC starts by scanning the root objects and marking them as live. This initial marking phase is done with a pause, but it is typically very short.

2. Concurrent Mark: While the application is running, Shenandoah GC continues marking live objects concurrently. It traces object graphs and identifies all reachable objects in the heap.

3. Concurrent Evacuation: After marking, Shenandoah GC proceeds with the evacuation phase. It identifies regions in the heap that contain garbage and reclaims them. Meanwhile, it also copies live objects to new regions.

4. Update References: During the evacuation phase, Shenandoah GC updates references to the newly copied objects. It ensures that the application's references point to the correct memory locations.

5. Final Mark: Shenandoah GC performs a final marking phase to identify any objects that have become live during the concurrent phases. This phase involves a short pause.

6. Cleanup: Finally, Shenandoah GC cleans up the remaining garbage and frees up memory.

Pros of using Shenandoah GC:

- Low pause times: Shenandoah GC reduces pause times significantly, resulting in improved application responsiveness and reduced latency.
- Scalability: It performs well with large heap sizes and can handle applications with substantial memory requirements.
- Concurrent execution: Shenandoah GC works concurrently with the application threads, allowing the application to continue running with minimal interruptions.

Cons of using Shenandoah GC:

- Increased CPU overhead: Shenandoah GC may introduce additional CPU overhead compared to other garbage collectors due to its concurrent execution.
- Young generation collection: Shenandoah GC focuses on the older generation, so it relies on a separate collector for the young generation, which may lead to some overhead.

Conclusion:

Shenandoah GC is a valuable option for applications that require low latency and consistent pause times. By employing concurrent evacuation, it significantly reduces pause times and improves application responsiveness. However, it's important to consider the specific characteristics of your application and evaluate the trade-offs, such as increased CPU overhead, before deciding to use Shenandoah GC. Conducting performance testing and analysis on your specific workload will help determine if Shenandoah GC is the right choice for your application.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate

. . .

Epsilon GC

Epsilon GC is a special-purpose garbage collector introduced in JDK 11. It is designed for specific use cases where garbage collection is not required or can be completely eliminated. Epsilon GC is an experimental feature that serves as a "no-op" garbage collector, meaning it does not perform any garbage collection activities. It is primarily used for performance testing, short-lived applications, or situations where the application manages memory explicitly.

Usage and Benefits:

Epsilon GC is used in scenarios where the application has a short lifespan or does not produce significant garbage. It is particularly useful in performance testing environments to measure the baseline performance of an application without any garbage collection overhead. By eliminating garbage collection entirely, Epsilon GC can provide insights into the true performance characteristics of an application and help identify potential bottlenecks unrelated to garbage collection.

How Epsilon GC Works:

Epsilon GC works in a straightforward manner. It is a “no-op” garbage collector, which means it does not perform any garbage collection activities. It allows objects to be allocated in memory without tracking their lifetime or performing any cleanup. As a result, memory allocations occur rapidly without any overhead associated with garbage collection.

Pros of using Epsilon GC:

1. Performance testing: Epsilon GC is ideal for performance testing environments where garbage collection overhead needs to be eliminated. It provides a baseline measurement of an application’s performance without any interference from garbage collection activities.

2. Short-lived applications: Applications that have a short lifespan and produce minimal garbage can benefit from using Epsilon GC. Since no garbage collection is performed, it avoids unnecessary overhead and allows the application to run with maximum efficiency.

3. Memory management control: Epsilon GC allows developers to have complete control over memory management. It enables explicit memory allocation and deallocation, which can be advantageous in certain specialized applications.

Cons of using Epsilon GC:

1. Memory leaks: Since Epsilon GC does not perform garbage collection, it does not reclaim memory allocated to objects. If an application has memory leaks or excessive memory usage, Epsilon GC will not free up memory automatically, leading to potential out-of-memory errors.

2. Limited use cases: Epsilon GC is not suitable for long-running applications or applications that generate a significant amount of garbage. It lacks the ability to reclaim memory and can result in memory exhaustion if not used appropriately.

3. Lack of runtime optimization: Epsilon GC does not provide any runtime optimizations or adaptive behaviors commonly found in other garbage collectors. It does not dynamically adjust its behavior based on the application's memory usage patterns.

Conclusion:

Epsilon GC serves as a specialized garbage collector for specific use cases where garbage collection is not required or can be completely eliminated. It is primarily used for performance testing, short-lived applications, or situations where explicit memory management is preferred. While Epsilon GC eliminates garbage collection overhead, it is important to consider the limitations and potential memory management challenges associated with its use. Careful analysis and consideration of the application's memory requirements and usage patterns are necessary to determine if Epsilon GC is suitable for a given scenario.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate
Epsilon GC	N/A	N/A	N/A	Very High	Performance testing, Memory allocation analysis	Very Low

. . .

Azul C4 GC

Azul C4 GC, also known as the Continuously Concurrent Compacting Collector, is a garbage collector designed for high-performance Java applications. It is developed and provided by Azul Systems, a company specializing in Java runtime technologies. Azul C4 GC aims to minimize pause times, improve throughput, and provide predictable performance for applications with large heaps and high memory requirements.

Usage and Benefits:

Azul C4 GC is particularly useful for applications that require low latency and consistent performance. It is commonly used in latency-sensitive industries such as financial services, e-commerce, and gaming, where even small pauses in application execution can have a significant impact. The main advantages of using Azul C4 GC include reduced pause times, improved application throughput, and enhanced scalability.

How Azul C4 GC Works:

Azul C4 GC employs a unique algorithm that enables it to perform garbage collection concurrently with the application's execution. It works in the following steps:

1. Initial marking: Azul C4 GC starts by identifying the live objects in the heap, marking them as live, and recording their references.
2. Concurrent marking: While the application continues to run, Azul C4 GC concurrently traces object references and updates the marking information. This allows it to keep track of live objects without requiring a stop-the-world pause.
3. Remark phase: Once the concurrent marking is complete, Azul C4 GC performs a remark phase to handle any changes in the object graph that may have occurred during the concurrent marking phase.

4. Concurrent relocation: Azul C4 GC then proceeds to relocate live objects in the heap, compacting them to eliminate fragmentation and improve memory locality. This relocation process is performed concurrently with the application, minimizing pause times.

5. Final remark: After the concurrent relocation, Azul C4 GC performs a final remark to capture any additional changes to the object graph.

Pros of using Azul C4 GC:

1. Low pause times: Azul C4 GC minimizes pause times by performing most of the garbage collection work concurrently with the application's execution. This results in improved application responsiveness and reduced impact on user experience.

2. Predictable performance: Azul C4 GC provides predictable performance characteristics, ensuring consistent application behavior even under high load and memory pressure. It aims to deliver consistent response times and throughput, crucial for latency-sensitive applications.

3. Scalability: Azul C4 GC is designed to handle large heaps and high memory requirements efficiently. It can scale effectively to accommodate applications with substantial memory footprints, allowing them to operate smoothly without significant pauses.

Cons of using Azul C4 GC:

1. Third-party dependency: As Azul C4 GC is a proprietary solution provided by Azul Systems, it introduces a dependency on a specific vendor. This can limit the portability of the application and tie it to Azul Systems' runtime environment.

2. Potential licensing costs: Depending on the specific usage and licensing terms, there may be associated costs for using Azul C4 GC. It is important to consider the licensing implications when evaluating the feasibility of adopting this garbage collector.

3. Configuration complexity: Configuring Azul C4 GC optimally for a specific application may require expertise and understanding of the underlying

algorithms. The complexity of tuning the collector to achieve optimal performance can be a challenge for developers.

Conclusion:

Azul C4 GC offers a compelling solution for applications that demand low latency, consistent performance, and efficient memory management. Its concurrent garbage collection algorithm minimizes pause times and provides predictable behavior, making it suitable for latency-sensitive industries. However, it is essential to consider vendor dependency, potential licensing costs, and the need for expert configuration when deciding to use Azul C4 GC in a Java application. Proper evaluation and understanding of the specific requirements.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate
Epsilon GC	N/A	N/A	N/A	Very High	Performance testing, Memory allocation analysis	Very Low
Azul C4 GC	Large	Very Short	High	Very High	Enterprise applications, Cloud environments	Low to moderate

. . .

IBM Metronome GC

IBM Metronome GC is a real-time garbage collector designed for Java applications that require strict determinism and low pause times. It is developed by IBM and focuses on providing predictable execution in real-time and embedded systems. Metronome GC is suitable for applications where meeting strict timing requirements is critical, such as robotics, avionics, and industrial control systems.

Usage and Benefits:

IBM Metronome GC is specifically designed for real-time applications that require deterministic behavior and minimal pause times. It offers several benefits, including:

1. Deterministic execution: Metronome GC guarantees predictable execution, ensuring that an application meets its timing requirements consistently. This is crucial in real-time systems where precise timing is essential for correct functionality.

2. Low pause times: The collector aims to minimize pause times by utilizing a concurrent and incremental garbage collection approach. It strives to keep application pauses short and predictable, allowing for smoother execution and responsiveness.

3. Real-time system support: Metronome GC is tailored for real-time and embedded systems, providing reliable and predictable behavior in environments with strict timing constraints.

How IBM Metronome GC Works:

Metronome GC operates on the principle of incremental and concurrent garbage collection. It follows these key steps:

1. Initial marking: The collector starts by identifying live objects by traversing the object graph, marking them as live, and recording their references.

2. Concurrent marking: While the application continues to execute, Metronome GC performs concurrent marking, tracing object references and updating the marking information. This process ensures the accurate identification of live objects without requiring a stop-the-world pause.

3. Incremental relocation: Once the concurrent marking phase is complete, Metronome GC performs an incremental relocation of live objects. It relocates a portion of objects at a time during application execution to minimize pause times. This process is repeated incrementally until all live objects are relocated.

4. Reference updates: During relocation, Metronome GC updates object references to reflect the new object locations accurately. This step ensures that the

application can access relocated objects correctly.

5. Collection completion: After all live objects are relocated, the garbage collector finalizes the relocation process and frees memory occupied by unreachable objects. This allows the application to utilize memory more efficiently.

Pros of using IBM Metronome GC:

1. Deterministic execution: Metronome GC guarantees predictable execution and adherence to strict timing requirements. It is well-suited for applications where consistent timing is critical for correct functioning.

2. Low pause times: The collector aims to keep pause times minimal, ensuring that the application remains responsive and meets its real-time requirements.

3. Real-time system support: Metronome GC is specifically designed for real-time and embedded systems, providing reliable and predictable behavior in such environments.

Cons of using IBM Metronome GC:

1. Specialized use case: Metronome GC is primarily intended for real-time systems with strict timing constraints. It may not be suitable or provide significant advantages for general-purpose applications that do not require deterministic behavior.

2. Potential complexity: Configuring and fine-tuning Metronome GC for optimal performance may require specialized knowledge and expertise. Developers need to have a deep understanding of real-time systems and the collector's intricacies.

3. Limited ecosystem support: As a specialized garbage collector, the ecosystem and tooling support for Metronome GC may be relatively limited compared to mainstream collectors like the ones provided by OpenJDK or Oracle.

Conclusion:

IBM Metronome GC offers deterministic execution and low pause times, making it suitable for real-time and embedded systems with strict timing requirements. It guarantees predictable behavior, ensuring consistent application performance.

However, it is essential to consider the specialized nature of Metronome GC and the associated complexity in configuration and ecosystem support.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate
Epsilon GC	N/A	N/A	N/A	Very High	Performance testing, Memory allocation analysis	Very Low
Azul C4 GC	Large	Very Short	High	Very High	Enterprise applications, Cloud environments	Low to moderate
IBM Metronome GC	N/A	Very Short	Very High	Very High	Real-time applications, Predictable latency requirements	Very Low

. . .

SAP GC

SAP Garbage Collector (GC) is a custom garbage collector developed by SAP for use with the SAP JVM (Java Virtual Machine). It is designed to optimize memory

management and garbage collection for SAP applications running on the SAP NetWeaver platform. SAP GC aims to improve application performance and scalability by efficiently managing memory and reducing pause times.

Usage and Benefits:

SAP GC is specifically tailored for SAP applications, which are often large-scale enterprise systems that handle extensive data and user loads. It offers several benefits, including:

1. Memory efficiency: SAP GC optimizes memory management by minimizing memory footprint and reducing unnecessary object allocations. This helps improve overall memory efficiency and reduces the risk of memory-related issues.

2. Reduced pause times: The collector is designed to minimize pause times during garbage collection. By employing concurrent and incremental collection techniques, SAP GC aims to keep pauses short and predictable, enabling applications to remain responsive and maintain consistent performance.

3. Scalability: SAP GC is optimized for scalability, allowing SAP applications to handle large data volumes and user loads effectively. It ensures efficient memory management even under high workloads, enabling applications to scale smoothly.

How SAP GC Works:

SAP GC utilizes a combination of generational and concurrent garbage collection techniques to manage memory efficiently. Here's an overview of its functioning in layman's terms:

1. Generational collection: SAP GC divides the heap into generations based on the age of objects. Younger objects are allocated in the young generation, while longer-lived objects are moved to the old generation. The young generation undergoes more frequent garbage collection, while the old generation is collected less frequently.

2. Concurrent collection: SAP GC performs concurrent garbage collection alongside the application's execution. While the application continues to run,

the collector identifies and marks live objects, ensuring that they are not mistakenly treated as garbage. This concurrent marking reduces the impact on application responsiveness.

3. Incremental collection: To minimize pause times, SAP GC performs garbage collection incrementally. It divides the collection process into small units and interleaves garbage collection with application execution. This approach allows the collector to spread the work over multiple short pauses, ensuring that the pauses do not significantly impact application performance.

4. Compaction: During garbage collection, SAP GC compacts the memory space by rearranging live objects, eliminating fragmentation, and improving memory locality. Compaction reduces the memory footprint and improves the efficiency of memory access.

Pros of using SAP GC:

1. Memory efficiency: SAP GC optimizes memory usage, reducing the memory footprint of SAP applications. This results in better overall performance and

resource utilization.

2. Reduced pause times: The concurrent and incremental collection techniques employed by SAP GC minimize pause times during garbage collection. This enables applications to maintain responsiveness and provide consistent performance.

3. Scalability: SAP GC is designed to handle large-scale enterprise applications with high data volumes and user loads. It ensures efficient memory management, allowing applications to scale effectively.

Cons of using SAP GC:

1. Limited to SAP applications: SAP GC is specifically designed for SAP applications running on the SAP NetWeaver platform. It may not provide significant advantages or be applicable to general-purpose Java applications outside the SAP ecosystem.

2. Vendor dependency: SAP GC is developed and maintained by SAP, which introduces a vendor dependency. Applications using SAP GC may need to rely on SAP's JVM and runtime environment.

3. Ecosystem support: The availability of tools, documentation, and community support for SAP GC may be more limited compared to mainstream garbage collectors.

Conclusion:

SAP GC is a custom garbage collector designed for SAP applications, aiming to optimize memory management, reduce pause times, and improve scalability. It offers memory efficiency, reduced pauses, and scalability benefits specifically tailored to SAP enterprise systems. However, its usage is limited to SAP applications, and it introduces a dependency on SAP's JVM and runtime environment. It is crucial to consider the specific requirements and compatibility with the SAP ecosystem before opting for SAP GC in a Java application.

GC Type	Heap Size Support	Pause Times	Throughput	Performance	Application	CPU Overhead
Serial GC	Small to medium	Longer	Low	Lower	Single-threaded applications, Development environments	Low
Parallel GC	Medium to large	Moderate	High	Higher	Batch processing, Scientific computing, Data analysis	Moderate
CMS GC	Medium to large	Moderate	Moderate	Moderate	Web applications, Medium-sized enterprise systems	Moderate
G1 GC	Medium to large	Short to medium	High	Higher	Mixed workloads, Large enterprise systems	Moderate to High
Z GC	Large	Very Short	High	Very High	Latency-sensitive applications, Large-scale systems	Low to moderate
Shenandoah GC	Medium to large	Very Short	High	Very High	Low-latency applications, Large-scale systems	Low to Moderate
Epsilon GC	N/A	N/A	N/A	Very High	Performance testing, Memory allocation analysis	Very Low
Azul C4 GC	Large	Very Short	High	Very High	Enterprise applications, Cloud environments	Low to moderate
IBM Metronome GC	N/A	Very Short	Very High	Very High	Real-time applications, Predictable latency requirements	Very Low
SAP GC	Large	Short to medium	High	High	Enterprise applications, SAP environments	Moderate to High

. . .

Conclusion

In conclusion, we have examined several popular garbage collectors in the Java ecosystem, including Serial GC, Parallel GC, CMS GC, G1 GC, ZGC, Shenandoah GC, Epsilon GC, Azul C4 GC, IBM Metronome GC, and SAP GC. Each garbage collector

has its own strengths and weaknesses, making them suitable for different scenarios and application requirements.

Here is a summary of the key points from the comparison:

- Serial GC: Suitable for small applications with limited resources and low pause time requirements. It operates using a single thread, which can limit scalability and performance.
- Parallel GC: Designed for multi-core systems, providing improved throughput and reduced garbage collection times. It is well-suited for applications with high computational loads.
- CMS GC: Known for its low-latency garbage collection, making it suitable for applications that prioritize responsiveness and have large heap sizes. However, it can introduce fragmentation and may have higher overall CPU usage.
- G1 GC: Provides a balance between throughput and latency by dividing the heap into smaller regions. It is well-suited for large applications with dynamic

memory requirements and can adapt to changing workloads.

- ZGC: Offers ultra-low pause times by performing concurrent garbage collection, making it suitable for applications with strict latency requirements. It can handle large heap sizes efficiently but may have a higher CPU overhead.
- Shenandoah GC: Designed to minimize pause times and provide predictable latencies, making it suitable for large applications with strict latency requirements. It operates concurrently and uses region-based garbage collection.
- Epsilon GC: Primarily used for performance testing and troubleshooting, as it does not perform any garbage collection. It is not suitable for production environments.
- Azul C4 GC: Offers low and predictable pause times, making it suitable for applications with strict latency requirements. It utilizes a concurrent and compacting garbage collection algorithm.

- IBM Metronome GC: A research-based garbage collector designed for real-time and deterministic applications, providing predictable and short pause times.
- SAP GC: Tailored specifically for SAP applications, it optimizes memory usage, reduces pause times, and improves scalability within the SAP ecosystem.

Java | Garbage Collection | Parallel Computing | Java Heap Memory | Java Gc

Follow